

---

# **Pachyderm Documentation**

***Release 1.9.7***

**Joe Doliner**

**Oct 29, 2019**



<b>1</b>	<b>Getting Started</b>	<b>3</b>
1.1	Local Installation . . . . .	3
1.2	Beginner Tutorial . . . . .	7
<b>2</b>	<b>Concepts</b>	<b>19</b>
2.1	Versioned Data Concepts . . . . .	19
2.2	Pipeline Concepts . . . . .	27
<b>3</b>	<b>Pachyderm Enterprise Edition Guide</b>	<b>41</b>
3.1	Overview . . . . .	41
3.2	Deploying Enterprise Edition . . . . .	44
3.3	Access Controls . . . . .	46
3.4	Advanced Statistics . . . . .	52
3.5	Using the S3 Gateway . . . . .	56
<b>4</b>	<b>Deploy Pachyderm</b>	<b>65</b>
4.1	Overview . . . . .	65
4.2	Getting Started with Pachyderm Hub . . . . .	65
4.3	Google Cloud Platform . . . . .	68
4.4	Deploy Pachyderm on Amazon AWS . . . . .	73
4.5	Azure . . . . .	81
4.6	OpenShift . . . . .	86
4.7	On Premises . . . . .	94
4.8	Create a Custom Pachyderm Deployment . . . . .	99
4.9	Connect to a Pachyderm cluster . . . . .	104
4.10	Deploy Pachyderm with TLS . . . . .	106
4.11	Custom Object Stores . . . . .	107
4.12	Import a Kubernetes Context . . . . .	110
4.13	Non-Default Namespaces . . . . .	111
4.14	RBAC . . . . .	111
4.15	Troubleshooting Deployments . . . . .	113
<b>5</b>	<b>Manage Pachyderm</b>	<b>119</b>
5.1	Manage Cluster Access . . . . .	119
5.2	Autoscaling a Pachyderm Cluster . . . . .	121
5.3	Data Management Best Practices . . . . .	122
5.4	Sharing GPU Resources . . . . .	123
5.5	Backup and Restore . . . . .	124

5.6	Upgrades and Migrations . . . . .	129
5.7	General Troubleshooting . . . . .	129
5.8	Troubleshooting Pipelines . . . . .	131
<b>6</b>	<b>How-Tos</b>	<b>137</b>
6.1	Individual Developer Workflow . . . . .	137
6.2	Team Developer Workflow . . . . .	139
6.3	Load Your Data Into Pachyderm . . . . .	140
6.4	Export Your Data From Pachyderm . . . . .	143
6.5	Split Data . . . . .	145
6.6	Combine, Merge, and Join Data . . . . .	154
6.7	Create a Machine Learning Workflow . . . . .	156
6.8	Run a Pipeline on a Specific Commit . . . . .	157
6.9	Update a Pipeline . . . . .	158
6.10	Processing Time-Windowed Data . . . . .	160
6.11	Delete Data . . . . .	163
6.12	Configure Distributed Computing . . . . .	165
6.13	Deferred Processing of Data . . . . .	166
6.14	Ingress and Egress Data from an External Object Store . . . . .	169
<b>7</b>	<b>Pipeline Specification</b>	<b>171</b>
7.1	JSON Manifest Format . . . . .	171
7.2	The Input Glob Pattern . . . . .	183
7.3	PPS Mounts and File Access . . . . .	184
<b>8</b>	<b>Environment Variables</b>	<b>185</b>
<b>9</b>	<b>Config Specification</b>	<b>187</b>
9.1	JSON format . . . . .	187
<b>10</b>	<b>Pachyderm language clients</b>	<b>189</b>
10.1	Go Client . . . . .	189
10.2	Python Client . . . . .	189
10.3	Scala Client . . . . .	189
10.4	Other languages . . . . .	190
<b>11</b>	<b>S3 Gateway API</b>	<b>191</b>
11.1	Authentication . . . . .	191
11.2	Operations on the Service . . . . .	191
11.3	Operations on Buckets . . . . .	191
11.4	Operations on Objects . . . . .	192
<b>12</b>	<b>Pachctl Command Line Tool</b>	<b>195</b>
12.1	pachctl . . . . .	195
12.2	pachctl auth . . . . .	195
12.3	pachctl auth activate . . . . .	196
12.4	pachctl auth check . . . . .	196
12.5	pachctl auth deactivate . . . . .	197
12.6	pachctl auth get-auth-token . . . . .	197
12.7	pachctl auth get-config . . . . .	197
12.8	pachctl auth get . . . . .	198
12.9	pachctl auth list-admins . . . . .	198
12.10	pachctl auth login . . . . .	199
12.11	pachctl auth logout . . . . .	199
12.12	pachctl auth modify-admins . . . . .	200

12.13	<code>pachctl auth set-config</code>	200
12.14	<code>pachctl auth set</code>	201
12.15	<code>pachctl auth use-auth-token</code>	201
12.16	<code>pachctl auth whoami</code>	201
12.17	<code>pachctl completion</code>	202
12.18	<code>pachctl config</code>	202
12.19	<code>pachctl config delete</code>	203
12.20	<code>pachctl config delete context</code>	203
12.21	<code>pachctl config get</code>	203
12.22	<code>pachctl config get active-context</code>	204
12.23	<code>pachctl config get context</code>	204
12.24	<code>pachctl config get metrics</code>	204
12.25	<code>pachctl config list</code>	205
12.26	<code>pachctl config list context</code>	205
12.27	<code>pachctl config set</code>	205
12.28	<code>pachctl config set active-context</code>	206
12.29	<code>pachctl config set context</code>	206
12.30	<code>pachctl config set metrics</code>	207
12.31	<code>pachctl config update</code>	207
12.32	<code>pachctl config update context</code>	207
12.33	<code>pachctl copy</code>	208
12.34	<code>pachctl copy file</code>	208
12.35	<code>pachctl create</code>	209
12.36	<code>pachctl create branch</code>	209
12.37	<code>pachctl create pipeline</code>	209
12.38	<code>pachctl create repo</code>	210
12.39	<code>pachctl debug</code>	211
12.40	<code>pachctl debug binary</code>	211
12.41	<code>pachctl debug dump</code>	211
12.42	<code>pachctl debug pprof</code>	212
12.43	<code>pachctl debug profile</code>	212
12.44	<code>pachctl delete</code>	212
12.45	<code>pachctl delete all</code>	213
12.46	<code>pachctl delete branch</code>	213
12.47	<code>pachctl delete commit</code>	214
12.48	<code>pachctl delete file</code>	214
12.49	<code>pachctl delete job</code>	214
12.50	<code>pachctl delete pipeline</code>	215
12.51	<code>pachctl delete repo</code>	215
12.52	<code>pachctl delete transaction</code>	216
12.53	<code>pachctl deploy</code>	216
12.54	<code>pachctl deploy amazon</code>	217
12.55	<code>pachctl deploy custom</code>	219
12.56	<code>pachctl deploy export-images</code>	221
12.57	<code>pachctl deploy google</code>	222
12.58	<code>pachctl deploy import-images</code>	224
12.59	<code>pachctl deploy list-images</code>	225
12.60	<code>pachctl deploy local</code>	227
12.61	<code>pachctl deploy microsoft</code>	228
12.62	<code>pachctl deploy storage</code>	230
12.63	<code>pachctl deploy storage amazon</code>	231
12.64	<code>pachctl deploy storage google</code>	233
12.65	<code>pachctl deploy storage microsoft</code>	234
12.66	<code>pachctl diff</code>	235

12.67	pachctl diff file	236
12.68	pachctl edit	237
12.69	pachctl edit pipeline	237
12.70	pachctl enterprise	237
12.71	pachctl enterprise activate	238
12.72	pachctl enterprise get-state	238
12.73	pachctl extract	239
12.74	pachctl extract pipeline	239
12.75	pachctl finish	240
12.76	pachctl finish commit	240
12.77	pachctl finish transaction	240
12.78	pachctl flush	241
12.79	pachctl flush commit	241
12.80	pachctl flush job	242
12.81	pachctl fsck	243
12.82	pachctl garbage-collect	243
12.83	pachctl get	244
12.84	pachctl get file	244
12.85	pachctl get object	245
12.86	pachctl get tag	245
12.87	pachctl glob	246
12.88	pachctl glob file	246
12.89	pachctl inspect	247
12.90	pachctl inspect cluster	247
12.91	pachctl inspect commit	247
12.92	pachctl inspect datum	248
12.93	pachctl inspect file	248
12.94	pachctl inspect job	249
12.95	pachctl inspect pipeline	249
12.96	pachctl inspect repo	250
12.97	pachctl inspect transaction	250
12.98	pachctl list	251
12.99	pachctl list branch	251
12.100	pachctl list commit	252
12.101	pachctl list datum	252
12.102	pachctl list file	253
12.103	pachctl list job	254
12.104	pachctl list pipeline	255
12.105	pachctl list repo	255
12.106	pachctl list transaction	256
12.107	pachctl logs	256
12.108	pachctl mount	257
12.109	pachctl port-forward	258
12.110	pachctl put	258
12.111	pachctl put file	259
12.112	pachctl restart	260
12.113	pachctl restart datum	261
12.114	pachctl restore	261
12.115	pachctl resume	262
12.116	pachctl resume transaction	262
12.117	pachctl run	262
12.118	pachctl run pipeline	263
12.119	pachctl start	263
12.120	pachctl start commit	264

12.121	pachctl start pipeline . . . . .	264
12.122	pachctl start transaction . . . . .	265
12.123	pachctl stop . . . . .	265
12.124	pachctl stop job . . . . .	265
12.125	pachctl stop pipeline . . . . .	266
12.126	pachctl stop transaction . . . . .	266
12.127	pachctl subscribe . . . . .	267
12.128	pachctl subscribe commit . . . . .	267
12.129	pachctl undeploy . . . . .	268
12.130	pachctl unmount . . . . .	268
12.131	pachctl update-dash . . . . .	269
12.132	pachctl update . . . . .	269
12.133	pachctl update pipeline . . . . .	270
12.134	pachctl update repo . . . . .	270
12.135	pachctl version . . . . .	271
<b>13</b>	<b>Examples</b>	<b>273</b>
13.1	OpenCV Edge Detection . . . . .	273
13.2	Word Count (Map/Reduce) . . . . .	273
13.3	Periodic Ingress from a Database . . . . .	273
13.4	Lazy Shuffle pipeline . . . . .	273
13.5	Variant Calling and Joint Genotyping with GATK . . . . .	274
13.6	Machine Learning . . . . .	274
<b>14</b>	<b>Setup for contributors</b>	<b>277</b>
14.1	General requirements . . . . .	277
14.2	Bash helpers . . . . .	277
14.3	Special macOS configuration . . . . .	277
14.4	Dev cluster . . . . .	278
14.5	pachctl . . . . .	278
14.6	Fully resetting . . . . .	279
<b>15</b>	<b>Gcloud cluster setup</b>	<b>281</b>
15.1	First steps . . . . .	281
15.2	gcloud . . . . .	281
15.3	kubectrl . . . . .	282
15.4	Pachyderm cluster deployment . . . . .	282
<b>16</b>	<b>Repo layout</b>	<b>283</b>
<b>17</b>	<b>Coding Conventions</b>	<b>287</b>
17.1	Shell . . . . .	287
17.2	Go . . . . .	287





Welcome to the Pachyderm documentation portal! Below you find guides and information for beginners and experienced Pachyderm users, as well as the Pachyderm API reference docs.

If you cannot find what you are looking for or have an issue that is not mentioned here, we'd love to hear from you either on [GitHub](#), in our [Users Slack channel](#), or by email at [support@pachyderm.io](mailto:support@pachyderm.io).

---

**Note:** If you are using a Pachyderm version 1.4 or earlier, you can find relevant documentation published in [our old documentation portal](#).

---



---

## Getting Started

---

Welcome to the documentation portal for first-time Pachyderm users! This page is a great place to start using Pachyderm. Complete the sections below to get a taste of what working with Pachyderm feels like.

### 1.1 Local Installation

This guide walks you through the steps to install Pachyderm on macOS®, Linux®, or Windows®. Local installation helps you to learn some of the Pachyderm basics and is not designed to be a production environment.

**Note:** Pachyderm supports the Docker runtime only. If you want to deploy Pachyderm on a system that uses another container runtime, ask for advice in our [Slack channel](#).

#### 1.1.1 Prerequisites

Before you can deploy Pachyderm, make sure you have the following programs installed on your computer:

- *Minikube*
- Oracle® VirtualBox™ or *Docker Desktop (v18.06+)*
- *Pachyderm Command Line Interface*

If you install Pachyderm on Windows 10 or later, you must have the following components installed in addition to the ones listed above:

- [Windows Subsystem for Linux \(WSL\)](#)

**Note:** For a Windows installation, use Minikube.

#### Using Minikube

On your local machine, you can run Pachyderm in a minikube virtual machine. Minikube is a tool that creates a single-node Kubernetes cluster. This limited installation is sufficient to try basic Pachyderm functionality and complete the Beginner Tutorial.

To configure Minikube, follow these steps:

1. Install minikube and VirtualBox in your operating system as described in the [Kubernetes documentation](#).
2. `Install kubectl`.
3. `Start minikube`:

```
minikube start
```

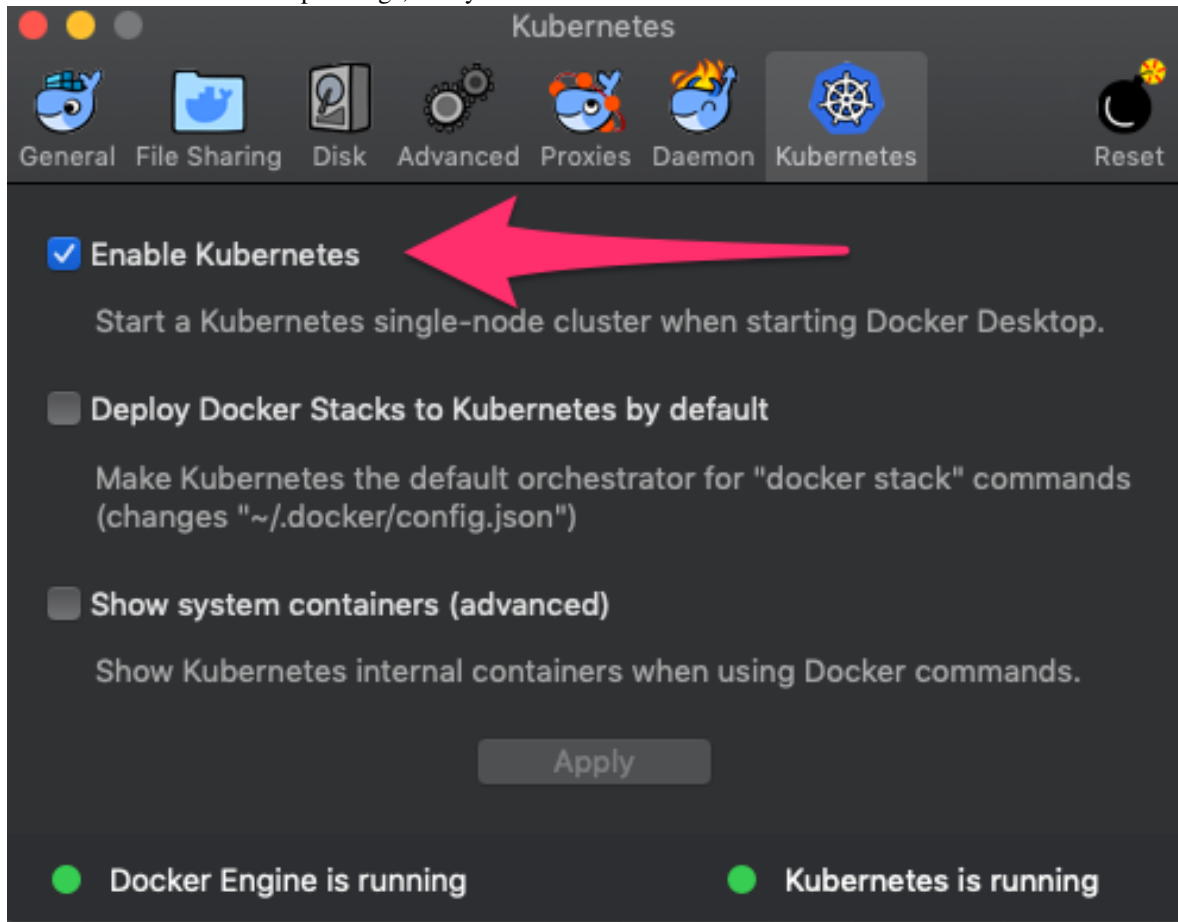
**Note:** Any time you want to stop and restart Pachyderm, run `minikube delete` and `minikube start`. Minikube is not meant to be a production environment and does not handle being restarted well without a full wipe.

## Docker Desktop

If you use Minikube, skip this section and proceed to [Install pachctl](#)

You can use Docker Desktop instead of Minikube on macOS or Linux by following these steps:

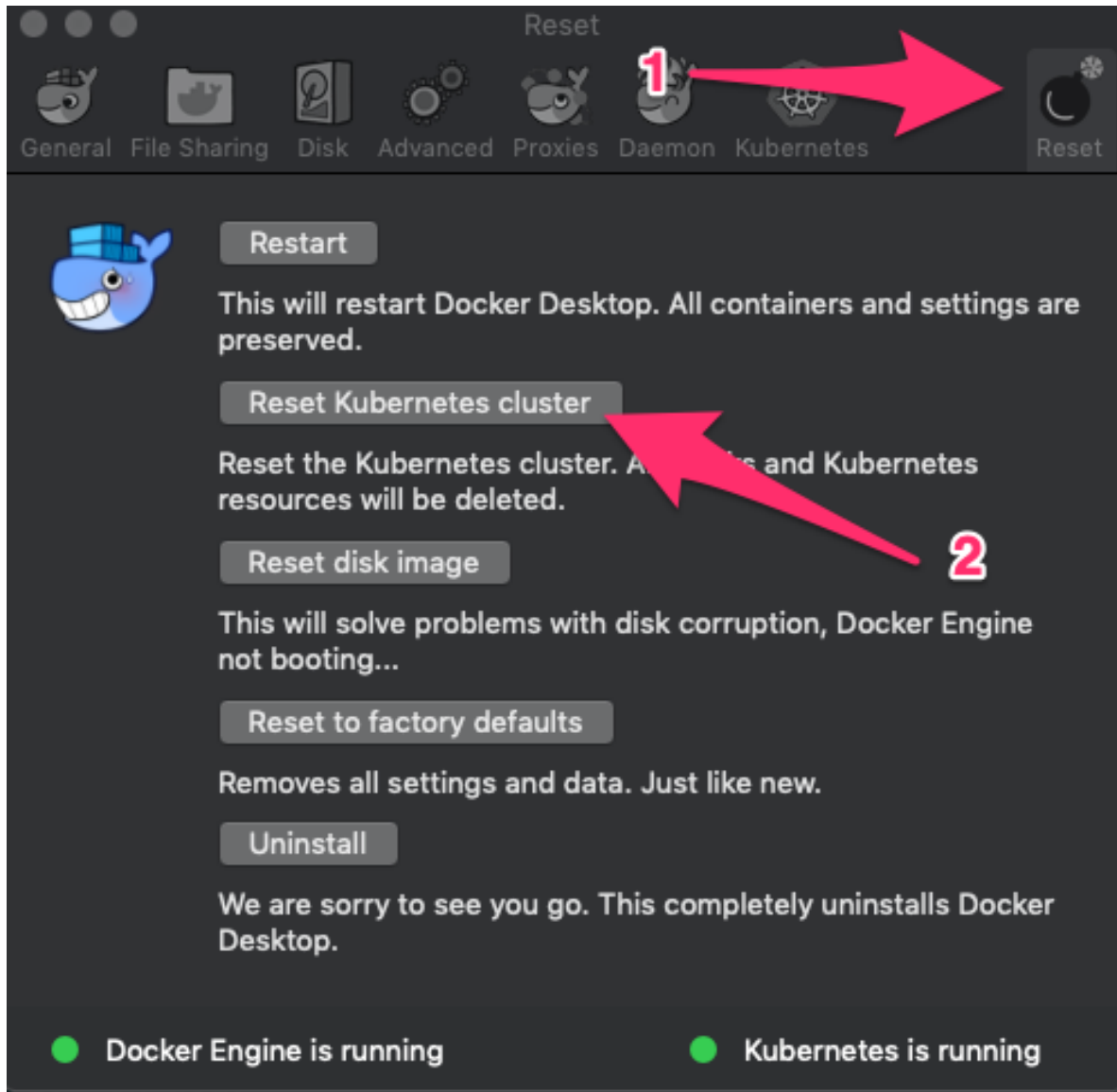
1. In the Docker Desktop settings, verify that Kubernetes is enabled:



2. From the command prompt, confirm that Kubernetes is running:

```
$ kubectl get all
NAME                TYPE          CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
service/kubernetes  ClusterIP     10.96.0.1     <none>       443/TCP    56d
```

- To reset your Kubernetes cluster that runs on Docker Desktop, click the **Reset** button in the **Preferences** sub-menu.



### Install pachctl

`pachctl` is a command-line utility that you can use to interact with a Pachyderm cluster.

To deploy Pachyderm locally, you need to have `pachctl` installed on your machine by following these steps:

1. Run the corresponding steps for your operating system:

- For macOS, run:

```
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.9
```

- For a Debian-based Linux 64-bit or Windows 10 or later running on WSL:

```
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/download/v1.9.7/pachctl_1.9.7_amd64.deb && sudo dpkg -i /tmp/pachctl.deb
```

- For all other Linux flavors:

```
$ curl -o /tmp/pachctl.tar.gz -L https://github.com/pachyderm/pachyderm/  
↪releases/download/v1.9.7/pachctl_1.9.7_linux_amd64.tar.gz && tar -xvf /tmp/  
↪pachctl.tar.gz -C /tmp && sudo cp /tmp/pachctl_1.9.7_linux_amd64/pachctl /  
↪usr/local/bin
```

2. Verify that installation was successful by running `pachctl version --client-only`:

```
$ pachctl version --client-only  
COMPONENT      VERSION  
pachctl         1.9.5
```

If you run `pachctl version` without `--client-only`, the command times out. This is expected behavior because `pachd` is not yet running.

### 1.1.2 Deploy Pachyderm

After you configure all the *Prerequisites*, deploy Pachyderm by following these steps:

- For macOS or Linux, run:

```
$ pachctl deploy local
```

This command generates a Pachyderm manifest and deploys Pachyderm on Kubernetes.

- For Windows:

1. Start WSL.
2. In WSL, run:

```
$ pachctl deploy local --dry-run > pachyderm.json
```

3. Copy the `pachyderm.json` file into your Pachyderm directory.
4. From the same directory, run:

```
kubectl create -f .\pachyderm.json
```

Because Pachyderm needs to pull the Pachyderm Docker image from DockerHub, it might take a few minutes for the Pachyderm pods status to change to `Running`.

1. Check the status of the Pachyderm pods by periodically running `kubectl get pods`. When Pachyderm is ready for use, all Pachyderm pods must be in the **Running** status.

```
$ kubectl get pods  
NAME                                READY    STATUS    RESTARTS    AGE  
dash-6c9dc97d9c-vb972              2/2      Running   0            6m  
etcd-7dbb489f44-9v5jj              1/1      Running   0            6m  
pachd-6c878bbc4c-f2h2c              1/1      Running   0            6m
```

**Note:** If you see a few restarts on the `pachd` nodes, that means that Kubernetes tried to bring up those pods before `etcd` was ready. Therefore, Kubernetes restarted those pods. You can safely ignore that message.

1. Run `pachctl version` to verify that `pachd` has been deployed.

```
$ pachctl version  
COMPONENT      VERSION  
pachctl         1.9.5  
pachd           1.9.5
```

2. Open a new terminal window.
3. Use port forwarding to access the Pachyderm dashboard.

```
$ pachctl port-forward
```

This command runs continuously and does not exit unless you interrupt it.

4. Alternatively, you can set up Pachyderm to directly connect to the Minikube instance:
  - (a) Get your Minikube IP address:

```
$ minikube ip
```

- (b) Configure Pachyderm to connect directly to the Minikube instance:

```
$ pachctl config update context pachctl config get active-context --pachd-address=minikube
ip:30650
```

### 1.1.3 Next Steps

After you install and configure Pachyderm, continue exploring Pachyderm:

- Complete the Beginner Tutorial to learn the basics of Pachyderm, such as adding data and building analysis pipelines.
- Explore the Pachyderm Dashboard. By default, Pachyderm deploys the Pachyderm Enterprise dashboard. You can use a FREE trial token to experiment with the dashboard. Point your browser to port 30080 on your minikube IP. Alternatively, if you cannot connect directly, enable port forwarding by running `pachctl port-forward`, and then point your browser to `localhost:30080`.

**See Also:**

- General Troubleshooting

## 1.2 Beginner Tutorial

Welcome to the beginner tutorial for Pachyderm! This tutorial takes about 15 minutes, and introduces you to the basic concepts of Pachyderm.

### 1.2.1 Image processing with OpenCV

This tutorial walks you through the deployment of a Pachyderm pipeline to do simple [edge detection](#) on a few images. Thanks to Pachyderm's built-in processing primitives, we'll be able to keep our code simple but still run the pipeline in a distributed, streaming fashion. Moreover, as new data is added, the pipeline will automatically process it and output the results.

If you hit any errors not covered in this guide, get help in our [public community Slack](#), submit an issue on [GitHub](#), or email us at [support@pachyderm.io](mailto:support@pachyderm.io). We are more than happy to help!

### Prerequisites

You must already have Pachyderm running locally, on a cloud platform of your choice, or in Pachyderm Hub.

If you have not yet installed Pachyderm, follow the instructions in one of these sections:

- Getting Started with Pachyderm Hub
- Deploy Pachyderm Locally
- Deploy Pachyderm in a Cloud Platform

### Create a Repo

A `repo` is the highest level data primitive in Pachyderm. Like many things in Pachyderm, it shares its name with a primitive in Git and is designed to behave analogously. Generally, repos should be dedicated to a single source of data such as log messages from a particular service, a users table, or training data for an ML model. Repos are dirt cheap so don't be shy about making tons of them.

For this demo, we'll simply create a repo called `images` to hold the data we want to process:

```
$ pachctl create repo images
$ pachctl list repo
NAME      CREATED          SIZE (MASTER)
images 7 seconds ago 0B
```

This shows that the repo has been successfully created, and the size of repo's HEAD commit on the master branch is 0B, since we haven't added anything to it yet.

### Adding Data to Pachyderm

Now that we've created a repo it's time to add some data. In Pachyderm, you write data to an explicit `commit` (again, similar to Git). Commits are immutable snapshots of your data which give Pachyderm its version control properties. Files can be added, removed, or updated in a given commit.

Let's start by just adding a file, in this case an image, to a new commit. We've provided some sample images for you that we host on Imgur.

We'll use the `put file` command along with the `-f` flag. `-f` can take either a local file, a URL, or a object storage bucket which it'll automatically scrape. In our case, we'll simply pass the URL.

Unlike Git, commits in Pachyderm must be explicitly started and finished as they can contain huge amounts of data and we don't want that much "dirty" data hanging around in an unpersisted state. `put file` automatically starts and finishes a commit for you so you can add files more easily. If you want to add many files over a period of time, you can do `start commit` and `finish commit` yourself.

We also specify the repo name "images", the branch name "master", and the file name: "liberty.png".

Here's an example atomic commit of the file `liberty.png` to the `images` repo's `master` branch:

```
$ pachctl put file images@master:liberty.png -f http://imgur.com/46Q8nDz.png
```

We can check to make sure the data we just added is in Pachyderm.

```
# If we list the repos, we can see that there is now data
$ pachctl list repo
NAME      CREATED          SIZE (MASTER)
images About a minute ago 57.27KiB
```



```
# We can view the commit we just created
$ pachctl list commit images
REPO    COMMIT                                PARENT STARTED      DURATION      SIZE
images d89758a7496a4c56920b0eaa7d7d3255 <none> 29 seconds ago Less than a second 57.
→27KiB

# And view the file in that commit
$ pachctl list file images@master
COMMIT                                NAME                TYPE COMMITTED      SIZE
d89758a7496a4c56920b0eaa7d7d3255 /liberty.png file About a minute ago 57.27KiB
```

We can also view the file we just added to Pachyderm. Since this is an image, we can't just print it out in the terminal, but the following commands will let you view it easily.

```
# on macOS
$ pachctl get file images@master:liberty.png | open -f -a /Applications/Preview.app

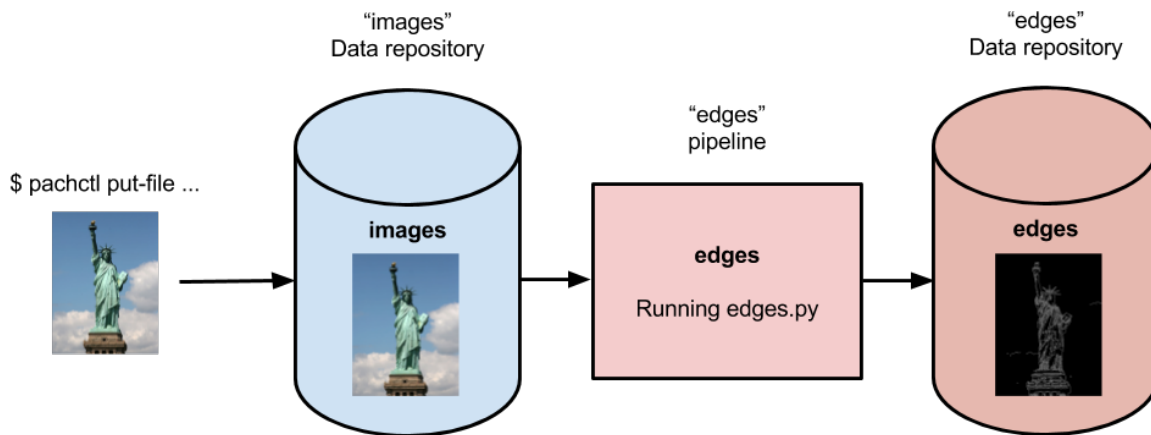
# on Linux
$ pachctl get file images@master:liberty.png | display
```

## Create a Pipeline

Now that we've got some data in our repo, it's time to do something with it. Pipelines are the core processing primitive in Pachyderm and they're specified with a JSON encoding. For this example, we've already created the pipeline for you and you can find the [code on Github](#).

When you want to create your own pipelines later, you can refer to the full [Pipeline Specification](#) to use more advanced options. Options include building your own code into a container instead of the pre-built Docker image we'll be using here.

For now, we're going to create a single pipeline that takes in images and does some simple edge detection.



Below is the pipeline spec and python code we're using. Let's walk through the details.

```
# edges.json
{
  "pipeline": {
    "name": "edges"
  },
  "transform": {
```

```
{
  "cmd": [ "python3", "/edges.py" ],
  "image": "pachyderm/opencv"
},
"input": {
  "pfs": {
    "repo": "images",
    "glob": "/*"
  }
}
}
```

Our pipeline spec contains a few simple sections. First is the pipeline name , edges. Then we have the transform which specifies the docker image we want to use, pachyderm/opencv (defaults to DockerHub as the registry), and the entry point edges.py . Lastly, we specify the input. Here we only have one PFS input, our images repo with a particular glob pattern.

The glob pattern defines how the input data can be broken up if we want to distribute our computation. /\* means that each file can be processed individually, which makes sense for images. Glob patterns are one of the most powerful features of Pachyderm so when you start creating your own pipelines, check out the *Pipeline Specification*.

```
# edges.py
import cv2
import numpy as np
from matplotlib import pyplot as plt
import os

# make_edges reads an image from /pfs/images and outputs the result of running
# edge detection on that image to /pfs/out. Note that /pfs/images and
# /pfs/out are special directories that Pachyderm injects into the container.
def make_edges(image):
    img = cv2.imread(image)
    tail = os.path.split(image)[1]
    edges = cv2.Canny(img,100,200)
    plt.imsave(os.path.join("/pfs/out", os.path.splitext(tail)[0]+'.png'), edges, cmap_
    => 'gray')

# walk /pfs/images and call make_edges on every file found
for dirpath, dirs, files in os.walk("/pfs/images"):
    for file in files:
        make_edges(os.path.join(dirpath, file))
```

We simply walk over all the images in /pfs/images , do our edge detection, and write to /pfs/out .

/pfs/images and /pfs/out are special local directories that Pachyderm creates within the container automatically. All the input data for a pipeline will be found in /pfs/<input\_repo\_name> and your code should always write out to /pfs/out . Pachyderm will automatically gather everything you write to /pfs/out and version it as this pipeline's output.

Now let's create the pipeline in Pachyderm:

```
$ pachctl create pipeline -f https://raw.githubusercontent.com/pachyderm/pachyderm/
    => master/examples/opencv/edges.json
```

## What Happens When You Create a Pipeline

Creating a pipeline tells Pachyderm to run your code on the data in your input repo (the HEAD commit) as well as **all future commits** that occur after the pipeline is created. Our repo already had a commit, so Pachyderm automatically

launched a `job` to process that data.

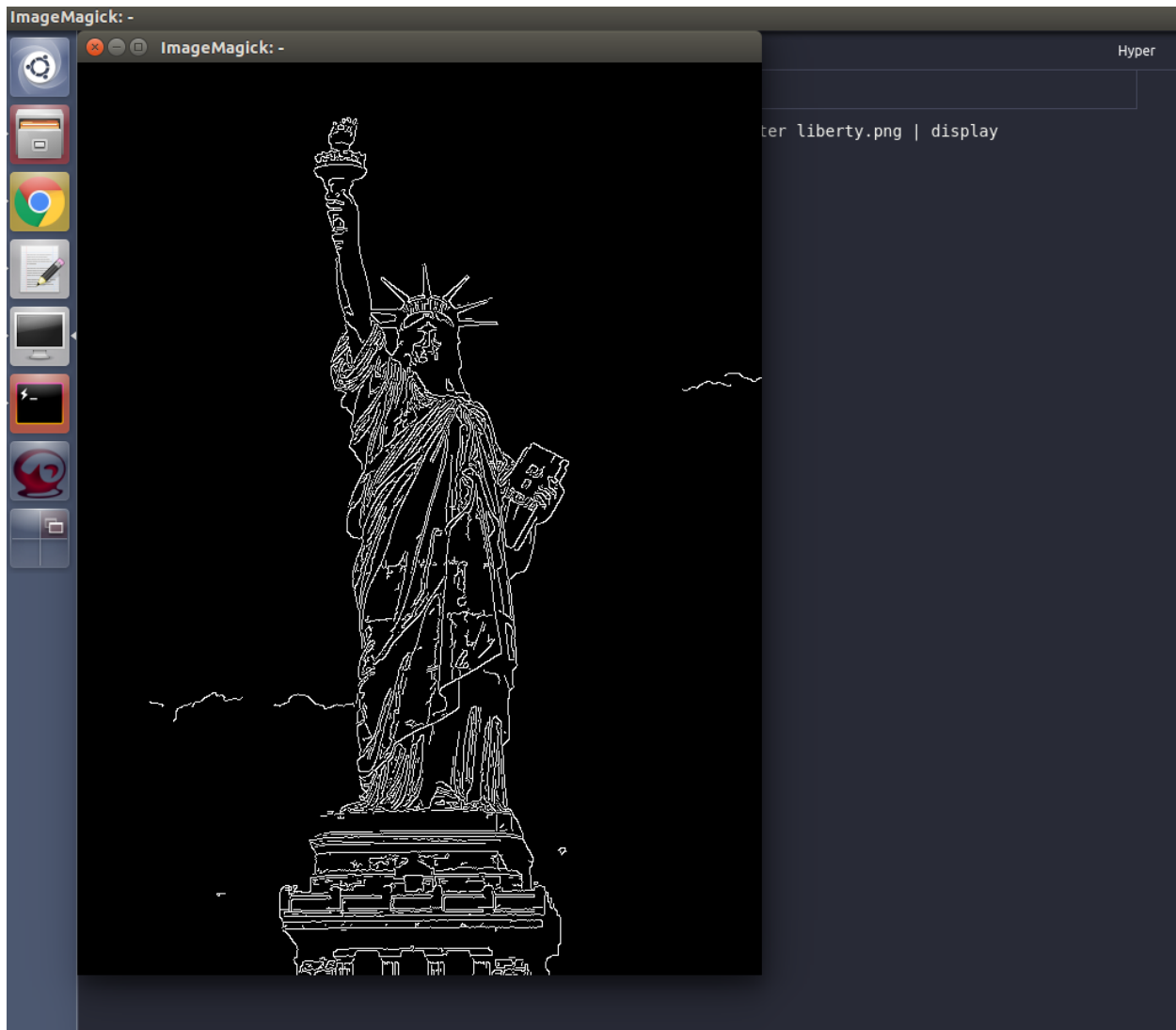
The first time Pachyderm runs a pipeline job, it needs to download the Docker image (specified in the pipeline spec) from the specified Docker registry (DockerHub in this case). This first run this might take a minute or so because of the image download, depending on your Internet connection. Subsequent runs will be much faster.

You can view the job with:

```
$ pachtctl list job
```

ID	PIPELINE	STARTED	DURATION	RESTART
0f6a53829eeb4ca193bb7944fe693700	edges	16 seconds ago	Less than a second	0

```
↪ + 0 / 1 57.27KiB 22.22KiB success
```



## Processing More Data

Pipelines will also automatically process the data from new commits as they are created. Think of pipelines as being subscribed to any new commits on their input repo(s). Also similar to Git, commits have a parental structure that tracks which files have changed. In this case we're going to be adding more images.

Let's create two new commits in a parental structure. To do this we will simply do two more `put file` commands and by specifying `master` as the branch, it'll automatically parent our commits onto each other. Branch names are just references to a particular HEAD commit.

```
$ pachctl put file images@master:AT-AT.png -f http://imgur.com/8MN9Kg0.png
$ pachctl put file images@master:kitten.png -f http://imgur.com/g2QnNqa.png
```

Adding a new commit of data will automatically trigger the pipeline to run on the new data we've added. We'll see corresponding jobs get started and commits to the output "edges" repo. Let's also view our new outputs.

```
# view the jobs that were kicked off
$ pachctl list job
```

ID	DL	UL	STATE	STARTED	DURATION	RESTART	PROGRESS
81ae47a802f14038b95f8f248cddbed2	↪DL			7 seconds ago	Less than a second	0	1 + 2 / 3
ce448c12d0dd4410b3a5ae0c0f07e1f9	↪102.4KiB	74.21KiB	success	16 seconds ago	Less than a second	0	1 + 1 / 2
490a28be32de491e942372018cd42460	↪78.7KiB	37.15KiB	success	9 minutes ago	35 seconds	0	1 + 0 / 1
	↪57.27KiB	22.22KiB	success				

```
# View the output data

# on macOS
$ pachctl get file edges@master:AT-AT.png | open -f -a /Applications/Preview.app

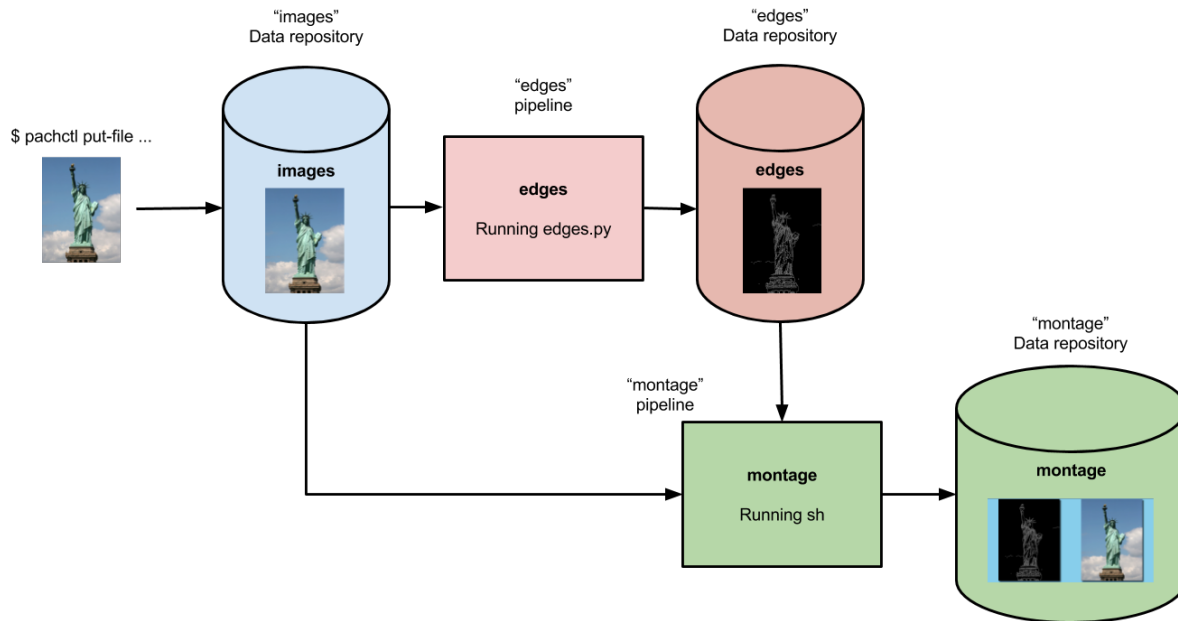
$ pachctl get file edges@master:kitten.png | open -f -a /Applications/Preview.app

# on Linux
$ pachctl get file edges@master:AT-AT.png | display

$ pachctl get file edges@master:kitten.png | display
```

## Adding Another Pipeline

We have successfully deployed and used a single stage Pachyderm pipeline. Now let's add a processing stage to illustrate a multi-stage Pachyderm pipeline. Specifically, let's add a `montage` pipeline that take our original and edge detected images and arranges them into a single montage of images:



Below is the pipeline spec for this new pipeline:

```
# montage.json
{
  "pipeline": {
    "name": "montage"
  }
}
```

```

},
"input": {
  "cross": [ {
    "pfs": {
      "glob": "/",
      "repo": "images"
    }
  },
  {
    "pfs": {
      "glob": "/",
      "repo": "edges"
    }
  }
]
},
"transform": {
  "cmd": [ "sh" ],
  "image": "v4tech/imagemagick",
  "stdin": [ "montage -shadow -background SkyBlue -geometry 300x300+2+2 $(find /pfs_
↪-type f | sort) /pfs/out/montage.png" ]
}
}

```

This `montage` pipeline spec is similar to our `edges` pipeline except for three differences: (1) we are using a different Docker image that has `imagemagick` installed, (2) we are executing a `sh` command with `stdin` instead of a python script, and (3) we have multiple input data repositories.

In the `montage` pipeline we are combining our multiple input data repositories using a `cross` pattern. This `cross` pattern creates a single pairing of our input images with our edge detected images. There are several interesting ways to combine data in Pachyderm, which are discussed [here](#) and [here](#).

We create the `montage` pipeline as before, with `pachctl`:

```

$ pachctl create pipeline -f https://raw.githubusercontent.com/pachyderm/pachyderm/
↪master/examples/opencv/montage.json

```

Pipeline creating triggers a job that generates a montage for all the current HEAD commits of the input repos:

```

$ pachctl list job
ID                               STARTED          DURATION          RESTART_
↪PROGRESS DL          UL          STATE
92cecc40c3144fd5b4e07603bb24b104 45 seconds ago 6 seconds          0          1 + 0 /_
↪1 371.9KiB 1.284MiB success
81ae47a802f14038b95f8f248cddb2 2 minutes ago Less than a second 0          1 + 2 /_
↪3 102.4KiB 74.21KiB success
ce448c12d0dd4410b3a5ae0c0f07e1f9 2 minutes ago Less than a second 0          1 + 1 /_
↪2 78.7KiB 37.15KiB success
490a28be32de491e942372018cd42460 11 minutes ago 35 seconds          0          1 + 0 /_
↪1 57.27KiB 22.22KiB success

```

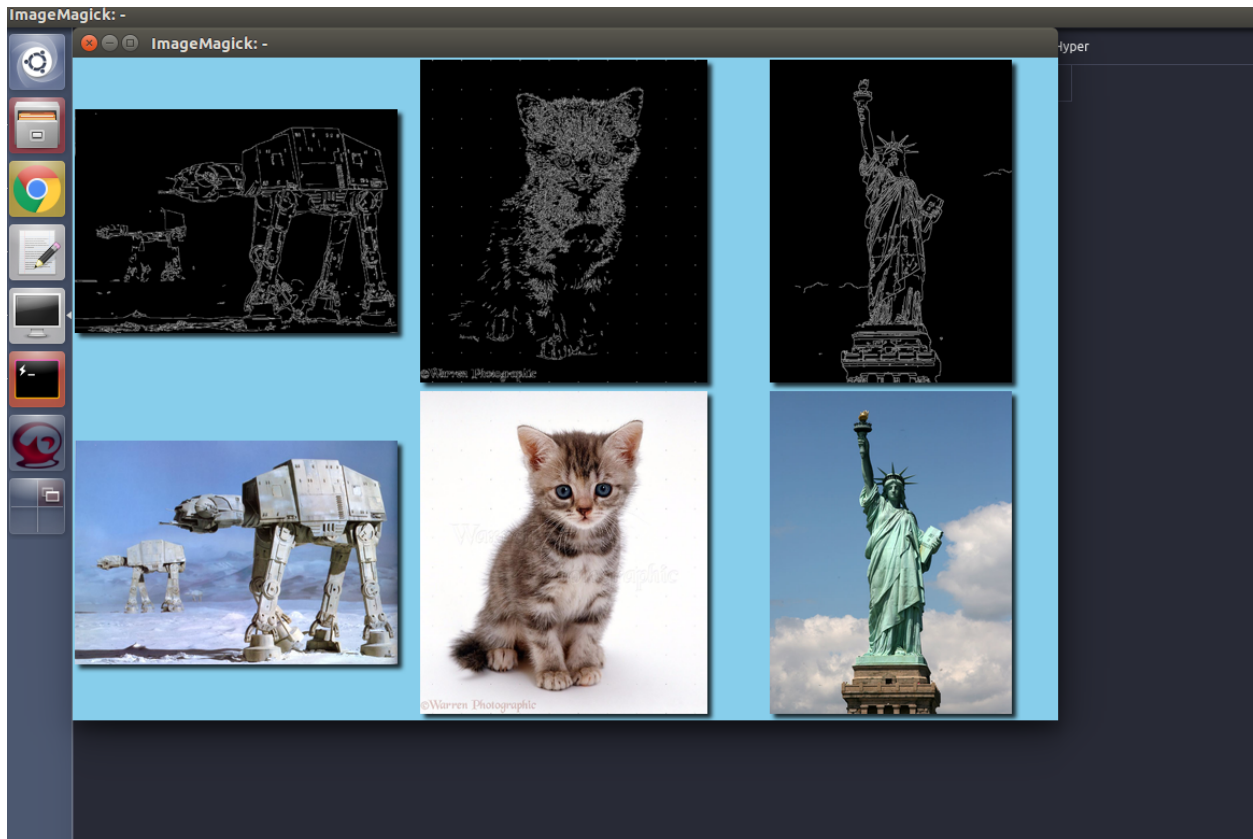
And you can view the generated montage image via:

```

# on macOS
$ pachctl get file montage@master:montage.png | open -f -a /Applications/Preview.app

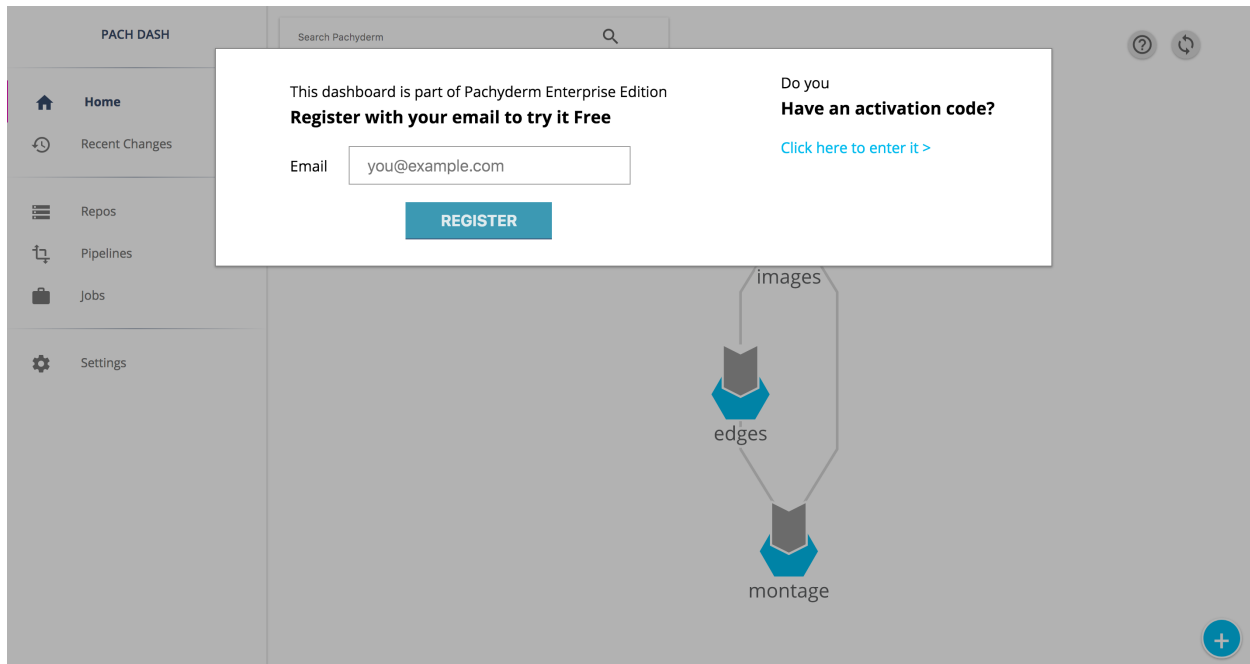
# on Linux
$ pachctl get file montage@master:montage.png | display

```

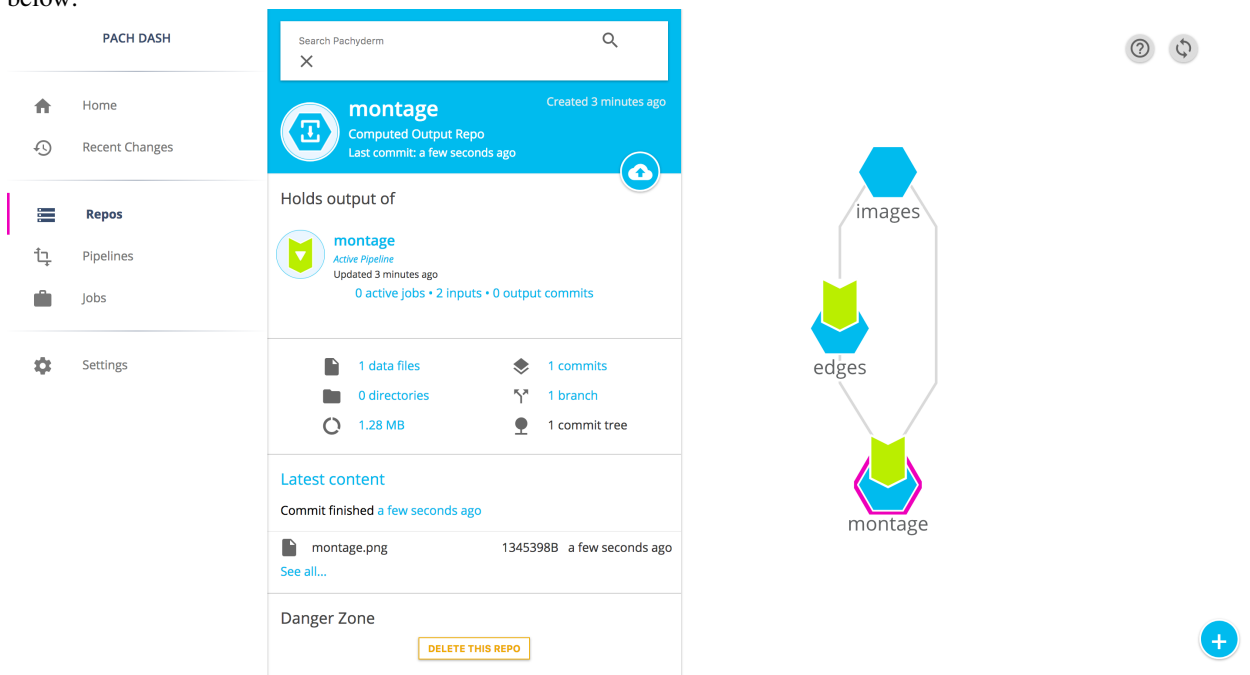


### 1.2.2 Exploring your DAG in the Pachyderm dashboard

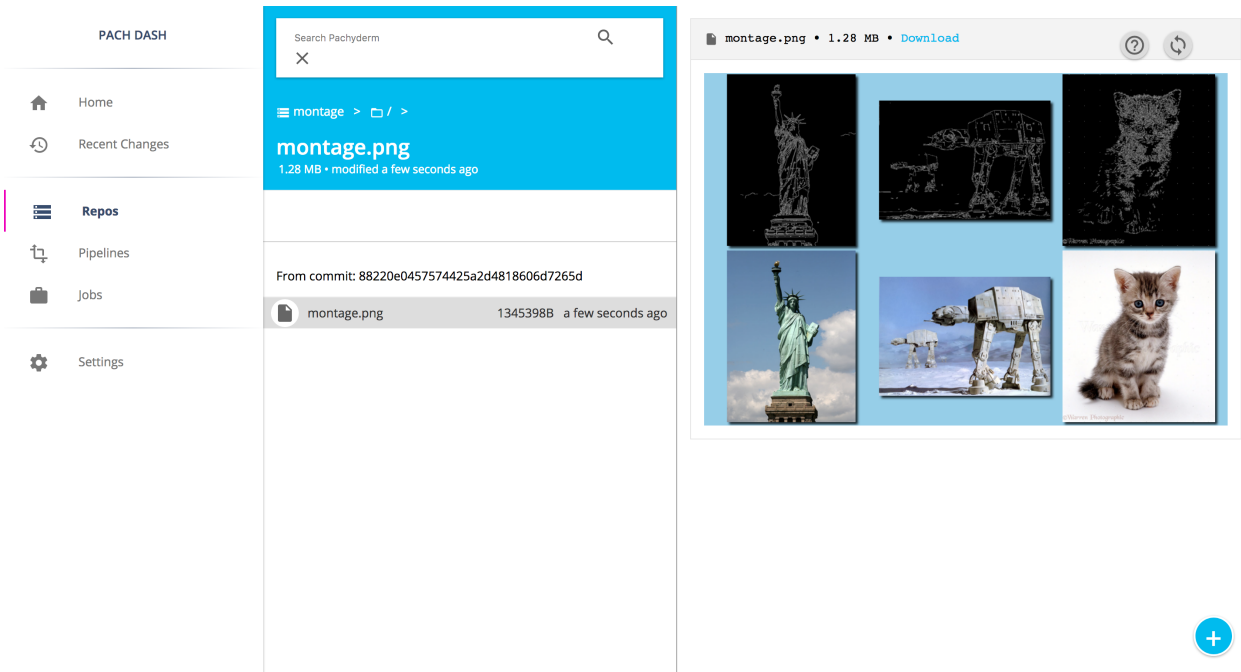
When you deployed Pachyderm locally, the Pachyderm Enterprise dashboard was also deployed by default. This dashboard will let you interactively explore your pipeline, visualize the structure of the pipeline, explore your data, debug jobs, etc. To access the dashboard visit `localhost:30080` in an Internet browser (e.g., Google Chrome). You should see something similar to this:



Enter your email address if you would like to obtain a free trial token for the dashboard. Upon entering this trial token, you will be able to see your pipeline structure and interactively explore the various pieces of your pipeline as pictured below:







### 1.2.3 Next Steps

Pachyderm is now running locally with data and a pipeline! To play with Pachyderm locally, you can use what you've learned to build on or change this pipeline. You can also dig in and learn more details about:

- [Deploying Pachyderm to the cloud or on prem](#)
- [Load Your Data Into Pachyderm](#)
- [../how-tos/working-with-pipelines](#)

We'd love to help and see what you come up with, so submit any issues/questions you come across on [GitHub](#), [Slack](#), or email at [support@pachyderm.io](mailto:support@pachyderm.io) if you want to show off anything nifty you've created!



---

## Concepts

---

Pachyderm is an enterprise-grade, open-source data science platform that makes explainable, repeatable, and scalable Machine Learning (ML) and Artificial Intelligence (AI) a reality. The Pachyderm platform brings together version control for data with the tools to build scalable end-to-end ML/AI pipelines while empowering users to develop their code in any language, framework, or tool of their choice. Pachyderm has been proven to be the ideal foundation for teams looking to use ML and AI to solve real-world problems reliably.

The Pachyderm platform includes the following main components:

- Pachyderm File System (PFS)
- Pachyderm pipelines

To start, you need to understand the foundational concepts of Pachyderm data versioning and pipeline semantics. After you have a good grasp of the basics, you can use advanced concepts and features for more complicated challenges.

This section describes the following Pachyderm concepts:

### 2.1 Versioned Data Concepts

Pachyderm data concepts describe version control primitives that you interact with when you use Pachyderm.

These concepts are similar to the Git version control system with a few notable exceptions. Because Pachyderm deals not only with plain text but also with binary files and large datasets, it does not manage the data in the same way as Git. When you use Git, you store a copy of the repository on your local machine. You work with that copy, apply your changes, and then send the changes to the upstream master copy of the repository where it gets merged.

The Pachyderm version control works slightly differently. In Pachyderm, only a centralized repository exists, and you do not store any local copies of that repository. Therefore, the merge, in the traditional Git meaning, does not occur.

Instead, your data can be continuously updated in the master branch of your repo, while you can experiment with specific data commits in a separate branch or branches. Because of this behavior, you cannot run into a merge conflict with Pachyderm.

The Pachyderm data versioning system has the following main concepts:

**Repository** A Pachyderm repository is the highest level data object. Typically, each dataset in Pachyderm is its own repository.

**Commit** A commit is an immutable snapshot of a repo at a particular point in time.

**Branch** A branch is an alias to a specific commit, or a pointer, that automatically moves as new data is submitted.

**File** Files and directories are actual data in your repository. Pachyderm supports any type, size, and number of files.

**Provenance** Provenance expresses the relationship between various commits, branches, and repositories. It helps you to track the origin of each commit.

Learn more about Pachyderm data concepts in the following sections:

### 2.1.1 Repository

A Pachyderm repository is a location where you store your data inside Pachyderm. A Pachyderm repository is a top-level data object that contains files and folders. Similar to Git, a Pachyderm repository tracks all changes to the data and creates a history of data modifications that you can access and review. You can store any type of file in a Pachyderm repo, including binary and plain text files.

Unlike a Git repository that stores history in a `.git` file in your copy of a Git repo, Pachyderm stores the history of your commits in a centralized location. Because of that, you do not run into merge conflicts as you often do with Git commits when you try to merge your `.git` history with the master copy of the repo. With large datasets resolving a merge conflict might not be possible.

A Pachyderm repository is the first entity that you configure when you want to add data to Pachyderm. You can create a repository by running the `pachctl create repo` command or by using the Pachyderm UI. After creating the repository, you can add your data by using the `pachctl put file` command.

The following types of repositories exist in Pachyderm:

- **Input repositories** Users or external applications outside of Pachyderm can add data to the input repositories for further processing.
- **Output repositories** Pachyderm automatically creates output repositories pipelines write results of computations into these repositories. Any data that is written to the `pfs/out` directory within your pipeline user container is written to that pipeline output repository.

You can view the list of repositories in your Pachyderm cluster by running the `pachctl list repo` command.

**Example:**

```
$ pachctl list repo
NAME          CREATED          SIZE (MASTER)
raw_data      6 hours ago     0B
```

The `pachctl inspect repo` command provides a more detailed overview of a specified repository.

**Example:**

```
$ pachctl inspect repo raw_data
Name: raw_data
Description: A raw data repository
Created: 6 hours ago
Size of HEAD on master: 5.121MiB
```

If you need to delete a repository, you can run the `pachctl delete` command. This command deletes all data and the information about the specified repository, such as commit history. The delete operation is irreversible and results in a complete cleanup of your Pachyderm repository. If you run the delete command with the `--all` flag, Pachyderm deletes all repositories in this cluster.

**See Also:**

- Pipelines

## 2.1.2 Commit

A commit is a snapshot that preserves the state of your data at a point in time. It represents a single set of changes to files or directories in your Pachyderm repository. A commit is a user-defined operation, which means that you can start a commit, make changes, and then close the commit after you are done.

Each commit has a unique identifier (ID) that you can reference in the `<repo>@<commitID or branch>` format. When you create a new commit, the previous commit on which the new commit is based becomes the parent of the new commit.

You can obtain information about commits in a repository by running `pachctl list commit <repo>` or `pachctl inspect commit <commitID>`. In Pachyderm, commits are atomic operations that capture a state of the files and directories in a repository. Unlike Git commits, Pachyderm commits are centralized and transactional. You can start a commit by running the `pachctl start commit` command, make changes to the repository, and close the commit by running the `pachctl finish commit` command. After the commit is finished, Pachyderm saves the new state of the repository.

When you *start*, or *open*, a commit, it means that you can make changes by using `put file`, `delete file`, or other commands. You can *finish*, or *close* a commit, which means the commit is immutable and cannot be changed.

The `pachctl list commit repo@branch` command. This command returns a timestamp, size, parent, and other information about the commit. The initial commit has `<none>` as a parent.

### Example:

```
pachctl list commit images@master
REPO  BRANCH COMMIT                                PARENT
↳STARTED      DURATION      SIZE
raw_data master 8248d97632874103823c7603fb8c851c 22cdb5ae05cb40868566586140ea5ed5 6
↳seconds ago   Less than a second 5.121MiB
raw_data master 22cdb5ae05cb40868566586140ea5ed5 <none> 33
↳minutes ago   Less than a second 2.561MiB
```

The `list commit <repo>` command displays all commits in all branches in the specified repository.

The `pachctl inspect commit` command enables you to view detailed information about a commit, such as the size, parent, and the original branch of the commit, as well as how long ago the commit was started and finished. The `--full-timestamps` flag, enables you to see the exact date and time of when the commit was opened and when it was finished. If you specify a branch instead of a specific commit, Pachyderm displays the information about the HEAD of the branch. For most commands, you can specify either a branch or a commit ID.

The most important information that the `pachctl inspect commit` command provides the origin of the commit, or its provenance. Typically, provenance can be tracked for the commits in the output repositories.

### Example:

```
$ pachctl inspect commit edges@master
Commit: edges@234fd5ea60ae4422b075f1945786ec5f
Original Branch: master
Parent: c5c9849ebb5849dc8bf37c4a925e3b20
Started: 12 seconds ago
Finished: 7 seconds ago
Size: 22.22KiB
Provenance: images@e55ab0f1c44544ecb93151e11867c6b5 (master) __spec__
↳@ede9d05de2584108b03593301f1fdf81 (edges)
```

The `delete commit` command enables you to delete opened and closed commits, which results in permanent loss of all the data *introduced* in those commits. The `delete commit` command makes it as the deleted commit never happened. If the deleted commit was the HEAD of the branch, its parent becomes the HEAD. You can only delete a commit from an input repository at the top of your commit history, also known as DAG. Deleting a commit in the

middle of a DAG breaks the provenance chain. When you delete a commit from the top of your DAG, Pachyderm automatically deletes all the commits that were created in downstream output repos by processing the deleted commit, making it as if the commit never existed.

Commit deletion is an irreversible operation that should be used with caution. An alternative and a much safer way to revert incorrect data changes is to move the HEAD of the branch or create a new commit that removes the incorrect data.

### Example:

```
$ pachctl delete commit raw_data@8248d97632874103823c7603fb8c851c
```

### See also:

- Provenance

## 2.1.3 Branch

A Pachyderm branch is a pointer, or an alias, to a commit that moves along with new commits as they are submitted. By default, when you create a repository, Pachyderm does not create any branches. Most users prefer to create a `master` branch by initiating the first commit and specifying the `master` branch in the `put file` command. Also, you can create additional branches to experiment with the data. Branches enable collaboration between teams of data scientists. However, many users find it sufficient to use the `master` branch for all their work. Although the concept of the branch is similar to Git branches, in most cases branches are not used as extensively as in source code version-control systems.

Each branch has a `HEAD` which references the latest commit in the branch. Pachyderm pipelines look at the `HEAD` of the branch for changes and, if they detect new changes, trigger a job. When you commit a new change, the `HEAD` of the branch moves to the latest commit.

To view a list of branches in a repo, run the `pachctl list branch` command.

### Example:

```
pachctl list branch images
BRANCH HEAD
master bb41c5fb83a14b69966a21c78a3c3b24
```

## 2.1.4 File

A file is a Unix filesystem object, which is a directory or file, that stores data. Unlike source code version-control systems that are most suitable for storing plain text files, you can store any type of file in Pachyderm, including binary files. Often, data scientists operate with comma-separated values (CSV), JavaScript Object Notation (JSON), images, and other plain text and binary file formats. Pachyderm supports all file sizes and formats and applies storage optimization techniques, such as deduplication, in the background.

To upload your files to a Pachyderm repository, run the `pachctl put file` command. By using the `pachctl put file` command, you can put both files and directories into a Pachyderm repository.

### Appending and overwriting

When you add a file by using the `pachctl put file` command, Pachyderm can either append the file to the already existing file or overwrite the existing file.

**Appending files** By default, when you put a file into a Pachyderm repository, and a file by the same name already exists in the repo, Pachyderm appends the new data to the existing file. For example, you have an `A.csv` file in a

repository. If you upload the same file to that repository, Pachyderm *appends* the data to the existing file, which results in the `A.csv` file having twice the data from its original size.

#### Example:

1. View the list of files:

```
$ pachctl list file images@master
NAME    TYPE SIZE
/A.csv  file 258B
```

2. Add the `A.csv` file once again:

```
$ pachctl put file images@master -f A.csv
```

3. Verify that the file has doubled in size:

```
$ pachctl list file images@master
NAME    TYPE SIZE
/A.csv  file 516B
```

**Overwriting files** When you enable the overwrite mode by using the `--overwrite` flag or `-o`, the file replaces the existing file instead of appending to it. For example, you have an `A.csv` file in the `images` repository. If you upload the same file to that repository with the `--overwrite` flag, Pachyderm *overwrites* the whole file.

#### Example:

1. View the list of files:

```
$ pachctl list file images@master
NAME    TYPE SIZE
/A.csv  file 258B
```

2. Add the `A.csv` file once again:

```
$ pachctl put file --overwrite images@master -f A.csv
```

3. Check the file size:

```
$ pachctl list file images@master
NAME    TYPE SIZE
/A.csv  file 258B
```

## 2.1.5 Provenance

Data versioning enables Pachyderm users to go back in time and see the state of a dataset or repository at a particular moment in time. Data provenance (from the French *provenir*, which means *the place of origin*), also known as data lineage, tracks the dependencies and relationships *between* datasets. Provenance answers not only the question of where the data comes from, but also how the data was transformed along the way. Data scientists use provenance in root cause analysis to improve their code, workflows, and understanding of the data and its implications on final results. Data scientists need to have confidence in the information with which they operate. They need to be able to reproduce the results and sometimes go through the whole data transformation process from scratch multiple times, which makes data provenance one of the most critical aspects of data analysis. If your computations result in unexpected numbers, the first place to look is the historical data that gives insights into possible flaws in the transformation chain or the data itself.

For example, when a bank decides on a mortgage application, many factors are taken into consideration, including the credit history, annual income, and loan size. This data goes through multiple automated steps of analysis with

numerous dependencies and decisions made along the way. If the final decision does not satisfy the applicant, the historical data is the first place to look for proof of authenticity, as well as for possible prejudice or model bias against the applicant. Data provenance creates a complete audit trail that enables data scientists to track the data from its origin through to the final decision and make appropriate changes that address issues. With the adoption of the General Data Protection Regulation (GDPR) compliance requirements, monitoring data lineage is becoming a necessity for many organizations that work with sensitive data.

Pachyderm implements provenance for both commits and repositories. You can track revisions of the data and understand the connection between the data stored in one repository and the results in the other repository.

Collaboration takes data provenance even further. Provenance enables teams of data scientists across the globe to build on each other work, share, transform, and update datasets while automatically maintaining a complete audit trail so that all results are reproducible.

The following diagram demonstrates how provenance works:

In the diagram above, you can see two input repositories called `params` and `data`. The `data` repository continuously collects data from an outside source. The training model pipeline combines the data from the first repository with the parameters in the second repository, runs them through the pipeline code, and collects the results in the output repo.

Provenance helps you to understand where commits in the output repo originates in. For example, in the diagram above, you can see that the commit `3b` was created from the commit `1b` from the `data` repository and the commit `2a` in the `params` repository. Similar, the commit `3a` was created from the commit `1a` from the `data` repository and the commit `2a` from the `params` repository.

### Tracking the Provenance Upstream

Pachyderm provides the `pachctl inspect commit` command that enables you to track the provenance of your commits and learn where the data in the repository originated.

#### Example:

```
$ pachctl inspect commit split@master
Commit: split@f71e42704b734598a89c02026c8f7d13
Original Branch: master
Started: 4 minutes ago
Finished: 3 minutes ago
Size: 0B
Provenance: __spec__@8c6440f52a2d4aa3980163e25557b4a1 (split) raw_
↳data@ccf82debb4b94ca3bfe165aca8d517c3 (master)
```

In the example above, you can see that the latest commit in the master branch of the `split` repository tracks back to the master branch in the `raw_data` repository. The `__spec__` provenance shows you which version of your code was run on the input commit `ccf82debb4b94ca3bfe165aca8d517c3` in the `raw_data` repository to produce the output commit `f71e42704b734598a89c02026c8f7d13` in the `split` repository.

### Tracking the Provenance Downstream

Pachyderm provides the `flush commit` command that enables you to track the provenance downstream. Tracking downstream means that instead of tracking the origin of a commit, you can learn in which output repository a particular input has resulted.

For example, you have the `ccf82debb4b94ca3bfe165aca8d517c3` commit in the `raw_data` repository. If you run the `pachctl flush commit` command for this commit, you can see in which repositories and commits that data resulted.



```
$ pachctl flush commit raw_data@ccf82debb4b94ca3bfe165aca8d517c3
REPO          BRANCH COMMIT          PARENT STARTED          DURATION
↪ SIZE
split         master f71e42704b734598a89c02026c8f7d13 <none> 52 minutes ago About a
↪minute 25B
split         stats  9b46d7abf9a74bf7bf66c77f2a0da4b1 <none> 52 minutes ago About a
↪minute 15.39MiB
pre_process master a99ab362dc944b108fb33544b2b24a8c <none> 48 minutes ago About a
↪minute 100B
```

## 2.1.6 History

Pachyderm implements rich version-control and history semantics. This section describes the core concepts and architecture of Pachyderm’s version control and the various ways to use the system to access historical data.

The following abstractions store the history of your data:

- **Commits**

In Pachyderm, commits are the core version-control primitive that is similar to Git commits. Commits represent an immutable snapshot of a filesystem and can be accessed with an ID. Commits have a parentage structure, where new commits inherit content from their parents. You can think of this parentage structure as of a linked list or a *chain of commits*. Commit IDs are useful if you want to have a static pointer to a snapshot of a filesystem. However, because they are static, their use is limited. Instead, you mostly work with branches.

- **Branches**

Branches are pointers to commits that are similar to Git branches. Typically, Branches have semantically meaningful names such as `master` and `staging`. Branches are mutable, and they move along a growing chain of commits as you commit to the branch, and can even be reassigned to any commit within the repo by using the `pachctl create branch` command. The commit that a branch points to is referred to as the branches *head*, and the head’s ancestors are referred to as *on the branch*. Branches can be substituted for commits in Pachyderm’s API and behave as if the head of the branch were passed. This allows you to deal with semantically meaningful names for commits that can be updated, rather than static opaque identifiers.

## Ancestry Syntax

Pachyderm’s commits and branches support a familiar Git syntax for referencing their history. A commit or branch parent can be referenced by adding a `^` to the end of the commit or branch. Similar to how `master` resolves to the head commit of `master`, `master^` resolves to the parent of the head commit. You can add multiple `^`s. For example, `master^^` resolves to the parent of the parent of the head commit of `master`, and so on. Similarly, `master^3` has the same meaning as `master^^^`.

Git supports two characters for ancestor references—`^` and `~`—with slightly different meanings. Pachyderm supports both characters as well, but their meaning is identical.

Also, Pachyderm also supports a type of ancestor reference that Git does not—forward references, these use a different special character `.` and resolve to commits on the beginning of commit chains. For example, `master.1` is the first (oldest) commit on the `master` branch, `master.2` is the second commit, and so on.

Resolving ancestry syntax requires traversing chains of commits high numbers passed to `^` and low numbers passed to `.`. These operations require traversing a large number of commits which might take a long time. If you plan to repeatedly access an ancestor, you might want to resolve that ancestor to a static commit ID with `pachctl inspect commit` and use that ID for future accesses.

## View the Filesystem Object History

Pachyderm enables you to view the history of filesystem objects by using the `--history` flag with the `pachctl list file` command. This flag takes a single argument, an integer, which indicates how many historical versions you want to display. For example, you can get the two most recent versions of a file with the following command:

```
$ pachctl list file repo@master:/file --history 2
COMMIT                                NAME      TYPE  COMMITTED      SIZE
73ba56144be94f5bad1ce64e6b96eade /file file  16 seconds ago 8B
c5026f053a7f482fbd719dadecec8f89 /file file  21 seconds ago 4B
```

This command might return a different result from if you run `pachctl list file repo@master:/file` followed by `pachctl list file repo@master^:/file`. The history flag looks for changes to the file, and the file might not be changed with every commit. Similar to the ancestry syntax above, because the history flag requires traversing through a linked list of commits, this operation can be expensive. You can get back the full history of a file by passing `all` to the history flag.

### Example:

```
$ pachctl list file edges@master:liberty.png --history all
COMMIT                                NAME      TYPE  COMMITTED      SIZE
ff479f3a639344daa9474e729619d258 /liberty.png file  23 hours ago 22.22KiB
```

## View the Pipeline History

Pipelines are the main processing primitive in Pachyderm. However, they expose version-control and history semantics similar to filesystem objects. This is largely because, under the hood, they are implemented in terms of filesystem objects. You can access previous versions of a pipeline by using the same ancestry syntax that works for commits and branches. For example, `pachctl inspect pipeline foo^` gives you the previous version of the pipeline `foo`. The `pachctl inspect pipeline foo.1` command returns the first ever version of that same pipeline. You can use this syntax in all operations and scripts that accept pipeline names.

To view historical versions of a pipeline use the `--history` flag with the `pachctl list pipeline` command:

```
$ pachctl list pipeline --history all
NAME      VERSION INPUT      CREATED      STATE / LAST JOB
Pipeline2 1      input2:/* 4 hours ago running / success
Pipeline1 3      input1:/* 4 hours ago running / success
Pipeline1 2      input1:/* 4 hours ago running / success
Pipeline1 1      input1:/* 4 hours ago running / success
```

A common operation with pipelines is reverting a pipeline to a previous. To revert a pipeline to a previous version, run the following command:

```
$ pachctl extract pipeline pipeline^ | pachctl create pipeline
```

## View the Job History

Jobs do not have versioning semantics associated with them. However, they are strongly associated with the pipelines that created them. Therefore, they inherit some of their versioning semantics. You can use the `-p <pipeline>` flag with the `pachctl list job` command to list all the jobs that were run for the latest version of the pipeline. To view a previous version of a pipeline you can add the caret symbol to the end of the pipeline name. For example `-p edges^`.

Furthermore you can get jobs from multiple versions of pipelines by passing the `--history` flag. For example, `pachctl list job --history all` returns all jobs from all versions of all pipelines.

To view job history, run the following command:

- By using the `-p` flag:

```
$ pachctl list job -p <pipeline^>
```

- By using the `history` flag:

```
$ pachctl list job --history all
```

## 2.2 Pipeline Concepts

Pachyderm Pipeline System (PPS) is the computational component of the Pachyderm platform that enables you to perform various transformations on your data. Pachyderm pipelines have the following main concepts:

**Pipeline** A pipeline is a job-spawner that waits for certain conditions to be met. Most commonly, this means watching one or more Pachyderm repositories for new commits of data. When new data arrives, a pipeline executes a user-defined piece of code to perform an operation and process the data. Each of these executions is called a job.

Pachyderm has the following special types of pipelines:

**Cron** A cron input enables you to trigger the pipeline code at a specific interval. This type of pipeline is useful for such tasks as web scraping, querying a database, and other similar operations where you do not want to wait for new data, but instead trigger the pipeline periodically.

**Join** A join pipeline enables you to join files that are stored in different Pachyderm repositories and match a particular file path pattern. Conceptually, joins are similar to the database's inner join operations, although they only match on file paths, not the actual file content.

**Service** A service is a special type of pipeline that instead of executing jobs and then waiting, permanently runs a serving data through an endpoint. For example, you can be serving an ML model or a REST API that can be queried. A service reads data from Pachyderm but does not have an output repo.

**Spout** A spout is a special type of pipeline for ingesting data from a data stream. A spout can subscribe to a message stream, such as Kafka or Amazon SQS, and ingest data when it receives a message. A spout does not have an input repo.

**Job** A job is an individual execution of a pipeline. A job can succeed or fail. Within a job, data and processing can be broken up into individual units of work called datums.

**Datum** A datum is the smallest indivisible unit of work within a job. Different datums can be processed in parallel within a job.

Read the sections below to learn more about these concepts:

### 2.2.1 Pipeline

A pipeline is a Pachyderm primitive that is responsible for reading data from a specified source, such as a Pachyderm repo, transforming it according to the pipeline configuration, and writing the result to an output repo. A pipeline subscribes to a branch in one or more input repositories. Every time the branch has a new commit, the pipeline executes a job that runs your code to completion and writes the results to a commit in the output repository. Every pipeline automatically creates an output repository by the same name as the pipeline. For example, a pipeline named `model` writes all results to the `model` output repo.

In Pachyderm, a Pipeline is an individual execution step. You can chain multiple pipelines together to create a directed acyclic graph (DAG).

A minimum pipeline specification must include the following parameters:

- `name` — The name of your data pipeline. Set a meaningful name for your pipeline, such as the name of the transformation that the pipeline performs. For example, `split` or `edges`. Pachyderm automatically creates an output repository with the same name. A pipeline name must be an alphanumeric string that is less than 63 characters long and can include dashes and underscores. No other special characters allowed.
- `input` — A location of the data that you want to process, such as a Pachyderm repository. You can specify multiple input repositories and set up the data to be combined in various ways. For more information, see [Cross](#) and [Union](#).

One very important property that is defined in the `input` field is the `glob` pattern that defines how Pachyderm breaks the data into individual processing units, called *datums*. For more information, see [Datum](#).

- `transform` — Specifies the code that you want to run against your data. The `transform` section must include an `image` field that defines the Docker image that you want to run, as well as a `cmd` field for the specific code within the container that you want to execute, such as a Python script.

**Example:**

```
{
  "pipeline": {
    "name": "wordcount"
  },
  "transform": {
    "image": "wordcount-image",
    "cmd": ["python3", "/my_python_code.py"]
  },
  "input": {
    "pfs": {
      "repo": "data",
      "glob": "/*"
    }
  }
}
```

Also, Pachyderm provides special types of pipelines that allow for other use cases, such as running periodically, serving an endpoint, or automatically ingesting data from a message stream.

You can configure the following special types of pipelines:

**Cron**

Pachyderm triggers pipelines when new changes appear in the input repository. However, if you want to trigger a pipeline based on time instead of upon arrival of input data, you can schedule such pipelines to run periodically by using the built-in `cron` input type.

Cron inputs are well suited for a variety of use cases, including the following:

- Scraping websites
- Making API calls
- Querying a database
- Retrieving a file from a location accessible through an S3 protocol or a File Transfer Protocol (FTP).

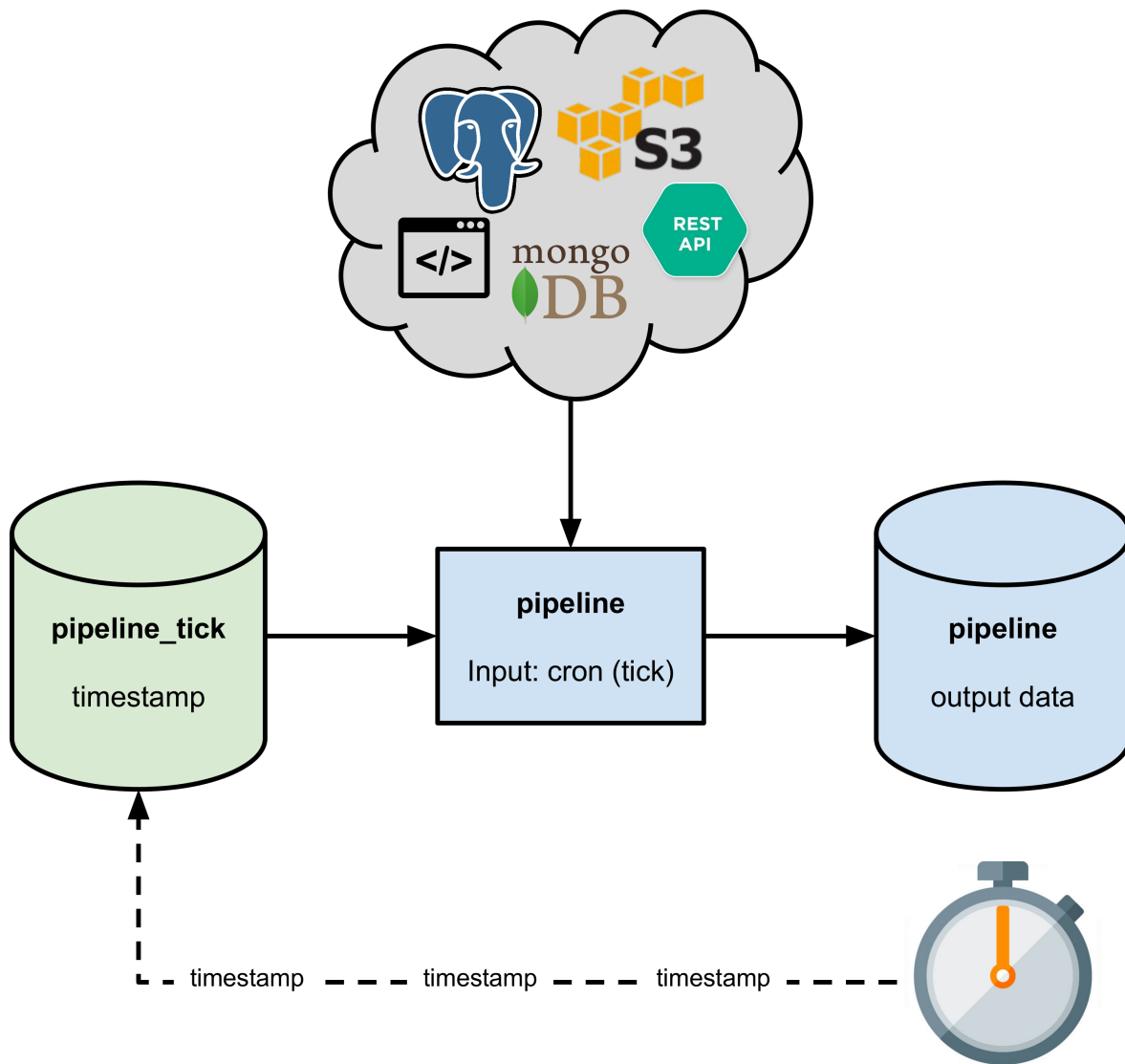
A minimum cron pipeline must include the following parameters:

## Example of a Cron Pipeline

For example, you want to query a database every ten seconds and update your dataset with the new data every time the pipeline is triggered. The following pipeline extract illustrates how you can specify this configuration:

```
"input": {
  "cron": {
    "name": "tick",
    "spec": "@every 10s"
  }
}
```

When you create this pipeline, Pachyderm creates a new input data repository that corresponds to the `cron` input. Then, Pachyderm automatically commits a timestamp file to the `cron` input repository every ten seconds, which triggers the pipeline.



The pipeline runs every ten seconds, queries the database, and updates its output. By default, each cron trigger adds a new tick file to the cron input repository, accumulating more datums over time. This behavior works for some

pipelines. For others, you can set the `--overwrite` flag to `true` to overwrite the timestamp file on each tick. To learn more about overwriting files within a commit as opposed to adding a new file, and how that affects the datums in the subsequent jobs, see [Datum processing](#)

**Example:**

```
"input": {
  "cron": {
    "name": "tick",
    "spec": "@every 10s",
    "overwrite": true
  }
}
```

**See Also:**

- [Periodic Ingress from MongoDB](#)

## Join

A join is a special type of pipeline input that enables you to combine files that reside in separate Pachyderm repositories and match a particular naming pattern. The join operator must be used in combination with a glob pattern that reflects a specific naming convention.

By analogy, a Pachyderm join is similar to a database *equi-join*, or *inner join* operation, but it matches on file paths only, not the contents of the files.

Unlike the cross input, which creates datums from every combination of files in each input repository, joins only create datums where there is a *match*. You can use joins to combine data from different Pachyderm repositories and ensure that only specific files from each repo are processed together.

When you configure a join input, you must specify a glob pattern that includes a capture group. The capture group defines the specific string in the file path that is used to match files in the other joined repos. Capture groups work analogously to the [regex capture group](#). You define the capture group inside parenthesis. Capture groups are numbered from left to right and can also be nested within each other. Numbering for nested capture groups is based on their opening parenthesis.

Below you can find a few examples of applying a glob pattern with a capture group to a file path. For example, if you have the following file path:

```
/foo/bar-123/ABC.txt
```

The following glob patterns in a joint input create the following capture groups:

Also, joins require you to specify a [replacement group](#) in the a `join_on` parameter to define which capture groups you want to try to match.

For example, `$1` indicates that you want Pachyderm to match based on capture group 1. Similarly, `$2` matches the capture group 2. `$1$2` means that it must match both capture groups 1 and 2.

If Pachyderm does not find any matching files, you get a zero-datum job.

You can test your glob pattern and capture groups by using the `pachctl glob file` command as described in [Glob Pattern](#).

## Example

For example, you have two repositories. One with sensor readings and the other with parameters. The repositories have the following structures:

- readings repo:

```
-- ID1234
  -- file1.txt
  -- file2.txt
  -- file3.txt
  -- file4.txt
  -- file5.txt
```

- parameters repo:

```
-- file1.txt
-- file2.txt
-- file3.txt
-- file4.txt
-- file5.txt
-- file6.txt
-- file7.txt
-- file8.txt
```

Pachyderm runs your code on only the pairs of files that match the glob pattern and capture groups.

The following example shows how you can use joins to group matching IDs:

```
{
  "pipeline": {
    "name": "joins"
  },
  "input": {
    "join": [
      {
        "pfs": {
          "repo": "readings",
          "branch": "master",
          "glob": "/*/(*) .txt",
          "join_on": "$1"
        }
      },
      {
        "pfs": {
          "repo": "parameters",
          "branch": "master",
          "glob": "/(*) .txt",
          "join_on": "$1"
        }
      }
    ]
  },
  "transform": {
    "cmd": [ "python3", "/joins.py" ],
    "image": "joins-example"
  }
}
```

The glob pattern for the readings repository, `/*/(*)`, indicates all matching files in the ID sub-directory. In the parameters repository, the glob pattern `/(*)` selects all the matching files in the root directory. All files with indices from 1 to 5 match. The files with indices from 6 to 8 do not match. Therefore, you only get five datums for this job.

To experiment further, see the full [joins example](#).

## Service

Service is a special type of pipeline that does not process data but provides a capability to expose it to the outside world. For example, you can use a service to serve a machine learning model as an API that has the most up-to-date version of your data.

The following pipeline spec extract is an example of how you can expose your Jupyter notebook as a service by adding a `service` field:

```
{
  "input": {
    "pfs": {
      "glob": "/",
      "repo": "input"
    }
  },
  "service": {
    "external_port": 30888,
    "internal_port": 8888
  },
  "transform": {
    "cmd": [
      "start-notebook.sh"
    ],
    "image": "jupyter/datascience-notebook"
  }
}
```

The service section specifies the following parameters:

### See Also

- Service

## Spout

A spout is a type of pipeline that ingests streaming data. Generally, you use spouts for situations when the interval between new data generation is large or sporadic, but the latency requirement to start the processing is short. Therefore, a regular pipeline with a cron input that polls for new data might not be an optimal solution.

Examples of streaming data include a message queue, a database transactions log, event notifications, and others. In spouts, your code runs continuously and writes the results to the pipeline's output location, `pfs/out`. Every time you create a complete `.tar` archive, Pachyderm creates a new commit and triggers the pipeline to process it.

One main difference from regular pipelines is that spouts ingest their data from outside sources. Therefore, they do not take an input.

Another important aspect is that in spouts, `pfs/out` is a *named pipe*, or *First in, First Out* (FIFO), and is not a directory like in standard pipelines. Unlike the traditional pipe, that is familiar to most Linux users, a *named pipe* enables two system processes to access the pipe simultaneously and gives one of the processes read-only and the other process write-only access. Therefore, the two processes can simultaneously read and write to the same pipe.

To create a spout pipeline, you need the following items:

- A source of streaming data
- A Docker container with your spout code that reads from the data source



- A spout pipeline specification file that uses the container

Your spout code performs the following actions:

1. Connects to the specified streaming data source.
2. Opens `/pfs/out` as a named pipe.
3. Reads the data from the streaming data source.
4. Packages the data into a `tar` stream.
5. Writes the `tar` stream into the `pfs/out` pipe. In case of transient errors produced by closing a previous write to the pipe, retries the write operation.
6. Closes the `tar` stream and connection to `/pfs/out`, which produces the commit.

A minimum spout specification must include the following parameters:

The following text is an example of a minimum specification:

**Note:** The `env` property is an optional argument. You can define your data stream source from within the container in which you run your script. For simplicity, in this example, `env` specifies the source of the Kafka host.

```
{
  "pipeline": {
    "name": "my-spout"
  },
  "transform": {
    "cmd": [ "go", "run", "./main.go" ],
    "image": "myaccount/myimage:0.1"
    "env": {
      "HOST": "kafkahost",
      "TOPIC": "mytopic",
      "PORT": "9092"
    },
  },
  "spout": {
    "overwrite": false
  }
}
```

#### See also

- Pipeline Specification

## 2.2.2 Job

A Pachyderm job is an execution of a pipeline that triggers when new data is detected in an input repository. Each job runs your code against the current commit and then submits the results to the output repository and creates a single output commit. A pipeline triggers a new job every time you submit new changes, a commit, into your input source.

Each job has the following stages:

## 2.2.3 Datum

A datum is the smallest indivisible unit of computation within a job. A job can have one, many, or no datums. Each datum is processed independently with a single execution of the user code, and then the results of all the datums are merged together to create the final output commit.

The number of datums for a job is defined by the glob pattern which you specify for each input. Think of datums as if you were telling Pachyderm how to divide your input data to efficiently distribute computation and only process the *new* data. You can configure a whole input repository to be one datum, each top-level filesystem object to be a separate datum, specific paths can be datums, and so on. Datums affect how Pachyderm distributes processing workloads and are instrumental in optimizing your configuration for best performance.

Pachyderm takes each datum and processes it in isolation on one of the pipeline worker nodes. You can define datums, workers, and other performance parameters through the corresponding fields in the pipeline specification.

To understand how datums affect data processing in Pachyderm, you need to understand the following subconcepts:

## Glob Pattern

Defining how your data is spread among workers is one of the most important aspects of distributed computation and is the fundamental idea around concepts such as Map and Reduce.

Instead of confining users to data-distribution patterns, such as Map, that splits everything as much as possible, and Reduce, that groups all the data, Pachyderm uses glob patterns to provide incredible flexibility to define data distribution.

You can configure a glob pattern for each PFS input in the input field of a pipeline specification. Pachyderm detects this parameter and divides the input data into individual *datums*.

You can think of each input repository as a filesystem where the glob pattern is applied to the root of the filesystem. The files and directories that match the glob pattern are considered datums. The Pachyderm's concept of glob patterns is similar to the Unix glob patterns. For example, the `ls *.md` command matches all files with the `.md` file extension.

In Pachyderm, the `/` and `*` indicators are most commonly used globs.

The following are examples of glob patterns that you can define:

- `/` — Pachyderm denotes the whole repository as a single datum and sends all of the input data to a single worker node to be processed together.
- `/*` — Pachyderm defines each top-level filesystem object, that is a file or a directory, in the input repo as a separate datum. For example, if you have a repository with ten files in it and no directory structure, Pachyderm identifies each file as a single datum and processes them independently.
- `/*/*` — Pachyderm processes each filesystem object in each subdirectory as a separate datum.

If you have more than one input repo in your pipeline, you can define a different glob pattern for each input repo. You can combine the datums from each input repo by using either the `cross` or `union` operator to create the final datums that your code processes. For more information, see Cross and Union.

## Example of Defining Datums

For example, you have the following directories:

```
/California
  /San-Francisco.json
  /Los-Angeles.json
  ...
/Colorado
  /Denver.json
  /Boulder.json
  ...
...
```

Each top-level directory represents a US state with a `json` file for each city in that state.

If you set `glob` pattern to `/`, every time you change anything in any of the files and directories or add a new file to the repository, Pachyderm processes the contents of the whole repository from scratch as a single datum. For example, if you add `Sacramento.json` to the `California/` directory, Pachyderm processes all files and folders in the repo as a single datum.

If you set `/*` as a `glob` pattern, Pachyderm processes the data for each state individually. It defines one datum per state, which means that all the cities for a given state are processed together by a single worker, but each state is processed independently. For example, if you add a new file `Sacramento.json` to the `California/` directory, Pachyderm processes the `California/` datum only.

If you set `/**/*.json`, Pachyderm processes each city as a single datum on a separate worker. For example, if you add the `Sacramento.json` file, Pachyderm processes the `Sacramento.json` file only.

Glob patterns also let you take only a particular directory or subset of directories as an input instead of the whole repo. For example, you can set `/California/*` to process only the data for the state of California. Therefore, if you add a new city in the `Colorado/` directory, Pachyderm ignore this change and does not start the pipeline. However, if you add `Sacramento.json` to the `California/` directory, Pachyderm processes the `California/` datum.

## Test glob patterns

You can use the `pachctl glob file` command to preview which filesystem objects a pipeline defines as datums. This command helps you to test various glob patterns before you use them in a pipeline.

### Example:

- If you set the `glob` property to `/`, Pachyderm detects all top-level filesystem objects in the `census_data` repository as one datum:

```
$ pachctl glob file census_data@master:/
NAME TYPE SIZE
/    dir  15.11GiB
```

- If you set the `glob` property to `/*`, Pachyderm detects each top-level filesystem object in the `census_data` repository as a separate datum:

```
$ pachctl glob file census_data@master:/*
NAME                TYPE SIZE
/California         dir  1.224GiB
/Colorado           dir   74GiB
/Connecticut        dir  13.81GiB
```

## Datum Processing

This section helps you to understand the following concepts:

- Pachyderm job stages
- Processing multiple datums
- Incremental processing

A datum is a Pachyderm abstraction that helps to optimize pipeline processing. A datum is a representation of a unit of work in your job that helps the job to run more efficiently. A datum determines how Pachyderm divide your data for parallel processing. Sometimes, all your input files need to be exposed to the job together. Other times, only small groups of files need to be processed together, and therefore, different groups of files can be processed in parallel.

In addition to parallelizing your data processing, datums leverage Pachyderm's versioning semantics to enable your pipelines to run much faster by avoiding repeated processing of unchanged datums. For example, if you have multiple datums and only one datum was modified, Pachyderm processes only the changed datum and skips re-processing the unchanged datums. This incremental behavior ensures efficient resource utilization.

Each Pachyderm job can process multiple datums, which can consist of one or many input and output files.

When you create a pipeline specification, the most important fields that you need to configure are in the `input` section. The `input` section is where you define the input data source for your pipeline. The required `glob` parameter defines the number of datums in the source repository. By using the `glob` parameter, you can configure everything in the input repository to be processed as a single datum or break it down to multiple datums for parallel processing. The way you break your input repository into datums directly affects incremental processing and your pipeline processing speed. For more information about glob patterns, see [Glob Pattern](#).

When new data comes in, a Pachyderm pipeline automatically starts a new job. Each Pachyderm job consists of the following stages:

1. Creation of input datums. In this stage, Pachyderm breaks input files into datums according to the glob pattern set in the pipeline specification.
2. Transformation. The pipeline executes your code to processes the datums.
3. Completion of output datums. Your transformation code finishes processing all datums and generates a set of output files for each datum.
4. Merge. Pachyderm combines all the output files with the same file path from each datum to create the final output files and complete the output commit of the job.

If you think about this process in terms of filesystem objects and processing abstractions, the following transformation happens:

**input files => input datums => output datums => output files**

To summarize all the mentioned above, the datum abstraction has the following important concepts that are crucial to understand:

- The *merging stage* during which compiling of the final output files from the output datums occurs. Because Pachyderm processes each output datum independently, two datums processed in parallel can output to the same file. When that happens, Pachyderm appends those results together as part of the merge step, as opposed to having one *clobber* the other, which might result in data loss. The files are merged without any particular order.
- The way Pachyderm handles the datums that have changed. If you overwrite, modify, or delete any file within a datum, Pachyderm identifies that entire datum as *changed*. While Pachyderm skips any unchanged datums, changed datums are processed in their entirety, and the new output results of that datum overwrite the previous version of that datum. The files in that newly generated output datum are then re-merged as usual, replacing any file chunks that were from the previous version of the datum.

The following examples demonstrate these fundamental concepts.

### Example 1: Output datums with no overlapping file paths

In the diagram below, you can see three input datums, which might consist of one or many input files. Because there are three input datums, the result in precisely three output datums. In this example, each output datum is just a single file with a unique name. Since none of the files in the output datums overlap, the merge step is trivial and you have three files in the final output commit.

If with your next commit you decide to change any file in `datum 3`, Pachyderm notes that there are no changes in `datum 1` and `datum 2` and skips processing on these datums. Pachyderm detects changes in `datum 3`, processes it, and writes the final output file `3`.

## Example 2: Output datums with overlapping file paths

Often, you want different output datums in a job to write to the same file path and then merge those results together. The merge step might include appending certain files together or overwriting outdated chunks of files.

In the diagram below, you have the same datums as in the previous section, but in this case, each output datum includes two files with overlapping names:

- `datum 1` results in files `1` and `3`.
- `datum 2` results in files `2` and `3`.
- `datum 3` results in files `2` and `1`.

Because these datums used different input data, the contents of `file 1` in `datum 1` are not the same as the contents of `file 1` in `datum 3`. Pachyderm then merges the contents of these two files together to create the final `file 1` that you see in the output commit. Similar merges need to happen for `file 2` and `file 3`.

For example, in a new commit, you decide to modify a file in `datum 2`. Because `datum 1` and `datum 3` are unchanged, Pachyderm skips processing these datums. Pachyderm detects that something has changed in `datum 2` and processes it. Let's say that based on the new input data, the output of `datum 2` now includes three files, `file 1`, `file 3`, and `file 4`. Pachyderm then needs to re-merge all these files together to create the new final output files.

The following transformations happen during this re-merge:

- `file 1` now has three chunks, the previous two from `datum 1` and `datum 3` which have not changed, but also the third new chunk from `datum 2`.
- `file 2` has a chunk from `datum 3`, while the chunk from `datum 2` has been deleted.
- `file 3` still has two chunks, but the new `file 3` chunk from `datum 2` replaces the previous one.
- `file 4` is new and only has one chunk.

## Cross and Union Inputs

Pachyderm enables you to combine multiple input repositories in a single pipeline by using the `union` and `cross` operators in the pipeline specification.

If you are familiar with [Set theory](#), you can think of `union` as a *disjoint union binary operator* and `cross` as a *cartesian product binary operator*. However, if you are unfamiliar with these concepts, it is still easy to understand how `cross` and `union` work in Pachyderm.

This section describes how to use `cross` and `union` in your pipelines and how you can optimize your code when you work with them.

## Union Input

The union input combines each of the datums in the input repos as one set of datums. The number of datums that are processed is the sum of all the datums in each repo.

For example, you have two input repos, A and B. Each of these repositories contain three files with the following names.

Repository A has the following structure:

```
A
-- 1.txt
-- 2.txt
-- 3.txt
```

Repository B has the following structure:

```
B
-- 4.txt
-- 5.txt
-- 6.txt
```

If you want your pipeline to process each file independently as a separate datum, use a glob pattern of `/*`. Each glob is applied to each input independently. The input section in the pipeline spec might have the following structure:

```
"input": {
  "union": [
    {
      "pfs": {
        "glob": "/*",
        "repo": "A"
      }
    },
    {
      "pfs": {
        "glob": "/*",
        "repo": "B"
      }
    }
  ]
}
```

In this example, each Pachyderm repository has those three files in the root directory, so three datums from each input. Therefore, the union of A and B has six datums in total. Your pipeline processes the following datums without any specific order:

```
/pfs/A/1.txt
/pfs/A/2.txt
/pfs/A/3.txt
/pfs/B/4.txt
/pfs/B/5.txt
/pfs/B/6.txt
```

**Note:** Each datum in a pipeline is processed independently by a single execution of your code. In this example, your code runs six times, and each datum is available to it one at a time. For example, your code processes `pfs/A/1.txt` in one of the runs and `pfs/B/5.txt` in a different run, and so on. In a union, two or more datums are never available to your code at the same time. You can simplify your union code by using the `name` property as described below.

## Simplifying the Union Pipelines Code

In the example above, your code needs to read into the `pfs/A` or `pfs/B` directory because only one of them is present in any given datum. To simplify your code, you can add the `name` field to the `pfs` object and give the same name to each of the input repos. For example, you can add the `name` field with the value `C` to the input repositories `A` and `B`:

```
"input": {
  "union": [
    {
      "pfs": {
        "name": "C",
        "glob": "/*",
        "repo": "A"
      }
    },
    {
      "pfs": {
        "name": "C",
        "glob": "/*",
        "repo": "B"
      }
    }
  ]
}
```

Then, in the pipeline, all datums appear in the same directory.

```
/pfs/C/1.txt # from A
/pfs/C/2.txt # from A
/pfs/C/3.txt # from A
/pfs/C/4.txt # from B
/pfs/C/5.txt # from B
/pfs/C/6.txt # from B
```

## Cross Input

In a cross input, Pachyderm exposes every combination of datums, or a cross-product, from each of your input repositories to your code in a single run. In other words, a cross input pairs every datum in one repository with each datum in another, creating sets of datums. Your transformation code is provided one of these sets at the time to process.

For example, you have repositories `A` and `B` with three datums, each with the following structure:

**Note:** For this example, the `glob` pattern is set to `/*`.

Repository `A` has three files at the top level:

```
A
-- 1.txt
-- 2.txt
-- 3.txt
```

Repository `B` has three files at the top level:

```
B
-- 4.txt
```

```
-- 5.txt
-- 6.txt
```

Because you have three datums in each repo, Pachyderm exposes a total of nine combinations of datums to your code.

**Important:** In cross pipelines, both `pfs/A` and `pfs/B` directories are visible during each code run.

```
Run 1: /pfs/A/1.txt
       /pfs/B/1.txt

Run 2: /pfs/A/1.txt
       /pfs/B/2.txt

...

Run 9: /pfs/A/3.txt
       /pfs/B/3.txt
```

**Note:** In cross inputs, if you use the `name` field, your two inputs cannot have the same name. This could cause file system collisions.

**See Also:**

- Cross Input
- Union Input
- Combining/Merging/Joining Data
- [Distributed hyperparameter tuning](#)



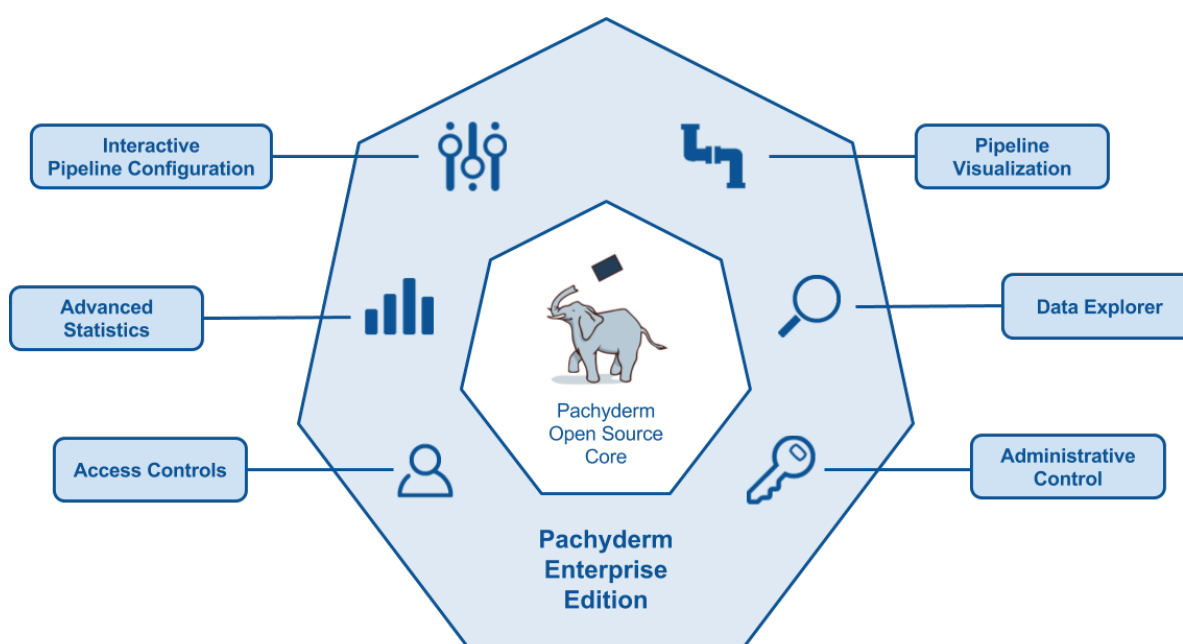
---

## Pachyderm Enterprise Edition Guide

---

This document describes the features that you can use in the Pachyderm Enterprise Edition.

### 3.1 Overview

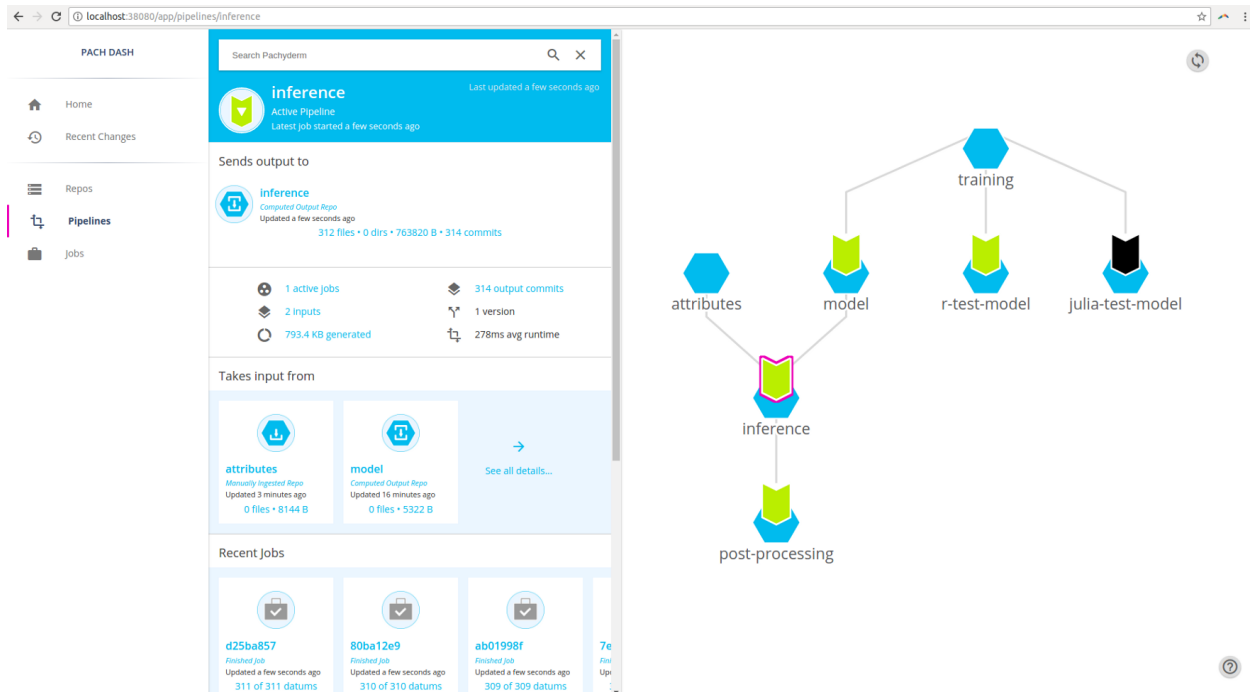


Pachyderm Enterprise Edition includes everything you need to scale and manage Pachyderm data pipelines in an enterprise setting. It delivers the most recent version of Pachyderm along with:

- Administrative and security features needed for enterprise-scale implementations of Pachyderm
- Visual and interactive interfaces to Pachyderm
- Detailed job and data statistics for faster development and data insight

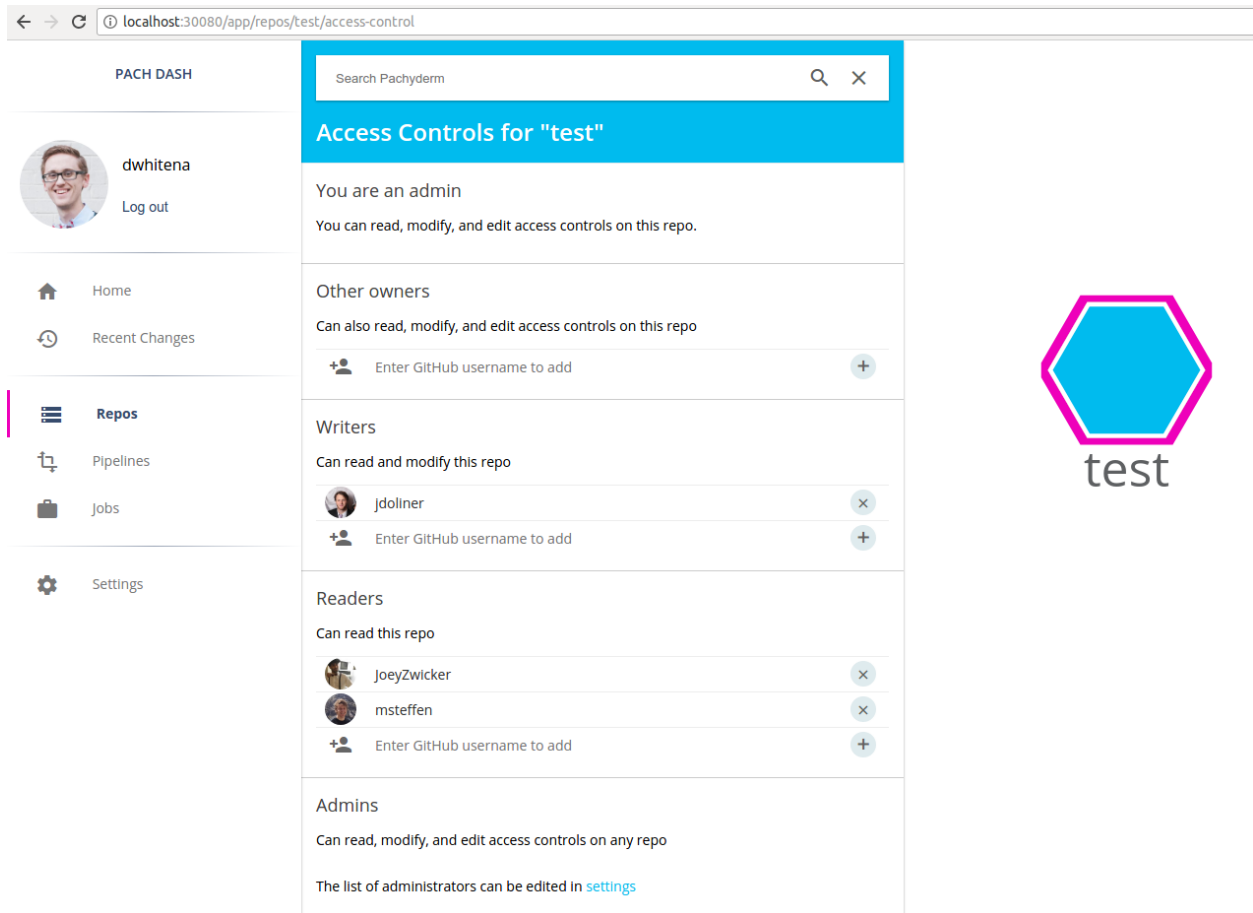
Pachyderm Enterprise Edition can be deployed easily on top of an existing or new deployment of Pachyderm, and we have engineers available to help enterprise customers get up and running very quickly. To get more information about Pachyderm Enterprise Edition, to ask questions, or to get access for evaluation, please contact us at [sales@pachyderm.io](mailto:sales@pachyderm.io) or on our [Slack](#).

### 3.1.1 Pipeline Visualization and Data Exploration



Pachyderm Enterprise Edition includes a full UI for visualizing pipelines and exploring data. Pachyderm Enterprise will automatically infer the structure of data scientists' DAG pipelines and display them visually. Data scientists and cluster admins can even click on individual segments of the pipelines to see what data is being processed, how many jobs have run, what images and commands are being run, and much more! Data scientists can also explore the versioned data in Pachyderm data repositories and see how the state of data has changed over time.

### 3.1.2 Access Controls



The screenshot displays the Pachyderm web interface at the URL `localhost:30080/app/repos/test/access-control`. The interface is divided into a left sidebar and a main content area.

**Left Sidebar:**

- PACH DASH** header.
- User profile for **dwhitena** with a **Log out** button.
- Navigation links: **Home**, **Recent Changes**, **Repos** (highlighted), **Pipelines**, **Jobs**, and **Settings**.

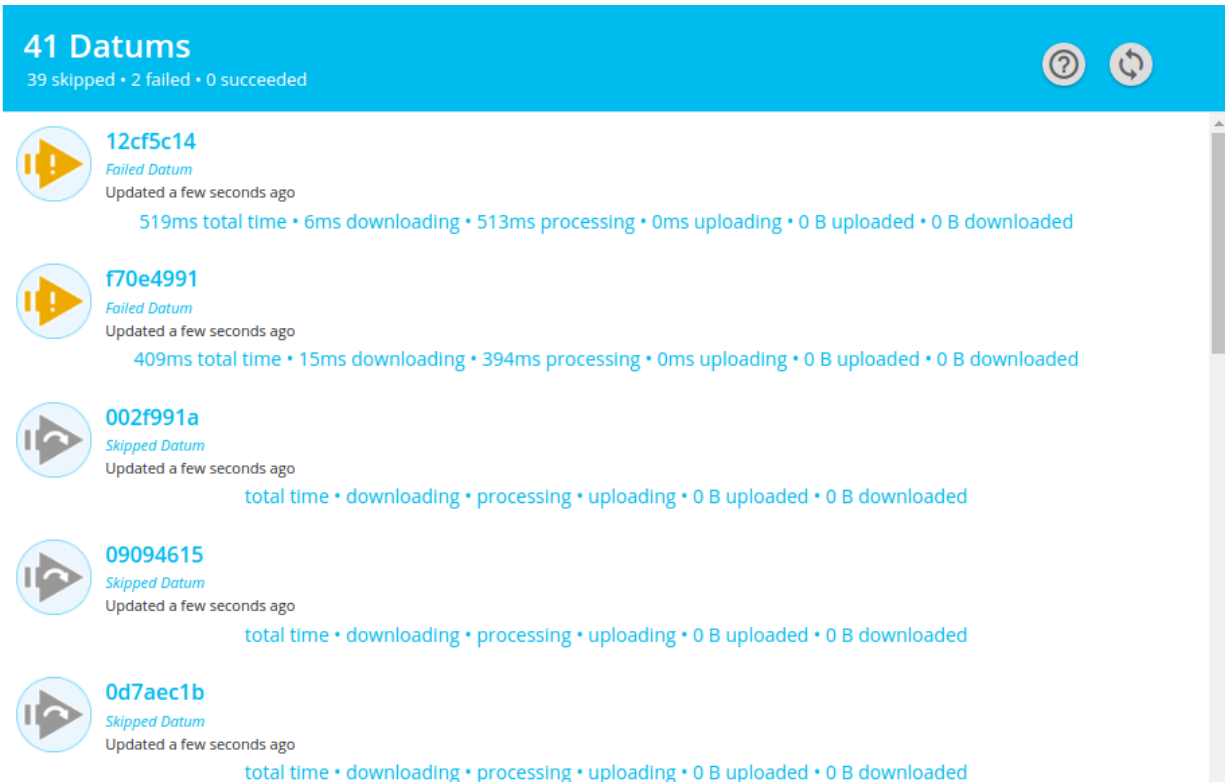
**Main Content Area:**

- Search Pachyderm** bar.
- Access Controls for "test"** header.
- You are an admin**: You can read, modify, and edit access controls on this repo.
- Other owners**: Can also read, modify, and edit access controls on this repo. Includes a button to **Enter GitHub username to add**.
- Writers**: Can read and modify this repo. Includes a list of users (e.g., **jdoliner**) and a button to **Enter GitHub username to add**.
- Readers**: Can read this repo. Includes a list of users (e.g., **JoeyZwicker**, **msteffen**) and a button to **Enter GitHub username to add**.
- Admins**: Can read, modify, and edit access controls on any repo. Includes a link to **settings** to edit the list of administrators.

To the right of the main content area, there is a large blue hexagon with a pink border and the word **test** below it.

Enterprise-scale deployments require access controls and multitenancy. Pachyderm Enterprise Edition gives teams the ability to control access to production pipelines, data, and configuration. Administrators can silo data, prevent unintended modifications to production pipelines, and support multiple data scientists or even multiple data science groups.

### 3.1.3 Advanced Statistics



Pachyderm Enterprise Edition gives data scientists advanced insights into their data, jobs, and results. For example, data scientists can see how much time jobs spend downloading/uploading data, what data was processed or skipped, and which workers were given particular datums. This information can be explored programmatically or via a number of charts and plots that help users parse the information quickly.

### 3.1.4 Administrative Controls, Interactive Pipeline Configuration

With Pachyderm Enterprise, cluster admins don't have to rely solely on command line tools and language libraries to configure and control Pachyderm. With new versions of our UI you can control, scale, and configure Pachyderm interactively.

### 3.1.5 S3Gateway

Pachyderm Enterprise Edition includes the s3gateway, an S3-like API for interacting with PFS content. With it, you can interact with PFS content with tools and libraries built to work with S3.

## 3.2 Deploying Enterprise Edition

To deploy and use Pachyderm's Enterprise Edition, you simply need to follow one of our guides to deploy Pachyderm and then *activate the Enterprise Edition*.

**Note** - Pachyderm's Enterprise dashboard is now deployed by default with Pachyderm. If you wish to deploy without the dashboard please use `pachctl deploy [command] --no-dashboard`

**Note** - You can get a FREE evaluation token for the enterprise edition on the landing page of the Enterprise dashboard.

### 3.2.1 Activating Pachyderm Enterprise Edition

There are two ways to activate Pachyderm’s enterprise features::

- *Activate Pachyderm Enterprise via the `pachctl` CLI*
- *Activate Pachyderm Enterprise via the dashboard*

For either method, you will need to have your Pachyderm Enterprise activation code available. You should have received this from Pachyderm sales/support when registering for the Enterprise Edition. If you are a new user evaluating Pachyderm, you can receive a FREE evaluation code on the landing page of the dashboard. Please contact [support@pachyderm.io](mailto:support@pachyderm.io) if you are having trouble locating your activation code.

#### Activate via the `pachctl` CLI

Assuming you followed one of our [deploy guides](#) and you have a Pachyderm cluster running, you should see that the state of your Pachyderm cluster is similar to the following:

```
$ kubectl get pods
NAME                                READY    STATUS    RESTARTS   AGE
dash-6c9dc97d9c-vb972              2/2      Running   0           6m
etcd-7dbb489f44-9v5jj              1/1      Running   0           6m
pachd-6c878bbc4c-f2h2c              1/1      Running   0           6m
```

You should also be able to connect to the Pachyderm cluster via the `pachctl` CLI:

```
$ pachctl version
COMPONENT    VERSION
pachctl      1.6.8
pachd        1.6.8
```

Activating the Enterprise features of Pachyderm is then as easy as:

```
$ pachctl enterprise activate <activation-code>
```

If this command returns no error, then the activation was successful. The state of the Enterprise activation can also be retrieved at any time via:

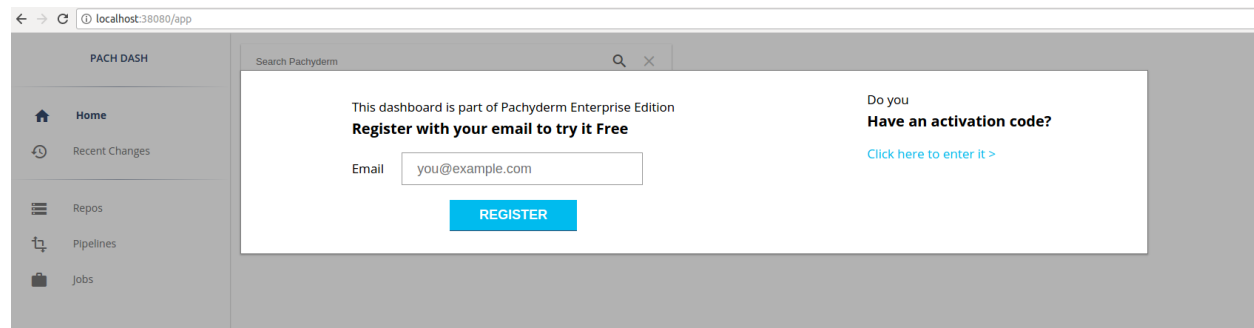
```
$ pachctl enterprise get-state
ACTIVE
```

#### Activate via the dashboard

You can active Enterprise Edition directly in the dashboard. There’s two ways to access the dashboard:

1. If you can directly connect, simply point your browser to port 30080 on your kubernetes cluster’s IP address.
2. You can enable port forwarding by calling `pachctl port-forward`, then point your browser to `localhost:30080`.

When you first visit the dashboard, it will prompt you for your activation code:



Once you enter your activation code, you should have full access to the Enterprise dashboard and your cluster will be an active Enterprise Edition cluster. This could be confirmed with:

```
$ pachctl enterprise get-state
ACTIVE
```

## 3.3 Access Controls

The access controls features of Pachyderm Enterprise let you create and manage users that interact with your Pachyderm cluster. You can restrict access to individual data repositories on a per user basis and, as a result, limit the subscription of pipelines to those data repositories.

This document guides you through the following sections:

### 3.3.1 Understanding Pachyderm access controls

If access controls are activated, each data repository, or repo, in Pachyderm has an Access Control List (ACL) associated with it. The ACL includes:

- **READERS** - users who can read the data versioned in the repo.
- **WRITERS** - users with **READER** access who can also submit additions, deletions, or modifications of data into the repo.
- **OWNERS** - users with **READER** and **WRITER** access who can also modify the repo's ACL.

Pachyderm defines the following account types:

- **GitHub user** is a user account that is associated with a GitHub account and logs in through the GitHub OAuth flow. If you do not use any third-party identity provider, you use this option. When a user tries to log in with a GitHub account, Pachyderm verifies the identity and sends a Pachyderm token for that account.
- **Robot user** is a user account that logs in with a pach-generated authentication token. Typically, you create a user in simplified workflow scenarios, such as initial SAML configuration.
- **Pipeline** is an account that Pachyderm creates for data pipelines. Pipelines inherit access control from its creator.
- **SAML user** is a user account that is associated with a Security Assertion Markup Language (SAML) identity provider. When a user tries to log in through a SAML ID provider, the system confirms the identity, associates that identity with a SAML identity provider account, and responds with the SAML identity provider token for that user. Pachyderm verifies the token, drops it, and creates a new internal token that encapsulates the information about the user.

By default, Pachyderm defines one hardcoded group called `admin`. Users in the `admin` group can perform any action on the cluster including appointing other admins. Furthermore, only the cluster admins can manage a repository without ACLs.

## Enabling access control

Before you enable access controls, make sure that you have activated Pachyderm Enterprise Edition as described in this guide.

To enable access controls, complete the following steps:

1. Verify the status of the Enterprise features by opening the Pachyderm dashboard in your browser or by running the following `pachctl` command:

```
$ pachctl enterprise get-state
ACTIVE
```

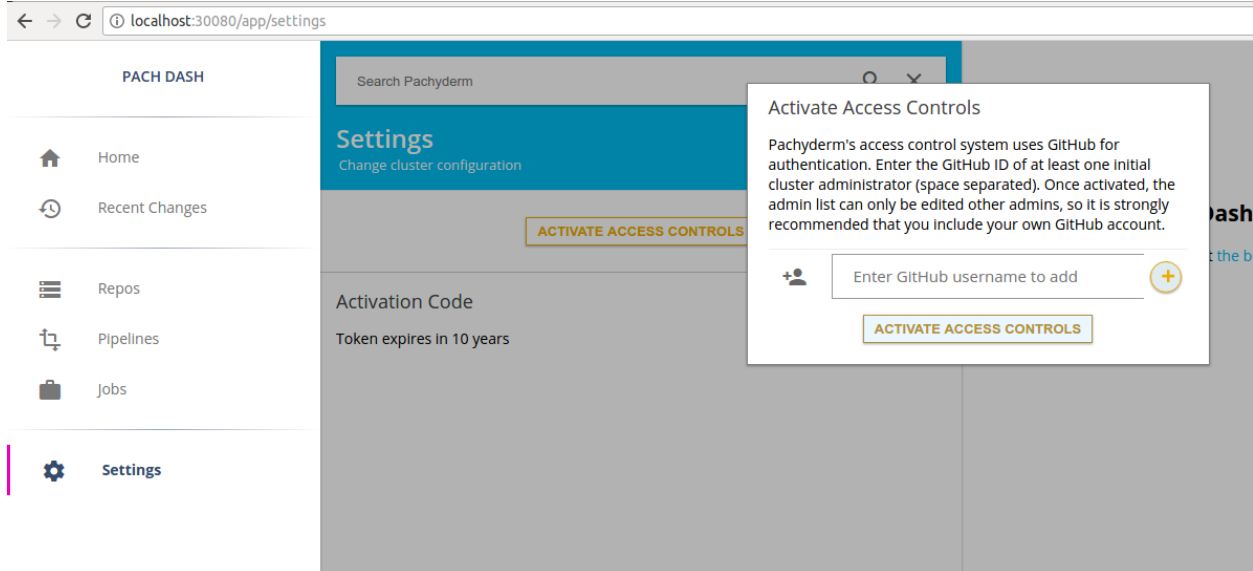
2. Activate the Enterprise access control features by completing the steps in one of these sections:

- [Activating Access Control with the Dashboard](#)
- [Activating Access Control with pachctl](#)

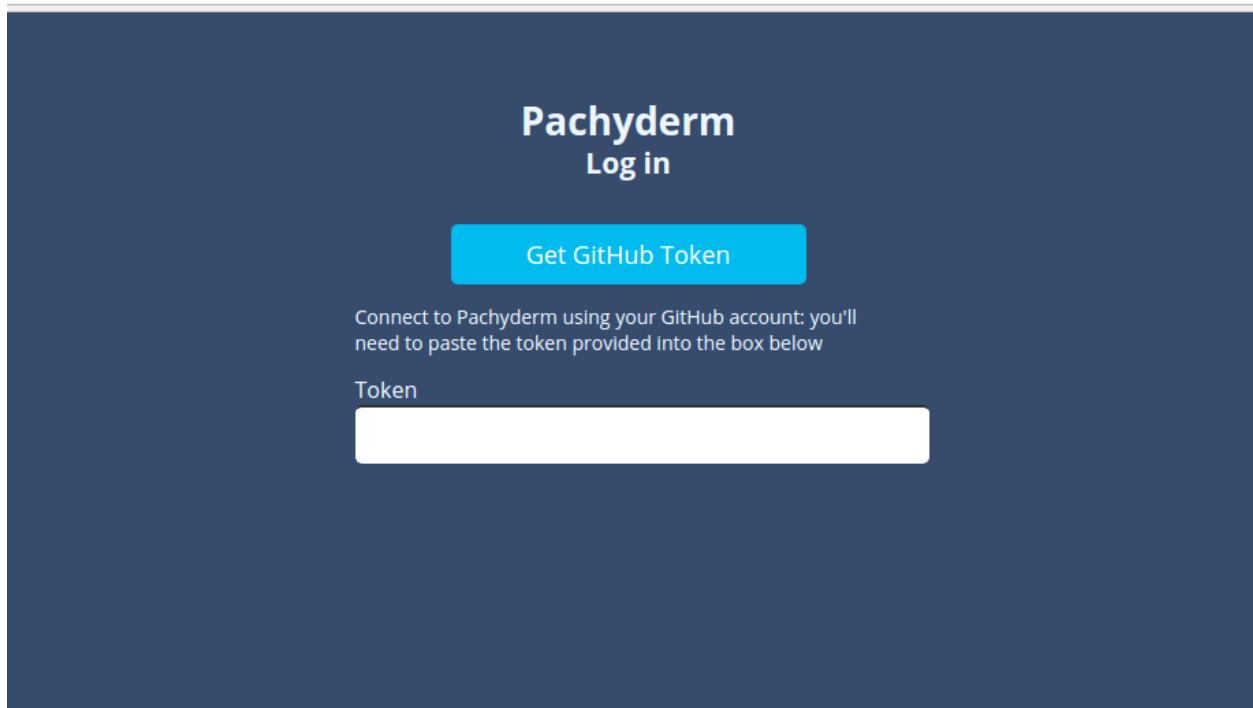
## Activating access controls with the dashboard

To activate access controls in the Pachyderm dashboard, complete the following steps:

1. Go to the Settings page.
2. Click the **Activate Access Controls** button. After you click the button, Pachyderm enables you to add GitHub users as cluster admins and activate access control:



After activating access controls, you should see the following screen that asks you to log in to Pachyderm:



### Activating access controls with `pachctl`

To activate access controls with `pachctl`, choose one of these options:

1. Activate access controls by specifying an initial admin user:

```
$ pachctl auth activate --initial-admin=<prefix>:<user>
```

**Note:** You must prefix the username with the appropriate account type, either `github:<user>` or `robot:<user>`. If you select the latter, Pachyderm generates and returns a Pachyderm auth token that might be used to authenticate as the initial robot admin by using `pachctl auth use-auth-token`. You can use this option when you cannot use GitHub as an identity provider.

1. Activate access controls with a GitHub account:

```
$ pachctl auth activate
```

Pachyderm prompts you to log in with your GitHub account. The GitHub account that you sign in with is the only admin until you add more by running `pachctl auth modify-admins`.

### Logging in to Pachyderm

After you activate access controls, log in to your cluster either through the dashboard or CLI. The CLI and the dashboard have independent login workflows:

- *Log in to the dashboard.*
- *Log in to the CLI.*

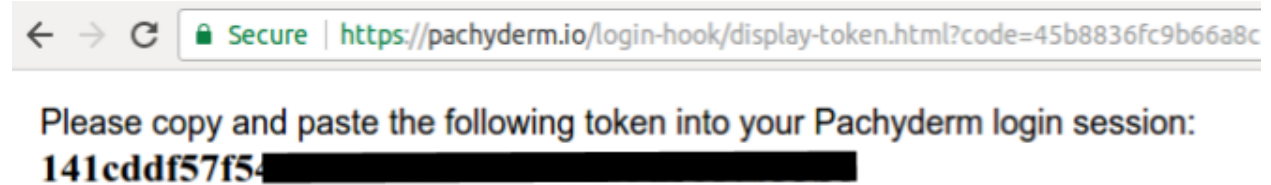


## Log in to the dashboard

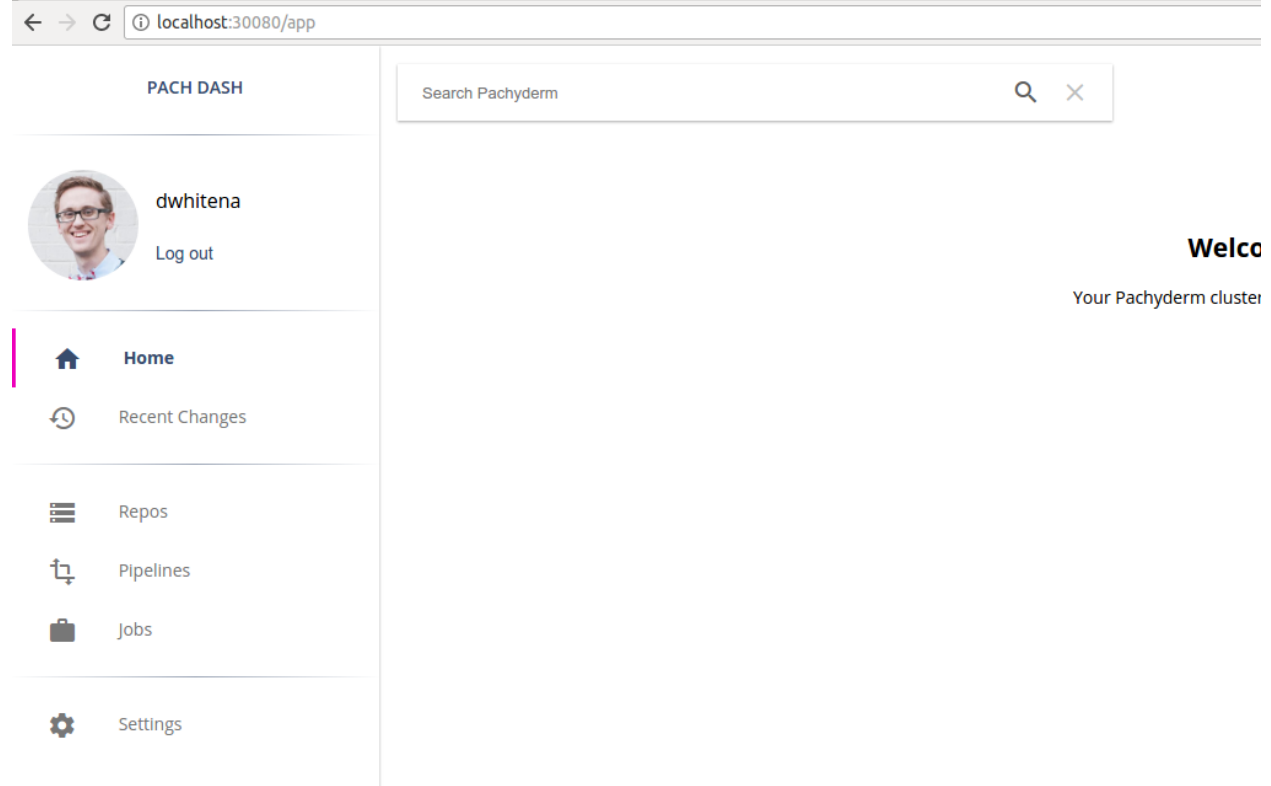
After you have activated access controls for Pachyderm, you need to log in to use the Pachyderm dashboard as shown above in [this section](#).

To log in to the dashboard, complete the following steps:

1. Click the **Get GitHub token** button. If you have not previously authorized Pachyderm on GitHub, an option to **Authorize Pachyderm** appears. After you authorize Pachyderm, a Pachyderm user token appears:



1. Copy and paste this token back into the Pachyderm login screen and press **Enter**. You are now logged in to Pachyderm, and you should see your GitHub avatar and an indication of your user in the upper left-hand corner of the dashboard:



## Log in to the CLI

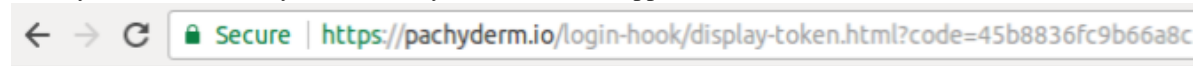
To log in to `pachctl`, complete the following steps:

1. Type the following command:

```
pachctl auth login
```

When you run this command, `pachctl` provides you with a GitHub link to authenticate as a GitHub user.

If you have not previously authorized Pachyderm on GitHub, an option to **Authorize Pachyderm** appears. After you authorize Pachyderm, a Pachyderm user token appears:



Please copy and paste the following token into your Pachyderm login session:

**141cddf57f54** [REDACTED]

2. Copy and paste this token back into the terminal and press enter.

You are now logged in to Pachyderm!

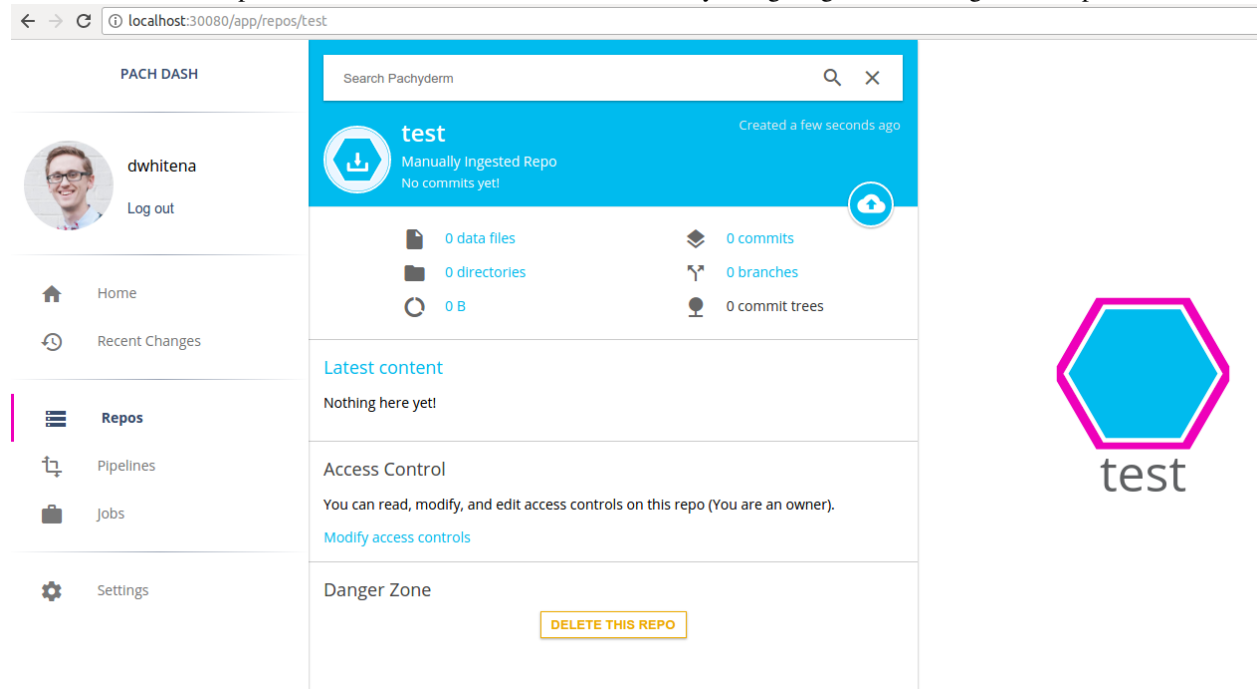
- (a) Alternatively, you can run the command:

```
pachctl auth use-auth-token
```

- (b) Paste an authentication token received from `pachctl auth activate --initial-admin=robot:<user>` or `pachctl auth get-auth-token`

## Manage and update user access

You can manage user access in the UI and CLI. For example, you are logged in to Pachyderm as the user `dwhitena` and have a repository called `test`. Because the user `dwhitena` created this repository, `dwhitena` has full OWNER-level access to the repo. You can confirm this in the dashboard by navigating to or clicking on the repo `test`:



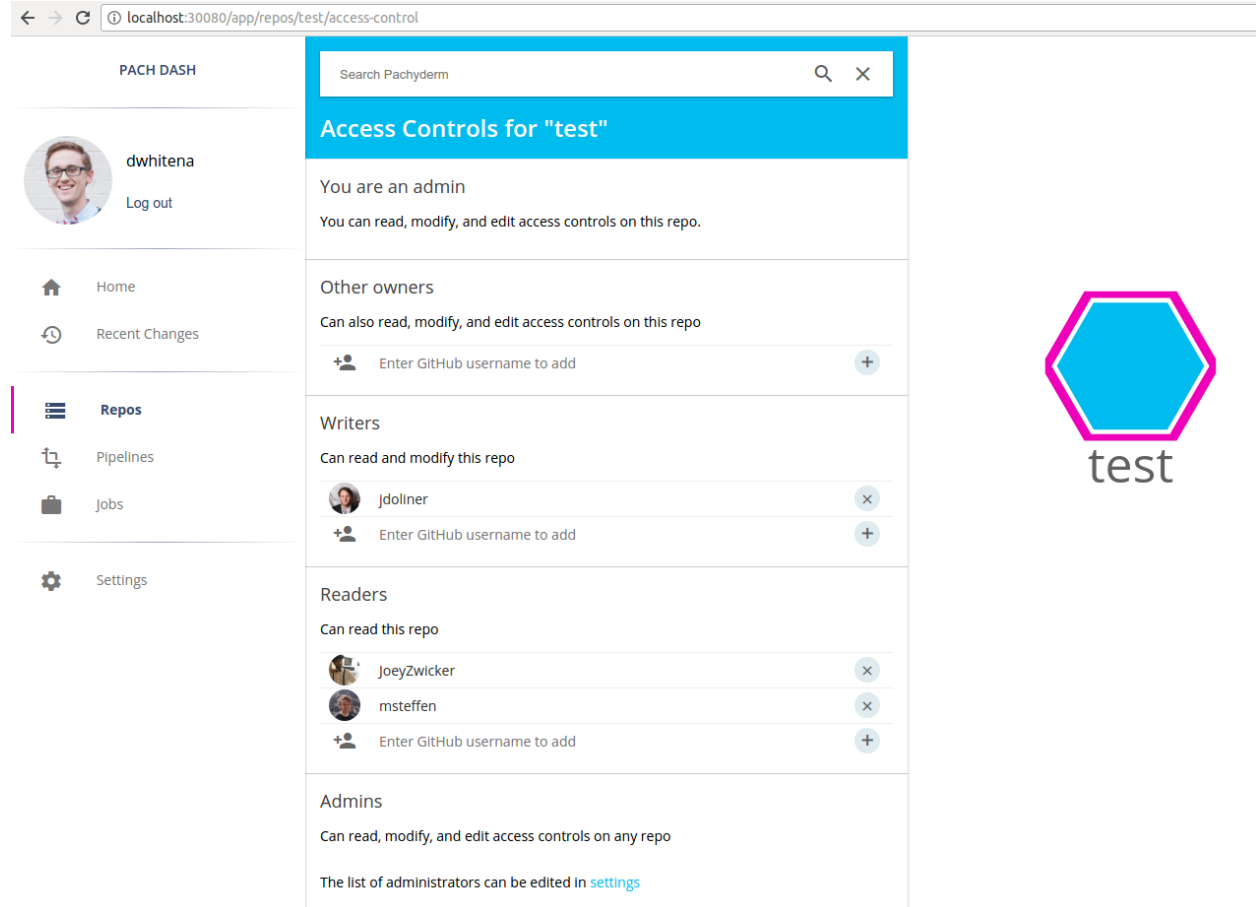
Alternatively, you can confirm your access by running the `pachctl auth get ...` command.

**Example:**

```
$ pachctl auth get dwhitena test`  
OWNER
```

An OWNER of `test` or a cluster admin can then set other user's level of access to the repo by using the `pachctl auth set ...` command or through the dashboard.

For example, to give the GitHub users `JoeyZwicker` and `msteffen` `READER`, but not `WRITER` or `OWNER`, access to `test` and `jdoliner` `WRITER`, but not `OWNER`, access, click on **Modify access controls** under the repo details in the dashboard. This functionality allows you to add the users easily one by one:



The screenshot shows the Pachyderm dashboard interface. The top navigation bar includes a search bar and a user profile for 'dwhitena' with a 'Log out' button. The left sidebar contains navigation links: Home, Recent Changes, Repos (highlighted), Pipelines, Jobs, and Settings. The main content area is titled 'Access Controls for "test"' and displays the following information:

- You are an admin:** You can read, modify, and edit access controls on this repo.
- Other owners:** Can also read, modify, and edit access controls on this repo. A button to 'Enter GitHub username to add' is present.
- Writers:** Can read and modify this repo. The user 'jdoliner' is listed as a writer. A button to 'Enter GitHub username to add' is present.
- Readers:** Can read this repo. The users 'JoeyZwicker' and 'msteffen' are listed as readers. A button to 'Enter GitHub username to add' is present.
- Admins:** Can read, modify, and edit access controls on any repo. A link to 'settings' is provided for editing administrators.

To the right of the dashboard, there is a blue hexagon logo with the word 'test' underneath it.

### Behavior of pipelines as related to access control

In Pachyderm, you do not explicitly grant users access to pipelines. Instead, pipelines infer access from their input and output repositories. To update a pipeline, you must have at least `READER` -level access to all pipeline inputs and at least `WRITER` -level access to the pipeline output. This is because pipelines read from their input repos and write to their output repos, and you cannot grant a pipeline more access than you have yourself.

- An OWNER, WRITER, or READER of a repo can subscribe a pipeline to that repo.
- When a user subscribes a pipeline to a repo, Pachyderm sets that user as an OWNER of that pipeline's output repo.
- If additional users need access to the output repository, the initial OWNER of a pipeline's output repo, or an admin, needs to configure these access rules.
- To update a pipeline, you must have WRITER access to the pipeline's output repos and READER access to the pipeline's input repos.

## Manage the Activation Code

When an enterprise activation code expires, an auth-activated Pachyderm cluster goes into an `admin-only` state. In this state, only admins have access to data that is in Pachyderm. This safety measure keeps sensitive data protected, even when an enterprise subscription becomes stale. As soon as the enterprise activation code is updated by using the dashboard or CLI, the Pachyderm cluster returns to its previous state.

When you deactivate access controls on a Pachyderm cluster by running `pachctl auth deactivate`, the cluster returns its original state that including the following changes:

- All ACLs are deleted.
- The cluster returns to being a blank slate in regards to access control. Everyone that can connect to Pachyderm can access and modify the data in all repos.
- No users are present in Pachyderm, and no one can log in to Pachyderm.

## 3.4 Advanced Statistics

To take advantage of the advanced statistics features in Pachyderm Enterprise Edition, you need to:

1. Run your pipelines on a Pachyderm cluster that has activated Enterprise features (see [Deploying Enterprise Edition](#) for more details).
2. Enable stats collection in your pipelines by including `"enable_stats": true` in your [pipeline specifications](#).

You will then be able to access the following information for any jobs corresponding to your pipelines:

- The amount of data that was uploaded and downloaded during the job and on a per-datum level (see [here](#) for info about Pachyderm datums).
- The time spend uploading and downloading data on a per-datum level.
- The amount of data uploaded and downloaded on a per-datum level.
- The total time spend processing on a per-datum level.
- Success/failure information on a per-datum level.
- The directory structure of input data that was seen by the job.

The primary and recommended way to view this information is via the Pachyderm Enterprise dashboard, which can be deployed as detailed [here](#). However, the same information is available through the `inspect datum` and `list datum pachctl` commands or through their language client equivalents.

**Note** - We recommend enabling stats for all of your pipeline and only disabling the feature for very stable, long-running pipelines. In most cases, the debugging/maintenance benefits of the stats data will outweigh any disadvantages of storing the extra data associated with the stats. Also note, none of your data is duplicated in producing the stats.

### 3.4.1 Enabling stats for a pipeline

As mentioned above, enabling stats collection for a pipeline is as simple as adding the `"enable_stats": true` field to a pipeline specification. For example, to enable stats collection for our [OpenCV demo pipeline](#), we would modify the pipeline specification as follows:

```
{
  "pipeline": {
    "name": "edges"
```

```

},
"input": {
  "pfs": {
    "glob": "/*",
    "repo": "images"
  }
},
"transform": {
  "cmd": [ "python3", "/edges.py" ],
  "image": "pachyderm/opencv"
},
"enable_stats": true
}

```

Once the pipeline has been created and you have utilized it to process data, you can confirm that stats are being collected with `list file`. There should now be stats data in the output repo of the pipeline under a branch called `stats`:

```

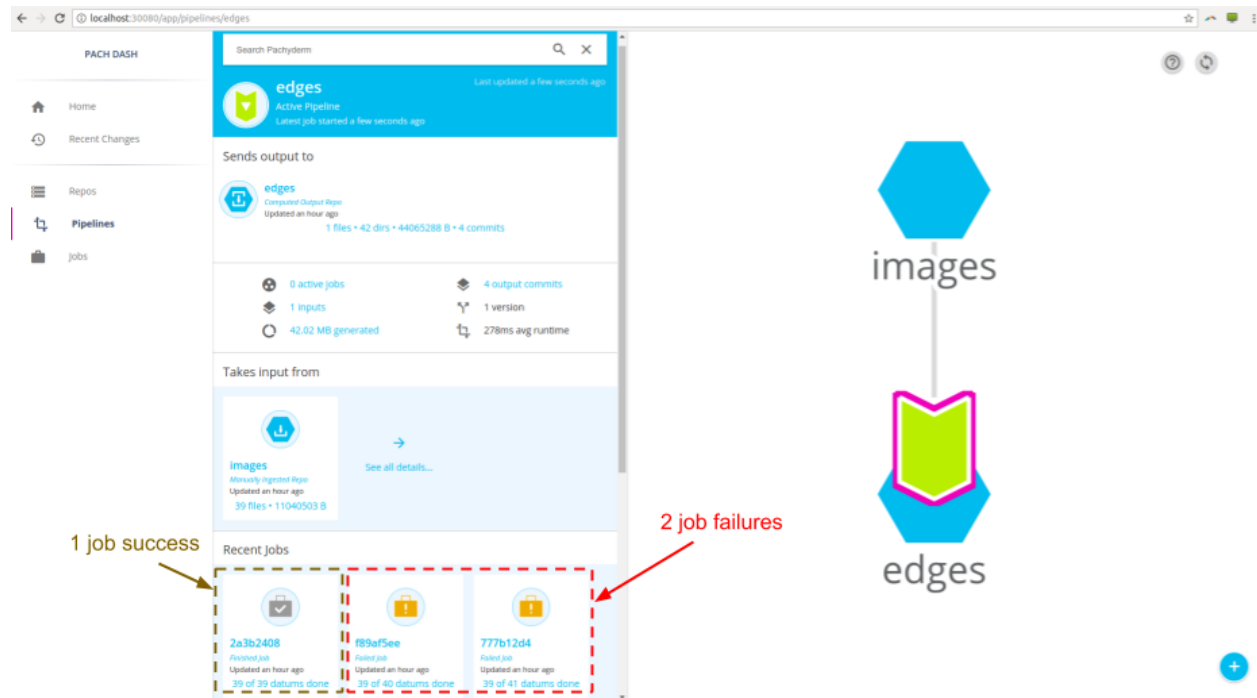
$ pachctl list file edges@stats
NAME                                                    TYPE
↪ SIZE
002f991aa9db9f0c44a92a30dff8ab22e788f86cc851bec80d5a74e05ad12868  dir
↪ 342.7KiB
0597f2df3f37f1bb5b9bcd6397841f30c62b2b009e79653f9a97f5f13432cf09  dir
↪ 1.177MiB
068fac9c3165421b4e54b358630acd2c29f23ebf293e04be5aa52c6750d3374e  dir
↪ 270.3KiB
0909461500ce508c330ca643f3103f964a383479097319dbf4954de99f92f9d9  dir
↪ 109.6KiB
etc...

```

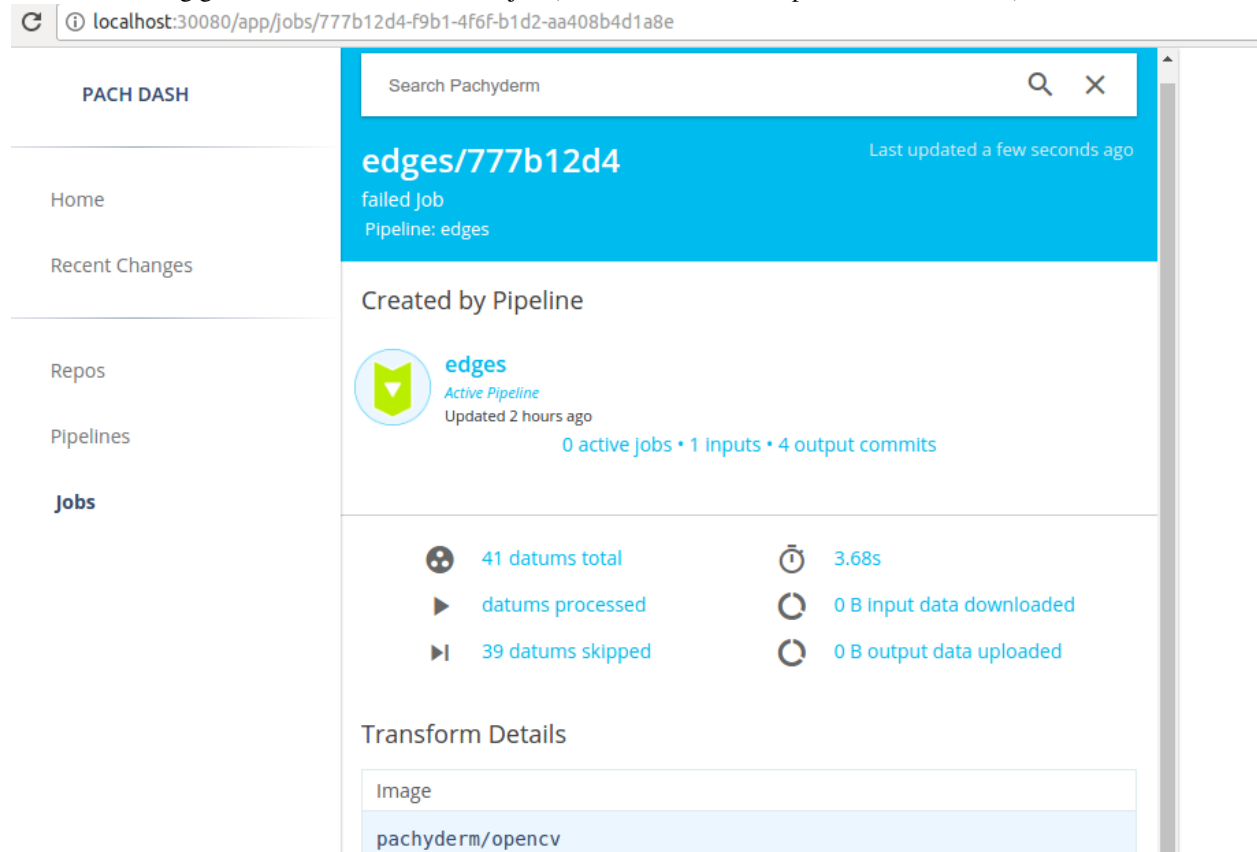
Don't worry too much about this view of the stats data. It just confirms that stats are being collected.

### 3.4.2 Accessing stats via the dashboard

Assuming that you have deployed and activated the Pachyderm Enterprise dashboard, you can explore your advanced statistics in just a few clicks. For example, if we navigate to our `edges` pipeline (specified above), we will see something similar to this:



In this example case, we can see that the pipeline has had 1 recent successful job and 2 recent job failures. Pachyderm advanced stats can be very helpful in debugging these job failures. When we click on one of the job failures we will see the following general stats about the failed job (total time, total data upload/download, etc.):



To get more granular per-datum stats (see [here](#) for info on Pachyderm datums), we can click on the 41 datums

total , which will reveal the following:



We can easily identify the exact datums that caused our pipeline to fail and the associated stats:

- Total time
- Time spent downloading data
- Time spent processing
- Time spent uploading data
- Amount of data downloaded
- Amount of data uploaded

If we need to, we can even go a level deeper and explore the exact details of a failed datum. Clicking on one of the failed datums will reveal the logs corresponding to the datum processing failure along with the exact input files of the datum:

The screenshot displays the Pachyderm web interface for a failed datum. At the top, a blue header shows the datum ID '12cf5c14' and its status 'failed Datum'. Below this, performance metrics are listed: 519ms total runtime, 6ms download time, 513ms processing time, and 0ms upload time. The 'Input Files' section is highlighted with a red arrow and text: 'View the input files corresponding to the datum'. The 'Created by Pipeline' section shows the pipeline 'edges/777b12d4' with a 'Failed job' icon and 'Updated 2 hours ago'. The 'As part of job' section shows the job 'edges' with an 'Active Pipeline' icon and 'Updated 2 hours ago'. The 'Logs from worker pipeline-edges-v1-q3x4q' section is expanded, showing 31 lines of logs. A red bracket on the right side of the logs is labeled 'Logs corresponding to the failure'. The logs show a traceback from a file named 'edgesec.py'.

## 3.5 Using the S3 Gateway

Pachyderm Enterprise includes an S3 gateway that enables you to interact with PFS storage through an HTTP application programming interface (API) that imitates the Amazon S3 Storage API. Therefore, with Pachyderm S3 gateway, you can interact with Pachyderm through tools and libraries designed to work with object stores. For example, you can use these tools:

- MinIO
- boto3
- S3cmd

When you deploy `pachd`, the S3 gateway starts automatically. However, the S3 gateway is an enterprise feature that is only available to paid customers or during the free trial evaluation.

The S3 gateway has some limitations that are outlined below. If you need richer access, use the PFS gRPC interface instead, or one of the [client drivers](#).

### 3.5.1 Authentication

If auth is enabled on the Pachyderm cluster, credentials must be passed with each s3 gateway endpoint using AWS' signature v2 or v4 methods. Object store tools and libraries provide built-in support for these methods, but they do not



work in the browser. When you use authentication, set the access and secret key to the same value; they are both the Pachyderm auth token used to issue the relevant PFS calls.

If auth is not enabled on the Pachyderm cluster, no credentials need to be passed to s3gateway requests.

### 3.5.2 Buckets

The S3 gateway presents each branch from every Pachyderm repository as an S3 bucket. For example, if you have a `master` branch in the `images` repository, an S3 tool sees `images@master` as the `master.images` S3 bucket.

### 3.5.3 Versioning

Most operations act on the HEAD of the given branch. However, if your object store library or tool supports versioning, you can get objects in non-HEAD commits by using the commit ID as the version.

### 3.5.4 Port Forwarding

If you do not have direct access to the Kubernetes cluster, you can use port forwarding instead. Simply run `pachctl port-forward`, which will allow you to access the s3 gateway through `localhost:30600`.

However, the Kubernetes port forwarder incurs substantial overhead and does not recover well from broken connections. Connecting to the cluster directly is therefore faster and more reliable.

### 3.5.5 Configure the S3 client

Before you can work with the S3 gateway, configure your S3 client to access Pachyderm. Complete the steps in one of the sections below that correspond to your S3 client.

#### Configure MinIO

If you are not using the MinIO client, skip this section.

To install and configure MinIO, complete the following steps:

1. Install the MinIO client on your platform as described on the [MinIO download page](#).
2. Verify that MinIO components are successfully installed by running the following command:

```
$ minio version
$ mc version
Version: 2019-07-11T19:31:28Z
Release-tag: RELEASE.2019-07-11T19-31-28Z
Commit-id: 31e5ac02bdbdbaf20a87683925041f406307cfb9
```

3. Set up the MinIO configuration file to use the `30600` port for your host:

```
vi ~/.mc/config.json
```

You should see a configuration similar to the following:

- For a minikube deployment, verify the `local` host configuration:

```
"local": {
  "url": "http://localhost:30600",
  "accessKey": "YOUR-PACHYDERM-AUTH-TOKEN",
  "secretKey": "YOUR-PACHYDERM-AUTH-TOKEN",
  "api": "S3v4",
  "lookup": "auto"
},
```

Set both the access key and secret key to your Pachyderm auth token. If auth is not enabled on the cluster, both should be empty strings.

## Configure the AWS CLI

If you are not using the AWS CLI, skip this section.

If you have not done so already, you need to install and configure the AWS CLI client on your machine. To configure the AWS CLI, complete the following steps:

1. Install the AWS CLI for your operating system as described in the [AWS documentation](#).
2. Verify that the AWS CLI is installed:

```
$ aws --version aws-cli/1.16.204 Python/2.7.16 Darwin/17.7.0 botocore/1.12.194
```

3. Configure AWS CLI:

```
$ aws configure
AWS Access Key ID: YOUR-PACHYDERM-AUTH-TOKEN
AWS Secret Access Key: YOUR-PACHYDERM-AUTH-TOKEN
Default region name:
Default output format [None]:
```

Note that both the access key and secret key should be set to your Pachyderm auth token. If auth is not enabled on the cluster, both should be empty strings.

## Configure S3cmd

If you are not using S3cmd, skip this section.

S3cmd is an open-source command line client that enables you to access S3 object store buckets. To configure S3cmd, complete the following steps:

1. If you do not have S3cmd installed on your machine, install it as described in the [S3cmd documentation](#). For example, in macOS, run:

```
$ brew install s3cmd
```

2. Verify that S3cmd is installed:

```
$ s3cmd --version
s3cmd version 2.0.2
```

3. Configure S3cmd to use Pachyderm:

```
$ s3cmd --configure
...
```

4. Fill all fields and specify the following settings for Pachyderm.

**Example:**

```
New settings:
Access Key: "YOUR-PACHYDERM-AUTH-TOKEN"
Secret Key: "YOUR-PACHYDERM-AUTH-TOKEN"
Default Region: US
S3 Endpoint: localhost:30600
DNS-style bucket+hostname:port template for accessing a bucket: localhost:30600/
→%(bucket)
Encryption password:
Path to GPG program: /usr/local/bin/gpg
Use HTTPS protocol: False
HTTP Proxy server name:
HTTP Proxy server port: 0
```

Set both the access key and secret key to your Pachyderm auth token. If auth is not enabled on the cluster, both should be empty strings.

### 3.5.6 Supported Operations

The Pachyderm S3 gateway supports the following operations:

- Create buckets: Creates a repo and branch.
- Delete buckets: Deletes a branch or a repo with all branches.
- List buckets: Lists all branches on all repos as S3 buckets.
- Write objects: Atomically overwrites a file on a branch.
- Remove objects: Atomically removes a file on a branch.
- List objects: Lists the files in the HEAD of a branch.
- Get objects: Gets file contents on a branch.

#### List Filesystem Objects

If you have configured your S3 client correctly, you should be able to see the list of filesystem objects in your Pachyderm repository by running an S3 client `ls` command.

To list filesystem objects, complete the following steps:

1. Verify that your S3 client can access all of your Pachyderm repositories:

- If you are using MinIO, type:

```
$ mc ls local
[2019-07-12 15:09:50 PDT]      0B master.train/
[2019-07-12 14:58:50 PDT]      0B master.pre_process/
[2019-07-12 14:58:09 PDT]      0B master.split/
[2019-07-12 14:58:09 PDT]      0B stats.split/
[2019-07-12 14:36:27 PDT]      0B master.raw_data/
```

- If you are using AWS, type:

```
$ aws --endpoint-url http://localhost:30600 s3 ls
2019-07-12 15:09:50 master.train
2019-07-12 14:58:50 master.pre_process
2019-07-12 14:58:09 master.split
2019-07-12 14:58:09 stats.split
2019-07-12 14:36:27 master.raw_data
```

- If you are using S3cmd, type:

```
$ s3cmd ls
2019-07-12 15:09 master.train
2019-07-12 14:58 master.pre_process
2019-07-12 14:58 master.split
2019-07-12 14:58 stats.split
2019-07-12 14:36 master.raw_data
```

## 2. List the contents of a repository:

- If you are using MinIO, type:

```
$ mc ls local/master.raw_data
[2019-07-19 12:11:37 PDT] 2.6MiB github_issues_medium.csv
```

- If you are using AWS, type:

```
$ aws --endpoint-url http://localhost:30600/ s3 ls s3://master.raw_data
2019-07-26 11:22:23 2685061 github_issues_medium.csv
```

- If you are using S3cmd, type:

```
$ s3cmd ls s3://master.raw_data/
2019-07-26 11:22 2685061 s3://master.raw_data/github_issues_medium.csv
```

## Create an S3 Bucket

You can create an S3 bucket in Pachyderm by using the AWS CLI or the MinIO client commands. The S3 bucket that you create is a branch in a repository in Pachyderm.

To create an S3 bucket, complete the following steps:

1. Use the `mb <host/branch.repo>` command to create a new S3 bucket, which is a repository with a branch in Pachyderm.

- If you are using MinIO, type:

```
$ mc mb local/master.test
Bucket created successfully `local/master.test`.
```

- If you are using AWS, type:

```
$ aws --endpoint-url http://localhost:30600/ s3 mb s3://master.test
make_bucket: master.test
```

- If you are using S3cmd, type:

```
$ s3cmd mb s3://master.test
```

This command creates the `test` repository with the `master` branch.

## 2. Verify that the S3 bucket has been successfully created:

- If you are using MinIO, type:

```
$ mc ls local
[2019-07-18 13:32:44 PDT]      0B master.test/
[2019-07-12 15:09:50 PDT]      0B master.train/
[2019-07-12 14:58:50 PDT]      0B master.pre_process/
[2019-07-12 14:58:09 PDT]      0B master.split/
[2019-07-12 14:58:09 PDT]      0B stats.split/
[2019-07-12 14:36:27 PDT]      0B master.raw_data/
```

- If you are using AWS, type:

```
$ aws --endpoint-url http://localhost:30600/ s3 ls
2019-07-26 11:35:28 master.test
2019-07-12 14:58:50 master.pre_process
2019-07-12 14:58:09 master.split
2019-07-12 14:58:09 stats.split
2019-07-12 14:36:27 master.raw_data
```

- If you are using S3cmd, type:

```
$ s3cmd ls
2019-07-26 11:35 master.test
2019-07-12 14:58 master.pre_process
2019-07-12 14:58 master.split
2019-07-12 14:58 stats.split
2019-07-12 14:36 master.raw_data
```

- You can also use the `pachctl list repo` command to view the list of repositories:

```
$ pachctl list repo
NAME                CREATED                SIZE (MASTER)
test                About an hour ago      0B
train               6 days ago             68.57MiB
pre_process         6 days ago             1.18MiB
split               6 days ago             1.019MiB
raw_data            6 days ago             2.561MiB
```

You should see the newly created repository in this list.

## Delete an S3 Bucket

You can delete an S3 bucket in Pachyderm from the AWS CLI or MinIO client by running the following command:

- If you are using MinIO, type:

```
$ mc rb local/master.test
Removed `local/master.test` successfully.
```

- If you are using AWS, type:

```
$ aws --endpoint-url http://localhost:30600/ s3 rb s3://master.test
remove_bucket: master.test
```

- If you are using S3cmd, type:

```
$ s3cmd rb s3://master.test
```

## Upload and Download File Objects

For input repositories at the top of your DAG, you can both add files to and download files from the repository.

When you add files, Pachyderm automatically overwrites the previous version of the file if it already exists. Uploading new files is not supported for output repositories, these are the repositories that are the output of a pipeline.

Not all the repositories that you see in the results of the `ls` command are input repositories that can be written to. Some of them might be read-only output repos. Check your pipeline specification to verify which repositories are the input repos.

To add a file to a repository, complete the following steps:

1. Run the `cp` command for your S3 client:

- If you are using MinIO, type:

```
$ mc cp test.csv local/master.raw_data/test.csv
test.csv:          62 B / 62 B    100.00% 206 B/s 0s
```

- If you are using AWS, type:

```
$ aws --endpoint-url http://localhost:30600/ s3 cp test.csv s3://master.raw_
↪data
upload: ./test.csv to s3://master.raw_data/test.csv
```

- If you are using S3cmd, type:

```
$ s3cmd cp test.csv s3://master.raw_data
```

These commands add the `test.csv` file to the `master` branch in the `raw_data` repository. `raw_data` is an input repository.

2. Check that the file was added:

- If you are using MinIO, type:

```
$ mc ls local/master.raw_data
[2019-07-19 12:11:37 PDT] 2.6MiB github_issues_medium.csv
[2019-07-19 12:11:37 PDT] 62B test.csv
```

- If you are using AWS, type:

```
$ aws --endpoint-url http://localhost:30600/ s3 ls s3://master.raw_data/
2019-07-19 12:11:37 2685061 github_issues_medium.csv
2019-07-19 12:11:37 62 test.csv
```

- If you are using S3cmd, type:

```
$ s3cmd ls s3://master.raw_data/
2019-07-19 12:11 2685061 github_issues_medium.csv
2019-07-19 12:11 62 test.csv
```

3. Download a file from MinIO to the current directory by running the following commands:

- If you are using MinIO, type:

```
$ mc cp local/master.raw_data/github_issues_medium.csv .
...hub_issues_medium.csv:  2.56 MiB / 2.56 MiB   100.00% 1.26 MiB/s 2s
```

- If you are using AWS, type:

```
$ aws --endpoint-url http://localhost:30600/ s3 cp s3://master.raw_data/test.
↪csv .
download: s3://master.raw_data/test.csv to ./test.csv
```

- If you are using S3cmd, type:

```
$ s3cmd cp s3://master.raw_data/test.csv .
```

## Remove a File Object

You can delete a file in the `HEAD` of a Pachyderm branch by using the MinIO command-line interface:

1. List the files in the input repository:

- If you are using MinIO, type:

```
$ mc ls local/master.raw_data/
[2019-07-19 12:11:37 PDT]  2.6MiB github_issues_medium.csv
[2019-07-19 12:11:37 PDT]    62B test.csv
```

- If you are using AWS, type:

```
$ aws --endpoint-url http://localhost:30600/ s3 ls s3://master.raw_data
2019-07-19 12:11:37    2685061 github_issues_medium.csv
2019-07-19 12:11:37         62 test.csv
```

- If you are using S3cmd, type:

```
$ s3cmd ls s3://master.raw_data
2019-07-19 12:11    2685061 github_issues_medium.csv
2019-07-19 12:11         62 test.csv
```

2. Delete a file from a repository. Example:

- If you are using MinIO, type:

```
$ mc rm local/master.raw_data/test.csv
Removing `local/master.raw_data/test.csv`.
```

- If you are using AWS, type:

```
$ aws --endpoint-url http://localhost:30600/ s3 rm s3://master.raw_data/test.
↪csv
delete: s3://master.raw_data/test.csv
```

- If you are using S3cmd, type:

```
$ s3cmd rm s3://master.raw_data/test.csv
```

### 3.5.7 Unsupported operations

Some of the S3 functionalities are not yet supported by Pachyderm. If you run any of these operations, Pachyderm returns a standard S3 `NotImplemented` error.

The S3 Gateway does not support the following S3 operations:

- Accelerate
- Analytics
- Object copying. PFS supports this functionality through gRPC.
- CORS configuration
- Encryption
- HTML form uploads
- Inventory
- Legal holds
- Lifecycles
- Logging
- Metrics
- Notifications
- Object locks
- Payment requests
- Policies
- Public access blocks
- Regions
- Replication
- Retention policies
- Tagging
- Torrents
- Website configuration



---

## Deploy Pachyderm

---

This section describes how to deploy Pachyderm on premises or on a supported cloud platform of your choice.

**Note:** Pachyderm supports the Docker runtime only. If you want to deploy Pachyderm on a system that uses another container runtime, ask for advice in our [Slack channel](#).

### 4.1 Overview

Pachyderm runs on [Kubernetes](#) and is backed by an object store of your choice. As such, Pachyderm can run on any platform that supports Kubernetes and an object store. These following docs cover common deployments and related topics:

- [Google Cloud Platform](#)
- [Amazon Web Services](#)
- [Azure](#)
- [OpenShift](#)
- [On Premises](#)
- [Custom Object Stores](#)
- [Migrations](#)
- [Upgrading Pachyderm Versions](#)
- [Non-Default Namespaces](#)
- [RBAC](#)

#### 4.1.1 Usage Metrics

Pachyderm automatically reports anonymized usage metrics. These metrics help us understand how people are using Pachyderm and make it better. They can be disabled by setting the env variable `METRICS` to `false` in the pachd container.

### 4.2 Getting Started with Pachyderm Hub

Pachyderm Hub is a platform for data scientists where you can version-control your data, build analysis pipelines, and track the provenance of your data science workflow.

This section walks you through the steps of creating a cluster in Pachyderm Hub so that you do not need to worry about the underlying infrastructure and can get started using Pachyderm right away.

Pachyderm Hub enables you to preview Pachyderm functionality free of charge by removing the burden of deploying Pachyderm locally or in a third-party cloud platform. Currently, Pachyderm Hub is in beta so clusters cannot be turned into production clusters and should only be used for easy development and testing. Production-grade functionality will be supported in later releases.

**Note:** We'd like to hear your feedback! Let us know what you think about Pachyderm Hub and help us make it better. Join our Slack channel.

### 4.2.1 How it Works

To get started, complete the following steps:

### 4.2.2 Log in

Pachyderm Hub uses GitHub OAuth as an identity provider. Therefore, to start using Pachyderm Hub, you need to log in by authorizing Pachyderm Hub with your GitHub account. If you do not have a GitHub account yet, create one by following the steps described in [Join GitHub](#).

To log in to Pachyderm Hub, complete the following steps:

1. Go to [hub.pachyderm.com](https://hub.pachyderm.com).
2. Click **Try for free**.
3. Authorize Pachyderm Hub with your GitHub account by typing your GitHub user name and password.
4. Proceed to *Step 1*.

### 4.2.3 Step 1: Create a Cluster

To get started, create a Pachyderm cluster on which your pipelines will run. A Pachyderm cluster runs on top of the underlying cloud infrastructure. In Pachyderm Hub, you can create a one-node cluster that you can use for a limited time.

To create a Pachyderm cluster, complete the following steps:

1. If you have not yet done so, log in to Pachyderm Hub.
2. Click **Create cluster**.
3. Type a name for your cluster. For example, `test1`.
4. Click **Create**.

Your cluster is provisioned instantly!

## Clusters


[Create cluster...](#)

NAME	STATUS	EXPIRES	
test-cluster-sv-cykhvod6pw	Running	in 4 hours	<a href="#">Dashboard</a> <a href="#">Connect</a>

**Note:** While Pachyderm maintains a few clusters that are instantly available, none may be available during periods of high traffic. If you see your cluster is in a *starting* state, you might have to wait a few minutes for it to be ready.

5. Proceed to [Step 2](#).

### 4.2.4 Step 2 - Connect to Your Cluster

Pachyderm Hub enables you to access your cluster through a command-line interface (CLI) called `pachctl` and the web interface called the Dashboard. Although you can perform most simple actions directly in the dashboard, `pachctl` provides full functionality. Most likely, you will use `pachctl` for any operation beyond the most basic workflow. Pachyderm recommends that you use `pachctl` for all data operations and the dashboard to view your data and graphical representation of your pipelines.

After you create a cluster, you need to go to the terminal on your computer and configure your CLI to connect to your cluster by installing `pachctl` and configuring your Pachyderm context. For more information about Pachyderm contexts, see [Connect by using a Pachyderm Context](#).

To set the correct Pachyderm context, you need to use the hostname of your cluster that is available in the Pachyderm Hub UI under **Connect**.

**Note:** `kubectl` commands are not supported for the clusters deployed on Pachyderm Hub.

To connect to your cluster, complete the following steps:

1. Install or upgrade `pachctl` as described in [Install pachctl](#).
2. Configure a Pachyderm context and log in to your cluster by using a one-time authentication token:
  - (a) In the Pachyderm Hub UI, click **Connect** next to your cluster.
  - (b) Copy, paste, and run the commands in the instructions in your terminal. These commands create a new Pachyderm context with your cluster details on your machine.

**Note:** If you get the following error, that means that your authentication token has expired:

```
error authenticating with Pachyderm cluster: /pachyderm_auth/auth-codes/
↪e14ccfafb35d4768f4a73b2dc9238b365492b88e98b76929d82ef0c6079e0027 not found
```

To get a new token, refresh the page. Then, use the new token to authenticate.

- (c) Verify that you have set the correct context:

```
$ pachctl config get active-context
```

3. Verify that you can run `pachctl` commands on your cluster:

- (a) Create a repo called `test` :

```
$ pachctl create repo test
```

- (b) Verify that the repo was created:

```
$ pachctl list repo
NAME      CREATED          SIZE (MASTER)  ACCESS LEVEL  OWNER
test      3 seconds ago   0B              OWNER
```

- (c) Go to the dashboard and verify that you can see the repo in the dashboard:

- i. In the Pachyderm Hub UI, click **Dashboard** next to your cluster. The dashboard opens in a new window.

The screenshot shows the Pachyderm Hub UI dashboard. On the left is a sidebar with the 'PACH DASH' header and a navigation menu including 'account-10' (with a 'Log out' link), 'Home', 'Recent Changes', 'Repos' (highlighted), 'Pipelines', 'Jobs', and 'Settings'. The main content area has a blue header with a search bar and a 'test' repo card. The card shows 'Created a few seconds ago', 'Manually Ingested Repo', and 'No commits yet'. Below the card are statistics: 'undefined data files', 'undefined directories', '0 B', '0 commits', '0 branches', and '0 commit trees'. There is an 'Ingest Data' button. The 'Latest content' section shows 'Commit finished a few seconds ago' and 'Nothing here yet!'. The 'Access Control' section states 'You can read, modify, and edit access controls on this repo (You are an owner)' and has a 'Modify access controls' link. The 'Danger Zone' section has a 'DELETE THIS REPO' button.



## 4.2.5 Next Steps

Congratulations! You have successfully deployed and configured a Pachyderm cluster in Pachyderm Hub. Now, you can try out our Beginners tutorial that walks you through the Pachyderm basics.

- [Beginner Tutorial](#)

## 4.3 Google Cloud Platform

Google Cloud Platform has excellent support for Kubernetes, and thus Pachyderm, through the [Google Kubernetes Engine](#) (GKE). The following guide will walk you through deploying a Pachyderm cluster on GCP.

### 4.3.1 Prerequisites

- Google Cloud SDK  $\geq 124.0.0$
- `kubectl`
- `pachctl`

If this is the first time you use the SDK, make sure to follow the [quick start guide](#). Note, this may update your `~/.bash_profile` and point your `$PATH` at the location where you extracted `google-cloud-sdk`. We recommend extracting the SDK to `~/bin`.

Note, you can also install `kubectl` installed via the Google Cloud SDK using:

```
$ gcloud components install kubectl
```

### 4.3.2 Deploy Kubernetes

To create a new Kubernetes cluster via GKE, run:

```
$ CLUSTER_NAME=<any unique name, e.g. "pach-cluster">

$ GCP_ZONE=<a GCP availability zone. e.g. "us-west1-a">

$ gcloud config set compute/zone ${GCP_ZONE}

$ gcloud config set container/cluster ${CLUSTER_NAME}

$ MACHINE_TYPE=<machine type for the k8s nodes, we recommend "n1-standard-4" or
↳larger>

# By default the following command spins up a 3-node cluster. You can change the
↳default with '--num-nodes VAL'.
$ gcloud container clusters create ${CLUSTER_NAME} --scopes storage-rw --machine-type
↳${MACHINE_TYPE}

# By default, GKE clusters have RBAC enabled. To allow 'pachctl deploy' to give the
↳'pachyderm' service account
# the requisite privileges via clusterrolebindings, you will need to grant *your user
↳account* the privileges
# needed to create those clusterrolebindings.
#
# Note that this command is simple and concise, but gives your user account more
↳privileges than necessary. See
# https://docs.pachyderm.io/en/latest/deployment/rbac.html for the complete list of
↳privileges that the
# pachyderm serviceaccount needs.
$ kubectl create clusterrolebinding cluster-admin-binding --clusterrole=cluster-admin
↳--user=$(gcloud config get-value account)
```

**Important Note:** You must create the Kubernetes cluster via the `gcloud` command-line tool rather than the Google Cloud Console, as it's currently only possible to grant the `storage-rw` scope via the command-line tool. Also note, you should deploy a 1.9.x cluster if possible to take full advantage of Pachyderm's latest features.

This may take a few minutes to start up. You can check the status on the [GCP Console](#). A `kubeconfig` entry will automatically be generated and set as the current context. As a sanity check, make sure your cluster is up and running via `kubectl`:

```
# List all pods in the kube-system namespace.
$ kubectl get pods -n kube-system
```

NAME	READY	STATUS	RESTARTS	
↪ AGE				
event-exporter-v0.1.7-5c4d9556cf-fd9j2	2/2	Running	0	└
↪ 1m				
fluentd-gcp-v2.0.9-68vhs	2/2	Running	0	└
↪ 1m				
fluentd-gcp-v2.0.9-fzfpw	2/2	Running	0	└
↪ 1m				
fluentd-gcp-v2.0.9-qvk8f	2/2	Running	0	└
↪ 1m				
heapster-v1.4.3-5fbfb6bf55-xgdwx	3/3	Running	0	└
↪ 55s				
kube-dns-778977457c-7hbrv	3/3	Running	0	└
↪ 1m				
kube-dns-778977457c-dpff4	3/3	Running	0	└
↪ 1m				
kube-dns-autoscaler-7db47cb9b7-gp5ns	1/1	Running	0	└
↪ 1m				
kube-proxy-gke-pach-cluster-default-pool-9762dc84-bzcz	1/1	Running	0	└
↪ 1m				
kube-proxy-gke-pach-cluster-default-pool-9762dc84-hqkr	1/1	Running	0	└
↪ 1m				
kube-proxy-gke-pach-cluster-default-pool-9762dc84-jcbg	1/1	Running	0	└
↪ 1m				
kubernetes-dashboard-768854d6dc-t75rp	1/1	Running	0	└
↪ 1m				
l7-default-backend-6497bcd4d-w72k5	1/1	Running	0	└
↪ 1m				

If you *don't* see something similar to the above output, you can point `kubectl` to the new cluster manually via:

```
# Update your kubeconfig to point at your newly created cluster.
$ gcloud container clusters get-credentials ${CLUSTER_NAME}
```

### 4.3.3 Deploy Pachyderm

To deploy Pachyderm we will need to:

1. *Create some storage resources,*
2. *Install the Pachyderm CLI tool, `pachctl`, and*
3. *Deploy Pachyderm on the k8s cluster*

#### Set up the Storage Resources

Pachyderm needs a [GCS bucket](#) and a [persistent disk](#) to function correctly. We can specify the size of the persistent disk, the bucket name, and create the bucket as follows:

```
# For the persistent disk, 10GB is a good size to start with.
# This stores PFS metadata. For reference, 1GB
# should work fine for 1000 commits on 1000 files.
$ STORAGE_SIZE=<the size of the volume that you are going to create, in GBs. e.g. "10"
↪ ">
```

```
# The Pachyderm bucket name needs to be globally unique across the entire GCP region.
$ BUCKET_NAME=<The name of the GCS bucket where your data will be stored>

# Create the bucket.
$ gsutil mb gs://${BUCKET_NAME}
```

To check that everything has been set up correctly, try:

```
$ gsutil ls
# You should see the bucket you created.
```

## Install pachctl

pachctl is a command-line utility for interacting with a Pachyderm cluster. You can install it locally as follows:

```
# For macOS:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.9

# For Linux (64 bit) or Window 10+ on WSL:
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
  ↳ download/v1.9.7/pachctl_1.9.7_amd64.deb && sudo dpkg -i /tmp/pachctl.deb
```

You can then run `pachctl version --client-only` to check that the installation was successful.

```
$ pachctl version --client-only
1.9.7
```

## Deploy Pachyderm on the k8s cluster

Now we're ready to deploy Pachyderm itself. This can be done in one command:

```
$ pachctl deploy google ${BUCKET_NAME} ${STORAGE_SIZE} --dynamic-etcd-nodes=1
serviceaccount "pachyderm" created
storageclass "etcd-storage-class" created
service "etcd-headless" created
statefulset "etcd" created
service "etcd" created
service "pachd" created
deployment "pachd" created
service "dash" created
deployment "dash" created
secret "pachyderm-storage-secret" created

Pachyderm is launching. Check its status with "kubectl get all"
Once launched, access the dashboard by running "pachctl port-forward"
```

Note, here we are using 1 etcd node to manage Pachyderm metadata. The number of etcd nodes can be adjusted as needed.

**Important Note:** If RBAC authorization is a requirement or you run into any RBAC errors please read our docs on the subject [here](#).

It may take a few minutes for the pachd nodes to be running because it's pulling containers from DockerHub. You can see the cluster status with `kubectl`, which should output the following when Pachyderm is up and running:

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
dash-482120938-np8cc               2/2     Running   0           4m
etcd-0                             1/1     Running   0           4m
pachd-3677268306-9sqm0             1/1     Running   0           4m
```

If you see a few restarts on the `pachd` pod, that's totally ok. That simply means that Kubernetes tried to bring up those containers before other components were ready, so it restarted them.

Finally, assuming your `pachd` is running as shown above, we need to set up forward a port so that `pachctl` can talk to the cluster.

```
# Forward the ports. We background this process because it blocks.
$ pachctl port-forward &
```

And you're done! You can test to make sure the cluster is working by trying `pachctl version` or even creating a new repo.

```
$ pachctl version
COMPONENT      VERSION
pachctl        1.9.7
pachd          1.9.7
```

## 4.3.4 Additional Tips for Performance Improvements

### Increasing Ingress Throughput

One way to improve Ingress performance is to restrict Pachd to a specific, more powerful node in the cluster. This is accomplished by the use of [node-taints](#) in GKE. By creating a node-taint for Pachd, we're telling the kubernetes scheduler that the only pod that should be on that node is Pachd. Once that's completed, you then deploy Pachyderm with the `--pachd-cpu-request` and `--pachd-memory-request` set to match the resources limits of the machine type. And finally, you'll modify the Pachd deployment such that it has an appropriate toleration:

```
tolerations:
- key: "dedicated"
  operator: "Equal"
  value: "pachd"
  effect: "NoSchedule"
```

### Increasing upload performance

The most straightforward approach to increasing upload performance is to simply [leverage SSD's as the boot disk](#) in your cluster as SSD's provide higher throughput and lower latency than standard disks. Additionally, you can increase the size of the SSD for further performance gains as IOPS improve with disk size.

### Increasing merge performance

Performance tweaks when it comes to merges can be done directly in the [Pachyderm pipeline spec](#). More specifically, you can increase the number of hashtrees (hashtree spec) in the pipeline spec. This number determines the number of shards for the filesystem metadata. In general this number should be lower than the number of workers (parallelism spec) and should not be increased unless merge time (the time before the job is done and after the number of processed datums + skipped datums is equal to the total datums) is too slow.



## 4.4 Deploy Pachyderm on Amazon AWS

Pachyderm can run in a Kubernetes cluster deployed in Amazon® Web Services (AWS), whether it is an Elastic Container Service (EKS) or a Kubernetes cluster deployed directly on EC2 by using a deployment tool. AWS removes the need to maintain the underlying virtual cloud. This advantage makes AWS a logical choice for organizations that decide to offload the cloud infrastructure operational burden to a third-party vendor. Pachyderm seamlessly integrates with Amazon EKS and runs in the same fashion as on your computer.

You can install Pachyderm on Amazon AWS by using one of the following options:

**Deploy Pachyderm on Amazon EKS** If you already have an Amazon EKS cluster, you can quickly deploy Pachyderm on top of it. If you are just starting with Amazon EKS, this section guides you through the EKS deployment process.

**Deploy Pachyderm on Amazon EC2 by using `kops`** Instead of EKS, you can deploy Kubernetes on AWS EC2 directly by using a Kubernetes deployment tool such as `kops` and then deploy Pachyderm on that Kubernetes cluster. If you deploy a cluster with `kops`, you remain responsible for the Kubernetes operations and maintenance.

**Deploy Pachyderm with CloudFront** Use this option in production environments that require high throughput and secure data delivery.

### 4.4.1 Deploy Pachyderm on Amazon EKS

Amazon EKS provides an easy way to deploy, configure, and manage Kubernetes clusters. If you want to avoid managing your Kubernetes infrastructure, EKS might be the right choice for your organization. Pachyderm seamlessly deploys on Amazon EKS.

#### Prerequisites

Before you can deploy Pachyderm on an EKS cluster, verify that you have the following prerequisites installed and configured:

- `kubectl`
- `AWS CLI`
- `eksctl`
- `aws-iam-authenticator`.
- `pachctl`

#### Deploy an EKS cluster by using `eksctl`

Use the `eksctl` tool to deploy an EKS cluster in your Amazon AWS environment. The `eksctl create cluster` command creates a virtual private cloud (VPC), a security group, and an IAM role for Kubernetes to create resources. For detailed instructions, see [Amazon documentation](#).

To deploy an EKS cluster, complete the following steps:

1. Deploy an EKS cluster:

```
eksctl create cluster --name <name> --version <version> \
--nodegroup-name <name> --node-type <vm-flavor> \
--nodes <number-of-nodes> --nodes-min <min-number-nodes> \
--nodes-max <max-number-nodes> --node-ami auto
```

**Example output:**

```
[ ] using region us-east-1
[ ] setting availability zones to [us-east-1a us-east-1f]
[ ] subnets for us-east-1a - public:192.168.0.0/19 private:192.168.64.0/19
[ ] subnets for us-east-1f - public:192.168.32.0/19 private:192.168.96.0/19
[ ] nodegroup "pachyderm-test-workers" will use "ami-0f2e8e5663e16b436"
→ [AmazonLinux2/1.13]
[ ] using Kubernetes version 1.13
[ ] creating EKS cluster "pachyderm-test-eks" in "us-east-1" region
[ ] will create 2 separate CloudFormation stacks for cluster itself and the
→ initial nodegroup
[ ] if you encounter any issues, check CloudFormation console or try 'eksctl
→ utils describe-stacks --region=us-east-1 --name=pachyderm-test-eks'
[ ] 2 sequential tasks: { create cluster control plane "svetkars-eks", create
→ nodegroup "pachyderm-test-workers" }
[ ] building cluster stack "eksctl-pachyderm-test-eks-cluster"
[ ] deploying stack "eksctl-pachyderm-test-eks-cluster"

...
[ ] EKS cluster "pachyderm-test" in "us-east-1" region is ready
```

**2. Verify the deployment:**

```
$ kubectl get all
NAME                                TYPE                CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
service/kubernetes                  ClusterIP           10.100.0.1    <none>         443/TCP    7m9s
```

**3. Deploy Pachyderm as described in Deploy Pachyderm on AWS.**

## 4.4.2 Deploy Kubernetes with kops

`kops` is one of the most popular open-source tools that enable you to deploy, manage, and upgrade a Kubernetes cluster in the cloud. By using `kops` you can quickly spin-up a highly-available Kubernetes cluster in a supported cloud platform.

### Prerequisites

Before you can deploy Pachyderm on Amazon AWS with `kops`, you must have the following components configured:

- Install [AWS CLI](#)
- Install [kubectl](#)
- Install [kops](#)
- Install [pachctl](#)
- Install [jq](#)
- Install [uuid](#)

### Configure kops

`kops`, which stands for *Kubernetes Operations*, is an open-source tool that deploys a production-grade Kubernetes cluster on a cloud environment of choice. You need to have access to the AWS Management console to add an Identity and Access Management (IAM) user for `kops`.

For more information about `kops`, see [kops AWS documentation](#). These instructions provide more details about configuring additional cluster parameters, such as enabling version control or encryption on your S3 bucket, and so on.

To configure `kops`, complete the following steps:

1. In the IAM console or by using the command line, create a `kops` group with the following permissions:
  - `AmazonEC2FullAccess`
  - `AmazonRoute53FullAccess`
  - `AmazonS3FullAccess`
  - `IAMFullAccess`
  - `AmazonVPCFullAccess`
2. Add a user that will create a Kubernetes cluster to that group.
3. In the list of users, select that user and navigate to the **Security credentials** tab.
4. Create an access key and save the access and secret keys in a location on your computer.
5. Configure an AWS CLI client:

```
$ aws configure
```

6. Use the access and secret keys to configure the AWS CLI client.
7. Create an S3 bucket for your cluster:

```
$ aws s3api create-bucket --bucket <name> --region <region>
```

**Example:**

```
$ aws s3api create-bucket --bucket test-pachyderm --region us-east-1
{
  "Location": "/test-pachyderm"
}
```

8. Optionally, configure DNS as described in [Configure DNS](#). In this example, a gossip-based cluster that ends with `k8s.local` is deployed.
9. Export the name of your cluster and the S3 bucket for the Kubernetes cluster as variables.

**Example:**

```
export NAME=test-pachyderm.k8s.local
export KOPS_STATE_STORE=s3://test-pachyderm
```

10. Create the cluster configuration:

```
kops create cluster --zones <region> ${NAME}
```

11. Optionally, edit your cluster:

```
kops edit cluster ${NAME}
```

12. Build and deploy the cluster:

```
kops update cluster ${NAME} --yes
```

The deployment might take some time.

13. Run `kops cluster validate` periodically to monitor cluster deployment. When `kops` finishes deploying the cluster, you should see the output similar to the following:

```
$ kops validate cluster
Using cluster from kubectl context: test-pachyderm.k8s.local

Validating cluster svetskars.k8s.local

INSTANCE          GROUPS
NAME              ROLE          MACHINETYPE  MIN  MAX  SUBNETS
master-us-west-2a Master        m3.medium    1    1    us-west-2a
nodes              Node          t2.medium    2    2    us-west-2a

NODE STATUS
NAME                                                    ROLE  READY
ip-172-20-45-231.us-west-2.compute.internal           node   True
ip-172-20-50-8.us-west-2.compute.internal             master True
ip-172-20-58-132.us-west-2.compute.internal           node   True
```

14. Proceed to Deploy Pachyderm on AWS.

### 4.4.3 Deploy Pachyderm on AWS

After you deploy Kubernetes cluster by using `kops` or `eksctl`, you can deploy Pachyderm on top of that cluster.

You need to complete the following steps to deploy Pachyderm:

1. Install `pachctl` as described in [Install pachctl](#).
2. Add stateful storage for Pachyderm as described in [Add Stateful Storage](#).
3. Deploy Pachyderm by using an [IAM role](#) (recommended) or [an access key](#).

#### Add Stateful Storage

Pachyderm requires the following types of persistent storage:

An S3 object store bucket for data. The S3 bucket name must be globally unique across the whole Amazon region. Therefore, add a descriptive prefix to the S3 bucket name, such as your username.

An Elastic Block Storage (EBS) persistent volume (PV) for Pachyderm metadata. Pachyderm recommends that you assign at least 10 GB for this persistent EBS volume. If you expect your cluster to be very long running a scale to thousands of jobs per commits, you might need to go add more storage. However, you can easily increase the size of the persistent volume later.

To add stateful storage, complete the following steps:

1. Set up the following system variables:
  - `BUCKET_NAME` — A globally unique S3 bucket name.
  - `STORAGE_SIZE` — The size of the persistent volume in GB. For example, 10 .
  - `AWS_REGION` — The AWS region of your Kubernetes cluster. For example, `us-west-2` and not `us-west-2a` .
1. Create an S3 bucket:

- If you are creating an S3 bucket in the `us-east-1` region, run the following command:

```
$ aws s3api create-bucket --bucket ${BUCKET_NAME} --region ${AWS_REGION}
```

- If you are creating an S3 bucket in any region but the `us-east-1` region, run the following command:

```
$ aws s3api create-bucket --bucket ${BUCKET_NAME} --region ${AWS_REGION} --
→create-bucket-configuration LocationConstraint=${AWS_REGION}
```

2. Verify that the S3 bucket was created:

```
$ aws s3api list-buckets --query 'Buckets[].Name'
```

## Deploy Pachyderm with an IAM Role

IAM roles provide better user management and security capabilities compared to access keys. If a malicious user gains access to an access key, your data might become compromised. Therefore, enterprises often opt out to use IAM roles rather than access keys for production deployments.

You need to configure the following IAM settings:

- The worker nodes on which Pachyderm is deployed must be associated with the IAM role that is assigned to the Kubernetes cluster. If you created your cluster by using `kops` or `eksctl` the nodes must have a dedicated IAM role already assigned.
- The IAM role must have access to the S3 bucket that you created for Pachyderm.
- The IAM role must have correct trust relationships.

You need to set a system variable `IAM_ROLE` to the name of the IAM role that you will use to deploy the cluster. This role is different from the Role ARN or the Instance Profile ARN of the role. It is the actual role name.

To deploy Pachyderm with an IAM role, complete the following steps:

1. Find the IAM role assigned to the cluster:
  - (a) Go to the AWS Management console.
  - (b) Select an EC2 instance in the Kubernetes cluster.
  - (c) Click **Description**.
  - (d) Find the **IAM Role** field.
2. Enable access to the S3 bucket for the IAM role:
  - (a) In the **IAM Role** field, click on the IAM role.
  - (b) In the **Permissions** tab, click **Edit policy**.
  - (c) Select the **JSON** tab.
  - (d) Append the following text to the end of the existing JSON:

```
{
  "Effect": "Allow",
  "Action": [
    "s3:ListBucket"
  ],
  "Resource": [
    "arn:aws:s3:::<your-bucket>"
  ]
}
```

```

    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "s3:PutObject",
      "s3:GetObject",
      "s3:DeleteObject"
    ],
    "Resource": [
      "arn:aws:s3:::<your-bucket>/*"
    ]
  }
}

```

Replace `<your-bucket>` with the name of your S3 bucket.

**Note:** For the EKS cluster, you might need to use the **Add inline policy** button and create a name for the new policy. The JSON above is inserted between the square brackets for the `Statement` element.

3. Set up trust relationships for the IAM role:

- (a) Click the **Trust relationships > Edit trust relationship**.
- (b) Ensure that you see a statement with `sts:AssumeRole`. Example:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}

```

4. Set the system variable `IAM_ROLE` to the IAM role name for the Pachyderm deployment.

5. Deploy Pachyderm:

```

$ pachctl deploy amazon ${BUCKET_NAME} ${AWS_REGION} ${STORAGE_SIZE} --dynamic-
→etcd-nodes=1 --iam-role ${IAM_ROLE}

```

The deployment takes some time. You can run `kubectl get pods` periodically to check the status of deployment. When Pachyderm is deployed, the command shows all pods as `READY`:

```

$ kubectl get pods
NAME                                READY    STATUS    RESTARTS    AGE
dash-6c9dc97d9c-89dv9              2/2     Running    0            1m
etcd-0                              1/1     Running    0            4m
pachd-65fd68d6d4-8vjq7             1/1     Running    0            4m

```

**Note:** If you see a few restarts on the `pachd` nodes, it means that Kubernetes tried to bring up those pods before `etcd` was ready. Therefore, Kubernetes restarted those pods. You can safely ignore this message.

6. Verify that the Pachyderm cluster is up and running:

```
$ pachctl version
```

COMPONENT	VERSION
pachctl	1.9.1
pachd	1.9.1

- If you want to access the Pachyderm UI or use the S3 gateway, you need to forward Pachyderm ports. Open a new terminal window and run the following command:

```
$ pachctl port-forward
```

## Deploy Pachyderm with an Access Key

When you installed `kops`, you created a dedicated IAM user with access credentials such as an access key and secret key. You can deploy Pachyderm by using the credentials of this IAM user directly. However, deploying Pachyderm with an access key might not satisfy your enterprise security requirements. Therefore, deploying with an IAM role is preferred.

To deploy Pachyderm with an access key, complete the following steps:

1. Run the following command to deploy your Pachyderm cluster:

```
$ pachctl deploy amazon ${BUCKET_NAME} ${AWS_REGION} ${STORAGE_SIZE} --dynamic-
→etcd-nodes=1 --credentials "${AWS_ACCESS_KEY_ID},${AWS_SECRET_ACCESS_KEY},"
```

The `,` at the end of the `credentials` flag in the deploy command is for an optional temporary AWS token. You might use such a token if you are just experimenting with Pachyderm. However, do not use this token in a production deployment.

The deployment takes some time. You can run `kubectl get pods` periodically to check the status of deployment. When Pachyderm is deployed, the command shows all pods as `READY`:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
dash-6c9dc97d9c-89dv9	2/2	Running	0	1m
etcd-0	1/1	Running	0	4m
pachd-65fd68d6d4-8vjq7	1/1	Running	0	4m

**Note:** If you see a few restarts on the `pachd` nodes, it means that Kubernetes tried to bring up those pods before `etcd` was ready. Therefore, Kubernetes restarted those pods. You can safely ignore this message.

2. Verify that the Pachyderm cluster is up and running:

```
$ pachctl version
```

COMPONENT	VERSION
pachctl	1.9.1
pachd	1.9.1

- If you want to access the Pachyderm UI or use S3 gateway, you need to forward Pachyderm ports. Open a new terminal window and run the following command:

```
$ pachctl port-forward
```

#### 4.4.4 Deploy a Pachyderm Cluster with CloudFront

After you have an EKS cluster or a Kubernetes cluster deployed with `kops` ready, you can integrate it with Amazon CloudFront™.

Amazon CloudFront is a content delivery network (CDN) that streams data to your website, service, or application securely and with great performance. Pachyderm recommends that you set up Pachyderm with CloudFront for all production deployments.

To deploy Pachyderm cluster with CloudFront, complete the following steps:

1. [Create a CloudFront Distribution](#)
2. Deploy Pachyderm with an IAM role
3. [Apply the CloudFront Key Pair](#)

##### Apply the CloudFront Key Pair

If you need to create signed URLs and signed cookies for the data that goes to Pachyderm, you need to configure your AWS account to use a valid CloudFront key pair. Only a root AWS account can generate these secure credentials. Therefore, you might need to request your IT department to create them for you.

For more information, see the [Amazon documentation](#).

The CloudFront key pair includes the following attributes:

- The private and public key. For this deployment, you only need the private key.
- The key pair ID. Typically, the key pair ID is recorded in the filename.

##### Example:

```
rsa-APKAXXXXXXXXXXXXXXXXXX.pem
pk-APKAXXXXXXXXXXXXXXXXXX.pem
```

The key-pair ID is `APKAXXXXXXXXXXXXXXXXXX`. The other file is the private key, which looks similar to the following text:

```
$ cat pk-APKAXXXXXXXXXXXXXXXXXX.pem
-----BEGIN RSA PRIVATE KEY-----
...
```

To apply this key pair to your CloudFront distribution, complete the following steps:

1. Download the `secure-cloudfront.sh` script from the Pachyderm repository:

```
$ curl -o secure-cloudfront.sh https://raw.githubusercontent.com/pachyderm/
↳ pachyderm/master/etc/deploy/cloudfront/secure-cloudfront.sh
```

2. Make the script executable:

```
$ chmod +x secure-cloudfront.sh
```

3. From the `deploy.log` file, obtain the S3 bucket name for your deployment and the CloudFront distribution ID.

4. Apply the key pair to your CloudFront distribution:

```
$ ./secure-cloudfront.sh --region us-west-2 --zone us-west-2c --bucket YYYY-
↳ pachyderm-store --cloudfront-distribution-id E1BEBVLIDYTLV --cloudfront-
↳ keypair-id APKAXXXXXXXXXXXXXXXXXX --cloudfront-private-key-file ~/Downloads/pk-
↳ APKAXXXXXXXXXXXXXXXXXX.pem
```



- Restart the `pachd` pod for the changes to take effect:

```
$ kubectl scale --replicas=0 deployment/pachd && kubectl scale --replicas=1
→ deployment/pachd && kubectl get pod
```

- Verify the setup by checking the `pachd` logs and confirming that Kubernetes uses the CloudFront credentials:

```
$ kubectl get pod
NAME                                READY    STATUS    RESTARTS   AGE
etcd-0                             1/1     Running   0           19h
etcd-1                             1/1     Running   0           19h
etcd-2                             1/1     Running   0           19h
pachd-2796595787-9x0qf             1/1     Running   0           16h

$ kubectl logs pachd-2796595787-9x0qf | grep cloudfront
2017-06-09T22:56:27Z INFO  AWS deployed with cloudfront distribution at
→ d3j9kenawdv8p0
2017-06-09T22:56:27Z INFO  Using cloudfront security credentials - keypair ID
→ (APKAXXXXXXXXXX) - to sign cloudfront URLs
```

## 4.5 Azure

You can deploy Pachyderm in a new or existing Microsoft® Azure® Kubernetes Service environment and use Azure's resource to run your Pachyderm workloads. To deploy Pachyderm to AKS, you need to:

- Install Prerequisites*
- Deploy Kubernetes*
- Deploy Pachyderm*

### 4.5.1 Install Prerequisites

Before you can deploy Pachyderm on Azure, you need to configure a few prerequisites on your client machine. If not explicitly specified, use the latest available version of the components listed below. Install the following prerequisites:

- [Azure CLI 2.0.1 or later](#)
- `jq`
- `kubectl`
- `pachctl`

#### Install `pachctl`

`pachctl` is a primary command-line utility for interacting with Pachyderm clusters. You can run the tool on Linux®, macOS®, and Microsoft® Windows® 10 or later operating systems and install it by using your favorite command line package manager. This section describes how you can install `pachctl` by using `brew` and `curl`.

If you are installing `pachctl` on Windows, you need to first install Windows Subsystem (WSL) for Linux.

To install `pachctl`, complete the following steps:

- To install on macOS by using `brew`, run the following command:

```
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.9
```

- To install on Linux 64-bit or Windows 10 or later, run the following command:

```
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
↳download/v1.9.7/pachctl_1.9.7_amd64.deb && sudo dpkg -i /tmp/pachctl.deb
```

1. Verify your installation by running `pachctl version`:

```
$ pachctl version --client-only
COMPONENT      VERSION
pachctl        1.9.0
```

### 4.5.2 Deploy Kubernetes

You can deploy Kubernetes on Azure by following the official [Azure Container Service documentation](#) or by following the steps in this section. When you deploy Kubernetes on Azure, you need to specify the following parameters:

To deploy Kubernetes on Azure, complete the following steps:

1. Log in to Azure:

```
$ az login
Note, we have launched a browser for you to login. For old experience with
device code, use "az login --use-device-code"
```

If you have not already logged in this command opens a browser window. Log in with your Azure credentials. After you log in, the following message appears in the command prompt:

```
You have logged in. Now let us find all the subscriptions to which you have
↳access...
[
  {
    "cloudName": "AzureCloud",
    "id": "your_id",
    "isDefault": true,
    "name": "Microsoft Azure Sponsorship",
    "state": "Enabled",
    "tenantId": "your_tenant_id",
    "user": {
      "name": "your_contact_id",
      "type": "user"
    }
  }
]
```

2. Create an Azure resource group.

```
$ az group create --name=${RESOURCE_GROUP} --location=${LOCATION}
```

#### Example:

```
$ az group create --name="test-group" --location=centralus
{
  "id": "/subscriptions/6c9f2e1e-0eba-4421-b4cc-172f959ee110/resourceGroups/pach-
↳resource-group",
  "location": "centralus",
```

```

"managedBy": null,
"name": "pach-resource-group",
"properties": {
  "provisioningState": "Succeeded"
},
"tags": null,
"type": null
}

```

### 3. Create an AKS cluster:

```

$ az aks create --resource-group ${RESOURCE_GROUP} --name ${CLUSTER_NAME} --
→generate-ssh-keys --node-vm-size ${NODE_SIZE}

```

#### Example:

```

$ az aks create --resource-group test-group --name test-cluster --generate-ssh-
→keys --node-vm-size Standard_DS4_v2
{
  "aadProfile": null,
  "addonProfiles": null,
  "agentPoolProfiles": [
    {
      "availabilityZones": null,
      "count": 3,
      "enableAutoScaling": null,
      "maxCount": null,
      "maxPods": 110,
      "minCount": null,
      "name": "nodepool1",
      "orchestratorVersion": "1.12.8",
      "osDiskSizeGb": 100,
      "osType": "Linux",
      "provisioningState": "Succeeded",
      "type": "AvailabilitySet",
      "vmSize": "Standard_DS4_v2",
      "vnetSubnetId": null
    }
  ],
  ...
}

```

### 4. Confirm the version of the Kubernetes server:

```

$ kubectl version
Client Version: version.Info{Major:"1", Minor:"13", GitVersion:"v1.13.4",
→GitCommit:"c27b913fddd1a6c480c229191a087698aa92f0b1", GitTreeState:"clean",
→BuildDate:"2019-03-01T23:36:43Z", GoVersion:"go1.12", Compiler:"gc", Platform:
→"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"13", GitVersion:"v1.13.4",
→GitCommit:"c27b913fddd1a6c480c229191a087698aa92f0b1", GitTreeState:"clean",
→BuildDate:"2019-02-28T13:30:26Z", GoVersion:"go1.11.5", Compiler:"gc", Platform:
→"linux/amd64"}

```

#### See also:

- [Azure Virtual Machine sizes](#)

### 4.5.3 Add storage resources

Pachyderm requires you to deploy an object store and a persistent volume in your cloud environment to function correctly. For best results, you need to use faster disk drives, such as *Premium SSD Managed Disks* that are available with the Azure Premium Storage offering.

You need to specify the following parameters when you create storage resources:

To create these resources, follow these steps:

1. Clone the [Pachyderm GitHub repo](#).
2. Change the directory to the root directory of the `pachyderm` repository.
3. Create an Azure storage account:

```
$ az storage account create \
  --resource-group="${RESOURCE_GROUP}" \
  --location="${LOCATION}" \
  --sku=Premium_LRS \
  --name="${STORAGE_ACCOUNT}" \
  --kind=BlockBlobStorage
```

**System response:**

```
{
  "accessTier": null,
  "creationTime": "2019-06-20T16:05:55.616832+00:00",
  "customDomain": null,
  "enableAzureFilesAadIntegration": null,
  "enableHttpsTrafficOnly": false,
  "encryption": {
    "keySource": "Microsoft.Storage",
    "keyVaultProperties": null,
    "services": {
      "blob": {
        "enabled": true,
      }
    }
  },
  ...
}
```

Make sure that you set Stock Keeping Unit (SKU) to `Premium_LRS` and the `kind` parameter is set to `BlockBlobStorage`. This configuration results in a storage that uses SSDs rather than standard Hard Disk Drives (HDD). If you set this parameter to an HDD-based storage option, your Pachyderm cluster will be too slow and might malfunction.

4. Verify that your storage account has been successfully created:

```
$ az storage account list
```

5. Build a Microsoft tool for creating Azure VMs from an image:

```
$ STORAGE_KEY="$(az storage account keys list \
  --account-name="${STORAGE_ACCOUNT}" \
  --resource-group="${RESOURCE_GROUP}" \
  --output=json \
  | jq '.[0].value' -r
)"
```

6. Find the generated key in the **Storage accounts > Access keys** section in the Azure Portal or by running the following command:

```
$ az storage account keys list --account-name=${STORAGE_ACCOUNT}
[
  {
    "keyName": "key1",
    "permissions": "Full",
    "value": ""
  }
]
```

**See Also**

- [Azure Storage](#)

## 4.5.4 Deploy Pachyderm

After you complete all the sections above, you can deploy Pachyderm on Azure. If you have previously tried to run Pachyderm locally, make sure that you are using the right Kubernetes context. Otherwise, you might accidentally deploy your cluster on Minikube.

1. Verify cluster context:

```
$ kubectl config current-context
```

This command should return the name of your Kubernetes cluster that runs on Azure.

- If you have a different contents displayed, configure `kubectl` to use your Azure configuration:

```
$ az aks get-credentials --resource-group ${RESOURCE_GROUP} --name ${CLUSTER_NAME}
Merged "${CLUSTER_NAME}" as current context in /Users/test-user/.kube/config
```

2. Run the following command:

```
$ pachctl deploy microsoft ${CONTAINER_NAME} ${STORAGE_ACCOUNT} ${STORAGE_KEY} $
↪ ${STORAGE_SIZE} --dynamic-etcd-nodes 1
```

**Example:**

```
$ pachctl deploy microsoft test-container teststorage <key> 10 --dynamic-etcd-
↪ nodes 1
serviceaccount/pachyderm configured
clusterrole.rbac.authorization.k8s.io/pachyderm configured
clusterrolebinding.rbac.authorization.k8s.io/pachyderm configured
service/etcd-headless created
statefulset.apps/etcd created
service/etcd configured
service/pachd configured
deployment.apps/pachd configured
service/dash configured
deployment.apps/dash configured
secret/pachyderm-storage-secret configured

Pachyderm is launching. Check its status with "kubectl get all"
Once launched, access the dashboard by running "pachctl port-forward"
```

Because Pachyderm pulls containers from DockerHub, it might take some time before the `pachd` pods start. You can check the status of the deployment by periodically running `kubectl get all`.

- When pachyderm is up and running, get the information about the pods:

```
$ kubectl get pods
NAME                                READY    STATUS    RESTARTS   AGE
dash-482120938-vdlg9              2/2     Running   0           54m
etcd-0                             1/1     Running   0           54m
pachd-1971105989-mjn61            1/1     Running   0           54m
```

**Note:** Sometimes Kubernetes tries to start `pachd` nodes before the `etcd` nodes are ready which might result in the `pachd` nodes restarting. You can safely ignore those restarts.

- To connect to the cluster from your local machine, such as your laptop, set up port forwarding to enable `pachctl` and cluster communication:

```
$ pachctl port-forward
```

- Verify that the cluster is up and running:

```
$ pachctl version
COMPONENT    VERSION
pachctl      1.9.0
pachd        1.9.0
```

## 4.6 OpenShift

**OpenShift** is a popular enterprise Kubernetes distribution. Pachyderm can run on OpenShift with a few small tweaks in the deployment process, which will be outlined below. Please see [known issues](#) below for currently issues with OpenShift deployments.

### 4.6.1 Prerequisites

Pachyderm needs a few things to install and run successfully in any Kubernetes environment

- A persistent volume, used by Pachyderm's `etcd` for storage of system metadata. The kind of PV you provision will be dependent on your infrastructure. For example, many on-premises deployments use Network File System (NFS) access to some kind of enterprise storage.
- An object store, used by Pachyderm's `pachd` for storing all your data. The object store you use will probably be dependent on where you're going to run OpenShift: S3 for [AWS](#), GCS for [Google Cloud Platform](#), Azure Blob Storage for [Azure](#), or a storage provider like Minio, EMC's ECS or Swift providing S3-compatible access to enterprise storage for on-premises deployment.
- Access to particular TCP/IP ports for communication.

#### Persistent volume

You'll need to create a persistent volume with enough space for the metadata associated with the data you plan to store Pachyderm. The `pachctl deploy` command for AWS, GCP and Azure creates persistent storage for you, when you follow the instructions below. A custom deploy can also create storage. We'll show you below how to take out the PV that's automatically created, in case you want to create it outside of the Pachyderm deployment and just consume it.

We're currently developing good rules of thumb for scaling this storage as your Pachyderm deployment grows, but it looks like 10G of disk space is sufficient for most purposes.

## Object store

Size your object store generously, once you start using Pachyderm, you'll start versioning all your data. You'll need four items to configure object storage

1. The access endpoint. For example, Minio's endpoints are usually something like `minio-server:9000`. Don't begin it with the protocol; it's an endpoint, not an url.
2. The bucket name you're dedicating to Pachyderm. Pachyderm will need exclusive access to this bucket.
3. The access key id for the object store. This is like a user name for logging into the object store.
4. The secret key for the object store. This is like the above user's password.

## TCP/IP ports

For more details on how Kubernetes networking and service definitions work, see the [Kubernetes services documentation](#).

### Incoming ports (port)

These are the ports internal to the containers, You'll find these on both the `pachd` and `dash` containers. OpenShift runs containers and pods as unprivileged users which don't have access to port numbers below 1024. Pachyderm's default manifests use ports below 1024, so you'll have to modify the manifests to use other port numbers. It's usually as easy as adding a "1" in front of the port numbers we use.

### Pod ports (targetPort)

This is the port exposed by the pod to Kubernetes, which is forwarded to the `port`. You should leave the `targetPort` set at 0 so it will match the `port` definition.

### External ports (nodePorts)

This is the port accessible from outside of Kubernetes. You probably don't need to change `nodePort` values unless your network security requirements or architecture requires you to change to another method of access. Please see the [Kubernetes services documentation](#) for details.

## 4.6.2 The OCPify script

A bash script that automates many of the substitutions below is available at [this gist](#). You can use it to modify a manifest created using the `--dry-run` flag to `pachctl deploy custom`, as detailed below, and then use this guide to ensure the modifications it makes are relevant to your OpenShift environment. It requires certain prerequisites, just as `jq` and `sponge`, found in [moreutils](#).

This script may be useful as a basis for automating redeployments of Pachyderm as needed.

### Best practices: Infrastructure as code

We highly encourage you to apply the best practices used in developing software to managing the deployment process.

1. Create scripts that automate as much of your processes as possible and keep them under version control.

2. Keep copies of all artifacts, such as manifests, produced by those scripts and keep those under version control.
3. Document your practices in the code and outside it.

### 4.6.3 Preparing to deploy Pachyderm

Things you'll need

1. Your PV. It can be created separately.
2. Your object store information
3. Your project in OpenShift
4. A text editor for editing your deployment manifest

### 4.6.4 Deploying Pachyderm

#### 1. Setting up PV and object stores

How you deploy Pachyderm on OpenShift is largely going to depend on where OpenShift is deployed. Below you'll find links to the documentation for each kind of deployment you can do. Follow the instructions there for setting up persistent volumes and object storage resources. Don't yet deploy your manifest, come back here after you've set up your PV and object store.

- OpenShift Deployed on [AWS](#)
- OpenShift Deployed on [GCP](#)
- OpenShift Deployed on [Azure](#)
- OpenShift Deployed [on-premise](#)

#### 2. Determine your role security policy

Pachyderm is deployed by default with cluster roles. Many institutional Openshift security policies require namespace-local roles rather than cluster roles. If your security policies require namespace-local roles, use the *`pachctl`* *deploy* command below with the *`--local-roles`* flag.

#### 3. Run the deploy command with `--dry-run`

Once you have your PV, object store, and project, you can create a manifest for editing using the `--dry-run` argument to `pachctl deploy`. That step is detailed in the deployment instructions for each type of deployment, above.

Below, find examples, with cluster roles and with namespace-local roles, using AWS elastic block storage as a persistent disk with a custom deploy. We'll show how to remove this PV in case you want to use a PV you create separately.

#### Cluster roles

```
$ pachctl deploy custom --persistent-disk aws --object-store s3 \  
    <pv-storage-name> <pv-storage-size> \  
    <s3-bucket-name> <s3-access-key-id> <s3-access-secret-key> <s3-access-endpoint-  
↪url> \  
    --static-etcd-volume=<pv-storage-name> > manifest.json
```



## Namespace-local roles

```
$ pachctl deploy custom --persistent-disk aws --object-store s3 \
  <pv-storage-name> <pv-storage-size> \
  <s3-bucket-name> <s3-access-key-id> <s3-access-secret-key> <s3-access-endpoint-
↪url> \
  --static-etcd-volume=<pv-storage-name> --local-roles > manifest.json
```

## 4. Modify pachd Service ports

In the deployment manifest, which we called `manifest.json`, above, find the stanza for the `pachd` Service. An example is shown below.

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "pachd",
    "namespace": "default",
    "creationTimestamp": null,
    "labels": {
      "app": "pachd",
      "suite": "pachyderm"
    },
    "annotations": {
      "prometheus.io/port": "9091",
      "prometheus.io/scrape": "true"
    }
  },
  "spec": {
    "ports": [
      {
        "name": "api-grpc-port",
        "port": 650,
        "targetPort": 0,
        "nodePort": 30650
      },
      {
        "name": "trace-port",
        "port": 651,
        "targetPort": 0,
        "nodePort": 30651
      },
      {
        "name": "api-http-port",
        "port": 652,
        "targetPort": 0,
        "nodePort": 30652
      },
      {
        "name": "saml-port",
        "port": 654,
        "targetPort": 0,
        "nodePort": 30654
      }
    ]
  }
}
```

```
        "name": "api-git-port",
        "port": 999,
        "targetPort": 0,
        "nodePort": 30999
    },
    {
        "name": "s3gateway-port",
        "port": 600,
        "targetPort": 0,
        "nodePort": 30600
    }
],
"selector": {
    "app": "pachd"
},
"type": "NodePort"
},
"status": {
    "loadBalancer": {}
}
}
```

While the nodePort declarations are fine, the port declarations are too low for OpenShift. Good example values are shown below.

```
"spec": {
  "ports": [
    {
      "name": "api-grpc-port",
      "port": 1650,
      "targetPort": 0,
      "nodePort": 30650
    },
    {
      "name": "trace-port",
      "port": 1651,
      "targetPort": 0,
      "nodePort": 30651
    },
    {
      "name": "api-http-port",
      "port": 1652,
      "targetPort": 0,
      "nodePort": 30652
    },
    {
      "name": "saml-port",
      "port": 1654,
      "targetPort": 0,
      "nodePort": 30654
    },
    {
      "name": "api-git-port",
      "port": 1999,
      "targetPort": 0,
      "nodePort": 30999
    },
    {
```

```

        "name": "s3gateway-port",
        "port": 1600,
        "targetPort": 0,
        "nodePort": 30600
    },
    ],

```

## 5. Modify pachd Deployment ports and add environment variables

In this case you're editing two parts of the `pachd` Deployment json. Here, we'll omit the example of the unmodified version. Instead, we'll show you the modified version.

### 5.1 pachd Deployment ports

The `pachd` Deployment also has a set of port numbers in the spec for the `pachd` container. Those must be modified to match the port numbers you set above for each port.

```

{
  "kind": "Deployment",
  "apiVersion": "apps/v1",
  "metadata": {
    "name": "pachd",
    "namespace": "default",
    "creationTimestamp": null,
    "labels": {
      "app": "pachd",
      "suite": "pachyderm"
    }
  },
  "spec": {
    "replicas": 1,
    "selector": {
      "matchLabels": {
        "app": "pachd",
        "suite": "pachyderm"
      }
    },
    "template": {
      "metadata": {
        "name": "pachd",
        "namespace": "default",
        "creationTimestamp": null,
        "labels": {
          "app": "pachd",
          "suite": "pachyderm"
        },
        "annotations": {
          "iam.amazonaws.com/role": ""
        }
      },
      "spec": {
        "volumes": [
          {
            "name": "pach-disk"
          }
        ]
      }
    }
  }
}

```

```
        {
          "name": "pachyderm-storage-secret",
          "secret": {
            "secretName": "pachyderm-storage-secret"
          }
        }
      ],
      "containers": [
        {
          "name": "pachd",
          "image": "pachyderm/pachd:1.9.0rc1",
          "ports": [
            {
              "name": "api-grpc-port",
              "containerPort": 1650,
              "protocol": "TCP"
            },
            {
              "name": "trace-port",
              "containerPort": 1651
            },
            {
              "name": "api-http-port",
              "containerPort": 1652,
              "protocol": "TCP"
            },
            {
              "name": "peer-port",
              "containerPort": 1653,
              "protocol": "TCP"
            },
            {
              "name": "api-git-port",
              "containerPort": 1999,
              "protocol": "TCP"
            },
            {
              "name": "saml-port",
              "containerPort": 1654,
              "protocol": "TCP"
            }
          ]
        }
      ],
```

## 5.2 Add environment variables

There are six environment variables necessary for OpenShift

1. `WORKER_USES_ROOT` : This controls whether worker pipelines run as the root user or not. You'll need to set it to `false`
2. `PORT` : This is the grpc port used by pachd for communication with `pachctl` and the api. It should be set to the same value you set for `api-grpc-port` above.
3. `PPROF_PORT` : This is used for Prometheus. It should be set to the same value as `trace-port` above.
4. `HTTP_PORT` : The port for the api proxy. It should be set to `api-http-port` above.

5. `PEER_PORT` : Used to coordinate `pachd` 's. Same as `peer-port` above.
6. `PPS_WORKER_GRPC_PORT` : Used to talk to pipelines. Should be set to a value above 1024. The example value of 1680 below is recommended.

The added values below are shown inserted above the `PACH_ROOT` value, which is typically the first value in this array. The rest of the stanza is omitted for clarity.

```
"env": [
  {
    "name": "WORKER_USES_ROOT",
    "value": "false"
  },
  {
    "name": "PORT",
    "value": "1650"
  },
  {
    "name": "PPROF_PORT",
    "value": "1651"
  },
  {
    "name": "HTTP_PORT",
    "value": "1652"
  },
  {
    "name": "PEER_PORT",
    "value": "1653"
  },
  {
    "name": "PPS_WORKER_GRPC_PORT",
    "value": "1680"
  },
  {
    "name": "PACH_ROOT",
    "value": "/pach"
  },
],
```

## 6. (Optional) Remove the PV created during the deploy command

If you're using a PV you've created separately, remove the PV that was added to your manifest by `pachctl deploy --dry-run`. Here's the example PV we created with the deploy command we used above, so you can recognize it.

```
{
  "kind": "PersistentVolume",
  "apiVersion": "v1",
  "metadata": {
    "name": "etcd-volume",
    "namespace": "default",
    "creationTimestamp": null,
    "labels": {
      "app": "etcd",
      "suite": "pachyderm"
    }
  },
  "spec": {
    "capacity": {
      "storage": "10Gi"
    }
  }
}
```

```
    },
    "awsElasticBlockStore": {
      "volumeID": "pach-disk",
      "fsType": "ext4"
    },
    "accessModes": [
      "ReadWriteOnce"
    ],
    "persistentVolumeReclaimPolicy": "Retain"
  },
  "status": {}
}
```

### 4.6.5 7. Deploy the Pachyderm manifest you modified.

```
$ oc create -f pachyderm.json
```

You can see the cluster status by using `oc get pods` as in upstream Kubernetes:

```
$ oc get pods
NAME                                READY    STATUS    RESTARTS   AGE
dash-6c9dc97d9c-89dv9              2/2     Running   0           1m
etcd-0                              1/1     Running   0           4m
pachd-65fd68d6d4-8vjq7              1/1     Running   0           4m
```

### Known issues

Problems related to OpenShift deployment are tracked in [issues with the “openshift” label](#).

## 4.7 On Premises

This document is broken down into the following sections, available at the links below

- [Introduction to on-premises deployments](#) takes you through what you need to know about Kubernetes, persistent volumes, object stores and best practices. That’s this page.
- Customizing your Pachyderm deployment for on-premises use details the various options of the `pachctl deploy custom ...` command for an on-premises deployment.
- Single-node Pachyderm deployment is the document you should read when deploying Pachyderm for personal, low-volume usage.
- Registries takes you through on-premises, private Docker registry configuration.
- Ingress details the Kubernetes ingress configuration you’d need for using `pachctl` and the dashboard outside of the Kubernetes cluster
- Non-cloud object stores discusses common configurations for on-premises object stores.

Need information on a particular flavor of Kubernetes or object store? Check out the [see also](#) section.

Troubleshooting a deployment? Check out [Troubleshooting Deployments](#).

### 4.7.1 Introduction

Deploying Pachyderm successfully on-premises requires a few prerequisites and some planning. Pachyderm is built on [Kubernetes](#). Before you can deploy Pachyderm, you or your Kubernetes administrator will need to perform the following actions:

1. *Deploy Kubernetes* on-premises.
2. *Deploy a Kubernetes persistent volume* that Pachyderm will use to store administrative data.
3. *Deploy an on-premises object store* using a storage provider like [MinIO](#), [EMC's ECS](#), or [SwiftStack](#) to provide S3-compatible access to your on-premises storage.
4. Create a Pachyderm manifest by running the `pachctl deploy custom` command with appropriate arguments and the `--dry-run` flag to create a Kubernetes manifest for the Pachyderm deployment.
5. *Edit the Pachyderm manifest* for your particular Kubernetes deployment

In this series of documents, we'll take you through the steps unique to Pachyderm. We assume you have some Kubernetes knowledge. We will point you to external resources for the general Kubernetes steps to give you background.

### 4.7.2 Best practices

#### Infrastructure as code

We highly encourage you to apply the best practices used in developing software to managing the deployment process.

1. Create scripts that automate as much of your processes as possible and keep them under version control.
2. Keep copies of all artifacts, such as manifests, produced by those scripts and keep those under version control.
3. Document your practices in the code and outside it.

#### Infrastructure in general

Be sure that you design your Kubernetes infrastructure in accordance with recommended guidelines. Don't mix on-premises Kubernetes and cloud-based storage. It's important that bandwidth to your storage deployment meet the guidelines of your storage provider.

### 4.7.3 Prerequisites

#### Software you will need

1. `kubectl`
2. `pachctl`

### 4.7.4 Setting up to deploy on-premises

#### Deploying Kubernetes

The Kubernetes docs have instructions for [deploying Kubernetes in a variety of on-premise scenarios](#). We recommend following one of these guides to get Kubernetes running on premise.

## Deploying a persistent volume

### Persistent volumes: how do they work?

A Kubernetes [persistent volume](#) is used by Pachyderm's `etcd` for storage of system metadata. In Kubernetes, [persistent volumes](#) are a mechanism for providing storage for consumption by the users of the cluster. They are provisioned by the cluster administrators. In a typical enterprise Kubernetes deployment, the administrators have configured persistent volumes that your Pachyderm deployment will consume by means of a [persistent volume claim](#) in the Pachyderm manifest you generate.

You can deploy PV's to Pachyderm using our command-line arguments in three ways: using a static PV, with StatefulSets, or with StatefulSets using a StorageClass.

If your administrators are using [selectors](#), or you want to use StorageClasses in a different way, you'll need to *edit the Pachyderm manifest* appropriately before applying it.

### Static PV

In this case, `etcd` will be deployed in Pachyderm as a [ReplicationController](#) with one (1) pod that uses a static PV. This is a common deployment for testing.

### StatefulSets

[StatefulSets](#) are a mechanism provided in Kubernetes 1.9 and newer to manage the deployment and scaling of applications. It uses either [Persistent Volume Provisioning](#) or pre-provisioned PV's.

If you're using StatefulSets in your Kubernetes cluster, you will need to find out the particulars of your cluster's PV configuration and *use appropriate flags to `pachctl deploy custom`*

### StorageClasses

If your administrators require specification of [classes](#) to consume persistent volumes, you will need to find out the particulars of your cluster's PV configuration and *use appropriate flags to `pachctl deploy custom`*.

## Common tasks to all types of PV deployments

### Sizing the PV

You'll need to use a PV with enough space for the metadata associated with the data you plan to store in Pachyderm. We're currently developing good rules of thumb for scaling this storage as your Pachyderm deployment grows, but it looks like 10G of disk space is sufficient for most purposes.

### Creating the PV

In the case of cloud-based deployments, the `pachctl deploy` command for AWS, GCP and Azure creates persistent volumes for you, when you follow the instructions for those infrastructures.

In the case of on-premises deployments, the kind of PV you provision will be dependent on what kind of storage your Kubernetes administrators have attached to your cluster and configured, and whether you are expected to consume that storage as a static PV, with Persistent Volume Provisioning or as a StorageClass.



For example, many on-premises deployments use Network File System (NFS) to access to some kind of enterprise storage. Persistent volumes are provisioned in Kubernetes like all things in Kubernetes: by means of a manifest. You can learn about creating [volumes](#) and [persistent volumes](#) in the Kubernetes documentation.

You or your Kubernetes administrators will be responsible for configuring the PVs you create to be consumable as static PV's, with Persistent Volume Provisioning or as a StorageClass.

## What you'll need for Pachyderm configuration of PV storage

Keep the information below at hand for when you run `pachctl deploy custom` further on

### Configuring with static volumes

You'll need the name of the PV and the amount of space you can use, in gigabytes. We'll refer to those, respectively, as `PVC_STORAGE_NAME` and `PVC_STORAGE_SIZE` further on. With this kind of PV, you'll use the flag `--static-etcd-volume` with `PVC_STORAGE_NAME` as its argument in your deployment.

Note: this will override any attempt to configure with StorageClasses, below.

### Configuring with StatefulSets

If you're deploying using [StatefulSets](#), you'll just need the amount of space you can use, in gigabytes, which we'll refer to as `PVC_STORAGE_SIZE` further on..

Note: The `--etcd-storage-class` flag and argument will be ignored if you use the flag `--static-etcd-volume` along with it.

### Configuring with StatefulSets using StorageClasses

If you're deploying using [StatefulSets](#) with [StorageClasses](#), you'll need the name of the storage class and the amount of space you can use, in gigabytes. We'll refer to those, respectively, as `PVC_STORAGECLASS` and `PVC_STORAGE_SIZE` further on. With this kind of PV, you'll use the flag `--etcd-storage-class` with `PVC_STORAGECLASS` as its argument in your deployment.

Note: The `--etcd-storage-class` flag and argument will be ignored if you use the flag `--static-etcd-volume` along with it.

## Deploying an object store

### Object store: what's it for?

An object store is used by Pachyderm's `pachd` for storing all your data. The object store you use must be accessible via a low-latency, high-bandwidth connection like [Gigabit](#) or [10G Ethernet](#).

For an on-premises deployment, it's not advisable to use a cloud-based storage mechanism. Don't deploy an on-premises Pachyderm cluster against cloud-based object stores such as S3 from [AWS](#), GCS from [Google Cloud Platform](#), Azure Blob Storage from [Azure](#).

## Object store prerequisites

Object stores are accessible using the S3 protocol, created by Amazon. Storage providers like [MinIO](#), [EMC's ECS](#), or [SwiftStack](#) provide S3-compatible access to enterprise storage for on-premises deployment. You can find links to instructions for providers of particular object stores in the *See also* section.

## Sizing the object store

Size your object store generously. Once you start using Pachyderm, you'll start versioning all your data. We're currently developing good rules of thumb for scaling your object store as your Pachyderm deployment grows, but it's a good idea to start with a large multiple of your current data set size.

## What you'll need for Pachyderm configuration of the object store

You'll need four items to configure the object store. We're prefixing each item with how we'll refer to it further on.

1. `OS_ENDPOINT` : The access endpoint. For example, MinIO's endpoints are usually something like `minio-server:9000`. Don't begin it with the protocol; it's an endpoint, not an url.
2. `OS_BUCKET_NAME` : The bucket name you're dedicating to Pachyderm. Pachyderm will need exclusive access to this bucket.
3. `OS_ACCESS_KEY_ID` : The access key id for the object store. This is like a user name for logging into the object store.
4. `OS_SECRET_KEY` : The secret key for the object store. This is like the above user's password.

Keep this information handy.

## Next step: creating a custom deploy manifest for Pachyderm

Once you have Kubernetes deployed, your persistent volume created, and your object store configured, it's time to create the Pachyderm manifest for deploying to Kubernetes.

## 4.7.5 See Also

### Kubernetes variants

- OpenShift

### Object storage variants

- EMC ECS
- MinIO
- SwiftStack

## 4.8 Create a Custom Pachyderm Deployment

Pachyderm provides the `pachctl deploy custom` command for creating customized deployments for cloud providers or on-premises use.

This section describes how to use `pachctl deploy custom` to create a manifest for a custom, on-premises deployment. Although deployment automation is out of the scope of this section, Pachyderm strongly encourages you to treat your infrastructure as code.

The topics in this section walk you through the process of using the available flags to create the following components of your Pachyderm infrastructure:

- A Pachyderm deployment using StatefulSets.
- An on-premises Kubernetes cluster with StatefulSets configured. It has the standard `etcd` StorageClass, along with access controls that limit the deployment to namespace-local roles only.
- An on-premises MinIO object store with the following parameters:
  - SSL is enabled.
  - Authentication requests are signed with the S3v4 signatures.
  - The endpoint is `minio:9000`.
  - The access key is `OBSIJRBE0PP2NO4QOA27`.
  - The secret key is `tftSlswRu7BJ86wekitnifILbZam1KYY3TG`.
  - The S3 bucket name is `pachyderm-bucket`.

After configuring these parameters, you save the output of the invocation to a configuration file that you can later use to deploy and configure your environment. For the purposes of our example, all scripts in that hypothetical infrastructure work with YAML manifests.

Complete the steps described in the following topics to deploy your custom environment:

### 4.8.1 Before You Begin

Before you start creating a custom deployment, verify that you have completed the following steps:

1. Read and complete the steps described in the Introduction in the *On-Premises* section. This section explains the differences between static persistent volumes, StatefulSets, and StatefulSets with StorageClasses. Also, it explains the meanings of the variables, such as `PVC_STORAGE_SIZE` and `OS_ENDPOINT` that are used in the examples below.
2. Install `kubectl`.
3. Install `pachctl`.
4. Proceed to Pachyderm Deployment Manifest.

### 4.8.2 Pachyderm Deployment Manifest

This section provides an overview of the Kubernetes manifest that you use to deploy your Pachyderm cluster. This section is provided for your reference and does not include configuration steps. If you are familiar with Kubernetes or do not have immediate questions about the configuration parameters, you can skip this section and proceed to Configuring Persistent Disk Parameters.

When you run the `pachctl deploy` command, Pachyderm generates a JSON-encoded Kubernetes manifest which consists of sections that describe a Pachyderm deployment.

Pachyderm deploys the following sets of application components:

- `pachd` : The main Pachyderm pod.
- `etcd` : The administrative datastore for `pachd`.
- `dash` : The web-based UI for Pachyderm Enterprise Edition.

**Example:**

```
pachctl deploy custom --persistent-disk <persistent disk backend> --object-store
↪<object store backend> \
    <persistent disk arg1> <persistent disk arg 2> \
    <object store arg 1> <object store arg 2> <object store arg 3> <object store_
↪arg 4> \
    [[--dynamic-etcd-nodes n] | [--static-etcd-volume <volume name>]]
    [optional flags]
```

As you can see in the example command above, you can run the `pachctl deploy custom` command with different flags that generate an appropriate manifest for your infrastructure. The flags broadly fall into the following categories:

**Kubernetes Manifest Parameters**

Your Kubernetes manifest includes sections that describe the configuration of your Pachyderm cluster.

The manifest includes the following sections:

**Roles and permissions manifests**

**Application-related manifests**

**Pachyderm pods manifests**

**Pachyderm Kubernetes secrets manifests**

### 4.8.3 Configuring Persistent Disk Parameters

Before reading this section, complete the steps in [Before You Begin](#).

To create a custom deployment, you need to configure persistent storage that Pachyderm will use to store metadata. You can do so by using the `--persistent-disk` flag that creates a PersistentVolume (PV) backend on a supported provider.

Pachyderm has automated configuration for styles of backend for the following major cloud providers:

- Amazon Web Services™ (AWS)
- Google Cloud Platform™ (GCP)
- Microsoft® Azure™

Choosing one of these providers creates a configuration close to what you need. After carefully reading the section below, consult with your Kubernetes administrators on which provider to choose. You might need to then edit your manifest manually, based on configuration information they provide to you.

For each of the providers above, the final configuration depends on which of the following flags you define:

- `--dynamic-etcd-nodes` . The `--dynamic-etcd-nodes` flag is used when your Kubernetes installation is configured to use StatefulSets. Many Kubernetes deployments use StatefulSets as a reliable solution that ensures the persistence of pod storage. Your on-premises Kubernetes installation might also be configured to use StatefulSets. The `--dynamic-etcd-nodes` flag specifies the number of `etcd` nodes that your deployment

creates. Pachyderm recommends that you keep this number at 1. If you want to change it, consult with your Pachyderm support team. This flag creates a `VolumeClaimTemplate` in the `etcd StatefulSet` that uses the standard `etcd-storage-class`.

**NOTE** Consult with your Kubernetes administrator about the `StorageClass` that you should use for `etcd` in your Kubernetes deployment. If you need to use a different than the default setting, you can use the `--etcd-storage-class` flag to specify the `StorageClass`.

- `--static-etcd-volume`. The `--static-etcd-volume` flag is used when your Kubernetes installation has not been configured to use `StatefulSets`. When you specify `--static-etcd-volume` flag, Pachyderm creates a static volume for `etcd`. Pachyderm creates a PV with a spec appropriate for each of the cloud providers:
- `aws`: `awsElasticBlockStore` for Amazon Web Services
- `google`: `gcePersistentDisk` for Google Cloud Storage
- `azure`: `azureDisk` for Microsoft Azure

As stated above, the specifics of one of these choices might not match precisely what your on-premises deployment requires. To determine the closest correct choices for your on-prem infrastructure, consult with your Kubernetes administrators. You might have to then edit your manifest manually, based on configuration information they provide to you.

### Example invocation with persistent disk parameters

This example on-premises cluster has `StatefulSets` enabled, with the standard `etcd` storage class configured. The deployment command uses the following flags:

```
pachctl deploy custom --persistent-disk aws --object-store <object store backend> \
  any-string 10 \
  <object store arg 1> <object store arg 2> <object store arg 3> <object store_
  ↪arg 4> \
  --dynamic-etcd-nodes 1
  [optional flags]
```

The `--persistent-disk` flag takes two arguments that you specify right after the single argument to the `--object-store` flag. Although the first argument is required, Pachyderm ignores it. Therefore, you can set it to any text value, such as `any-string` in the example above. The second argument is the size, in gigabytes (GB), that Pachyderm requests for the `etcd` disk. A good value for most deployments is 10.

After completing the steps described in this section, proceed to [Configuring Object Store](#).

## 4.8.4 Configuring Object Store

Before reading this section, complete the steps in [Configuring Persistent Disk Parameters](#).

You can use the `--object-store` flag to configure Pachyderm to use an s3 storage protocol to access the configured object store. This configuration uses the Amazon S3 driver to access your on-premises object store, regardless of the vendor, since the Amazon S3 API is the standard with which every object store is designed to work.

The S3 API has two different extant versions of *signature styles*, which are how the object store validates client requests. S3v4 is the most current version, but many S3v2 object-store servers are still in use. Because support for S3v2 is scheduled to be deprecated, Pachyderm recommends that you use S3v4 in all deployments.

If you need to access an object store that uses S3v2 signatures, you can specify the `--isS3V2` flag. This parameter configures Pachyderm to use the MinIO driver, which allows the use of the older signature. This `--isS3V2` flag dis-

ables SSL for connections to the object store with the `minio` driver. You can re-enable it with the `-s` or `--secure` flag. You may also manually edit the `pachyderm-storage-secret` Kubernetes manifest.

The `--object-store` flag takes four required configuration arguments. Place these arguments immediately after the persistent disk parameters:

- `bucket-name` : The name of the bucket, without the `s3://` prefix or a trailing forward slash (`/`).
- `access-key` : The user access ID that is used to access the object store.
- `secret-key` : The associated password that is used with the user access ID to access the object store.
- `endpoint` : The hostname and port that are used to access the object store, in `<hostname>:<port>` format.

### Example Invocation with a PV and Object Store

This example on-premises cluster uses an on-premises MinIO object store with the following configuration parameters:

- An on-premises MinIO object store with the following parameters:
  - SSL is enabled
  - S3v4 signatures
  - The endpoint is `minio:9000`
  - The access key is `OBSIJRBE0PP2NO4QOA27`
  - The secret key is `tfteslswRu7BJ86wekitnifILbZam1KYY3TG`
  - A bucket named `pachyderm-bucket`

The deployment command uses the following flags:

```
pachctl deploy custom --persistent-disk aws --object-store s3 \
  any-string 10 \
  pachyderm-bucket 'OBSIJRBE0PP2NO4QOA27' 'tfteslswRu7BJ86wekitnifILbZam1KYY3TG'
↪ 'minio:9000' \
  --dynamic-etcd-nodes 1
  [optional flags]
```

In the example command above, some of the arguments might contain characters that the shell could interpret. Those are enclosed in single-quotes.

**Note:** Because the `deploy custom` command ignores the first configuration argument for the `--persistent-disk` flag, you can specify any string. For more information, see [Configuring Persistent Disk Parameters](#)

After completing the steps described in this section, proceed to [Create a Complete Configuration](#).

## 4.8.5 Create a Complete Configuration

Before reading this section, complete the steps in [Configuring Object Store](#).

The following is a complete deploy command example of a custom deployment. The command generates the manifest and saves it as a YAML configuration file. Also, the command includes the `local-roles` flag to scope the deployment to the `pachyderm` service account access permissions.

Run the following command to deploy your example cluster:

```
pachctl deploy custom --persistent-disk aws --object-store s3 \
  foobar 10 \
  pachyderm-bucket 'OBSIJRBE0PP2NO4QOA27' 'tfteSlsWRu7BJ86wekitnifILbZam1KYY3TG'
→ 'minio:9000' \
  --dynamic-etcd-nodes 1
  --local-roles --output yaml --dry-run > custom_deploy.yaml
```

For more information about the contents of the `custom_deploy.yaml` file, see Pachyderm Deployment Manifest.

## Deploy Your Cluster

You can either deploy manifests that you have created above or edit them to customize them further, before deploying.

If you decide to edit your manifest, you must consult with an experienced Kubernetes administrator. If you are attempting a highly customized deployment, use one of the Pachyderm support resources listed below.

To deploy your configuration, run the following command:

```
$ kubectl apply -f ./custom_deploy.yaml
```

### 4.8.6 Additional flags

This section describes all the additional flags that you can use to configure your custom deployment:

#### Access to Kubernetes resources flags:

- `--local-roles`: You can use the `--local-roles` flag to change the kind of role the pachyderm service account uses from cluster-wide (`ClusterRole`) to namespace-specific (`Role`). Using `--local-roles` inhibits your ability to use the `coefficient parallelism` feature. After you set the `--local-roles` flag, you might see a message similar to this in the `pachd` pod Kubernetes logs:

```
ERROR unable to access kubernetes nodeslist, Pachyderm will continue to work
but it will not be possible to use COEFFICIENT parallelism. error: nodes is
forbidden: User "system:serviceaccount:pachyderm-test-1:pachyderm" cannot
list nodes at the cluster scope
```

#### Resource requests and limits flags:

Larger deployments might require you to configure more resources for `pachd` and `etcd` or set higher limits for transient workloads. The following flags set attributes which are passed on to Kubernetes directly through the produced manifest.

- `--etcd-cpu-request`: The number of CPU cores that Kubernetes allocates to `etcd`. Fractions are allowed.
- `--etcd-memory-request`: The amount of memory that Kubernetes allocates to `etcd`. The SI suffixes are accepted as possible values.
- `--no-guaranteed`: Turn off QoS for `etcd` and `pachd`. Do not use this flag in production environments.
- `--pachd-cpu-request`: The number of CPU cores that Kubernetes allocates to `pachd`. Fractions are allowed.
- `--pachd-memory-request`: The amount of memory that Kubernetes allocates to `pachd`. This flag accepts the SI suffixes.
- `--shards`: The maximum number of `pachd` nodes allowed in the cluster. Increasing this number from the default value of 16 might result in degraded performance.

**Note:** Do not modify the default values of these flags for production deployments without consulting with Pachyderm support.

**Enterprise Edition flags:**

- `--dash-image` : The Docker image for the Pachyderm Enterprise Edition dashboard.
- `--image-pull-secret` : The name of a Kubernetes secret that Pachyderm uses to pull from a private Docker registry.
- `--no-dashboard` : Skip the creation of a manifest for the Enterprise Edition dashboard.
- `--registry` : The registry for Docker images.
- `--tls` : A string in the "`<cert path>, <key path>`" format with the signed TLS certificate that is used for encrypting `pachd` communications.

**Output formats flags:**

- `--dry-run` : Create a manifest and send it to standard output, but do not deploy to Kubernetes.
- `-o` or `--output` : An output format. You can choose from JSON (default) or YAML.

**Logging flags:**

- `log-level` : The `pachd` verbosity level, from most verbose to least. You can set this parameter to `debug`, `info`, or `error`.
- `-v` or `--verbose` : Controls the verbosity of the `pachctl` invocation.

## 4.9 Connect to a Pachyderm cluster

After you deploy a Pachyderm cluster, you can continue to use the command-line interface, connect to the Pachyderm dashboard, or configure third-party applications to access your cluster programmatically. Often all you need to do is just continue to use the command-line interface to create repositories, pipelines, and upload your data. At other times, you might have multiple Pachyderm clusters deployed and need to switch between them to perform management operations.

You do not need to configure anything specific to start using the Pachyderm CLI right after deployment. However, the Pachyderm dashboard and the S3 gateway require explicit port-forwarding or direct access through an externally exposed IP address and port.

This section describes the various options available for you to connect to your Pachyderm cluster.

### 4.9.1 Connect to a Local Cluster

If you are just exploring Pachyderm, use port-forwarding to access both `pachd` and the Pachyderm dashboard.

By default, Pachyderm enables port-forwarding from `pachctl` to `pachd`. If you do not want to use port-forwarding for `pachctl` operations, configure a `pachd_address` as described in *Connect by using a Pachyderm context*.

To connect to the Pachyderm dashboard, you can either use port-forwarding, or the IP address of the virtual machine on which your Kubernetes cluster is running.

The following example shows how to access a Pachyderm cluster that runs in `minikube`.

To connect to a Pachyderm dashboard, complete the following steps:

- To use port-forwarding, run:



```
$ pachctl port-forward
```

- To use the IP address of the VM:‘

1. Get the minikube IP address:

```
$ minikube ip
```

2. Point your browser to the following address:

```
<minikube_ip>:30080
```

## 4.9.2 Connect to a Cluster Deployed on a Cloud Platform

As in a local cluster deployment, a Pachyderm cluster deployed on a cloud platform has implicit port-forwarding enabled. This means that, if you are connected to the cluster so that `kubectl` works, `pachctl` can communicate with `pachd` without any additional configuration. Other services still need explicit port forwarding. For example, to access the Pachyderm dashboard, you need to run `pachctl port-forward`. Since a Pachyderm cluster deployed on a cloud platform is more likely to become a production deployment, configuring a `pachd_address` as described in *Connect by using a Pachyderm context* is the preferred way.

## 4.9.3 Connect by using a Pachyderm context

You can specify an IP address that you use to connect to the Pachyderm UI and the S3 gateway by storing that address in the Pachyderm configuration file as the `pachd_address` parameter. If you have already deployed a Pachyderm cluster, you can set a Pachyderm IP address by updating your cluster configuration file.

This configuration is supported for deployments that do not have a firewall set up between the Pachyderm cluster deployed in the cloud and the client machine. Defining a dedicated `pachd` IP address and host is a more reliable way that might also result in better performance compared to port-forwarding. Therefore, Pachyderm recommends that you use contexts in all production environments.

This configuration requires that you deploy an ingress controller and a reliable security method to protect the traffic between the Pachyderm pods and your client machine. Remember that exposing your traffic through a public ingress might create a security issue. Therefore, if you expose your `pachd` endpoint, you need to make sure that you take steps to protect the endpoint and traffic against common container security threats. Port-forwarding might be an alternative which might result in sufficient performance if you place the data that is consumed by your pipeline in object store buckets located in the same region.

For more information about Pachyderm contexts, see [Manage Cluster Access](#).

To connect by using a Pachyderm context, complete the following steps:

1. Get the current context:

```
$ pachctl config get active-context
```

This command returns the name of the currently active context. Use this as the argument to the command below.

If no IP address is set up for this cluster, you get the following output:

```
$ pachctl config get context <name>
{
}
```

2. Set up `pachd_address` :

```
$ pachctl config update context <name> --pachd-address <host:port>
```

**Example:**

```
$ pachctl config update context local --pachd-address 192.168.1.15:30650
```

**Note:** By default, the `pachd` port is 30650 .

3. Verify that the configuration has been updated:

```
$ pachctl config get context local
{
  "pachd_address": "192.168.1.15:30650"
}
```

## 4.9.4 Connect by Using Port-Forwarding

The Pachyderm port-forwarding is the simplest way to enable cluster access and test basic Pachyderm functionality. Pachyderm automatically starts port-forwarding from `pachctl` to `pachd` . Therefore, the traffic from the local machine goes to the `pachd` endpoint through the Kubernetes API. However, to open a persistent tunnel to other ports, including the Pachyderm dashboard, git and authentication hooks, the built-in HTTP file API, and other, you need to run port-forwarding explicitly.

Also, if you are connecting with port-forward, you are using the `0.0.0.0` . Therefore, if you are using a proxy, it needs to handle that appropriately.

Although port-forwarding is convenient for testing, for production environments, this connection might be too slow and unreliable. The speed of the port-forwarded traffic is limited to 1 MB/s . Therefore, if you experience high latency with `put file` and `get file` operations, or if you anticipate high throughput in your Pachyderm environment, you need to enable ingress access to your Kubernetes cluster. Each cloud provider has its own requirements and procedures for enabling ingress controller and load balancing traffic at the application layer for better performance.

Remember that exposing your traffic through a public ingress might create a security issue. Therefore, if you expose your `pachd` endpoint, you need to make that you take steps to protect the endpoint and traffic against common container security threats. Port-forwarding might be an alternative which might provide satisfactory performance if you place the data that is consumed by your pipeline in object store buckets located in the same region.

To enable port-forwarding, complete the following steps:

1. Open a new terminal window.
2. Run:

```
$ pachctl port-forward
```

This command does not stop unless you manually interrupt it. You can run other `pachctl` commands from another window. If any of your `pachctl` commands hang, verify if the `kubectl port-forwarding` has had issues that prevent `pachctl port-forward` from running properly.

## 4.10 Deploy Pachyderm with TLS

You can deploy your Pachyderm cluster with Transport Layer Security(TLS) enabled to ensure your cluster communications are protected from external attackers, and all the communication parties are verified by means of a trusted certificate and a private key. For many organizations, TLS is a security requirement that ensures integrity of their data.

Before you can enable TLS, you need to obtain a certificate from a trusted CA, such as Let's Encrypt, Cloudflare, or other. You can enable TLS during the deployment of your Pachyderm cluster by providing a path to your CA certificate and your private key by using the `--tls` flag with the `pachctl deploy` command.

```
$ pachctl deploy <platform> --tls "<path/to/cert>,<path/to/key>"
```

**Note:** The paths to the certificate and to the key must be specified exactly as shown in the example above — in double quotes, separated by a comma, and without a space.

After you deploy Pachyderm, to connect through `pachctl` by using a trusted certificate, you need to configure the `pachd_address` in the Pachyderm context with the cluster IP address that starts with `grpcs://`. You can do so by running the following command:

**Example:**

```
echo '{"pachd_address": "grpcs://<cluster-ip>:31400"}' | pachctl config
pachctl config update context `p config get active-context` --pachd_address "grpcs://
↪<cluster-ip>:31400"
```

**See Also:**

- Connect by using a Pachyderm context

## 4.11 Custom Object Stores

In other sections of this guide we have demonstrated how to deploy Pachyderm in a single cloud using that cloud's object store offering. However, Pachyderm can be backed by any object store, and you are not restricted to the object store service provided by the cloud in which you are deploying.

As long as you are running an object store that has an S3 compatible API, you can easily deploy Pachyderm in a way that will allow you to back Pachyderm by that object store. For example, we have seen Pachyderm be backed by [Minio](#), [GlusterFS](#), [Ceph](#), and more.

To deploy Pachyderm with your choice of object store in Google, Azure, or AWS, see the below guides. To deploy Pachyderm on premise with a custom object store, see the [on premise docs](#).

### 4.11.1 Common Prerequisites

1. A working Kubernetes cluster and `kubectl`.
2. An account on or running instance of an object store with an S3 compatible API. You should be able to get an ID, secret, bucket name, and endpoint that point to this object store.

### 4.11.2 Google + Custom Object Store

Additional prerequisites:

- [Google Cloud SDK](#) `>= 124.0.0` - If this is the first time you use the SDK, make sure to follow the [quick start guide](#).

First, we need to create a persistent disk for Pachyderm's metadata:

```
# Name this whatever you want, we chose pach-disk as a default
$ STORAGE_NAME=pach-disk

# For a demo you should only need 10 GB. This stores PFS metadata. For reference, 1GB
```

```
# should work for 1000 commits on 1000 files.
$ STORAGE_SIZE=[the size of the volume that you are going to create, in GBs. e.g. "10
↪"]

# Create the disk.
gcloud compute disks create --size=${STORAGE_SIZE}GB ${STORAGE_NAME}
```

Then we can deploy Pachyderm:

```
pachctl deploy custom --persistent-disk google --object-store s3 ${STORAGE_NAME} $
↪ ${STORAGE_SIZE} <object store bucket> <object store id> <object store secret>
↪ <object store endpoint> --static-etcd-volume=${STORAGE_NAME}
```

### 4.11.3 AWS + Custom Object Store

Additional prerequisites:

- **AWS CLI** - have it installed and have your **AWS credentials** configured.

First, we need to create a persistent disk for Pachyderm's metadata:

```
# We recommend between 1 and 10 GB. This stores PFS metadata. For reference 1GB
# should work for 1000 commits on 1000 files.
$ STORAGE_SIZE=[the size of the EBS volume that you are going to create, in GBs. e.g.
↪ "10"]

$ AWS_REGION=[the AWS region of your Kubernetes cluster. e.g. "us-west-2" (not us-
↪ west-2a)]

$ AWS_AVAILABILITY_ZONE=[the AWS availability zone of your Kubernetes cluster. e.g.
↪ "us-west-2a"]

# Create the volume.
$ aws ec2 create-volume --size ${STORAGE_SIZE} --region ${AWS_REGION} --availability-
↪ zone ${AWS_AVAILABILITY_ZONE} --volume-type gp2

# Store the volume ID.
$ aws ec2 describe-volumes
$ STORAGE_NAME=[volume id]
```

The we can deploy Pachyderm:

```
pachctl deploy custom --persistent-disk aws --object-store s3 ${STORAGE_NAME} $
↪ ${STORAGE_SIZE} <object store bucket> <object store id> <object store secret>
↪ <object store endpoint> --static-etcd-volume=${STORAGE_NAME}
```

### 4.11.4 Azure + Custom Object Store

Additional prerequisites:

- Install **Azure CLI** `>= 2.0.1`
- Install **jq**
- Clone [github.com/pachyderm/pachyderm](https://github.com/pachyderm/pachyderm) and work from the root of that project.

First, we need to create a persistent disk for Pachyderm's metadata. To do this, start by declaring some environmental variables:

```
# Needs to be globally unique across the entire Azure location
$ RESOURCE_GROUP=[The name of the resource group where the Azure resources will be_
↳ organized]

$ LOCATION=[The Azure region of your Kubernetes cluster. e.g. "West US2"]

# Needs to be globally unique across the entire Azure location
$ STORAGE_ACCOUNT=[The name of the storage account where your data will be stored]

# Needs to end in a ".vhd" extension
$ STORAGE_NAME=pach-disk.vhd

# We recommend between 1 and 10 GB. This stores PFS metadata. For reference 1GB
# should work for 1000 commits on 1000 files.
$ STORAGE_SIZE=[the size of the data disk volume that you are going to create, in GBs.
↳ e.g. "10"]
```

And then run:

```
# Create a resource group
$ az group create --name=${RESOURCE_GROUP} --location=${LOCATION}

# Create azure storage account
az storage account create \
  --resource-group="${RESOURCE_GROUP}" \
  --location="${LOCATION}" \
  --sku=Standard_LRS \
  --name="${STORAGE_ACCOUNT}" \
  --kind=Storage

# Build microsoft tool for creating Azure VMs from an image
$ STORAGE_KEY="$(az storage account keys list \
  --account-name="${STORAGE_ACCOUNT}" \
  --resource-group="${RESOURCE_GROUP}" \
  --output=json \
  | jq .[0].value -r
)"
$ make docker-build-microsoft-vhd
$ VOLUME_URI="$(docker run -it microsoft-vhd \
  "${STORAGE_ACCOUNT}" \
  "${STORAGE_KEY}" \
  "${CONTAINER_NAME}" \
  "${STORAGE_NAME}" \
  "${STORAGE_SIZE}G"
)"
```

To check that everything has been setup correctly, try:

```
$ az storage account list | jq '.[0].name'
```

The we can deploy Pachyderm:

```
pachctl deploy custom --persistent-disk azure --object-store s3 ${VOLUME_URI} $
↳ ${STORAGE_SIZE} <object store bucket> <object store id> <object store secret>
↳ <object store endpoint> --static-etcd-volume=${VOLUME_URI}
```

## 4.12 Import a Kubernetes Context

**Note:** The steps in this section apply to your configuration only if you deployed Pachyderm from a manifest created by the `pachctl deploy` command with the `--dry-run` flag. If you did not use the `--dry-run` flag, skip this section.

When you run the `pachctl deploy` command with `--dry-run` flag, instead of immediately deploying a cluster, the command creates a Kubernetes deployment manifest that you can further edit and later use to deploy a Pachyderm cluster.

You can use that manifest with a standard `kubectl apply` command to deploy Pachyderm. For example, if you have created a manifest called `test-manifest.yaml`, you can deploy a Pachyderm cluster by running the following command:

```
kubectl apply -f test-manifest.yaml
```

Typically, when you run `pachctl deploy`, Pachyderm creates a new Pachyderm context with the information from the current [Kubernetes context](#) embedded into it.

When you use the `--dry-run` flag, the Pachyderm context is not created. Therefore, if you deploy a Pachyderm cluster from a manifest that you have created earlier, you need to manually create a new Pachyderm context with the embedded current Kubernetes context and activate that context.

To import a Kubernetes context, complete the following steps:

1. Deploy a Pachyderm cluster from the Kubernetes manifest that you have created when you ran the `pachctl deploy` command with the `--dry-run` flag:

```
$ kubectl apply -f <manifest.yaml>
clusterrole.rbac.authorization.k8s.io/pachyderm configured
clusterrolebinding.rbac.authorization.k8s.io/pachyderm configured
deployment.apps/etcd configured
service/etcd configured
service/pachd configured
deployment.apps/pachd configured
service/dash configured
deployment.apps/dash configured
secret/pachyderm-storage-secret configured
```

2. Verify that the cluster was successfully deployed:

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
dash-64c868cc8b-j79d6              2/2     Running   0           20h
etcd-6865455568-tm5tf              1/1     Running   0           20h
pachd-6464d985c7-dqgzg              1/1     Running   0           70s
```

You must see all the `dash`, `etcd`, and `pachd` pods running.

3. Create a new Pachyderm context with the embedded Kubernetes context:

```
$ pachctl config set context <new-pachyderm-context> -k `kubectl config current-
→context`
```

4. Verify that the context was successfully created and view the context parameters:

**Example:**

```
$ pachctl config get context test-context
{
  "source": "IMPORTED",
  "cluster_name": "minikube",
  "auth_info": "minikube",
  "namespace": "default"
}
```

5. Activate the new Pachyderm context:

```
$ pachctl config set active-context <new-pachyderm-context>
```

6. Verify that the new context has been activated:

```
$ pachctl config get active-context
```

## 4.13 Non-Default Namespaces

Often, production deploys of Pachyderm involve deploying Pachyderm to a non-default namespace. This helps administrators of the cluster more easily manage Pachyderm components alongside other things that might be running inside of Kubernetes (DataDog, TensorFlow Serving, etc.).

To deploy Pachyderm to a non-default namespace, you just need to create that namespace with `kubectl` and then add the `--namespace` flag to your deploy command:

```
$ kubectl create namespace pachyderm
$ kubectl config set-context $(kubectl config current-context) --namespace=pachyderm
$ pachctl deploy <args> --namespace pachyderm
```

After the Pachyderm pods are up and running, you should see something similar to:

```
$ kubectl get pods
NAME                                READY    STATUS    RESTARTS   AGE
dash-68578d4bb4-mmrbj              2/2      Running   0           3m
etcd-69fcfb5fcf-dgc8j              1/1      Running   0           3m
pachd-784bdf7cd7-7dzxr              1/1      Running   0           3m
```

## 4.14 RBAC

Pachyderm has support for Kubernetes Role-Based Access Controls (RBAC) and is a default part of all Pachyderm deployments. For most users, you shouldn't have any issues as Pachyderm takes care of setting all the RBAC permissions automatically. However, if you are deploying Pachyderm on a cluster that your company owns, security policies might not allow certain RBAC permissions by default. Therefore, it's suggested that you contact your Kubernetes admin and provide the following to ensure you don't encounter any permissions issues:

### Pachyderm Permission Requirements

```
Rules: []rbacv1.PolicyRule{{
  APIGroups: []string{"",
  Verbs:      []string{"get", "list", "watch"},
  Resources:  []string{"nodes", "pods", "pods/log", "endpoints"},
}, {
  APIGroups: []string{"",
```

```

Verbs:      []string{"get", "list", "watch", "create", "update", "delete"},
Resources:  []string{"replicationcontrollers", "services"},
}, {
  APIGroups: []string{"",
  Verbs:      []string{"get", "list", "watch", "create", "update", "delete"},
  Resources:  []string{"secrets"},
  ResourceNames: []string{client.StorageSecretName},
}},

```

## 4.14.1 RBAC and DNS

Kubernetes currently (as of 1.8.0) has a bug that prevents kube-dns from working with RBAC. Not having DNS will make Pachyderm effectively unusable. You can tell if you're being affected by the bug like so:

```

$ kubectl get all --namespace=kube-system
NAME                                DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
deploy/kube-dns                     1          1          1              0            3m

NAME                                DESIRED    CURRENT    READY         AGE
rs/kube-dns-86f6f55dd5              1          1          0             3m

NAME                                READY      STATUS      RESTARTS      AGE
po/kube-addon-manager-oryx          1/1        Running     0             3m
po/kube-dns-86f6f55dd5-xksnb        2/3        Running     4             3m
po/kubernetes-dashboard-bzjjh       1/1        Running     0             3m
po/storage-provisioner              1/1        Running     0             3m

NAME                                DESIRED    CURRENT    READY         AGE
rc/kubernetes-dashboard              1          1          1             3m

NAME                                TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
↪AGE
svc/kube-dns                        ClusterIP    10.96.0.10    <none>          53/UDP,53/TCP    3m
svc/kubernetes-dashboard            NodePort     10.97.194.16  <none>          80:30000/TCP     3m

```

Notice how po/kubernetes-dashboard-bzjjh only has 2/3 pods ready and has 4 restarts. To fix this do:

```

kubectl -n kube-system create sa kube-dns
kubectl -n kube-system patch deploy/kube-dns -p '{"spec": {"template": {"spec": {
↪"serviceAccountName": "kube-dns"}}}}'

```

this will tell Kubernetes that kube-dns should use the appropriate ServiceAccount. Kubernetes creates the ServiceAccount, it just doesn't actually use it.

## 4.14.2 RBAC Permissions on GKE

If you're deploying Pachyderm on GKE and run into the following error:

```

Error from server (Forbidden): error when creating "STDIN": clusterroles.rbac.
↪authorization.k8s.io "pachyderm" is forbidden: attempt to grant extra privileges:

```

Run the following and redeploy Pachyderm:

```

kubectl create clusterrolebinding cluster-admin-binding --clusterrole=cluster-admin --
↪user=$(gcloud config get-value account)

```



## 4.15 Troubleshooting Deployments

Here are some common issues by symptom related to certain deploys.

- *General Pachyderm cluster deployment*
- Environment-specific
  - *AWS*
    - \* *Can't connect to the Pachyderm cluster after a rolling update*
    - \* *The one shot deploy script, `aws.sh`, never completes*
    - \* *VPC limit exceeded*
    - \* *GPU node never appears*
  - Google - coming soon...
  - Azure - coming soon...

### 4.15.1 General Pachyderm cluster deployment

- *Pod stuck in `CrashLoopBackoff`*
- *Pod stuck in `CrashLoopBackoff` - with error attaching volume*
- [

#### Pod stuck in `CrashLoopBackoff`

##### Symptoms

The pachd pod keeps crashing/restarting:

```
$ kubectl get all
NAME                                READY    STATUS                RESTARTS   AGE
po/etcd-281005231-qlkzw            1/1     Running               0          7m
po/pachd-1333950811-0sm1p          0/1     CrashLoopBackOff      6          7m
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
svc/etcd	100.70.40.162	<nodes>	2379:30938/TCP	7m
svc/kubernetes	100.64.0.1	<none>	443/TCP	9m
svc/pachd	100.70.227.151	<nodes>	650:30650/TCP,651:30651/TCP	7m

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
deploy/etcd	1	1	1	1	7m
deploy/pachd	1	1	1	0	7m

NAME	DESIRED	CURRENT	READY	AGE
rs/etcd-281005231	1	1	1	7m
rs/pachd-1333950811	1	1	0	7m

## Recourse

First describe the pod:

```
$ kubectl describe po/pachd-1333950811-0smlp
```

If you see an error including `Error attaching EBS volume` or similar, see the recourse for that error here under the corresponding section below. If you don't see that error, but do see something like:

```
1m    3s    9    {kubelet ip-172-20-48-123.us-west-2.compute.internal}
↪ Warning    FailedSync    Error syncing pod, skipping: failed to "StartContainer"
↪ for "pachd" with CrashLoopBackOff: "Back-off 2m40s restarting failed
↪ container=pachd pod=pachd-1333950811-0smlp_default (a92b6665-506a-11e7-8e07-
↪ 02e3d74c49ac) "
```

it means Kubernetes tried running `pachd`, but `pachd` generated an internal error. To see the specifics of this internal error, check the logs for the `pachd` pod:

```
$ kubectl logs po/pachd-1333950811-0smlp
```

**Note:** If you're using a log aggregator service (e.g. the default in GKE), you won't see any logs when using `kubectl logs ...` in this way. You will need to look at your logs UI (e.g. in GKE's case the stackdriver console).

These logs will most likely reveal the issue directly, or at the very least, a good indicator as to what's causing the problem. For example, you might see, `BucketRegionError: incorrect region, the bucket is not in 'us-west-2' region`. In that case, your object store bucket is in a different region than your pachyderm cluster and the fix would be to recreate the bucket in the same region as your pachyderm cluster.

If the error / recourse isn't obvious from the error message, post the error as well as the `pachd` logs in our Slack channel, or open a [GitHub Issue](#) and provide the necessary details prompted by the issue template. Please do be sure provide these logs either way as it is extremely helpful in resolving the issue.

## Pod stuck in `CrashLoopBackoff` - with error attaching volume

### Symptoms

A pod (could be the `pachd` pod or a worker pod) fails to startup, and is stuck in `CrashLoopBackoff`. If you execute `kubectl describe po/pachd-xxxx`, you'll see an error message like the following at the bottom of the output:

```
30s    30s    1    {attachdetach }    Warning
↪ FailedMount    Failed to attach volume "etcd-volume" on node "ip-172-20-44-17.us-
↪ west-2.compute.internal" with: Error attaching EBS volume "vol-0c1d403ac05096dfe"
↪ to instance "i-0a12e00c0f3fb047d": VolumeInUse: vol-0c1d403ac05096dfe is already
↪ attached to an instance
```

This would indicate that the `peristent volume claim` is failing to get attached to the node in your kubernetes cluster.

### Recourse

Your best bet is to manually detach the volume and restart the pod.

For example, to resolve this issue when Pachyderm is deployed to AWS, pull up your AWS web console and look up the node mentioned in the error message (`ip-172-20-44-17.us-west-2.compute.internal` in our case). Then on the

bottom pane for the attached volume. Follow the link to the attached volume, and detach the volume. You may need to “Force Detach” it.

Once it’s detached (and marked as available). Restart the pod by killing it, e.g:

```
$kubectl delete po/pachd-xxx
```

It will take a moment for a new pod to get scheduled.

## 4.15.2 AWS Deployment

### Can’t connect to the Pachyderm cluster after a rolling update

#### Symptom

After running `kops rolling-update`, `kubectl` (and/or `pachctl`) all requests hang and you can’t connect to the cluster.

#### Recourse

First get your cluster name. You can easily locate that information by running `kops get clusters`. If you used the one shot deployment[[http://docs.pachyderm.io/en/latest/deployment/amazon\\_web\\_services.html#one-shot-script](http://docs.pachyderm.io/en/latest/deployment/amazon_web_services.html#one-shot-script)], you can also get this info in the deploy logs you created by `aws.sh`.

Then you’ll need to grab the new public IP address of your master node. The master node will be named something like `master-us-west-2a.masters.somerandomstring.kubernetes.com`

Update the etc hosts entry in `/etc/hosts` such that the api endpoint reflects the new IP, e.g:

```
54.178.87.68 api.somerandomstring.kubernetes.com
```

### One shot script never completes

#### Symptom

The `aws.sh` one shot deploy script hangs on the line:

```
Retrieving ec2 instance list to get k8s master domain name (may take a minute)
```

If it’s been more than 10 minutes, there’s likely an error.

#### Recourse

Check the AWS web console / autoscale group / activity history. You have probably hit an instance limit. To confirm, open the AWS web console for EC2 and check to see if you have any instances with names like:

```
master-us-west-2a.masters.tfgpu.kubernetes.com
nodes.tfgpu.kubernetes.com
```

If you don't see instances similar to the ones above the next thing to do is to navigate to "Auto Scaling Groups" in the left hand menu. Then find the ASG with your cluster name:

```
master-us-west-2a.masters.tfgpu.kubernetes.com
```

Look at the "Activity History" in the lower pane. More than likely, you'll see a "Failed" error message describing why it failed to provision the VM. You're probably run into an instance limit for your account for this region. If you're spinning up a GPU node, make sure that your region supports the instance type you're trying to spin up.

A successful provisioning message looks like:

```
Successful
Launching a new EC2 instance: i-03422f3d32658e90c
2017 June 13 10:19:29 UTC-7
2017 June 13 10:20:33 UTC-7
Description:DescriptionLaunching a new EC2 instance: i-03422f3d32658e90c
Cause:CauseAt 2017-06-13T17:19:15Z a user request created an AutoScalingGroup
↳changing the desired capacity from 0 to 1. At 2017-06-13T17:19:28Z an instance was
↳started in response to a difference between desired and actual capacity, increasing
↳the capacity from 0 to 1.
```

While a failed one looks like:

```
Failed
Launching a new EC2 instance
2017 June 12 13:21:49 UTC-7
2017 June 12 13:21:49 UTC-7
Description:DescriptionLaunching a new EC2 instance. Status Reason: You have
↳requested more instances (1) than your current instance limit of 0 allows for the
↳specified instance type. Please visit http://aws.amazon.com/contact-us/ec2-request
↳to request an adjustment to this limit. Launching EC2 instance failed.
Cause:CauseAt 2017-06-12T20:21:47Z an instance was started in response to a
↳difference between desired and actual capacity, increasing the capacity from 0 to 1.
```

## VPC Limit Exceeded

### Symptom

When running `aws.sh` or otherwise deploying with `kops`, you will see:

```
W0426 17:28:10.435315 26463 executor.go:109] error running task "VPC/5120cf0c-
↳pachydermcluster.kubernetes.com" (3s remaining to succeed): error creating VPC:
↳VpcLimitExceeded: The maximum number of VPCs has been reached.
```

### Recourse

You'll need to increase your VPC limit or delete some existing VPCs that are not in use. On the AWS web console navigate to the VPC service. Make sure you're in the same region where you're attempting to deploy.

It's not uncommon (depending on how you tear down clusters) for the VPCs not to be deleted. You'll see a list of VPCs here with cluster names, e.g. `aee6b566-pachydermcluster.kubernetes.com`. For clusters that you know are no longer in use, you can delete the VPC here.

## GPU Node Never Appears

### Symptom

After running `kops edit ig gpunodes` and `kops update` (as outlined [here](#)) the GPU node never appears, which can be confirmed via the AWS web console.

### Recourse

It's likely you have hit an instance limit for the GPU instance type you're using, or it's possible that AWS doesn't support that instance type in the current region.

Follow [these instructions](#) to check for and update Instance Limits. If this region doesn't support your instance type, you'll see an error message like:

```
Failed
Launching a new EC2 instance
2017 June 12 13:21:49 UTC-7
2017 June 12 13:21:49 UTC-7
Description:DescriptionLaunching a new EC2 instance. Status Reason: You have
↳requested more instances (1) than your current instance limit of 0 allows for the
↳specified instance type. Please visit http://aws.amazon.com/contact-us/ec2-request
↳to request an adjustment to this limit. Launching EC2 instance failed.
Cause:CauseAt 2017-06-12T20:21:47Z an instance was started in response to a
↳difference between desired and actual capacity, increasing the capacity from 0 to 1.
```



---

## Manage Pachyderm

---

This document describes basic management operations that you need to perform in Pachyderm.

### 5.1 Manage Cluster Access

Pachyderm contexts enable you to store configuration parameters for multiple Pachyderm clusters in a single configuration file saved at `~/.pachyderm/config.json`. This file stores the information about all Pachyderm clusters that you have deployed from your machine locally or on a remote server.

For example, if you have a cluster that is deployed locally in `minikube` and another one deployed on Amazon EKS, configurations for these clusters are stored in that `config.json` file. By default, all local cluster configurations have the `local` prefix. If you have multiple local clusters, Pachyderm adds a consecutive number to the `local` prefix of each cluster.

The following text is an example of a Pachyderm `config.json` file:

```
{
  "user_id": "b4fe4317-be21-4836-824f-6661c68b8fba",
  "v2": {
    "active_context": "local-1",
    "contexts": {
      "default": {},
      "local": {},
      "local-1": {},
    },
    "metrics": true
  }
}
```

#### 5.1.1 View the Active Context

When you have multiple Pachyderm clusters, you can switch between them by setting the current context. The active context is the cluster that you interact with when you run `pachctl` commands.

To view active context, type:

- View the active context:

```
$ pachctl config get active-context
local-1
```

- List all contexts and view the current context:

```
$ pachctl config list context
ACTIVE  NAME
        default
        local
        *  local-1
```

The active context is marked with an asterisk.

## 5.1.2 Change the Active Context

To change the active context, type `pachctl config set active-context <name>`.

Also, you can set the `PACH_CONTEXT` environmental variable that overrides the active context.

**Example:**

```
export PACH_CONTEXT=local1
```

## 5.1.3 Create a New Context

When you deploy a new Pachyderm cluster, a new context that points to the new cluster is created automatically.

In addition, you can create a new context by providing your parameters through the standard input stream (`stdin`) in your terminal. Specify the parameters as a comma-separated list enclosed in curly brackets.

To create a new context with specific parameters, complete the following steps:

1. Create a new Pachyderm context with a specific `pachd` IP address and a client certificate:

```
$ echo '{"pachd_address":"10.10.10.130:650", "server_cas":"key.pem"}' | pachctl
↪config set context new-local
Reading from stdin
```

**Note:** By default, the `pachd` port is 30650.

2. Verify your configuration by running the following command:

```
$ pachctl config get context new-local
{
  "pachd_address": "10.10.10.130:650",
  "server_cas": "key.pem"
}
```

## 5.1.4 Update an Existing Context

You can update an existing context with new parameters, such as a Pachyderm IP address, certificate authority (CA), and others. For the list of parameters, see [Pachyderm Config Specification](#).

To update the Active Context, run the following commands:

1. Update the context with a new `pachd` address:

```
$ pachctl config update context local-1 --pachd-address 10.10.10.131
```

**Note:** The `pachctl config update` command supports the `--pachd-address` flag only.



2. Verify that the context has been updated:

```
$ pachctl config get context local-1
{
  "pachd_address": "10.10.10.131"
}
```

3. Alternatively, you can update multiple properties by using an `echo` script:

**Example:**

```
$ echo '{"pachd_address":"10.10.10.132", "server_cas":"key.pem"}' | pachctl
↪config set context local-1 --overwrite
Reading from stdin.
```

4. Verify that the changes were applied:

```
$ pachctl config get context local-1
{
  "pachd_address": "10.10.10.132",
  "server_cas": "key.pem"
}
```

## 5.2 Autoscaling a Pachyderm Cluster

There are 2 levels of autoscaling in Pachyderm:

- Pachyderm can scale down workers when they're not in use.
- Cloud providers can scale workers down/up based on resource utilization (most often CPU).

### 5.2.1 Pachyderm Autoscaling of Workers

Refer to the `scaleDownThreshold` field in the pipeline specification. This allows you to specify a time window after which idle workers are removed. If new inputs come in on the pipeline corresponding to those deleted workers, they get scaled back up.

### 5.2.2 Cloud Provider Autoscaling

Out of the box, autoscaling at the cloud provider layer doesn't work well with Pachyderm. However, if configured properly, cloud provider autoscaling can complement Pachyderm autoscaling of workers.

#### Default Behavior with Cloud Autoscaling

Normally when you create a pipeline, Pachyderm asks the k8s cluster how many nodes are available. Pachyderm then uses that number as the default value for the pipeline's parallelism. (To read more about parallelism, [refer to the distributed processing docs](#)).

If you have cloud provider autoscaling activated, it is possible that your number of nodes will be scaled down to a few or maybe even a single node. A pipeline created on this cluster would have a default parallelism will be set to this low value (e.g., 1 or 2). Then, once the autoscale group notices that more nodes are needed, the parallelism of the pipeline won't increase, and you won't actually make effective use of those new nodes.

## Configuration of Pipelines to Complement Cloud Autoscaling

The goal of Cloud autoscaling is to:

- To schedule nodes only as the processing demand necessitates it.

The goals of Pachyderm worker autoscaling are:

- To make sure your job uses a maximum amount of parallelism.
- To ensure that you process the job efficiently.

Thus, to accomplish both of these goals, we recommend:

- Setting a `constant`, high level of parallelism. Specifically, setting the constant parallelism to the number of workers you will need when your pipeline is active.
- Setting the `cpu` and/or `mem` resource requirements in the `resource_requests` field on your pipeline.

To determine the right values for `cpu` / `mem`, first set these values rather high. Then use the monitoring tools that come with your cloud provider (or [try out our monitoring deployment](#)) so you can see the actual CPU/mem utilization per pod.

### Example Scenario

Let's say you have a certain pipeline with a constant parallelism set to 16. Let's also assume that you've set `cpu` to `1.0` and your instance type has 4 cores.

When a commit of data is made to the input of the pipeline, your cluster might be in a scaled down state (e.g., 2 nodes running). After accounting for the pachyderm services (`pachd` and `etcd`), ~6 cores are available with 2 nodes. K8s then schedules 6 of your workers. That accounts for all 8 of the CPUs across the nodes in your instance group. Your autoscale group then notices that all instances are being heavily utilized, and subsequently scales up to 5 nodes total. Now the rest of your workers get spun up (k8s can now schedule them), and your job proceeds.

This type of setup is best suited for long running jobs, or jobs that take a lot of CPU time. Such jobs give the cloud autoscaling mechanisms time to scale up, while still having data that needs to be processed when the new nodes are up and running.

## 5.3 Data Management Best Practices

This document discusses best practices for minimizing the space needed to store your Pachyderm data, increasing the performance of your data processing as related to data organization, and general good ideas when you are using Pachyderm to version/process your data.

- *Shuffling files*
- *Garbage collection*
- *Setting a root volume size*

### 5.3.1 Shuffling files

Certain pipelines simply shuffle files around (e.g., organizing files into buckets). If you find yourself writing a pipeline that does a lot of copying, such as [Time Windowing](#), it probably falls into this category.

The best way to shuffle files, especially large files, is to create **symlinks** in the output directory that point to files in the input directory.

For instance, to move a file `log.txt` to `logs/log.txt`, you might be tempted to write a `transform` like this:

```
cp /pfs/input/log.txt /pfs/out/logs/log.txt
```

However, it's more efficient to create a symlink:

```
ln -s /pfs/input/log.txt /pfs/out/logs/log.txt
```

Under the hood, Pachyderm is smart enough to recognize that the output file simply symlinks to a file that already exists in Pachyderm, and therefore skips the upload altogether.

Note that if your shuffling pipeline only needs the names of the input files but not their content, you can use `empty_files: true`. That way, your shuffling pipeline can skip both the download and the upload. An example for this type of shuffle pipeline is [here](#)

### 5.3.2 Garbage collection

When a file/commit/repo is deleted, the data is not immediately removed from the underlying storage system (e.g. S3) for performance and architectural reasons. This is similar to how when you delete a file on your computer, the file is not necessarily wiped from disk immediately.

To actually remove the data, you may need to manually invoke garbage collection. The easiest way to do it is through `pachctl garbage-collect`. Currently `pachctl garbage-collect` can only be started when there are no active jobs running. You also need to ensure that there's no ongoing `put file`. Garbage collection puts the cluster into a readonly mode where no new jobs can be created and no data can be added.

### 5.3.3 Setting a root volume size

When planning and configuring your Pachyderm deploy, you need to make sure that each node's root volume is big enough to accommodate your total processing bandwidth. Specifically, you should calculate the bandwidth for your expected running jobs as follows:

```
(storage needed per datum) x (number of datums being processed simultaneously) / ↵
↵ (number of nodes)
```

Here, the storage needed per datum should be the storage needed for the largest "datum" you expect to process anywhere on your DAG plus the size of the output files that will be written for that datum. If your root volume size is not large enough, pipelines might fail when downloading the input. The pod would get evicted and rescheduled to a different node, where the same thing will happen (assuming that node had a similar volume). This scenario is further discussed [here](#).

## 5.4 Sharing GPU Resources

Often times, teams are running big ML models on instances with GPU resources.

GPU instances are expensive! You want to make sure that you're utilizing the GPUs you're paying for!

### 5.4.1 Without configuration

To deploy a pipeline that relies on GPU, you'll already have set the `gpu` resource requirement in the pipeline specification. But Pachyderm workers by default are long lived ... the worker is spun up and waits for new input. That works great for pipelines that are processing a lot of new incoming commits.

For ML workflows, especially during the development cycle, you probably will see lower volume of input commits. Which means that you could have your pipeline workers ‘taking’ the GPU resource as far as k8s is concerned, but ‘idling’ as far as you’re concerned.

Let’s use an example.

Let’s say your cluster has a single GPU node with 2 GPUs. Let’s say you have a pipeline running that requires 1 GPU. You’ve trained some models, and found the results were surprising. You suspect your feature extraction code, and are delving into debugging that stage of your pipeline. Meanwhile, the worker you’ve spun up for your GPU training job is sitting idle, but telling k8s it’s using the GPU instance.

Now your coworker is actively trying to develop their GPU model with their pipeline. Their model requires 2 GPUs. But your pipeline is still marked as using 1 GPU, so their pipeline can’t run!

## 5.4.2 Configuring your pipelines to share GPUs

Whenever you have a limited amount of a resource on your cluster (in this case GPU), you want to make sure you’ve specified how much of that resource you need via the `resource_requests` as [part of your pipeline specification](#). But, you also need to make sure you set the `standby` field to `true` so that if your pipeline is not getting used, the worker pods get spun down and you free the GPU resource.

## 5.5 Backup and Restore

### 5.5.1 Contents

- *Introduction*
- *Backup & restore concepts*
- *General backup procedure*
  - *1. Pause all pipeline and data loading/unloading operations*
  - *2. Extract a pachyderm backup*
  - *3. Restart all pipeline and data loading operations*
- *General restore procedure*
  - *Restore your backup to a pachyderm cluster, same version*
- *Notes and Design Considerations*
  - *Loading data from other sources into pachyderm*
  - *Note about releases prior to Pachyderm 1.7*

### 5.5.2 Introduction

Since release 1.7, Pachyderm provides the commands `pachctl extract` and `pachctl restore` to backup and restore the state of a Pachyderm cluster. (Please see [the note below about releases prior to Pachyderm 1.7.](#))

The `pachctl extract` command requires that all pipeline and data loading activity into Pachyderm stop before the extract occurs. This enables Pachyderm to create a consistent, point-in-time backup. In this document, we’ll talk about how to create such a backup and restore it to another Pachyderm instance.

Extract and restore commands are currently used to migrate between minor and major releases of Pachyderm, so it's important to understand how to perform them properly. In addition, there are a few design points and operational techniques that data engineers should take into consideration when creating complex pachyderm deployments to minimize disruptions to production pipelines.

In this document, we'll take you through the steps to backup and restore a cluster, migrate an existing cluster to a newer minor or major release, and elaborate on some of those design and operations considerations.

### 5.5.3 Backup & restore concepts

Backing up Pachyderm involves the persistent volume (PV) that `etcd` uses for administrative data and the object store bucket that holds Pachyderm's actual data. Restoring involves populating that PV and object store with data to recreate a Pachyderm cluster.

### 5.5.4 General backup procedure

#### 1. Pause all pipeline and data loading/unloading operations

##### Pausing pipelines

From the directed acyclic graphs (DAG) that define your pachyderm cluster, stop each pipeline. You can either run a multiline shell command, shown below, or you must, for each pipeline, manually run the `pachctl stop pipeline` command.

```
pachctl stop pipeline <pipeline-name>
```

You can confirm each pipeline is paused using the `pachctl list pipeline` command

```
pachctl list pipeline
```

Alternatively, a useful shell script for running `stop pipeline` on all pipelines is included below. It may be necessary to install the utilities used in the script, like `jq` and `xargs`, on your system.

```
pachctl list pipeline --raw \
| jq -r '.pipeline.name' \
| xargs -P3 -n1 -I{} pachctl stop pipeline {}
```

It's also a useful practice, for simple to moderately complex deployments, to keep a terminal window up showing the state of all running kubernetes pods.

```
watch -n 5 kubectl get pods
```

You may need to install the `watch` and `kubectl` commands on your system, and configure `kubectl` to point at the cluster that Pachyderm is running in.

##### Pausing data loading operations

**Input repositories** or **input repos** in pachyderm are repositories created with the `pachctl create repo` command. They're designed to be the repos at the top of a directed acyclic graph of pipelines. Pipelines have their own output repos associated with them, and are not considered input repos. If there are any processes external to pachyderm that put data into input repos using any method (the Pachyderm APIs, `pachctl put file`, etc.), they need to be paused. See [Loading data from other sources into pachyderm](#) below for design considerations for those processes that will minimize downtime during a restore or migration.

Alternatively, you can use the following commands to stop all data loading into Pachyderm from outside processes.

```
# Once you have stopped all running pachyderm pipelines, such as with this command,
# $ pachctl list pipeline --raw \
#   | jq -r '.pipeline.name' \
#   | xargs -P3 -n1 -I{} pachctl stop pipeline {}

# all pipelines in your cluster should be suspended. To stop all
# data loading processes, we're going to modify the pachd Kubernetes service so that
# it only accepts traffic on port 30649 (instead of the usual 30650). This way,
# any background users and services that send requests to your Pachyderm cluster
# while 'extract' is running will not interfere with the process
#
# Backup the Pachyderm service spec, in case you need to restore it quickly
$ kubectl get svc/pachd -o json >pach_service_backup_30650.json

# Modify the service to accept traffic on port 30649
# Note that you'll likely also need to modify your cloud provider's firewall
# rules to allow traffic on this port
$ kubectl get svc/pachd -o json | sed 's/30650/30649/g' | kubectl apply -f -

# Modify your environment so that *you* can talk to pachd on this new port
$ pachctl config update context `pachctl config get active-context` --pachd-address=
→<cluster ip>:30649

# Make sure you can talk to pachd (if not, firewall rules are a common culprit)
$ pachctl version
COMPONENT      VERSION
pachctl        1.9.5
pachd          1.9.5
```

## 2. Extract a pachyderm backup

You can use `pachctl extract` alone or in combination with cloning/snapshotting services.

### Using `pachctl extract`

Using the `pachctl extract` command, create the backup you need.

```
pachctl extract > path/to/your/backup/file
```

You can also use the `-u` or `--url` flag to put the backup directly into an object store.

```
pachctl extract --url s3://...
```

If you are planning on backing up the object store using its own built-in clone operation, be sure to add the `--no-objects` flag to the `pachctl extract` command.

### Using your cloud provider's clone and snapshot services

You should follow your cloud provider's recommendation for backing up these resources. Here are some pointers to the relevant documentation.

## Snapshotting persistent volumes

For example, here are official guides on creating snapshots of persistent volumes on Google Cloud Platform, Amazon Web Services (AWS) and Microsoft Azure, respectively:

- [Creating snapshots of GCE persistent volumes](#)
- [Creating snapshots of Elastic Block Store \(EBS\) volumes](#)
- [Creating snapshots of Azure Virtual Hard Disk volumes](#)

For on-premises Kubernetes deployments, check the vendor documentation for your PV implementation on backing up and restoring.

## Cloning object stores

Below, we give an example using the Amazon Web Services CLI to clone one bucket to another, [taken from the documentation for that command](#). Similar commands are available for [Google Cloud](#) and [Azure blob storage](#).

```
aws s3 sync s3://mybucket s3://mybucket2
```

For on-premises Kubernetes deployments, check the vendor documentation for your on-premises object store for details on backing up and restoring a bucket.

## Combining cloning, snapshots and extract/restore

You can use `pachctl extract` command with the `--no-objects` flag to exclude the object store, and use an object store snapshot or clone command to back up the object store. You can run the two commands at the same time. For example, on Amazon Web Services, the following commands can be run simultaneously.

```
aws s3 sync s3://mybucket s3://mybucket2
pachctl extract --no-objects --url s3://anotherbucket
```

## Use case: minimizing downtime during a migration

The above cloning/snapshotting technique is recommended when doing a migration where minimizing downtime is desirable, as it allows the duplicated object store to be the basis of the upgraded, new cluster instead of requiring Pachyderm to extract the data from object store.

## 3. Restart all pipeline and data loading operations

Once the backup is complete, restart all paused pipelines and data loading operations.

From the directed acyclic graphs (DAG) that define your pachyderm cluster, start each pipeline. You can either run a multiline shell command, shown below, or you must, for each pipeline, manually run the `pachctl start pipeline` command.

```
pachctl start pipeline <pipeline-name>
```

You can confirm each pipeline is started using the `list pipeline` command

```
pachctl list pipeline
```

A useful shell script for running `start pipeline` on all pipelines is included below. It may be necessary to install the utilities used in the script, like `jq` and `xargs`, on your system.

```
pachctl list pipeline --raw \
| jq -r '.pipeline.name' \
| xargs -P3 -n1 -I{} pachctl start pipeline {}
```

If you used the port-changing technique, [above](#), to stop all data loading into Pachyderm from outside processes, you should change the ports back.

```
# Once you have restarted all running pachyderm pipelines, such as with this command,
# $ pachctl list pipeline --raw \
# | jq -r '.pipeline.name' \
# | xargs -P3 -n1 -I{} pachctl start pipeline {}

# all pipelines in your cluster should be restarted. To restart all data loading
# processes, we're going to change the pachd Kubernetes service so that
# it only accepts traffic on port 30650 again (from 30649).
#
# Backup the Pachyderm service spec, in case you need to restore it quickly
$ kubectl get svc/pachd -o json >pach_service_backup_30649.json

# Modify the service to accept traffic on port 30650, again
$ kubectl get svc/pachd -o json | sed 's/30649/30650/g' | kubectl apply -f -

# Modify your environment so that *you* can talk to pachd on the old port
$ pachctl config update context `pachctl config get active-context` --pachd-address=
↪<cluster ip>:30650

# Make sure you can talk to pachd (if not, firewall rules are a common culprit)
$ pachctl version
COMPONENT      VERSION
pachctl        1.9.5
pachd          1.9.5
```

## 5.5.5 General restore procedure

### Restore your backup to a pachyderm cluster, same version

Spin up a Pachyderm cluster and run `pachctl restore` with the backup you created earlier.

```
pachctl restore < path/to/your/backup/file
```

You can also use the `-u` or `--url` flag to get the backup directly from the object store you placed it in

```
pachctl restore --url s3://...
```

## 5.5.6 Notes and design considerations

### Loading data from other sources into Pachyderm

When writing systems that place data into Pachyderm input repos (see [above](#) for a definition of ‘input repo’), it is important to provide ways of ‘pausing’ output while queueing any data output requests to be output when the systems are ‘resumed’. This allows all Pachyderm processing to be stopped while the extract takes place.

In addition, it is desirable for systems that load data into Pachyderm have a mechanism for replaying a queue from any checkpoint in time. This is useful when doing migrations from one release to another, where you would like to minimize downtime of a production Pachyderm system. After an extract, the old system is kept running with the



checkpoint established while a duplicate, upgraded pachyderm cluster is migrated with duplicated data. Transactions that occur while the migrated, upgraded cluster is being brought up are not lost, and can be replayed into this new cluster to reestablish state and minimize downtime.

### Note about releases prior to Pachyderm 1.7

Pachyderm 1.7 is the first version to support `extract` and `restore`. To bridge the gap to previous Pachyderm versions, we've made a final 1.6 release, 1.6.10, which backports the `extract` and `restore` functionality to the 1.6 series of releases.

Pachyderm 1.6.10 requires no migration from other 1.6.x versions. You can simply `pachctl undeploy` and then `pachctl deploy` after upgrading `pachctl` to version 1.6.10. After 1.6.10 is deployed you should make a backup using `pachctl extract` and then upgrade `pachctl` again, to 1.7.0. Finally you can `pachctl deploy ...` with `pachctl 1.7.0` to trigger the migration.

## 5.6 Upgrades and Migrations

As new versions of Pachyderm are released, you may need to update your cluster to get access to bug fixes and new features. These updates fall into two categories, which are covered in detail at the links below:

- Upgrades - An upgrade is moving between point releases within the same major release (e.g. 1.7.2 → 1.7.3). Upgrades are typically a simple process that require little to no downtime.
- Migrations – A migration what you must perform to move between major releases such as 1.8.7 → 1.9.0.

*Important:* Performing an *upgrade* when going between *major releases* may lead to corrupted data. *You must perform a migration when going between major releases!*

Whether you're doing an upgrade or migration, it is recommended you backup Pachyderm prior. That will guarantee you can restore your cluster to its previous, good state.

## 5.7 General Troubleshooting

Here are some common issues by symptom along with steps to resolve them.

- *Connecting to a Pachyderm cluster*
  - *Cannot connect via `pachctl` - context deadline exceeded*
  - *Certificate error when using `kubectl`*
  - *Uploads/downloads are slow*

---

### 5.7.1 Connecting to a Pachyderm Cluster

#### Cannot connect via `pachctl` - context deadline exceeded

##### Symptom

You may be using the `pachd` address config value or environment variable to specify how `pachctl` talks to your Pachyderm cluster, or you may be forwarding the pachyderm port. In any event, you might see something similar to:

```
$ pachctl version
COMPONENT      VERSION
pachctl        1.9.5
context deadline exceeded
```

Also, you might get this message if `pachd` is not running.

## Recourse

It's possible that the connection is just taking a while. Occasionally this can happen if your cluster is far away (deployed in a region across the country). Check your internet connection.

It's also possible that you haven't poked a hole in the firewall to access the node on this port. Usually to do that you adjust a security rule (in AWS parlance a security group). For example, on AWS, if you find your node in the web console and click on it, you should see a link to the associated security group. Inspect that group. There should be a way to "add a rule" to the group. You'll want to enable TCP access (ingress) on port 30650. You'll usually be asked which incoming IPs should be whitelisted. You can choose to use your own, or enable it for everyone (0.0.0.0/0).

## Certificate Error When Using Kubectl

### Symptom

This can happen on any request using `kubectl` (e.g. `kubectl get all`). In particular you'll see:

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1.6.4", GitCommit:
↪ "d6f433224538d4f9ca2f7ae19b252e6fcb66a3ae", GitTreeState:"clean", BuildDate:"2017-
↪ 05-19T20:41:24Z", GoVersion:"go1.8.1", Compiler:"gc", Platform:"darwin/amd64"}
Unable to connect to the server: x509: certificate signed by unknown authority
```

## Recourse

Check if you're on any sort of VPN or other egress proxy that would break SSL. Also, there is a possibility that your credentials have expired. In the case where you're using GKE and gcloud, renew your credentials via:

```
$ kubectl get all
Unable to connect to the server: x509: certificate signed by unknown authority
$ gcloud container clusters get-credentials my-cluster-name-dev
Fetching cluster endpoint and auth data.
kubeconfig entry generated for my-cluster-name-dev.
$ kubectl config current-context
gke_my-org_us-east1-b_my-cluster-name-dev
```

## Uploads/Downloads are Slow

### Symptom

Any `pachctl put file` or `pachctl get file` commands are slow.

## Recourse

If you do not explicitly set the pachd address config value, `pachctl` will default to using port forwarding, which throttles traffic to ~1MB/s. If you need to do large downloads/uploads you should consider using pachd address config value. You'll also want to make sure you've allowed ingress access through any firewalls to your k8s cluster.

## 5.8 Troubleshooting Pipelines

### 5.8.1 Introduction

Job failures can occur for a variety of reasons, but they generally categorize into 3 failure types:

1. *User-code-related*: An error in the user code running inside the container or the json pipeline config.
2. *Data-related*: A problem with the input data such as incorrect file type or file name.
3. *System- or infrastructure-related*: An error in Pachyderm or Kubernetes such as missing credentials, transient network errors, or resource constraints (for example, out-of-memory–OOM–killed).

In this document, we'll show you the tools for determining what kind of failure it is. For each of the failure modes, we'll describe Pachyderm's and Kubernetes's specific retry and error-reporting behaviors as well as typical user triaging methodologies.

Failed jobs in a pipeline will propagate information to downstream pipelines with empty commits to preserve provenance and make tracing the failed job easier. A failed job is no longer running.

In this document, we'll describe what you'll see, how Pachyderm will respond, and techniques for triaging each of those three categories of failure.

At the bottom of the document, we'll provide specific troubleshooting steps for *specific scenarios*.

- *Pipeline exists but never runs*
- All your pods / jobs get evicted

### Determining the kind of failure

First off, you can see the status of Pachyderm's jobs with `pachctl list job`, which will show you the status of all jobs. For a failed job, use `pachctl inspect job <job-id>` to find out more about the failure. The different categories of failures are addressed below.

### User Code Failures

When there's an error in user code, the typical error message you'll see is

```
failed to process datum <UUID> with error: <user code error>
```

This means pachyderm successfully got to the point where it was running user code, but that code exited with a non-zero error code. If any datum in a pipeline fails, the entire job will be marked as failed, but datums that did not fail will not need to be reprocessed on future jobs. You can use `pachctl inspect datum <job-id> <datum-id>` or `pachctl logs` with the `--pipeline`, `--job` or `--datum` flags to get more details.

There are some cases where users may want mark a datum as successful even for a non-zero error code by setting the `transform.accept_return_code` field in the pipeline config.

### Retries

Pachyderm will automatically retry user code three (3) times before marking the datum as failed. This mitigates datums failing for transient connection reasons.

### Triage

`pachctl logs --job=<job_ID>` or `pachctl logs --pipeline=<pipeline_name>` will print out any logs from your user code to help you triage the issue. Kubernetes will rotate logs occasionally so if nothing is being returned, you'll need to make sure that you have a persistent log collection tool running in your cluster. If you set `enable_stats:true` in your pachyderm pipeline, pachyderm will persist the user logs for you.

In cases where user code is failing, changes first need to be made to the code and followed by updating the pachyderm pipeline. This involves building a new docker container with the corrected code, modifying the pachyderm pipeline config to use the new image, and then calling `pachctl update pipeline -f updated_pipeline_config.json`. Depending on the issue/error, user may or may not want to also include the `--reprocess` flag with `update pipeline`.

### Data Failures

When there's an error in the data, this will typically manifest in a user code error such as

```
failed to process datum <UUID> with error: <user code error>
```

This means pachyderm successfully got to the point where it was running user code, but that code exited with a non-zero error code, usually due to being unable to find a file or a path, a misformatted file, or incorrect fields/data within a file. If any datum in a pipeline fails, the entire job will be marked as failed. Datums that did not fail will not need to be reprocessed on future jobs.

### Retries

Just like with user code failures, Pachyderm will automatically retry running a datum 3 times before marking the datum as failed. This mitigates datums failing for transient connection reasons.

### Triage

Data failures can be triaged in a few different way depending on the nature of the failure and design of the pipeline.

In some cases, where malformed datums are expected to happen occasionally, they can be “swallowed” (e.g. marked as successful using `transform.accept_return_codes` or written out to a “failed\_datums” directory and handled within user code). This would simply require the necessary updates to the user code and pipeline config as described above. For cases where your code detects bad input data, a “dead letter queue” design pattern may be needed. Many pachyderm developers use a special directory in each output repo for “bad data” and pipelines with globs for detecting bad data direct that data for automated and manual intervention.

Pachyderm's engineering team is working on changes to the Pachyderm Pipeline System in a future release that may make implementation of design patterns like this easier. [Take a look at the pipeline design changes for pachyderm 1.9](#)

If a few files as part of the input commit are causing the failure, they can simply be removed from the HEAD commit with `start commit, delete file, finish commit`. The files can also be corrected in this manner as well. This method is similar to a revert in Git – the “bad” data will still live in the older commits in Pachyderm, but will not be part of the HEAD commit and therefore not processed by the pipeline.

If the entire commit is bad and you just want to remove it forever as if it never happened, `delete commit` will both remove that commit and all downstream commits and jobs that were created as downstream effects of that input data.

## System-level Failures

System-level failures are the most varied and often hardest to debug. We'll outline a few common patterns and triage steps. Generally, you'll need to look at deeper logs to find these errors using `pachctl logs --pipeline=<pipeline_name> --raw` and/or `--master` and `kubectl logs pod <pod_name>`.

Here are some of the most common system-level failures:

- Malformed or missing credentials such that a pipeline cannot connect to object storage, registry, or other external service. In the best case, you'll see `permission denied` errors, but in some cases you'll only see "does not exist" errors (this is common reading from object stores)
- Out-of-memory (OOM) killed or other resource constraint issues such as not being able to schedule pods on available cluster resources.
- Network issues trying to connect Pachd, etcd, or other internal or external resources
- Failure to find or pull a docker image from the registry

## Retries

For system-level failures, Pachyderm or Kubernetes will generally continually retry the operation with exponential backoff. If a job is stuck in a given state (e.g. starting, merging) or a pod is in `CrashLoopBackoff`, those are common signs of a system-level failure mode.

## Triage

Triaging system failures varies as widely as the issues do themselves. Here are options for the common issues mentioned previously.

- Credentials: check your secrets in k8s, make sure they're added correctly to the pipeline config, and double check your roles/perms within the cluster
- OOM: Increase the memory limit/request or node size for your pipeline. If you are very resource constrained, making your datums smaller to require less resources may be necessary.
- Network: Check to make sure etcd and pachd are up and running, that k8s DNS is correctly configured for pods to resolve each other and outside resources, firewalls and other networking configurations allow k8s components to reach each other, and ingress controllers are configured correctly
- Check your container image name in the pipeline config and `image_pull_secret`.

## 5.8.2 Specific scenarios

### All your pods / jobs get evicted

#### Symptom

Running:

```
$ kubectl get all
```

shows a bunch of pods that are marked `Evicted`. If you `kubectl describe ...` one of those evicted pods, you see an error saying that it was evicted due to disk pressure.

### Recourse

Your nodes are not configured with a big enough root volume size. You need to make sure that each node's root volume is big enough to store the biggest datum you expect to process anywhere on your DAG plus the size of the output files that will be written for that datum.

Let's say you have a repo with 100 folders. You have a single pipeline with this repo as an input, and the glob pattern is `/*`. That means each folder will be processed as a single datum. If the biggest folder is 50GB and your pipeline's output is about 3 times as big, then your root volume size needs to be bigger than:

```
50 GB (to accommodate the input) + 50 GB x 3 (to accommodate the output) = 200GB
```

In this case we would recommend 250GB to be safe. If your root volume size is less than 50GB (many defaults are 20GB), this pipeline will fail when downloading the input. The pod may get evicted and rescheduled to a different node, where the same thing will happen.

### Pipeline exists but never runs

#### Symptom

You can see the pipeline via:

```
$ pachctl list pipeline
```

But if you look at the job via:

```
$ pachctl list job
```

It's marked as running with `0/0` datums having been processed. If you inspect the job via:

```
$ pachctl inspect job
```

You don't see any worker set. E.g:

Worker Status:			
WORKER	JOB	DATUM	STARTED
...			

If you do `kubectl get pod` you see the worker pod for your pipeline, e.g:

```
po/pipeline-foo-5-v1-273zc
```

But it's state is `Pending` or `CrashLoopBackoff`.

### Recourse

First make sure that there is no parent job still running. Do `pachctl list job | grep yourPipelineName` to see if there are pending jobs on this pipeline that were kicked off prior to your

job. A parent job is the job that corresponds to the parent output commit of this pipeline. A job will block until all parent jobs complete.

If there are no parent jobs that are still running, then continue debugging:

Describe the pod via:

```
$kubect1 describe po/pipeline-foo-5-v1-273zc
```

If the state is `CrashLoopBackoff`, you're looking for a descriptive error message. One such cause for this behavior might be if you specified an image for your pipeline that does not exist.

If the state is `Pending` it's likely the cluster doesn't have enough resources. In this case, you'll see a `could not schedule` type of error message which should describe which resource you're low on. This is more likely to happen if you've set resource requests (`cpu/mem/gpu`) for your pipelines. In this case, you'll just need to scale up your resources. If you deployed using `kops`, you'll want to do edit the instance group, e.g. `kops edit ig nodes . . .` and up the number of nodes. If you didn't use `kops` to deploy, you can use your cloud provider's auto scaling groups to increase the size of your instance group. Either way, it can take up to 10 minutes for the changes to go into effect.

You can read more about autoscaling [here](#)





This section includes how-tos that describe best practices of data operations in Pachyderm, including the following topics:

## 6.1 Individual Developer Workflow

A typical Pachyderm workflow involves multiple iterations of experimenting with your code and pipeline specs. Before you read this section, make sure that you understand basic Pachyderm pipeline concepts described in Concepts.

### 6.1.1 How it works

Working with Pachyderm includes multiple iterations of the following steps:

#### 6.1.2 Step 1: Write Your Analysis Code

Because Pachyderm is completely language-agnostic, the code that is used to process data in Pachyderm can be written in any language and can use any libraries of choice. Whether your code is as simple as a bash command or as complicated as a TensorFlow neural network, it needs to be built with all the required dependencies into a container that can run anywhere, including inside of Pachyderm. See [Examples](#).

Your code does not have to import any special Pachyderm functionality or libraries. However, it must meet the following requirements:

- **Read files from a local file system.** Pachyderm automatically mounts each input data repository as `/pfs/<repo_name>` in the running containers of your Docker image. Therefore, the code that you write needs to read input data from this directory, similar to any other file system.

Because Pachyderm automatically spreads data across parallel containers, your analysis code does not have to deal with data sharding or parallelization. For example, if you have four containers that run your Python code, Pachyderm automatically supplies 1/4 of the input data to `/pfs/<repo_name>` in each running container. These workload balancing settings can be adjusted as needed through Pachyderm tunable parameters in the pipeline specification.

- **Write files into a local file system**, such as saving results. Your code must write to the `/pfs/out` directory that Pachyderm mounts in all of your running containers. Similar to reading data, your code does not have to manage parallelization or sharding.

### 6.1.3 Step 2: Build Your Docker Image

When you create a Pachyderm pipeline, you need to specify a Docker image that includes the code or binary that you want to run. Therefore, every time you modify your code, you need to build a new Docker image, push it to your image registry, and update the image tag in the pipeline spec. This section describes one way of building Docker images, but if you have your own routine, feel free to apply it.

To build an image, you need to create a `Dockerfile`. However, do not use the `CMD` field in your `Dockerfile` to specify the commands that you want to run. Instead, you add them in the `cmd` field in your pipeline specification. Pachyderm runs these commands inside the container during the job execution rather than relying on Docker to run them. The reason is that Pachyderm cannot execute your code immediately when your container starts, so it runs a shim process in your container instead, and then, it calls your pipeline specification's `cmd` from there.

After building your image, you need to upload the image into a public or private image registry, such as [DockerHub](#) or other.

Alternatively, you can use the Pachyderm's built-in functionality to tag, build, and push images by running the `pachctl update pipeline` command with the `--build` and `--push-images` flags. For more information, see [updating pipelines](#).

To build a Docker image, complete the following steps:

1. If you do not have a registry, create one with a preferred provider. If you decide to use DockerHub, follow the [Docker Hub Quickstart](#) to create a repository for your project.
2. Create a `Dockerfile` for your project. See the [OpenCV example](#).

**Note:** The above `Dockerfile` example is provided for your reference only. Your `Dockerfile` might look completely different.

3. Log in to an image registry.

- If you use DockerHub, run:

```
docker login --username=<dockerhub-username> --password=<dockerhub-password>  
↪<dockerhub-fqdn>
```

4. Build a new image from the `Dockerfile` by specifying a tag:

```
docker build -t <IMAGE>:<TAG> .
```

5. Push your image to your image registry.

- If you use DockerHub, run:

```
docker push <image>:tag
```

For more information about building Docker images, see [Docker documentation](#).

### 6.1.4 Step 3: Create a Pipeline

Pachyderm's pipeline specifications store the configuration information about the Docker image and code that Pachyderm should run. Pipeline specifications are stored in JSON format. As soon as you create a pipeline, Pachyderm immediately spins a pod or pods on a Kubernetes worker node in which pipeline code runs. By default, after the pipeline finishes running, the pods continue to run while waiting for the new data to be committed into the Pachyderm input repository. You can configure this parameter, as well as many others, in the pipeline specification.

A minimum pipeline specification must include the following parameters:

- `name`

- transform
- parallelism
- input

You can store your pipeline locally or in a remote location, such as a GitHub repository.

To create a Pipeline, complete the following steps:

1. Create a pipeline specification. Here is an example of a pipeline spec:

```
# my-pipeline.json
{
  "pipeline": {
    "name": "my-pipeline"
  },
  "transform": {
    "image": "my-pipeline-image",
    "cmd": ["/binary", "/pfs/data", "/pfs/out"]
  },
  "input": {
    "pfs": {
      "repo": "data",
      "glob": "/*"
    }
  }
}
```

2. Create a Pachyderm pipeline from the spec:

```
$ pachctl create pipeline -f my-pipeline.json
```

You can specify a local file or a file stored in a remote location, such as a GitHub repository. For example, <https://raw.githubusercontent.com/pachyderm/pachyderm/master/examples/opencv/edges.json>.

**See also:**

- Pipeline Specification

## 6.2 Team Developer Workflow

This section describes an example of how you can incorporate Pachyderm into your existing enterprise infrastructure. If you are just starting to use Pachyderm and not setting up automation for your Pachyderm build processes, see Individual Developer Workflow.

Pachyderm is a powerful system for providing data provenance and scalable processing to data scientists and engineers. You can make it even more powerful by integrating it with your existing continuous integration and continuous deployment (CI/CD) workflows and systems. This section walks you through the CI/CD processes that you can create to enable Pachyderm collaboration within your data science and engineering groups.

As you write code, you test it in containers and notebooks against sample data that you store in Pachyderm repos or mount it locally. Your code runs in development pipelines in Pachyderm. Pachyderm provides capabilities that assist with day-to-day development practices, including the `--build` and `--push-images` flags to the `pachctl update pipeline` command that enable you to build and push images to a Docker registry.

Although initial CI setup might require extra effort on your side, in the long run, it brings significant benefits to your team, including the following:

- Simplified workflow for data scientists. Data scientists do not need to be aware of the complexity of the underlying containerized infrastructure. They can follow an established Git process, and the CI platform takes care of the Docker build and push process behind the scenes.
- Your CI platform can run additional unit tests against the submitted code before creating the build.
- Flexibility in tagging Docker images, such as specifying a custom name and tag or using the commit SHA for tagging.

The following diagram demonstrates automated Pachyderm development workflow:

The automated developer workflow includes the following steps:

1. A new commit triggers a Git hook.

Typically, Pachyderm users store the following artifacts in a Git repository:

- A `Dockerfile` that you use to build local images.
- A `pipeline.json` specification file that you can use in a `Makefile` to create local builds, as well as in the CI/CD workflows.
- The code that performs data transformations.

A [commit hook in Git](#) for your repository triggers the CI/CD process. It uses the information in your pipeline specification for subsequent steps.

2. Build an image.

Your CI process automatically starts the build of a Docker container image based on your code and the `Dockerfile`.

3. Push the image tagged with commit ID to an image registry.

Your CI process pushes a Docker image created in Step 2 to your preferred image registry. When a data scientist submits their code to Git, a CI process uses the `Dockerfile` in the repository to build, tag with a Git commit SHA, and push the container to your image registry.

4. Update the pipeline spec with the tagged image.

In this step, your CI/CD infrastructure uses your updated `pipeline.json` specification and fills in the Git commit SHA for the version of the image that must be used in this pipeline. Then, it runs the `pachctl update pipeline` command to push the updated pipeline specification to Pachyderm. After that, Pachyderm pulls a new image from the registry automatically. When the production pipeline is updated with the `pipeline.json` file that has the correct image tag in it, Pachyderm restarts all pods for this pipeline with the new image automatically.

## 6.3 Load Your Data Into Pachyderm

Before you read this section, make sure that you are familiar with the Data Concepts and Pipeline Concepts.

The data that you commit to Pachyderm is stored in an object store of your choice, such as Amazon S3, MinIO, Google Cloud Storage, or other. Pachyderm records the cryptographic hash (SHA) of each portion of your data and stores it as a commit with a unique identifier (ID). Although the data is stored as an unstructured blob, Pachyderm enables you to interact with versioned data as you typically do in a standard file system.

Pachyderm stores versioned data in repositories which can contain one or multiple files, as well as files arranged in directories. Regardless of the repository structure, Pachyderm versions the state of each data repository as the data changes over time.

To put data into Pachyderm, a commit must be *started*, or *opened*. Data can then be put into Pachyderm as part of that open commit and is available once the commit is *finished* or *closed*.

Pachyderm provides the following options to load data:

- By using the `pachctl put file` command. This option is great for testing, development, integration with CI/CD, and for users who prefer scripting. See `Use put file`.
- By creating a pipeline to pull data from an outside source. Because Pachyderm pipelines can be any arbitrary code that runs in a Docker container, you can call out to external APIs or data sources and pull in data from there. Your pipeline code can be triggered on-demand or continuously with the following special types of pipelines:
  - **Spout:** A spout enables you to continuously load streaming data from a streaming data source, such as a messaging system or message queue into Pachyderm. See `Spout`.
  - **Cron:** A cron triggers your pipeline periodically based on the interval that you configure in your pipeline spec. See `Cron`.

**Note:** Pipelines enable you to do much more than just ingressing data into Pachyderm. Pipelines can run all kinds of data transformations on your input data sources, such as a Pachyderm repository, and be configured to run your code automatically as new data is committed. For more information, see `Pipeline`.

- By using a Pachyderm language client. This option is ideal for Go or Python users who want to push data into Pachyderm from services or applications written in those languages. If you did not find your favorite language in the list of supported language clients, Pachyderm uses a protobuf API which supports many other languages. See `Pachyderm Language Clients`.

If you are using the Pachyderm Enterprise version, you can use these additional options:

- By using the S3 gateway. This option is great to use with the existing tools and libraries that interact with S3-compatible object stores. See `Using the S3 Gateway`.
- By using the Pachyderm dashboard. The Pachyderm Enterprise dashboard provides a convenient way to upload data right from the UI.

**Note:** In the Pachyderm UI, you can only specify an S3 data source. Uploading data from your local device is not supported.

### 6.3.1 Load Your Data by Using `pachctl`

The `pachctl put file` command enables you to do everything from loading local files into Pachyderm to pulling data from an existing object store bucket and extracting data from a website. With `pachctl put file`, you can append new data to the existing data or overwrite the existing data. All these options can be configured by using the flags available with this command. Run `pachctl put file --help` to view the complete list of flags that you can specify.

To load your data into Pachyderm by using `pachctl`, you first need to create one or more data repositories. Then, you can use the `pachctl put file` command to put your data into the created repository.

In Pachyderm, you can *start* and *finish* commits. If you just run `pachctl put file` and no open commit exists, Pachyderm starts a new commit, adds the data to which you specified the path in your command, and finishes the commit. This is called an atomic commit.

Alternatively, you can run `pachctl start commit` to start a new commit. Then, add your data in multiple `put file` calls, and finally, when ready, close the commit by running `pachctl finish commit`.

To load your data into a repository, complete the following steps:

1. Create a Pachyderm repository:

```
$ pachctl create repo <repo name>
```

## 2. Select from the following options:

- To start and finish an atomic commit, run:

```
$ pachctl put file <repo>@<branch>:</path/to/file1> -f <file1>
```

- To start a commit and add data in iterations:

### (a) Start a commit:

```
$ pachctl start commit <repo>@<branch>
```

### (b) Put your data:

```
$ pachctl put file <repo>@<branch>:</path/to/file1> -f <file1>
```

### (c) Work on your changes, and when ready, put more data:

```
$ pachctl put file <repo>@<branch>:</path/to/file2> -f <file2>
```

### (d) Close the commit:

```
$ pachctl finish commit <repo>@<branch>
```

## 6.3.2 Filepath Format

In Pachyderm, you specify the path to file by using the `-f` option. A path to file can be a local path or a URL to an external resource. You can add multiple files or directories by using the `-i` option. To add contents of a directory, use the `-r` flag.

The following table provides examples of `pachctl put file` commands with various filepaths and data sources:

- Put data from a URL:

```
$ pachctl put file <repo>@<branch>:</path/to/file> -f http://url_path
```

- Put data from an object store. You can use `s3://`, `gcs://`, or `as://` in your filepath:

```
$ pachctl put file <repo>@<branch>:</path/to/file> -f s3://object_store_url
```

**Note:** If you are configuring a local cluster to access an S3 bucket, you need to first deploy a Kubernetes `Secret` for the selected object store.

- Add multiple files at once by using the `-i` option or multiple `-f` flags. In the case of `-i`, the target file must be a list of files, paths, or URLs that you want to input all at once:

```
$ pachctl put file <repo>@<branch> -i <file containing list of files, paths, or  
→URLs>
```

- Input data from stdin into a data repository by using a pipe:

```
$ echo "data" | pachctl put file <repo>@<branch> -f </path/to/file>
```

- Add an entire directory or all of the contents at a particular URL, either HTTP(S) or object store URL, `s3://`, `gcs://`, and `as://`, by using the recursive flag, `-r`:

```
$ pachctl put file <repo>@<branch> -r -f <dir>
```

### 6.3.3 Loading Your Data Partially

Depending on your use case, you might decide not to import all of your data into Pachyderm but only store and apply version control to some of it. For example, if you have a 10 PB dataset, loading the whole dataset into Pachyderm is a costly operation that takes a lot of time and resources. To optimize performance and the use of resources, you might decide to load some of this data into Pachyderm, leaving the rest of it in its original source.

One possible way of doing this is by adding a metadata file with a URL to the specific file or directory in your dataset to a Pachyderm repository and refer to that file in your pipeline. Your pipeline code would read the URL or path in the external data source and retrieve that data as needed for processing instead of needing to preload it all into a Pachyderm repo. This method works particularly well for mostly immutable data because in this case, Pachyderm will not keep versions of the source file, but it will keep track and provenance of the resulting output commits in its version-control system.

## 6.4 Export Your Data From Pachyderm

After you build a pipeline, you probably want to see the results that the pipeline has produced. Every commit into an input repository results in a corresponding commit into an output repository.

To access the results of a pipeline, you can use one of the following methods:

- By running the `pachctl get file` command. This command returns the contents of the specified file. To get the list of files in a repo, you should first run the `pachctl list file` command. See [Export Your Data with pachctl](#).
- By configuring the pipeline. A pipeline can push or expose output data to external sources. You can configure the following data exporting methods in a Pachyderm pipeline:
  - An `egress` property enables you to export your data to an external datastore, such as Amazon S3, Google Cloud Storage, and others. See [Export data by using egress](#).
  - A service. A Pachyderm service exposes the results of the pipeline processing on a specific port in the form of a dashboard or similar endpoint. See [Service](#).
  - Configure your code to connect to an external data source. Because a pipeline is a Docker container that runs your code, you can egress your data to any data source, even to those that the `egress` field does not support, by connecting to that source from within your code.
- By using the S3 gateway. Pachyderm Enterprise users can reuse their existing tools and libraries that work with object store to export their data with the S3 gateway. See [Using the S3 Gateway](#).

### 6.4.1 Export Your Data with pachctl

The `pachctl get file` command enables you to get the contents of a file in a Pachyderm repository. You need to know the file path to specify it in the command.

To export your data with pachctl:

1. Get the list of files in the repository:

```
$ pachctl list file <repo>@<branch>
```

**Example:**

```
$ pachctl list commit data@master
REPO  BRANCH COMMIT                                PARENT
↳STARTED          DURATION          SIZE
data master 230103d3c6bd45b483ab6d0b7ae858d5 f82b76f463ca4799817717a49ab74fac 2
↳seconds ago Less than a second 750B
data master f82b76f463ca4799817717a49ab74fac <none> 40
↳seconds ago Less than a second 375B
```

**2. Get the contents of a specific file:**

```
pachctl get file <repo>@<branch>:<path/to/file>
```

**Example:**

```
$ pachctl get file data@master:user_data.csv
1,cyukhtin0@stumbleupon.com,144.155.176.12
2,csisneros1@over-blog.com,26.119.26.5
3,jeye2@instagram.com,13.165.230.106
4,rnollet3@hexun.com,58.52.147.83
5,bposkitt4@irs.gov,51.247.120.167
6,vvenmore5@hubpages.com,161.189.245.212
7,lcoyte6@ask.com,56.13.147.134
8,atuke7@psu.edu,78.178.247.163
9,nmorrell18@howstuffworks.com,28.172.10.170
10,afynn9@google.com.au,166.14.112.65
```

Also, you can view the parent, grandparent, and any previous revision by using the caret (^) symbol with a number that corresponds to an ancestor in sequence:

- To view a parent of a commit:

- (a) List files in the parent commit:

```
$ pachctl list commit <repo>@<branch-or-commit>^:<path/to/file>
```

- (b) Get the contents of a file:

```
$ pachctl get file <repo>@<branch-or-commit>^:<path/to/file>
```

- To view an <n> parent of a commit:

- (a) List files in the parent commit:

```
$ pachctl list commit <repo>@<branch-or-commit>^<n>:<path/to/file>
```

**Example:**

```
NAME          TYPE SIZE
/user_data.csv file 375B
```

- (b) Get the contents of a file:

```
$ pachctl get file <repo>@<branch-or-commit>^<n>:<path/to/file>
```

**Example:**



```
$ pachctl get file datas@master^4:user_data.csv
```

You can specify any number in the `^<n>` notation. If the file exists in that commit, Pachyderm returns it. If the file does not exist in that revision, Pachyderm displays the following message:

```
$ pachctl get file <repo>@<branch-or-commit>^<n>:<path/to/file>
file "<path/to/file>" not found
```

## 6.4.2 Export Your Data with `egress`

The `egress` field in the Pachyderm pipeline specification enables you to push the results of a pipeline to an external datastore such as Amazon S3, Google Cloud Storage, or Azure Blob Storage. After the user code has finished running, but before the job is marked as successful, Pachyderm pushes the data to the specified destination.

You can specify the following `egress` protocols for the corresponding storage:

**Note:** Use the horizontal scroll bar in the table below to view full descriptions and syntax.

**Example:**

```
"egress": {
  "URL": "s3://bucket/dir"
},
```

## 6.5 Split Data

Pachyderm provides functionality that enables you to split your data while it is being loaded into a Pachyderm input repository which helps to optimize pipeline processing time and resource utilization.

Splitting data helps you to address the following use cases:

- **Optimize data processing.** If you have large size files, it might take Pachyderm a significant amount of time to process them. In addition, Pachyderm considers such a file as a single datum, and every time you apply even a minor change to that datum, Pachyderm processes the whole file from scratch.
- **Increase diff granularity.** Pachyderm does not create per-line diffs that display line-by-line changes. Instead, Pachyderm provides per-file diffing. Therefore, if all of your data is in one huge file, you might not be able to see what has changed in that file. Breaking up the file to smaller chunks addresses this issue.

This section provides examples of how you can use the `--split` command that breaks up your data into smaller chunks, called *split-files* and what happens when you update your data.

This section includes the following topics:

### 6.5.1 Splitting Data for Distributed Processing

Before you read this section, make sure that you understand the concepts described in [Distributed Computing](#).

Pachyderm enables you to parallelize computations over data as long as that data can be split up into multiple *datums*. However, in many cases, you might have a dataset that you want or need to commit into Pachyderm as a single file rather than a bunch of smaller files that are easily mapped to datums, such as one file per record. For such cases, Pachyderm provides an easy way to prepare your dataset for subsequent distributed computing by splitting it upon uploading to a Pachyderm repository.

In this example, you have a dataset that consists of information about your users and a repository called `user`. This data is in CSV format in a single file called `user_data.csv` with one record per line:

```
$ head user_data.csv
1,cyukhtin0@stumbleupon.com,144.155.176.12
2,csisneros1@over-blog.com,26.119.26.5
3,jeye2@instagram.com,13.165.230.106
4,rnollet3@hexun.com,58.52.147.83
5,bposkitt4@irs.gov,51.247.120.167
6,vvenmore5@hubpages.com,161.189.245.212
7,lcoyte6@ask.com,56.13.147.134
8,atuke7@psu.edu,78.178.247.163
9,nmorrell18@howstuffworks.com,28.172.10.170
10,afynn9@google.com.au,166.14.112.65
```

If you put this data into Pachyderm as a single file, Pachyderm processes them a single datum. It cannot process each of these user records in parallel as separate datums. Potentially, you can manually separate these user records into standalone files before you commit them into the `users` repository or through a pipeline stage dedicated to this splitting task. However, Pachyderm provides an optimized way of completing this task.

The `put file` API includes an option for splitting the file into separate datums automatically. You can use the `--split` flag with the `put file` command.

To complete this example, follow the steps below:

1. Create a `users` repository by running:

```
$ pachctl create repo users
```

2. Create a file called `user_data.csv` with the contents listed above.
3. Put your `user_data.csv` file into Pachyderm and automatically split it into separate datums for each line:

```
$ pachctl put file users@master -f user_data.csv --split line --target-file-
→datums 1
```

The `--split line` argument specifies that Pachyderm splits this file into lines, and the `--target-file-datums 1` argument specifies that each resulting file must include at most one datum or one line.

4. View the list of files in the master branch of the `users` repository:

```
$ pachctl list file users@master
NAME                                TYPE      SIZE
user_data.csv    dir              5.346 KiB
```

If you run `pachctl list file` command for the master branch in the `users` repository, Pachyderm still shows the `user_data.csv` entity to you as one entity in the repo. However, this entity is now a directory that contains all of the split records.

5. To view the detailed information about the `user_data.csv` file, run the command with the file name specified after a colon:

```
$ pachctl list file users@master:user_data.csv
NAME                                TYPE      SIZE
user_data.csv/000000000000000000  file      43 B
user_data.csv/000000000000000001  file      39 B
user_data.csv/000000000000000002  file      37 B
user_data.csv/000000000000000003  file      34 B
user_data.csv/000000000000000004  file      35 B
```

```
user_data.csv/000000000000000005  file      41 B
user_data.csv/000000000000000006  file      32 B
etc...
```

Then, a pipeline that takes the repo `users` as input with a glob pattern of `/user_data.csv/*` processes each user record, such as each line in the CSV file in parallel.

## JSON and Text File Splitting Examples

Pachyderm supports this type of splitting for lines or JSON blobs as well. See the examples below.

- Split a `json` file on `json` blobs by putting each `json` blob into a separate file.

```
$ pachctl put file users@master -f user_data.json --split json --target-file-
→datums 1
```

- Split a `json` file on `json` blobs by putting three `json` blobs into each split file.

```
$ pachctl put file users@master -f user_data.json --split json --target-file-
→datums 3
```

- Split a file on lines by putting each 100-bytes chunk into the split files.

```
$ pachctl put file users@master -f user_data.txt --split line --target-file-bytes
→100
```

## Specifying a Header

If your data has a common header, you can specify it manually by using `pachctl put file` with the `--header-records` flag. You can use this functionality with JSON and CSV data.

To specify a header, complete the following steps:

1. Create a new or use an existing data file. For example, the `user_data.csv` from the section above with the following header:

```
NUMBER, EMAIL, IP_ADDRESS
```

2. Create a new repository or use an existing one:

```
$ pachctl create repo users
```

3. Put your file into the repository by separating the header from other lines:

```
$ pachctl put file users@master -f user_data.csv --split=csv --header-records=1 --
→target-file-datums=1
```

4. Verify that the file was added and split:

```
$ pachctl list file users@master:/user_data.csv
```

**Example:**

NAME	TYPE	SIZE
/user_data.csv/000000000000000000	file	70B
/user_data.csv/000000000000000001	file	66B
/user_data.csv/000000000000000002	file	64B
/user_data.csv/000000000000000003	file	61B
/user_data.csv/000000000000000004	file	62B
/user_data.csv/000000000000000005	file	68B
/user_data.csv/000000000000000006	file	59B
/user_data.csv/000000000000000007	file	59B
/user_data.csv/000000000000000008	file	71B
/user_data.csv/000000000000000009	file	65B

5. Get the first file from the repository:

```
$ pachctl get file users@master:/user_data.csv/000000000000000000
NUMBER, EMAIL, IP_ADDRESS
1, cyukhtin0@stumbleupon.com, 144.155.176.12
```

6. Get all files:

```
$ pachctl get file users@master:/user_data.csv/*
NUMBER, EMAIL, IP_ADDRESS
1, cyukhtin0@stumbleupon.com, 144.155.176.12
2, csisneros1@over-blog.com, 26.119.26.5
3, jeye2@instagram.com, 13.165.230.106
4, rnollet3@hexun.com, 58.52.147.83
5, bposkitt4@irs.gov, 51.247.120.167
6, vvenmore5@hubpages.com, 161.189.245.212
7, lcoyte6@ask.com, 56.13.147.134
8, atuke7@psu.edu, 78.178.247.163
9, nmorrell18@howstuffworks.com, 28.172.10.170
10, afynn9@google.com.au, 166.14.112.65
```

For more information, type `pachctl put file --help`.

## Ingesting PostgreSQL data

Pachyderm supports direct data ingestion from PostgreSQL. You need first extract your database into a script file by using `pg_dump` and then add the data from the file into Pachyderm by running the `pachctl put file` with the `--split` flag.

When you use `pachctl put file --split sql ...`, Pachyderm splits your `pgdump` file into three parts - the header, rows, and the footer. The header contains all the SQL statements in the `pgdump` file that set up the schema and tables. The rows are split into individual files, or if you specify the `--target-file-datums` or `--target-file-bytes`, multiple rows per file. The footer contains the remaining SQL statements for setting up the tables.

The header and footer are stored in the directory that contains the rows. If you request a `get file` on that directory, you get just the header and footer. If you request an individual file, you see the header, the row or rows, and the footer. If you request all the files with a glob pattern, for example, `/directoryname/*`, you receive the header, all the rows, and the footer recreating the full `pgdump`. Therefore, you can construct full or partial `pgdump` files so that you can load full or partial datasets.

To put your PostgreSQL data into Pachyderm, complete the following steps:

1. Generate a `pgdump` file:

**Example:**

```
$ pg_dump -t users -f users.pgdump
```

## 2. View the pgdump file

### Example:

```
$ cat users.pgdump
--
-- PostgreSQL database dump
--

-- Dumped from database version 9.5.12
-- Dumped by pg_dump version 9.5.12

SET statement_timeout = 0;
SET lock_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET client_min_messages = warning;
SET row_security = off;

SET default_tablespace = '';

SET default_with_oids = false;

--
-- Name: users; Type: TABLE; Schema: public; Owner: postgres
--

CREATE TABLE public.users (
    id integer NOT NULL,
    name text NOT NULL,
    saying text NOT NULL
);

ALTER TABLE public.users OWNER TO postgres;

--
-- Data for Name: users; Type: TABLE DATA; Schema: public; Owner: postgres
--

COPY public.users (id, name, saying) FROM stdin;
0    wile E Coyote    ...
1    road runner  \\.
\\.

--
-- PostgreSQL database dump complete
--
```

## 3. Ingest the SQL data by using the pachctl put file command with the --split file:

```
$ pachctl put file data@master -f users.pgdump --split sql
$ pachctl put file data@master:users --split sql -f users.pgdump
```

## 4. View the information about your repository:

```
$ pachctl list file data@master
NAME          TYPE SIZE
users         dir  914B
```

The `users.pgdump` file is added to the master branch in the `data` repository.

5. View the information about the `users.pgdump` file:

```
$ pachctl list file data@master:users
NAME                                     TYPE SIZE
/users/00000000000000000000           file 20B
/users/0000000000000000000001         file 18B
```

6. In your pipeline, where you have started and forked PostgreSQL, you can load the data by running the following or a similar script:

```
$ cat /pfs/data/users/* | sudo -u postgres psql
```

By using the glob pattern `/*`, this code loads each raw PostgreSQL chunk into your PostgreSQL instance for processing by your pipeline.

**Tip:** For this use case, you might want to use `--target-file-datums` or `--target-file-bytes` because these commands enable your queries to run against many rows at a time.

## 6.5.2 Adjusting Data Processing by Splitting Data

Before you read this section, make sure that you understand the concepts described in [File](#), [Glob Pattern](#), [Pipeline Specification](#), and [Working with Pipelines](#).

Unlike source code version-control systems, such as Git, that mostly store and version text files, Pachyderm does not perform intra-file diffing. This means that if you have a 10,000 lines CSV file and change a single line in that file, a pipeline that is subscribed to the repository where that file is located processes the whole file. You can adjust this behavior by splitting your file upon loading into chunks.

Pachyderm applies diffing at per file level. Therefore, if one bit of a file changes, Pachyderm identifies that change as a new file. Similarly, Pachyderm can only distribute computation at the level of a single file. If your data is stored in one large file, it can only be processed by a single worker, which might affect performance.

To optimize performance and processing time, you might want to break up large files into smaller chunks while Pachyderm uploads data to an input repository. For simple data types, you can run the `pachctl put file` command with the `--split` flag. For more complex splitting pattern, such as when you work with `.avro` or other binary formats, you need to manually split your data either at ingest or by configuring splitting in a Pachyderm pipeline.

### Using Split and Target-File Flags

For common file types that are often used in data science, such as CSV, line-delimited text files, JavaScript Object Notation (JSON) files, Pachyderm includes the `--split`, `--target-file-bytes`, and `--target-file-datums` flags.

**Note:** In this section, a chunk of data is called a *split-file*.

If you specify both `--target-file-datums` and `--target-file-bytes` flags, Pachyderm creates split-files until it hits one of the constraints.

**See also:**

- Splitting Data for Distributed Processing <../cookbook/splitting.html#pg-dump-sql-supply>

### Example: Splitting a File

In this example, you have a 50-line file called `my-data.txt`. You create a repository named `line-data` and load `my-data.txt` into that repository. Then, you can analyze how the data is split in the repository.

To complete this example, perform the following steps:

1. Create a file with fifty lines named `my-data.txt`. You can add random lines, such as numbers from one to fifty, or US states, or anything else.

#### Examples:

```
Zero
One
Two
Three
...
Fifty
```

2. Create a repository called `line-data`:

```
$ pachctl create repo line-data
$ pachctl put file line-data@master -f my-data.txt --split line
```

3. List the filesystem objects in the repository:

```
$ pachctl list file line-data@master
NAME          TYPE  SIZE
/my-data.txt  dir   1.071KiB
```

**Important:** The `pachctl list file` command shows that the line-oriented file `my-data.txt` that was uploaded has been transformed into a directory that includes the chunks of the original `my-data.txt` file. Each chunk is put into a split-file and given a 16-character filename, left-padded with 0. Pachyderm numbers each filename sequentially in hexadecimal. We modify the command to list the contents of “`my-data.txt`”, and the output reveals the naming structure.

4. List the files in the `my-data.txt` directory:

```
$ pachctl list file line-data@master my-data.txt
NAME                                     TYPE  SIZE
/my-data.txt/0000000000000000          file  21B
/my-data.txt/0000000000000001          file  22B
/my-data.txt/0000000000000002          file  24B
/my-data.txt/0000000000000003          file  21B
...
NAME                                     TYPE  SIZE
/my-data.txt/0000000000000031          file  22B
```

### Example: Appending to files with `--split`

Combining `--split` with the default *append* behavior of `pachctl put file` enables flexible and scalable processing of record-oriented file data from external, legacy systems.

Pachyderm ensures that only the newly added data gets processed when you append to an existing file by using `--split`. Pachyderm optimizes storage utilization by automatically deduplicating each split-file. If you split a large file with many duplicate lines or objects with identical hashes might use less space in PFS than it does as a single file outside of PFS.

To complete this example, follow these steps:

1. Create a file `count.txt` with the following lines:

```
Zero
One
Two
Three
Four
```

2. Put the `count.txt` file into a Pachyderm repository called `raw_data`:

```
$ pachctl put file -f count.txt raw_data@master --split line
```

This command splits the `count.txt` file by line and creates a separate file with one line in each file. Also, this operation creates five datums that are processed by the pipelines that use this repository as input.

3. View the repository contents:

```
$ pachctl list file raw_data@master NAME TYPE SIZE /count.txt dir 24B
```

Pachyderm created a directory called `count.txt`.

4. View the contents of the `count.txt` directory:

```
$ pachctl list file raw_data@master:count.txt
NAME                                TYPE SIZE
/count.txt/000000000000000000    file 4B
/count.txt/000000000000000001    file 4B
/count.txt/000000000000000002    file 6B
/count.txt/000000000000000003    file 5B
/count.txt/000000000000000004    file 5B
```

In the output above, you can see that Pachyderm created five split-files from the original `count.txt` file. Each file has one line from the original `count.txt`. You can check the contents of each file by running the `pachctl get file` command. For example, to get the contents of `count.txt/000000000000000000`, run the following command:

```
```bash
$ pachctl get file raw_data@master:count.txt/000000000000000000
Zero
```

This operation creates five datums that are processed by the pipelines that use this repository as input.

5. Create a one-line file called `more-count.txt` with the following content:

```
Five
```

6. Load this file into Pachyderm by appending it to the `count.txt` file:

```
$ pachctl put file raw_data@master:count.txt -f more-count.txt --split line
```

**Note:** If you do not specify `--split` flag while appending to a file that was previously split, Pachyderm displays the following error message:



```
could not put file at "/count.txt"; a file of type directory is already there
```

7. Verify that another file was added:

```
$ pachctl list file raw_data@master:count.txt
NAME                                TYPE SIZE
/count.txt/000000000000000000    file 4B
/count.txt/000000000000000001    file 4B
/count.txt/000000000000000002    file 6B
/count.txt/000000000000000003    file 5B
/count.txt/000000000000000004    file 5B
/count.txt/000000000000000005    file 4B

The `count.txt/000000000000000005` file was added to the input
repository. Pachyderm considers
this new file as a separate datum. Therefore, pipelines process
only that datum instead of all the chunks of `count.txt`.
```

8. Get the contents of the `count.txt/000000000000000005` file:

```
$ pachctl get file raw_data@master:count.txt/000000000000000005
Five
```

### Example: Overwriting Files with `--split`

The behavior of Pachyderm when a file loaded with `--split` is overwritten is simple to explain but subtle in its implications. Most importantly, it can have major implications when new rows are inserted within the file as opposed to just being appended to the end. The loaded file is split into those sequentially-named files, as shown above. If any of those resulting split-files hashes differently than the one it is replacing, it causes the Pachyderm Pipeline System to process that data. This can have significant consequences for downstream processing.

To complete this example, follow these steps:

1. Create a file `count.txt` with the following lines:

```
One
Two
Three
Four
Five
```

2. Put the file into a Pachyderm repository called `raw_data`:

```
$ pachctl put file -f count.txt raw_data@master --split line
```

This command splits the `count.txt` file by line and creates a separate file with one line in each file. Also, this operation creates five datums that are processed by the pipelines that use this repository as input.

3. View the repository contents:

```
$ pachctl list file raw_data@master NAME TYPE SIZE /count.txt dir 24B
```

Pachyderm created a directory called `count.txt`.

4. View the contents of the `count.txt` directory:

```
$ pachctl list file raw_data@master:count.txt
NAME                                TYPE SIZE
/count.txt/000000000000000000    file 4B
/count.txt/000000000000000001    file 4B
/count.txt/000000000000000002    file 6B
/count.txt/000000000000000003    file 5B
/count.txt/000000000000000004    file 5B
```

In the output above, you can see that Pachyderm created five split-files from the original `count.txt` file. Each file has one line from the original `count.txt` file. You can check the contents of each file by running the `pachctl get file` command. For example, to get the contents of `count.txt/000000000000000000`, run the following command:

```
```bash
$ pachctl get file raw_data@master:count.txt/000000000000000000
One
```

5. In your local directory, modify the original `count.txt` file by inserting the word *Zero* on the first line:

```
Zero
One
Two
Three
Four
Five
```

6. Upload the updated `count.txt` file into the `raw_data` repository by using the `--split` and `--overwrite` flags:

```
$ pachctl put file -f count.txt raw_data@master:count.txt --split line --overwrite
```

Because Pachyderm takes the file name into account when hashing data for a pipeline, it considers every single split-file as new, and the pipelines that use this repository as input process all six datums.

7. List the directory:

```
$ pachctl list file raw_data@master:count.txt
NAME                                TYPE SIZE
/count.txt/000000000000000000    file 5B
/count.txt/000000000000000001    file 4B
/count.txt/000000000000000002    file 4B
/count.txt/000000000000000003    file 6B
/count.txt/000000000000000004    file 5B
/count.txt/000000000000000005    file 5B
```

The `/count.txt/000000000000000000` file now has the newly added *Zero* line. To verify the contents of the file, run:

```
$ pachctl get file raw_data@master:count.txt/000000000000000000
Zero
```

See also:

- Splitting Data

## 6.6 Combine, Merge, and Join Data

Before you read this section, make sure that you understand the concepts described in Distributed Processing

In some of your projects, you might need to match datums from multiple data repositories to process, join, or aggregate data. For example, you might need to process together multiple records that correspond to a certain user, experiment, or device.

In these cases, you can create pipelines that perform the following steps:

More specifically, you need to create the following pipelines:

1. Create a pipeline that groups all of the records for a specific key and index.
2. Create another pipeline that takes that grouped output and performs the merging, joining, or other processing for the group.

You can use these two data-combining pipelines for merging or grouped processing of data from various experiments, devices, and so on. You can also apply the same pattern to perform distributed joins of tabular data or data from database tables. For example, you can join user email records together with user IP records on the key and index of a user ID.

You can parallelize each of the stages across workers to scale with the size of your data and the number of data sources that you want to merge.

**Tip:** If your data is not split into separate files for each record, you can split it automatically as described in [Splitting Data for Distributed Processing](#).

## 6.6.1 Group Matching Records

The first pipeline that you create groups the records that need to be processed together.

In this example, you have two repositories A and B with JSON records. These repositories might correspond to two experiments, two geographic regions, two different devices that generate data, or other.

The following diagram displays the first pipeline:

The repository A has the following structure:

```
$ pachctl list file A@master
NAME          TYPE      SIZE
1.json        file      39 B
2.json        file      39 B
3.json        file      39 B
```

The repository B has the following structure:

```
$ pachctl list file B@master
NAME          TYPE      SIZE
1.json        file      39 B
2.json        file      39 B
3.json        file      39 B
```

If you want to process A/1.json with B/1.json to merge their contents or otherwise process them together, you need to group each set of JSON records into respective datums that the pipelines that you create in [Process grouped records](#) can process together.

The grouping pipeline takes a union of A and B as inputs, each with glob pattern /\*. While the pipeline processes a JSON file, the data is copied to a folder in the output that corresponds to the key and index for that record. In this example, it is just the number in the file name. Pachyderm also renames the files to unique names that correspond to the source:

```
/1
  A.json
  B.json
/2
  A.json
  B.json
/3
  A.json
  B.json
```

When you group your data, set the following parameters in the pipeline specification:

- In the `pfs` section, set `"empty_files": true` to avoid unnecessary downloads of data.
- Use symlinks to avoid unnecessary uploads of data and unnecessary data duplication.

## 6.6.2 Process Grouped Records

After you group the records together by using the grouping pipeline, you can use a merging pipeline on the `group` repository with a glob pattern of `/*`. By using the glob pattern of `/*` the pipeline can process each grouping of records in parallel.

The following diagram displays the second pipeline:

The second pipeline performs merging, aggregation, or other processing on the respective grouping of records. It can also output each respective result to the root of the output directory:

```
$ pachctl list file merge@master
NAME                TYPE      SIZE
result_1.json       file      39 B
result_2.json       file      39 B
result_3.json       file      39 B
```

## 6.7 Create a Machine Learning Workflow

Because Pachyderm is a language and framework agnostic and platform, and because it easily distributes analysis over large data sets, data scientists can use any tooling for creating machine learning workflows. Even if that tooling is not familiar to the rest of an engineering organization, data scientists can autonomously develop and deploy scalable solutions by using containers. Moreover, Pachyderm's pipeline logic paired with data versioning make any results reproducible for debugging purposes or during the development of improvements to a model.

For maximum leverage of Pachyderm's built functionality, Pachyderm recommends that you combine model training processes, persisted models, and model utilization processes, such as making inferences or generating results, into a single Pachyderm pipeline Directed Acyclic Graph (DAG).

Such a pipeline enables you to achieve the following goals:

- Keep a rigorous historical record of which models were used on what data to produce which results.
- Automatically update online ML models when training data or parameterization changes.
- Easily revert to other versions of an ML model when a new model does not produce an expected result or when *bad data* is introduced into a training data set.

The following diagram demonstrates an ML pipeline:

You can update the training dataset at any time to automatically train a new persisted model. Also, you can use any language or framework, including Apache Spark™, Tensorflow™, scikit-learn™, or other, and output any format of persisted model, such as pickle, XML, POJO, or other. Regardless of the framework, Pachyderm versions the model so that you can track the data that was used to train each model.

Pachyderm processes new data coming into the input repository with the updated model. Also, you can recompute old predictions with the updated model, or test new models on previously input and versioned data. This feature enables you to avoid manual updates to historical results or swapping ML models in production.

For examples of ML workflows in Pachyderm see Machine Learning Examples.

## 6.8 Run a Pipeline on a Specific Commit

Sometimes you need to see the result of merging different commits to analyze and identify correct combinations and potential flows in data collection and processing. Or, you just want to rerun a failed job manually.

Pachyderm enables you to run your pipelines with old commits or in different handpicked combinations of old commits that are stored in separate repositories or branches. This functionality is particularly useful for cross-pipelines, but you can also use it with other types of pipelines.

For example, you have branches A, B, and C. Each of these branches has had commits on it at various times, and each new commit triggered a new job from the pipeline.

Your branches have the following commit history:

- Branch A has commits A1, A2, A3, and A4.
- Branch B has commits B1 and B2.
- Branch C has commits C1, C2, and C3.

For example, you need to see the result of the pipeline with the combination of data after A4, B1, and C2 were committed. But none of the output commits were triggered on this particular combination.

To get the result of this combination, you can run the `pachctl run pipeline cross-pipe` command.

### Example:

```
$ pachctl run pipeline cross-pipe A4 B1 C2
```

This command triggers a new job that creates a commit on the pipeline's output branch with the result of that combination of data.

Because A4 is the head of branch A, you can also omit the A4 commit in the command and specify only the C2 and B1 commits:

```
$ pachctl run pipeline cross-pipe C2 B1
```

Pachyderm automatically uses the head for any branch that did not have a commit specified. The order in which you specify the commits does not matter. Pachyderm knows how to match them to the right place.

Also, you can reference the head commit of a branch by using the branch name.

### Example:

```
$ pachctl run pipeline cross-pipe A B1 C2
```

This behaviour implies that if you want to re-run the pipeline on the most recent commits, you can just run `pachctl run pipeline cross-pipe`.

If you try to use a commit from a branch that is not an input branch, `pachctl` returns an error. Similarly, specifying multiple commits from the same branch results in error.

You do not need to run the `pachctl run pipeline cross-pipe` command to initiate a newly created pipeline. Pachyderm runs the new pipelines automatically as you add new commits to the corresponding input branches.

## 6.9 Update a Pipeline

While working with your data, you often need to modify an existing pipeline with new transformation code or pipeline parameters. For example, when you develop a machine learning model, you might need to try different versions of your model while your training data stays relatively constant. To make changes to a pipeline, you need to use the `pachctl update pipeline` command.

### 6.9.1 Update Your Pipeline Specification

If you need to update your pipeline specification, such as change the parallelism settings, add an input repository, or other, you need to update your JSON file and then run the `pachctl update pipeline` command. By default, when you update your code, the new pipeline specification does not reprocess the data that has already been processed. Instead, it processes only the new data that you submit to the input repo. If you want to run the changes in your pipeline against the data in the `HEAD` commit of your input repo, use the `--reprocess` flag. After that, the updated pipeline continues to process new input data. Previous results remain accessible through the corresponding commit IDs.

To update a pipeline specification, complete the following steps:

1. Make the changes in your pipeline specification JSON file.
2. Update the pipeline with the new configuration:

```
$ pachctl update pipeline -f pipeline.json
```

Similar to `create pipeline`, `update pipeline` with the `-f` flag can also take a URL if your JSON manifest is hosted on GitHub or other remote location.

### 6.9.2 Update the Code in a Pipeline

The `pachctl update pipeline` updates the code that you use in one or more of your pipelines. To apply your code changes, you need to build a new Docker image and push it to your Docker image registry.

You can either use your registry instructions to build and push your new image or push the new image by using the flags built into the `pachctl update pipeline` command. Both procedures achieve the same goal, and it is entirely a matter of a personal preference which one of them to follow. If you do not have a build-push process that you already follow, you might prefer to use Pachyderm's built-in functionality.

To create a new image by using the Pachyderm commands, you need to use the `--build` flag with the `pachctl update pipeline` command. By default, if you do not specify a registry with the `--registry` flag, Pachyderm uses [DockerHub](#). When you build your image with Pachyderm, it assigns a random tag to your new image.

If you use a private registry or any other registry that is different from the default value, use the `--registry` flag to specify it. Make sure that you specify the private registry in the pipeline specification.

For example, if you want to push a `pachyderm/opencv` image to a registry located at `localhost:5000`, you need to add this in your pipeline spec:

```
"image": "localhost:5000/pachyderm/opencv"
```

Also, to be able to build and push images, you need to make sure that the Docker daemon is running. Depending on your operating system and the Docker distribution that you use, steps for enabling it might vary.

To update the code in your pipeline, complete the following steps:

1. Make the code changes.
2. Verify that the Docker daemon is running:

```
$ docker ps
```

- If you get an error message similar to the following:

```
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the
↪docker daemon running?
```

enable the Docker daemon. To enable the Docker daemon, see the Docker documentation for your operating system and platform. For example, if you use `minikube` on macOS, run the following command:

```
$ eval $(minikube docker-env)
```

3. Build, tag, and push the new image to your image registry:

- If you prefer to use Pachyderm commands:

- (a) Run the following command:

```
$ pachctl update pipeline -f <pipeline name> --build --registry <registry>
↪ --username <registry user>
```

If you use DockerHub, omit the `--registry` flag.

**Example:**

```
$ pachctl update pipeline -f edges.json --build --username testuser
```

- (b) When prompted, type your image registry password:

**Example:**

```
Password for docker.io/testuser: Building pachyderm/opencv:
↪fle0239fce5441c483b09de425f06b40, this may take a while.
```

- If you prefer to use instructions for your image registry:

- (a) Build, tag, and push a new image as described in the image registry documentation. For example, if you use DockerHub, see [Docker Documentation](#).
- (b) Update the pipeline:

```
$ pachctl update pipeline -f <pipeline.json>
```

## 6.10 Processing Time-Windowed Data

Before you read this section, make sure that you understand the concepts described in the following sections:

- Datum
- Distributed Computing
- Individual Developer Workflow

If you are analyzing data that is changing over time, you might need to analyze historical data. For example, you might need to examine *the last two weeks of data*, *January's data*, or some other moving or static time window of data.

Pachyderm provides the following approaches to this task:

1. *Fixed time windows* - for rigid, fixed time windows, such as months (Jan, Feb, and so on) or days—01-01-17, 01-02-17, and so on).
2. *Moving time windows*
  - for rolling time windows of data, such as three-day windows or two-week windows.

### 6.10.1 Fixed Time Windows

Datum is the basic unit of data partitioning in Pachyderm. The glob pattern property in the pipeline specification defines a datum. When you analyze data within fixed time windows, such as the data that corresponds to fixed calendar dates, Pachyderm recommends that you organize your data repositories so that each of the time windows that you plan to analyze corresponds to a separate file or directory in your repository, and therefore, Pachyderm processes it as a separate datum.

Organizing your repository as described above, enables you to do the following:

- Analyze each time window in parallel.
- Only re-process data within a time window when that data, or a corresponding data pipeline, changes.

For example, if you have monthly time windows of sales data stored in JSON format that needs to be analyzed, you can create a `sales` data repository with the following data:

```
sales
-- January
|  -- 01-01-17.json
|  -- 01-02-17.json
|  -- ...
-- February
|  -- 01-01-17.json
|  -- 01-02-17.json
|  -- ...
-- March
|  -- 01-01-17.json
|  -- 01-02-17.json
|  -- ...
```

When you run a pipeline with `sales` as an input repository and a glob pattern of `/*`, Pachyderm processes each month's worth of sales data in parallel if workers are available. When you add new data into a subset of the months or add data into a new month, for example, May, Pachyderm processes only these updated datums.

More generally, this structure enables you to create the following types of pipelines:

- Pipelines that aggregate or otherwise process daily data on a monthly basis by using the `/*` glob pattern.



- Pipelines that only analyze a particular month's data by using a `/subdir/*` or `/subdir/` glob pattern. For example, `/January/*` or `/January/`.
- Pipelines that process data on daily by using the `/*/*` glob pattern.
- Any combination of the above.

## 6.10.2 Moving or Rolling Time Windows

In some cases, you need to run analyses for moving or rolling time windows that do not correspond to certain calendar months or days. For example, you might need to analyze the last three days of data, the three days of data before that, or similar. In other words, you need to run an analysis for every rolling length of time.

For rolling or moving time windows, there are a couple of recommended patterns:

1. Bin your data in repository folders for each of the moving time windows.
2. Maintain a time-windowed set of data that corresponds to the latest of the moving time windows.

### Bin Data into Moving Time Windows

In this method of processing rolling time windows, you create the following two-pipeline DAGs to analyze time windows efficiently:

By splitting this analysis into two pipelines, you can benefit from using parallelism at the file level. In other words, *Pipeline 1* can be easily parallelized for each file, and *Pipeline 2* can be parallelized per bin. This structure enables easy pipeline scaling as the number of files increases.

For example, you have three-day moving time windows, and you want to analyze three-day moving windows of sales data. In the first repo, called `sales`, you commit data for the first day of sales:

```
sales
-- 01-01-17.json
```

In the first pipeline, you specify to bin this data into a directory that corresponds to the first rolling time window from 01-01-17 to 01-03-17:

```
binned_sales
-- 01-01-17_to_01-03-17
  -- 01-01-17.json
```

When the next day's worth of sales is committed, that data lands in the `sales` repository:

```
sales
-- 01-01-17.json
-- 01-02-17.json
```

Then, the first pipeline executes again to bin the 01-02-17 data into relevant bins. In this case, the data is placed in the previously created bin named 01-01-17 to 01-03-17. However, the data also goes to the bin that stores the data that is received starting on 01-02-17:

```
binned_sales
-- 01-01-17_to_01-03-17
|  -- 01-01-17.json
|  -- 01-02-17.json
-- 01-02-17_to_01-04-17
  -- 01-02-17.json
```

As more and more daily data is added, your repository structure starting to looks as follows:

```

binned_sales
-- 01-01-17_to_01-03-17
|  -- 01-01-17.json
|  -- 01-02-17.json
|  -- 01-03-17.json
-- 01-02-17_to_01-04-17
|  -- 01-02-17.json
|  -- 01-03-17.json
|  -- 01-04-17.json
-- 01-03-17_to_01-05-17
|  -- 01-03-17.json
|  -- 01-04-17.json
|  -- 01-05-17.json
-- ...

```

The following diagram describes how data accumulates in the repository over time:

Your second pipeline can then process these bins in parallel according to the glob pattern of `/*` or as described further. Both pipelines can be easily parallelized.

In the above directory structure, it might seem that data is duplicated. However, under the hood, Pachyderm deduplicates all of these files and maintains a space-efficient representation of your data. The binning of the data is merely a structural re-arrangement to enable you process these types of moving time windows.

It might also seem as if Pachyderm performs unnecessary data transfers over the network to bin files. However, Pachyderm ensures that these data operations do not require transferring data over the network.

## Maintaining a Single Time-Windowed Data Set

The advantage of the binning pattern above is that any of the moving time windows are available for processing. They can be compared, aggregated, and combined in any way, and any results or aggregations are kept in sync with updates to the bins. However, you do need to create a process to maintain the binning directory structure.

There is another pattern for moving time windows that avoids the binning of the above approach and maintains an up-to-date version of a moving time-windowed data set. This approach involves the creation of the following pipelines:

For example, you have three-day moving time windows, and you want to analyze three-day moving windows of sales data. The input data is stored in the `sales` repository:

```

sales
-- 01-01-17.json
-- 01-02-17.json
-- 01-03-17.json
-- 01-04-17.json

```

When the January 4th file, `01-04-17.json`, is committed, the first pipeline pulls out the last three days of data and arranges it in the following order:

```

last_three_days
-- 01-02-17.json
-- 01-03-17.json
-- 01-04-17.json

```

When the January 5th file, `01-05-17.json`, is committed into the `sales` repository:

```
sales
-- 01-01-17.json
-- 01-02-17.json
-- 01-03-17.json
-- 01-04-17.json
-- 01-05-17.json
```

the first pipeline updates the moving window:

```
last_three_days
-- 01-03-17.json
-- 01-04-17.json
-- 01-05-17.json
```

The analysis that you need to run on the moving windowed dataset in `moving_sales_window` can use the `/` or `/*` glob pattern, depending on whether you need to process all of the time-windowed files together or if they can be processed in parallel.

**Warning** - When you create this type of moving time-windowed data set, the concept of *now* or *today* is relative. You must define the time based on your use case. For example, by configuring to use UTC. Do not use functions such as `time.now()` to determine the current time. The actual time when this pipeline runs might vary.

## 6.11 Delete Data

If *bad* data was committed into a Pachyderm input repository, your pipeline might result in an error. In this case, you might need to delete this data to resolve the issue. Depending on the nature of the bad data and whether or not the bad data is in the HEAD of the branch, you can perform one of the following actions:

- *Delete the HEAD of a Branch.* If the incorrect data was added in the latest commit and no additional data was committed since then, follow the steps in this section to fix the HEAD of the corrupted branch.
- *Delete non-HEAD Commits.* If after committing the incorrect data, you have added more data to the same branch, follow the steps in this section to delete corrupted files.
- *Delete sensitive data.* If the bad commit included sensitive data that you need immediately and completely erase from Pachyderm, follow the steps in this section to purge data.

### 6.11.1 Delete the HEAD of a Branch

If you have just committed incorrect, corrupt, or otherwise bad data to a branch in a Pachyderm repository, the HEAD of your branch, or the latest commit is bad. Users who read from that commit might be misled, and pipelines subscribed to it might fail or produce bad downstream output. You can solve this issue by running the `pachctl delete commit` command.

To fix a broken HEAD, complete the following steps:

```
$ pachctl delete commit <repo>@<branch-or-commit-id>
```

When you delete a bad commit, Pachyderm performs the following actions:

- Deletes the commit metadata.
- Changes HEADs of all the branches that had the bad commit as their HEAD to the bad commit's parent. If the bad commit does not have a parent, Pachyderm sets the branch's HEAD to `nil`.

- If the bad commit has children, sets their parents to the deleted commit parent. If the deleted commit does not have a parent, then the children commit parents are set to `nil`.
- Deletes all the jobs that were triggered by the bad commit. Also, Pachyderm interrupts all running jobs, including not only the jobs that use the bad commit as a direct input but also the ones farther downstream in your DAG.
- Deletes the output commits from the deleted jobs. All the actions listed above are applied to those commits as well.

### 6.11.2 Delete Old Commits

If you have committed more data to the branch after the bad data was added, you can try to delete the commit as described in *Deleting the HEAD of a Branch*. However, unless the subsequent commits overwrote or deleted the bad data, the bad data might still be present in the children commits. Deleting a commit does not modify its children.

In Git terms, `pachctl delete commit` is equivalent to squashing a commit out of existence, such as with the `git reset --hard` command. The `delete commit` command is not equivalent to reverting a commit in Git. The reason for this behavior is that the semantics of revert can get ambiguous when the files that are being reverted have been otherwise modified. Because Pachyderm is a centralized system and the volume of data that you typically store in Pachyderm is large, merge conflicts can quickly become untenable. Therefore, Pachyderm prevents merge conflicts entirely.

To resolve issues with the commits that are not at the tip of the branch, you can try to delete the children commits. However, those commits might also have the data that you might want to keep.

To delete a file in an older commit, complete the following steps:

1. Start a new commit:

```
$ pachctl start commit <repo>@<branch>
```

2. Delete all corrupted files from the newly opened commit:

```
$ pachctl delete file <repo>@<branch or commitID>:/path/to/files
```

3. Finish the commit:

```
$ pachctl finish commit <repo>@<branch>
```

4. Delete the initial bad commit and all its children up to the newly finished commit.

Depending on how you use Pachyderm, the final step might be optional. After you finish the commit, the HEADs of all your branches converge to correct results as downstream jobs finish. However, deleting those commits cleans up your commit history and ensures that the errant data is not available when non-HEAD versions of the data is read.

### 6.11.3 Delete Sensitive Data

When you delete data as described in Delete Old Commits, Pachyderm does not immediately delete it from the physical disk. Instead, Pachyderm deletes references to the underlying data and later performs garbage collection. That is when the data is truly erased from the disk.

If you have accidentally committed sensitive data and you need to ensure that it is immediately erased and inaccessible, complete the following steps:

1. Delete all the references to data as described in Delete Old Commits.

## 2. Run `garbage-collect` :

```
$ pachctl garbage-collect
```

To make garbage collection more comprehensive, increase the amount of memory that is used during the garbage collection operation by specifying the `--memory` flag. The default value is 10 MB.

## 6.12 Configure Distributed Computing

Distributing your computations across multiple workers is a fundamental part of any big data processing. When you build production-scale pipelines, you need to adjust the number of workers and resources that are allocated to each job to optimize throughput.

A Pachyderm worker is an identical Kubernetes pod that runs the Docker image that you specified in the pipeline spec. Your analysis code does not affect how Pachyderm distributes the workload among workers. Instead, Pachyderm spreads out the data that needs to be processed across the various workers and makes that data available for your code.

When you create a pipeline, Pachyderm spins up worker pods that continuously run in the cluster waiting for new data to be available for processing. You can change this behavior by setting `"standby" : true`. Therefore, you do not need to recreate and schedule workers for every new job.

For each job, all the datums are queued up and then distributed across the available workers. When a worker finishes processing its datum, it grabs a new datum from the queue until all the datums complete processing. If a worker pod crashes, its datums are redistributed to other workers for maximum fault tolerance.

The following animation shows how distributed computing works:

In the diagram above, you have three Pachyderm worker pods that process your data. When a pod finishes processing a datum, it automatically takes another datum from the queue to process it. Datums might be different in size and, therefore, some of them might be processed faster than others.

Each datum goes through the following processing phases inside a Pachyderm worker pod:

When a datum completes a phase, the Pachyderm worker moves it to the next one while another datum from the queue takes its place in the processing sequence.

The following animation displays what happens inside a pod during the datum processing:

You can control the number of worker pods that Pachyderm runs in a pipeline by defining the `parallelism` parameter in the pipeline specification.

```
"parallelism_spec": {
  // Exactly one of these two fields should be set
  "constant": int
  "coefficient": double
```

Pachyderm has the following parallelism strategies that you can set in the pipeline spec:

By default, Pachyderm sets `parallelism` to `"constant": 1`, which means that it spawns one worker per Kubernetes node for this pipeline.

**See also:**

- Glob Pattern
- Pipeline Specification

## 6.13 Deferred Processing of Data

While a Pachyderm pipeline is running, it processes any new data that you commit to its input branch. However, in some cases, you want to commit data more frequently than you want to process it.

Because Pachyderm pipelines do not reprocess the data that has already been processed, in most cases, this is not an issue. But, some pipelines might need to process everything from scratch. For example, you might want to commit data every hour, but only want to retrain a machine learning model on that data daily because it needs to train on all the data from scratch.

In these cases, you can leverage a massive performance benefit from deferred processing. This section covers how to achieve that and control what gets processed.

Pachyderm controls what is being processed by using the *filesystem*, rather than at the pipeline level. Although pipelines are inflexible, they are simple and always try to process the data at the heads of their input branches. In contrast, the filesystem is very flexible and gives you the ability to commit data in different places and then efficiently move and rename the data so that it gets processed when you want.

### 6.13.1 Configure a Staging Branch in an Input repository

When you want to load data into Pachyderm without triggering a pipeline, you can upload it to a staging branch and then submit accumulated changes in one batch by re-pointing the `HEAD` of your `master` branch to a commit in the staging branch.

Although, in this section, the branch in which you consolidate changes is called `staging`, you can name it as you like. Also, you can have multiple staging branches. For example, `dev1`, `dev2`, and so on.

In the example below, the repository that is created called `data`.

To configure a staging branch, complete the following steps:

1. Create a repository. For example, `data`.

```
$ pachctl create repo data
```

2. Create a `master` branch.

```
$ pachctl create branch data@master
```

3. View the created branch:

```
$ pachctl list branch data
BRANCH HEAD
master -
```

**Note:** No HEAD means that nothing has yet been committed into this branch. When you commit data to the `master` branch, the pipeline immediately starts a job to process it. However, if you want to commit something without immediately processing it, you need to commit it to a different branch.

4. Commit a file to the staging branch:

```
$ pachctl put file data@staging -f <file>
```

Pachyderm automatically creates the `staging` branch. Your repo now has 2 branches, `staging` and `master`. In this example, the `staging` name is used, but you can name the branch as you want.

5. Verify that the branches were created:

```
$ pachctl list branch data
BRANCH HEAD
staging f3506f0fab6e483e8338754081109e69
master -
```

The `master` branch still does not have a `HEAD` commit, but the new branch, `staging`, does. There still have been no jobs, because there are no pipelines that take `staging` as inputs. You can continue to commit to `staging` to add new data to the branch, and the pipeline will not process anything.

6. When you are ready to process the data, update the `master` branch to point it to the head of the `staging` branch:

```
$ pachctl create branch data@master --head staging
```

7. List your branches to verify that the `master` branch has a `HEAD` commit:

```
$ pachctl list branch
staging f3506f0fab6e483e8338754081109e69
master f3506f0fab6e483e8338754081109e69
```

The `master` and `staging` branches now have the same `HEAD` commit. This means that your pipeline has data to process.

8. Verify that the pipeline has new jobs:

```
$ pachctl list job
ID PIPELINE STARTED DURATION
↪RESTART PROGRESS DL UL STATE
061b0ef8f44f41bab5247420b4e62ca2 test 32 seconds ago Less than a second 0
↪ 6 + 0 / 6 108B 24B success
```

You should see one job that Pachyderm created for all the changes you have submitted to the `staging` branch. While the commits to the `staging` branch are ancestors of the current `HEAD` in `master`, they were never the actual `HEAD` of `master` themselves, so they do not get processed. This behavior works for most of the use cases because commits in Pachyderm are generally additive, so processing the `HEAD` commit also processes data from previous commits.

### 6.13.2 Process Specific Commits

Sometimes you want to process specific intermediary commits that are not in the `HEAD` of the branch. To do this, you need to set `master` to have these commits as `HEAD`. For example, if you submitted ten commits in the `staging` branch and you want to process the seventh, third, and most recent commits, you need to run the following commands respectively:

```
$ pachctl create branch data@master --head staging^7
$ pachctl create branch data@master --head staging^3
$ pachctl create branch data@master --head staging
```

When you run the commands above, Pachyderm creates a job for each of the commands one after another. Therefore, when one job is completed, Pachyderm starts the next one. To verify that Pachyderm created jobs for these commands, run `pachctl list job`.

## Change the HEAD of your Branch

You can move backward to previous commits as easily as advancing to the latest commits. For example, if you want to change the final output to be the result of processing `staging^1`, you can *roll back* your HEAD commit by running the following command:

```
$ pachctl create branch data@master --head staging^1
```

This command starts a new job to process `staging^1`. The HEAD commit on your output repo will be the result of processing `staging^1` instead of `staging`.

### 6.13.3 Copy Files from One Branch to Another

Using a staging branch allows you to defer processing. To use this functionality you need to know your input commits in advance. However, sometimes you want to be able to commit data in an ad-hoc, disorganized manner and then organize it later. Instead of pointing your `master` branch to a commit in a staging branch, you can copy individual files from `staging` to `master`. When you run `copy file`, Pachyderm only copies references to the files and does not move the actual data for the files around.

To copy files from one branch to another, complete the following steps:

1. Start a commit:

```
$ pachctl start commit data@master
```

2. Copy files:

```
$ pachctl copy file data@staging:file1 data@master:file1
$ pachctl copy file data@staging:file2 data@master:file2
...
```

3. Close the commit:

```
$ pachctl finish commit data@master
```

Also, you can run `pachctl delete file` and `pachctl put file` while the commit is open if you want to remove something from the parent commit or add something that is not stored anywhere else.

### 6.13.4 Deferred Processing in Output Repositories

You can perform same deferred processing operations with data in output repositories. To do so, rather than committing to a `staging` branch, configure the `output_branch` field in your pipeline specification.

To configure deferred processing in an output repository, complete the following steps:

1. In the pipeline specification, add the `output_branch` field with the name of the branch in which you want to accumulate your data before processing:

**Example:**

```
"output_branch": "staging"
```

2. When you want to process data, run:

```
$ pachctl create-branch pipeline master --head staging
```



### 6.13.5 Automate Branch Switching

Typically, repointing from one branch to another happens when a certain condition is met. For example, you might want to repoint your branch when you have a specific number of commits, or when the amount of unprocessed data reaches a certain size, or at a specific time interval, such as daily, or other. To configure this functionality, you need to create a Kubernetes application that uses Pachyderm APIs and watches the repositories for the specified condition. When the condition is met, the application switches the Pachyderm branch from `staging` to `master`.

## 6.14 Ingress and Egress Data from an External Object Store

Occasionally, you might need to download data from or upload data to an object store that runs in a different cloud platform. For example, you might be running a Pachyderm cluster in Microsoft Azure, but you need to ingress files from an S3 bucket that resides on Amazon AWS.

You can configure Pachyderm to work with an external object store by using the following methods:

- Ingress data from an external object store by using the `pachctl put file` with a URL to the S3 bucket. Example:

```
$ pachctl put file repo@branch -f <s3://my_bucket/file>
```

- Egress data to an external object store by configuring the `egress` files in the pipeline specification. Example:

```
# pipeline.json
"egress": {
  "URL": "s3://bucket/dir"
```

### 6.14.1 Configure Credentials

You can configure Pachyderm to ingress and egress from and to any number of supported cloud object stores, including Amazon S3, Microsoft Azure Blob storage, and Google Cloud Storage. You need to provide Pachyderm with the credentials to communicate with the selected cloud provider.

The credentials are stored in a [Kubernetes secret](#) and share the same security properties.

**Note:** For each cloud provider, parameters and configuration steps might vary.

To provide Pachyderm with the object store credentials, complete the following steps:

1. Deploy object storage:

```
$ pachctl deploy storage <storage-provider> ...
```

2. In the command above, specify `aws`, `google`, or `azure` as a storage provider.
3. Depending on the storage provider, configure the required parameters. Run `pachctl deploy storage <backend> --help` for more information.

For example, if you select `aws`, you need to specify the following parameters:

```
$ pachctl deploy storage aws <region> <bucket-name> <access key id> <secret_
↪access key>
```

See also:

- Custom Object Store

- [Create a Custom Pachyderm Deployment](#)
- [Pipeline Specification](#)

---

## Pipeline Specification

---

This document discusses each of the fields present in a pipeline specification. To see how to use a pipeline spec to create a pipeline, refer to the `pachctl create pipeline doc`.

### 7.1 JSON Manifest Format

```
{
  "pipeline": {
    "name": string
  },
  "description": string,
  "transform": {
    "image": string,
    "cmd": [ string ],
    "stdin": [ string ],
    "err_cmd": [ string ],
    "err_stdin": [ string ],
    "env": {
      string: string
    },
    "secrets": [ {
      "name": string,
      "mount_path": string
    } ],
    {
      "name": string,
      "env_var": string,
      "key": string
    } ],
    "image_pull_secrets": [ string ],
    "accept_return_code": [ int ],
    "debug": bool,
    "user": string,
    "working_dir": string,
  },
  "parallelism_spec": {
    // Set at most one of the following:
    "constant": int,
    "coefficient": number
  },
  "hashtree_spec": {
```

```
"constant": int,
},
"resource_requests": {
  "memory": string,
  "cpu": number,
  "disk": string,
},
"resource_limits": {
  "memory": string,
  "cpu": number,
  "gpu": {
    "type": string,
    "number": int
  }
  "disk": string,
},
"datum_timeout": string,
"datum_tries": int,
"job_timeout": string,
"input": {
  <"pfs", "cross", "union", "cron", or "git" see below>
},
"output_branch": string,
"egress": {
  "URL": "s3://bucket/dir"
},
"standby": bool,
"cache_size": string,
"enable_stats": bool,
"service": {
  "internal_port": int,
  "external_port": int
},
"spout": {
  "overwrite": bool
}
\\ Optionally, you can combine a spout with a service:
"service": {
  "internal_port": int,
  "external_port": int,
  "annotations": {
    "foo": "bar"
  }
}
},
"max_queue_size": int,
"chunk_spec": {
  "number": int,
  "size_bytes": int
},
"scheduling_spec": {
  "node_selector": {string: string},
  "priority_class_name": string
},
"pod_spec": string,
"pod_patch": string,
}
```

-----

"pfs" input

```
-----
"pfs": {
  "name": string,
  "repo": string,
  "branch": string,
  "glob": string,
  "lazy" bool,
  "empty_files": bool
}
```

"cross" or "union" input

```
-----
"cross" or "union": [
  {
    "pfs": {
      "name": string,
      "repo": string,
      "branch": string,
      "glob": string,
      "lazy" bool,
      "empty_files": bool
    }
  },
  {
    "pfs": {
      "name": string,
      "repo": string,
      "branch": string,
      "glob": string,
      "lazy" bool,
      "empty_files": bool
    }
  }
  ...
]
```

"cron" input

```
-----
"cron": {
  "name": string,
  "spec": string,
  "repo": string,
  "start": time,
  "overwrite": bool
}
```

"join" input

```
-----
"join": [
  {
```

```
    "pfs": {
      "name": string,
      "repo": string,
      "branch": string,
      "glob": string,
      "join_on": string
      "lazy": bool
      "empty_files": bool
    }
  },
  {
    "pfs": {
      "name": string,
      "repo": string,
      "branch": string,
      "glob": string,
      "join_on": string
      "lazy": bool
      "empty_files": bool
    }
  }
]
```

-----  
"git" input  
-----

```
"git": {
  "URL": string,
  "name": string,
  "branch": string
}
```

In practice, you rarely need to specify all the fields. Most fields either come with sensible defaults or can be empty. The following text is an example of a minimum spec:

```
{
  "pipeline": {
    "name": "wordcount"
  },
  "transform": {
    "image": "wordcount-image",
    "cmd": ["/binary", "/pfs/data", "/pfs/out"]
  },
  "input": {
    "pfs": {
      "repo": "data",
      "glob": "/*"
    }
  }
}
```

### 7.1.1 Name (required)

`pipeline.name` is the name of the pipeline that you are creating. Each pipeline needs to have a unique name.

Pipeline names must meet the following prerequisites:

- Include only alphanumeric characters, `_` and `-`.
- Begin or end with only alphanumeric characters (not `_` or `-`).
- Not exceed 50 characters in length.

### 7.1.2 Description (optional)

`description` is an optional text field where you can add information about the pipeline.

### 7.1.3 Transform (required)

`transform.image` is the name of the Docker image that your jobs use.

`transform.cmd` is the command passed to the Docker run invocation. Similarly to Docker, `cmd` is not run inside a shell which means that wildcard globbing (`*`), pipes (`|`), and file redirects (`>` and `>>`) do not work. To specify these settings, you can set `cmd` to be a shell of your choice, such as `sh` and pass a shell script to `stdin`.

`transform.stdin` is an array of lines that are sent to your command on `stdin`. Lines do not have to end in newline characters.

`transform.err_cmd` is an optional command that is executed on failed datums. If the `err_cmd` is successful and returns 0 error code, it does not prevent the job from succeeding. This behavior means that `transform.err_cmd` can be used to ignore failed datums while still writing successful datums to the output repo, instead of failing the whole job when some datums fail. The `transform.err_cmd` command has the same limitations as `transform.cmd`.

`transform.err_stdin` is an array of lines that are sent to your error command on `stdin`. Lines do not have to end in newline characters.

`transform.env` is a key-value map of environment variables that Pachyderm injects into the container.

**Note:** There are environment variables that are automatically injected into the container, for a comprehensive list of them see the [Environment Variables](#) section below.

`transform.secrets` is an array of secrets. You can use the secrets to embed sensitive data, such as credentials. The secrets reference Kubernetes secrets by name and specify a path to map the secrets or an environment variable (`env_var`) that the value should be bound to. Secrets must set `name` which should be the name of a secret in Kubernetes. Secrets must also specify either `mount_path` or `env_var` and `key`. See more information about Kubernetes secrets [here](#).

`transform.image_pull_secrets` is an array of image pull secrets, image pull secrets are similar to secrets except that they are mounted before the containers are created so they can be used to provide credentials for image pulling. For example, if you are using a private Docker registry for your images, you can specify it by running the following command:

```
$ kubectl create secret docker-registry myregistrykey --docker-server=DOCKER_REGISTRY_
↪SERVER --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-
↪email=DOCKER_EMAIL
```

And then, notify your pipeline about it by using `"image_pull_secrets": [ "myregistrykey" ]`. Read more about image pull secrets [here](#).

`transform.accept_return_code` is an array of return codes, such as exit codes from your Docker command that are considered acceptable. If your Docker command exits with one of the codes in this array, it is considered a successful run to set job status. 0 is always considered a successful exit code.

`transform.debug` turns on added debug logging for the pipeline.

`transform.user` sets the user that your code runs as, this can also be accomplished with a `USER` directive in your `Dockerfile`.

`transform.working_dir` sets the directory that your command runs from. You can also specify the `WORKDIR` directive in your `Dockerfile`.

`transform.dockerfile` is the path to the `Dockerfile` used with the `--build` flag. This defaults to `./Dockerfile`.

## 7.1.4 Parallelism Spec (optional)

`parallelism_spec` describes how Pachyderm parallelizes your pipeline. Currently, Pachyderm has two parallelism strategies: `constant` and `coefficient`.

If you set the `constant` field, Pachyderm starts the number of workers that you specify. For example, set `"constant":10` to use 10 workers.

If you set the `coefficient` field, Pachyderm starts a number of workers that is a multiple of your Kubernetes cluster's size. For example, if your Kubernetes cluster has 10 nodes, and you set `"coefficient": 0.5`, Pachyderm starts five workers. If you set it to 2.0, Pachyderm starts 20 workers (two per Kubernetes node).

The default if left unset is `"constant=1"`.

## 7.1.5 Resource Requests (optional)

`resource_requests` describes the amount of resources you expect the workers for a given pipeline to consume. Knowing this in advance lets Pachyderm schedule big jobs on separate machines, so that they do not conflict and either slow down or die.

The `memory` field is a string that describes the amount of memory, in bytes, each worker needs (with allowed SI suffixes (M, K, G, Mi, Ki, Gi, and so on). For example, a worker that needs to read a 1GB file into memory might set `"memory": "1.2G"` with a little extra for the code to use in addition to the file. Workers for this pipeline will be placed on machines with at least 1.2GB of free memory, and other large workers will be prevented from using it (if they also set their `resource_requests`).

The `cpu` field is a number that describes the amount of CPU time in `cpu seconds/real seconds` that each worker needs. Setting `"cpu": 0.5` indicates that the worker should get 500ms of CPU time per second. Setting `"cpu": 2` indicates that the worker gets 2000ms of CPU time per second. In other words, it is using 2 CPUs, though worker threads might spend 500ms on four physical CPUs instead of one second on two physical CPUs.

The `disk` field is a string that describes the amount of ephemeral disk space, in bytes, each worker needs with allowed SI suffixes (M, K, G, Mi, Ki, Gi, and so on).

In both cases, the resource requests are not upper bounds. If the worker uses more memory than it is requested, it does not mean that it will be shut down. However, if the whole node runs out of memory, Kubernetes starts deleting pods that have been placed on it and exceeded their memory request, to reclaim memory. To prevent deletion of your worker node, you must set your `memory` request to a sufficiently large value. However, if the total memory requested by all workers in the system is too large, Kubernetes cannot schedule new workers because no machine has enough unclaimed memory. `cpu` works similarly, but for CPU time.

By default, workers are scheduled with an effective resource request of 0 (to avoid scheduling problems that prevent users from being unable to run pipelines). This means that if a node runs out of memory, any such worker might be terminated.

For more information about resource requests and limits see the [Kubernetes docs](#) on the subject.



### 7.1.6 Resource Limits (optional)

`resource_limits` describes the upper threshold of allowed resources a given worker can consume. If a worker exceeds this value, it will be evicted.

The `gpu` field is a number that describes how many GPUs each worker needs. Only whole number are supported, Kubernetes does not allow multiplexing of GPUs. Unlike the other resource fields, GPUs only have meaning in Limits, by requesting a GPU the worker will have sole access to that GPU while it is running. It's recommended to enable `standby` if you are using GPUs so other processes in the cluster will have access to the GPUs while the pipeline has nothing to process. For more information about scheduling GPUs see the [Kubernetes docs](#) on the subject.

### 7.1.7 Datum Timeout (optional)

`datum_timeout` is a string (e.g. `1s`, `5m`, or `15h`) that determines the maximum execution time allowed per datum. So no matter what your parallelism or number of datums, no single datum is allowed to exceed this value.

### 7.1.8 Datum Tries (optional)

`datum_tries` is a int (e.g. `1`, `2`, or `3`) that determines the number of retries that a job should attempt given failure was observed. Only failed datums are retries in retry attempt. The the operation succeeds in retry attempts then job is successful, otherwise the job is marked as failure.

### 7.1.9 Job Timeout (optional)

`job_timeout` is a string (e.g. `1s`, `5m`, or `15h`) that determines the maximum execution time allowed for a job. It differs from `datum_timeout` in that the limit gets applied across all workers and all datums. That means that you'll need to keep in mind the parallelism, total number of datums, and execution time per datum when setting this value. Keep in mind that the number of datums may change over jobs. Some new commits may have a bunch of new files (and so new datums). Some may have fewer.

### 7.1.10 Input (required)

`input` specifies repos that will be visible to the jobs during runtime. Commits to these repos will automatically trigger the pipeline to create new jobs to process them. Input is a recursive type, there are multiple different kinds of inputs which can be combined together. The `input` object is a container for the different input types with a field for each, only one of these fields be set for any instantiation of the object.

```
{
  "pfs": pfs_input,
  "union": union_input,
  "cross": cross_input,
  "cron": cron_input
}
```

#### PFS Input

**Note:** Atom inputs were renamed to PFS inputs in version 1.8.1. If you are using an older version of Pachyderm, replace every instance of `pfs` with `atom` in the code below.

PFS inputs are the simplest inputs, they take input from a single branch on a single repo.

```
{
  "name": string,
  "repo": string,
  "branch": string,
  "glob": string,
  "lazy" bool,
  "empty_files": bool
}
```

`input.pfs.name` is the name of the input. An input with the name `XXX` is visible under the path `/pfs/XXX` when a job runs. Input names must be unique if the inputs are crossed, but they may be duplicated between `PFSInput`s that are combined by using the `union` operator. This is because when `PFSInput`s are combined, you only ever see a datum from one input at a time. Overlapping the names of combined inputs allows you to write simpler code since you no longer need to consider which input directory a particular datum comes from. If an input's name is not specified, it defaults to the name of the repo. Therefore, if you have two crossed inputs from the same repo, you must give at least one of them a unique name.

`input.pfs.repo` is the name of the Pachyderm repository with the data that you want to join with other data.

`input.pfs.branch` is the `branch` to watch for commits. If left blank, Pachyderm sets this value to `master`.

`input.pfs.glob` is a glob pattern that is used to determine how the input data is partitioned.

`input.pfs.lazy` controls how the data is exposed to jobs. The default is `false` which means the job eagerly downloads the data it needs to process and exposes it as normal files on disk. If `lazy` is set to `true`, data is exposed as named pipes instead, and no data is downloaded until the job opens the pipe and reads it. If the pipe is never opened, then no data is downloaded.

Some applications do not work with pipes. For example, pipes do not support applications that makes `syscalls` such as `Seek`. Applications that can work with pipes must use them since they are more performant. The difference will be especially notable if the job only reads a subset of the files that are available to it.

**Note:** `lazy` currently does not support datums that contain more than 10000 files.

`input.pfs.empty_files` controls how files are exposed to jobs. If set to `true`, it causes files from this PFS to be presented as empty files. This is useful in shuffle pipelines where you want to read the names of files and reorganize them by using symlinks.

## Union Input

Union inputs take the union of other inputs. In the example below, each input includes individual datums, such as if `foo` and `bar` were in the same repository with the glob pattern set to `/*`. Alternatively, each of these datums might have come from separate repositories with the glob pattern set to `/` and being the only filesystem objects in these repositories.

inputA	inputB	inputA	inputB
foo	fizz	foo	
bar	buzz	fizz	
		bar	
		buzz	

The union inputs do not take a name and maintain the names of the sub-inputs. In the example above, you would see files under `/pfs/inputA/...` or `/pfs/inputB/...`, but never both at the same time. When you write code to address this behavior, make sure that your code first determines which input directory is present. Starting with Pachyderm 1.5.3, we recommend that you give your inputs the same `Name`. That way your code only needs to handle data being present in that directory. This only works if your code does not need to be aware of which of the underlying inputs the data comes from.

`input.union` is an array of inputs to combine. The inputs do not have to be `pfs` inputs. They can also be `union` and `cross` inputs. Although, there is no reason to take a union of unions because union is associative.

## Cross Input

Cross inputs create the cross product of other inputs. In other words, a cross input creates tuples of the datums in the inputs. In the example below, each input includes individual datums, such as if `foo` and `bar` were in the same repository with the glob pattern set to `/*`. Alternatively, each of these datums might have come from separate repositories with the glob pattern set to `/` and being the only filesystem objects in these repositories.

inputA	inputB	inputA inputB
foo	fizz	(foo, fizz)
bar	buzz	(foo, buzz)
		(bar, fizz)
		(bar, buzz)

The cross inputs above do not take a name and maintain the names of the sub-inputs. In the example above, you would see files under `/pfs/inputA/...` and `/pfs/inputB/...`.

`input.cross` is an array of inputs to cross. The inputs do not have to be `pfs` inputs. They can also be `union` and `cross` inputs. Although, there is no reason to take a union of unions because union is associative.

## Cron Input

Cron inputs allow you to trigger pipelines based on time. A Cron input is based on the Unix utility called `cron`. When you create a pipeline with one or more Cron inputs, `pachd` creates a repo for each of them. The start time for Cron input is specified in its spec. When a Cron input triggers, `pachd` commits a single file, named by the current [RFC 3339 timestamp](#) to the repo which contains the time which satisfied the spec.

```
{
  "name": string,
  "spec": string,
  "repo": string,
  "start": time,
  "overwrite": bool
}
```

`input.cron.name` is the name for the input. Its semantics is similar to those of `input.pfs.name`. Except that it is not optional.

`input.cron.spec` is a cron expression which specifies the schedule on which to trigger the pipeline. To learn more about how to write schedules, see the [Wikipedia page on cron](#). Pachyderm supports non-standard schedules, such as `"@daily"`.

`input.cron.repo` is the repo which Pachyderm creates for the input. This parameter is optional. If you do not specify this parameter, then `"<pipeline-name>_<input-name>"` is used by default.

`input.cron.start` is the time to start counting from for the input. This parameter is optional. If you do not specify this parameter, then the time when the pipeline was created is used by default. Specifying a time enables you to run on matching times from the past or skip times from the present and only start running on matching times in the future. Format the time value according to [RFC 3339](#).

`input.cron.overwrite` is a flag to specify whether you want the timestamp file to be overwritten on each tick. This parameter is optional, and if you do not specify it, it defaults to simply writing new files on each tick. By default, `pachd` expects only the new information to be written out on each tick and combines that data with the data from

the previous ticks. If "overwrite" is set to `true`, it expects the full dataset to be written out for each tick and replaces previous outputs with the new data written out.

## Join Input

A join input enables you to join files that are stored in separate Pachyderm repositories and that match a configured glob pattern. A join input must have the `glob` and `join_on` parameters configured to work properly. A join can combine multiple PFS inputs.

You can specify the following parameters for the `join` input.

- `input.pfs.name` — the name of the PFS input that appears in the `INPUT` field when you run the `pachctl list job` command. If an input name is not specified, it defaults to the name of the repo.
- `input.pfs.repo` — see the description in *PFS Input*. the name of the Pachyderm repository with the data that you want to join with other data.
- `input.pfs.branch` — see the description in *PFS Input*.
- `input.pfs.glob` — a wildcard pattern that defines how a dataset is broken up into datums for further processing. When you use a glob pattern in joins, it creates a naming convention that Pachyderm uses to join files. In other words, Pachyderm joins the files that are named according to the glob pattern and skips those that are not.

You can specify the glob pattern for joins in a parenthesis to create one or multiple capture groups. A capture group can include one or multiple characters. Use standard UNIX globbing characters to create capture, groups, including the following:

- `?` — matches a single character in a filepath. For example, you have files named `file000.txt`, `file001.txt`, `file002.txt`, and so on. You can set the glob pattern to `/file(?) (?) (?)` and the `join_on` key to `$2`, so that Pachyderm matches only the files that have same second character.
- `*` — any number of characters in the filepath. For example, if you set your capture group to `/(*)`, Pachyderm matches all files in the root directory.

If you do not specify a correct `glob` pattern, Pachyderm performs the `cross` input operation instead of `join`.

- `input.pfs.lazy` — see the description in *PFS Input*.
- `input.pfs.empty_files` — see the description in *PFS Input*.

## Git Input (alpha feature)

Git inputs allow you to pull code from a public git URL and execute that code as part of your pipeline. A pipeline with a Git Input will get triggered (i.e. will see a new input commit and will spawn a job) whenever you commit to your git repository.

**Note:** This only works on cloud deployments, not local clusters.

`input.git.URL` must be a URL of the form: `https://github.com/foo/bar.git`

`input.git.name` is the name for the input, its semantics are similar to those of `input.pfs.name`. It is optional.

`input.git.branch` is the name of the git branch to use as input.

Git inputs also require some additional configuration. In order for new commits on your git repository to correspond to new commits on the Pachyderm Git Input repo, we need to setup a git webhook. At the moment, only GitHub is supported. (Though if you ask nicely, we can add support for GitLab or BitBucket).

1. Create your Pachyderm pipeline with the Git Input.
2. To get the URL of the webhook to your cluster, do `pachctl inspect pipeline` on your pipeline. You should see a `Githook URL` field with a URL set. Note - this will only work if you've deployed to a cloud provider (e.g. AWS, GKE). If you see `pending` as the value (and you've deployed on a cloud provider), it's possible that the service is still being provisioned. You can check `kubectl get svc` to make sure you see the `githook` service running.
3. To setup the GitHub webhook, navigate to:

```
https://github.com/<your_org>/<your_repo>/settings/hooks/new
```

Or navigate to webhooks under settings. Then you'll want to copy the `Githook URL` into the 'Payload URL' field.

### 7.1.11 Output Branch (optional)

This is the branch where the pipeline outputs new commits. By default, it's "master".

### 7.1.12 Egress (optional)

`egress` allows you to push the results of a Pipeline to an external data store such as s3, Google Cloud Storage or Azure Storage. Data will be pushed after the user code has finished running but before the job is marked as successful.

For more information, see [Exporting Data by using egress](#)

### 7.1.13 Standby (optional)

`standby` indicates that the pipeline should be put into "standby" when there's no data for it to process. A pipeline in standby will have no pods running and thus will consume no resources, it's state will be displayed as "standby".

Standby replaces `scale_down_threshold` from releases prior to 1.7.1.

### 7.1.14 Cache Size (optional)

`cache_size` controls how much cache a pipeline's sidecar containers use. In general, your pipeline's performance will increase with the cache size, but only up to a certain point depending on your workload.

Every worker in every pipeline has a limited-functionality `pachd` server running adjacent to it, which proxies PFS reads and writes (this prevents thundering herds when jobs start and end, which is when all of a pipeline's workers are reading from and writing to PFS simultaneously). Part of what these "sidecar" `pachd` servers do is cache PFS reads. If a pipeline has a cross input, and a worker is downloading the same datum from one branch of the input repeatedly, then the cache can speed up processing significantly.

### 7.1.15 Enable Stats (optional)

`enable_stats` turns on stat tracking for the pipeline. This will cause the pipeline to commit to a second branch in its output repo called "stats". This branch will have information about each datum that is processed including: timing information, size information, logs and a `/pfs` snapshot. This information can be accessed through the `inspect datum` and `list datum` `pachctl` commands and through the webUI.

Note: enabling stats will use extra storage for logs and timing information. However it will not use as much extra storage as it appears to due to the fact that snapshots of the `/pfs` directory, which are generally the largest thing stored, don't actually require extra storage because the data is already stored in the input repos.

### 7.1.16 Service (alpha feature, optional)

`service` specifies that the pipeline should be treated as a long running service rather than a data transformation. This means that `transform.cmd` is not expected to exit, if it does it will be restarted. Furthermore, the service is exposed outside the container using a Kubernetes service. `"internal_port"` should be a port that the user code binds to inside the container, `"external_port"` is the port on which it is exposed through the `NodePorts` functionality of Kubernetes services. After a service has been created, you should be able to access it at `http://<kubernetes-host>:<external_port>`.

### 7.1.17 Spout (optional)

`spout` is a type of pipeline that processes streaming data. Unlike a union or cross pipeline, a spout pipeline does not have a PFS input. Instead, it opens a Linux *named pipe* into the source of the streaming data. Your pipeline can be either a spout or a service and not both. Therefore, if you added the `service` as a top-level object in your pipeline, you cannot add `spout`. However, you can expose a service from inside of a spout pipeline by specifying it as a field in the `spout spec`. Then, Kubernetes creates a service endpoint that you can expose externally. You can get the information about the service by running `kubectl get services`.

For more information, see Spouts.

### 7.1.18 Max Queue Size (optional)

`max_queue_size` specifies that maximum number of datums that a worker should hold in its processing queue at a given time (after processing its entire queue, a worker “checkpoints” its progress by writing to persistent storage). The default value is 1 which means workers will only hold onto the value that they’re currently processing.

Increasing this value can improve pipeline performance, as that allows workers to simultaneously download, process and upload different datums at the same time (and reduces the total time spent on checkpointing). Decreasing this value can make jobs more robust to failed workers, as work gets checkpointed more often, and a failing worker will not lose as much progress. Setting this value too high can also cause problems if you have `lazy` inputs, as there’s a cap of 10,000 `lazy` files per worker and multiple datums that are running all count against this limit.

### 7.1.19 Chunk Spec (optional)

`chunk_spec` specifies how a pipeline should chunk its datums.

`chunk_spec.number` if nonzero, specifies that each chunk should contain `number` datums. Chunks may contain fewer if the total number of datums don’t divide evenly.

`chunk_spec.size_bytes`, if nonzero, specifies a target size for each chunk of datums. Chunks may be larger or smaller than `size_bytes`, but will usually be pretty close to `size_bytes` in size.

### 7.1.20 Scheduling Spec (optional)

`scheduling_spec` specifies how the pods for a pipeline should be scheduled.

`scheduling_spec.node_selector` allows you to select which nodes your pipeline will run on. Refer to the [Kubernetes docs](#) on node selectors for more information about how this works.

`scheduling_spec.priority_class_name` allows you to select the priority class for the pipeline, which will how Kubernetes chooses to schedule and deschedule the pipeline. Refer to the [Kubernetes docs](#) on priority and preemption for more information about how this works.

### 7.1.21 Pod Spec (optional)

`pod_spec` is an advanced option that allows you to set fields in the pod spec that haven't been explicitly exposed in the rest of the pipeline spec. A good way to figure out what JSON you should pass is to create a pod in Kubernetes with the proper settings, then do:

```
kubectl get po/<pod-name> -o json | jq .spec
```

this will give you a correctly formatted piece of JSON, you should then remove the extraneous fields that Kubernetes injects or that can be set else where.

The JSON is applied after the other parameters for the `pod_spec` have already been set as a [JSON Merge Patch](#). This means that you can modify things such as the storage and user containers.

### 7.1.22 Pod Patch (optional)

`pod_patch` is similar to `pod_spec` above but is applied as a [JSON Patch](#). Note, this means that the process outlined above of modifying an existing pod spec and then manually blanking unchanged fields won't work, you'll need to create a correctly formatted patch by diffing the two pod specs.

## 7.2 The Input Glob Pattern

Each PFS input needs to specify a glob pattern.

Pachyderm uses the glob pattern to determine how many “datums” an input consists of. Datums are the unit of parallelism in Pachyderm. That is, Pachyderm attempts to process datums in parallel whenever possible.

Intuitively, you may think of the input repo as a file system, and you are applying the glob pattern to the root of the file system. The files and directories that match the glob pattern are considered datums.

For instance, let's say your input repo has the following structure:

```
/foo-1
/foo-2
/bar
  /bar-1
  /bar-2
```

Now let's consider what the following glob patterns would match respectively:

- `/` : this pattern matches `/` , the root directory itself, meaning all the data would be a single large datum.
- `/*` : this pattern matches everything under the root directory given us 3 datums: `/foo-1` , `/foo-2` , and everything under the directory `/bar` .
- `/bar/*` : this pattern matches files only under the `/bar` directory: `/bar-1` and `/bar-2`
- `/foo*` : this pattern matches files under the root directory that start with the characters `foo`
- `/**/*.*` : this pattern matches everything that's two levels deep relative to the root: `/bar/bar-1` and `/bar/bar-2`

The datums are defined as whichever files or directories match by the glob pattern. For instance, if we used `/*` , then the job will process three datums (potentially in parallel): `/foo-1` , `/foo-2` , and `/bar` . Both the `bar-1` and `bar-2` files within the directory `bar` would be grouped together and always processed by the same worker.

## 7.3 PPS Mounts and File Access

### 7.3.1 Mount Paths

The root mount point is at `/pfs`, which contains:

- `/pfs/input_name` which is where you would find the datum.
  - Each input will be found here by its name, which defaults to the repo name if not specified.
- `/pfs/out` which is where you write any output.



---

## Environment Variables

---

There are several environment variables that get injected into the user code before it runs. They are:

- `PACH_JOB_ID` the id the currently run job.
- `PACH_OUTPUT_COMMIT_ID` the id of the commit being outputted to.
- For each input there will be an environment variable with the same name defined to the path of the file for that input. For example if you are accessing an input called `foo` from the path `/pfs/foo` which contains a file called `bar` then the environment variable `foo` will have the value `/pfs/foo/bar`. The path in the environment variable is the path which matched the glob pattern, even if the file is a directory, ie if your glob pattern is `/*` it would match a directory `/bar`, the value of `$foo` would then be `/pfs/foo/bar`. With a glob pattern of `/*/*` you would match the files contained in `/bar` and thus the value of `foo` would be `/pfs/foo/bar/quux`.
- For each input there will be an environment variable named `input_COMMIT` indicating the id of the commit being used for that input.

In addition to these environment variables Kubernetes also injects others for Services that are running inside the cluster. These allow you to connect to those outside services, which can be powerful but also can be hard to reason about, as processing might be retried multiple times. For example if your code writes a row to a database that row may be written multiple times due to retries. Interaction with outside services should be *idempotent* to prevent unexpected behavior. Furthermore, one of the running services that your code can connect to is Pachyderm itself, this is generally not recommended as very little of the Pachyderm API is idempotent, but in some specific cases it can be a viable approach.



---

## Config Specification

---

This document outlines the fields in pachyderm configs. This should act as a reference. If you wish to change a config value, you should do so via `pachctl config`.

### 9.1 JSON format

```
{
  "user_id": string,
  "v2": {
    "active_context": string,
    "contexts": {
      string: {
        "pachd_address": string,
        "server_cas": string,
        "session_token": string,
        "active_transaction": string
      },
      ...
    },
    "metrics": bool
  }
}
```

If a field is not set, it will be omitted from JSON entirely. Following is an example of a simple config:

```
{
  "user_id": "514cbe16-e615-46fe-92d9-3156f12885d7",
  "v2": {
    "active_context": "default",
    "contexts": {
      "default": {}
    },
    "metrics": true
  }
}
```

Following is a walk-through of all the fields.

### 9.1.1 User ID

A UUID giving a unique ID for this user for metrics.

### 9.1.2 Metrics

Whether metrics is enabled.

### 9.1.3 Active Context

`v2.active_context` specifies the name of the currently active pachyderm context, as specified in `v2.contexts`.

### 9.1.4 Contexts

A map of context names to their configurations. Pachyderm contexts are akin to kubernetes contexts (and in fact reference the kubernetes context that they're associated with.)

### Pachd Address

A `host:port` specification for connecting to pachd. If this is set, pachyderm will directly connect to the cluster, rather than resorting to kubernetes' port forwarding. If you can set this (because there's no firewall between you and the cluster), you should, as kubernetes' port forwarder is not designed to handle large amounts of data.

### Server CAs

Trusted root certificates for the cluster, formatted as a base64-encoded PEM. This is only set when TLS is enabled.

### Session token

A secret token identifying the current pachctl user within their pachyderm cluster. This is included in all RPCs sent by pachctl, and used to determine if pachctl actions are authorized. This is only set when auth is enabled.

### Active transaction

The currently active transaction for batching together pachctl commands. This can be set or cleared via many of the `pachctl * transaction` commands.

---

## Pachyderm language clients

---

### 10.1 Go Client

The Go client is officially supported by the Pachyderm team. It implements almost all of the functionality that is provided with the `pachctl` CLI tool, and, thus, you can easily integrated operations like `put file` into your applications.

For more info, check out the [godocs](#).

**Note** - A compatible version of `grpc` is needed when using the Go client. You can deduce the compatible version from our [vendor.json](#) file, where you will see something like:

```
{
  "checksumSHA1": "mEyChIkG797MtkrJQXW8X/qZ0l0=",
  "path": "google.golang.org/grpc",
  "revision": "21f8ed309495401e6fd79b3a9fd549582aed1b4c",
  "revisionTime": "2017-01-27T15:26:01Z"
},
```

You can then get this version via:

```
go get google.golang.org/grpc
cd $GOPATH/src/google.golang.org/grpc
git checkout 21f8ed309495401e6fd79b3a9fd549582aed1b4c
```

### 10.2 Python Client

The Python client is officially supported by the Pachyderm team. It implements almost all of the functionality that is provided with the `pachctl` CLI tool, and, thus, you can easily integrated operations like `put file` into your applications.

For more info, check out [the repo](#).

### 10.3 Scala Client

Our users are currently working on a Scala client for Pachyderm. Please contact us if you are interested in helping with this or testing it out.

## 10.4 Other languages

Pachyderm uses a simple [protocol buffer API](#). Protobufs support [a bunch of other languages](#), any of which can be used to programmatically use Pachyderm. We haven't built clients for them yet, but it's not too hard. It's an easy way to contribute to Pachyderm if you're looking to get involved.

---

## S3 Gateway API

---

This outlines the HTTP API exposed by the s3 gateway and any peculiarities relative to S3. The operations largely mirror those documented in S3's [official docs](#).

Generally, you would not call these endpoints directly, but rather use a tool or library designed to work with S3-like APIs. Because of that, some working knowledge of S3 and HTTP is assumed.

### 11.1 Authentication

If authentication is not enabled on the Pachyderm cluster, S3 gateway endpoints can be hit without passing auth credentials.

If authentication is enabled, credentials must be passed using AWS' [signature v2](#) or [signature v4](#) methods. For both methods, set the AWS access key and secret key to the same value. Both values are the Pachyderm auth token that is used to issue the relevant PFS calls. One or both signature methods are built into most s3 tools and libraries already, so you do not need to configure these methods manually.

### 11.2 Operations on the Service

#### 11.2.1 GET Service

Route: GET /.

Lists all of the branches across all of the repos as S3 buckets.

### 11.3 Operations on Buckets

Buckets are represented via `branch.repo`, e.g. the `master.images` bucket corresponds to the `master` branch of the `images` repo.

#### 11.3.1 DELETE Bucket

Route: DELETE /<branch>.<repo>/.

Deletes the branch. If it is the last branch in the repo, the repo is also deleted. Unlike S3, you can delete non-empty branches.

### 11.3.2 GET Bucket (List Objects) Version 1

Route: GET /<branch>.<repo>/

Only S3's list objects v1 is supported.

PFS directories are represented via `CommonPrefixes`. This largely mirrors how S3 is used in practice, but leads to a couple of differences:

- If you set the `delimiter` parameter, it must be `/`.
- Empty directories are included in listed results.

With regard to listed results:

- Due to PFS peculiarities, the `LastModified` field references when the most recent commit to the branch happened, which may or may not have modified the specific object listed.
- The `HTTP ETag` field does not use MD5, but is a cryptographically secure hash of the file contents.
- The S3 `StorageClass` and `Owner` fields always have the same filler value.

### 11.3.3 GET Bucket location

Route: GET /<branch>.<repo>/?location

This will always serve the same location for every bucket, but the endpoint is implemented to provide better compatibility with S3 clients.

### 11.3.4 List Multipart Uploads

Route: GET /<branch>.<repo>/?uploads

Lists the in-progress multipart uploads in the given branch. The `delimiter` query parameter is not supported.

### 11.3.5 PUT Bucket

Route: PUT /<branch>.<repo>/

If the repo does not exist, it is created. If the branch does not exist, it is likewise created. As per S3's behavior in some regions (but not all), trying to create the same bucket twice will return a `BucketAlreadyOwnedByYou` error.

## 11.4 Operations on Objects

Object operations act upon the HEAD commit of branches. Authorization-gated PFS branches are not supported.

### 11.4.1 Delete Multiple Objects

Route: POST /<branch>.<repo>/?delete.

Deletes multiple files specified in the request payload.



### 11.4.2 DELETE Object

Route: DELETE /<branch>.<repo>/<filepath>.

Deletes the PFS file `filepath` in an atomic commit on the HEAD of `branch`.

### 11.4.3 GET Object

Route: GET /<branch>.<repo>/<filepath>.

There is support for range queries and conditional requests, however error response bodies for bad requests using these headers are not standard S3 XML.

With regard to HTTP response headers:

- Due to PFS peculiarities, the HTTP `Last-Modified` header references when the most recent commit to the branch happened, which may or may not have modified this specific object.
- The HTTP `ETag` does not use MD5, but is a cryptographically secure hash of the file contents.

### 11.4.4 PUT Object

Route: PUT /<branch>.<repo>/<filepath>.

Writes the PFS file at `filepath` in an atomic commit on the HEAD of `branch`. Any existing file content is overwritten. Unlike S3, there is no limit to upload size.

Unlike s3, a 64mb max size is not enforced on this endpoint. Especially as the file upload size gets larger, we recommend setting the `Content-MD5` request header to ensure data integrity.

### 11.4.5 Abort Multipart Upload

Route: DELETE /<branch>.<repo>?uploadId=<uploadId>

Aborts an in-progress multipart upload.

### 11.4.6 Complete Multipart Upload

Route: POST /<branch>.<repo>?uploadId=<uploadId>

Completes a multipart upload. If ETags are included in the request payload, they must be of the same format as returned by the s3 gateway when the multipart chunks are included. If they are md5 hashes or any other hash algorithm, they are ignored.

### 11.4.7 Initiate Multipart Upload

Route: POST /<branch>.<repo>?uploads

Initiates a multipart upload.

### 11.4.8 List Parts

Route: GET /<branch>.<repo>?uploadId=<uploadId>

Lists the parts of an in-progress multipart upload.

### 11.4.9 Upload Part

Route: PUT /<branch>.<repo>?uploadId=<uploadId>&partNumber=<partNumber>

Uploads a chunk of a multipart upload.

---

## Pachctl Command Line Tool

---

This document describes Pachyderm Command Line Interface (CLI) tool *pachctl*.

### 12.1 pachctl

#### 12.1.1 Synopsis

Access the Pachyderm API.

Environment variables: `PACHD_ADDRESS=:`, the pachd server to connect to (e.g. `127.0.0.1:30650`). `PACH_CONFIG=`, the path where pachctl will attempt to load your pach config. `JAEGER_ENDPOINT=:`, the Jaeger server to connect to, if `PACH_TRACE` is set `PACH_TRACE={true,false}`, If true, and `JAEGER_ENDPOINT` is set, attach a Jaeger trace to any outgoing RPCs

#### 12.1.2 Options

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

### 12.2 pachctl auth

Auth commands manage access to data in a Pachyderm cluster

#### 12.2.1 Synopsis

Auth commands manage access to data in a Pachyderm cluster

#### 12.2.2 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.3 pachctl auth activate

Activate Pachyderm’s auth system

### 12.3.1 Synopsis

Activate Pachyderm’s auth system, and restrict access to existing data to the user running the command (or the argument to `--initial-admin`), who will be the first cluster admin

```
pachctl auth activate
```

### 12.3.2 Options

<code>--initial-admin string</code>	The subject (robot user <b>or</b> github user) who will be the first cluster admin; the user running <code>'activate'</code> will identify <b>as</b> this user once auth <b>is</b> active. If you <b>set</b> <code>'initial-admin'</code> to a robot user, pachctl will <b>print</b> that robot user's Pachyderm <b>token</b> ; this token is effectively a root token, <b>and if</b> it's lost you will be <b>locked out of your</b> cluster
-------------------------------------	---

### 12.3.3 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.4 pachctl auth check

Check whether you have reader/writer/etc-level access to ‘repo’

### 12.4.1 Synopsis

Check whether you have reader/writer/etc-level access to ‘repo’. For example, ‘`pachctl auth check reader private-data`’ prints “true” if the you have at least “reader” access to the repo “private-data” (you could be a reader, writer, or owner). Unlike `pachctl auth get`, you do not need to have access to ‘repo’ to discover your own access level.

```
pachctl auth check (none|reader|writer|owner) <repo>
```

### 12.4.2 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.5 pachctl auth deactivate

Delete all ACLs, tokens, and admins, and deactivate Pachyderm auth

### 12.5.1 Synopsis

Deactivate Pachyderm’s auth system, which will delete ALL auth tokens, ACLs and admins, and expose all data in the cluster to any user with cluster access. Use with caution.

```
pachctl auth deactivate
```

### 12.5.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.6 pachctl auth get-auth-token

Get an auth token that authenticates the holder as “username”

### 12.6.1 Synopsis

Get an auth token that authenticates the holder as “username”; this can only be called by cluster admins

```
pachctl auth get-auth-token <username>
```

### 12.6.2 Options

```
-q, --quiet    if set, only print the resulting token (if successful). This is
↪useful for scripting, as the output can be piped to use-auth-token
```

### 12.6.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.7 pachctl auth get-config

Retrieve Pachyderm’s current auth configuration

## 12.7.1 Synopsis

Retrieve Pachyderm’s current auth configuration

```
pachctl auth get-config
```

## 12.7.2 Options

```
-o, --output-format string    output format ("json" or "yaml") (default "json")
```

## 12.7.3 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

# 12.8 pachctl auth get

Get the ACL for ‘repo’ or the access that ‘username’ has to ‘repo’

## 12.8.1 Synopsis

Get the ACL for ‘repo’ or the access that ‘username’ has to ‘repo’. For example, ‘pachctl auth get github-alice private-data’ prints “reader”, “writer”, “owner”, or “none”, depending on the privileges that “github-alice” has in “repo”. Currently all Pachyderm authentication uses GitHub OAuth, so ‘username’ must be a GitHub username

```
pachctl auth get [<username>] <repo>
```

## 12.8.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

# 12.9 pachctl auth list-admins

List the current cluster admins

## 12.9.1 Synopsis

List the current cluster admins

```
pachctl auth list-admins
```

## 12.9.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.10 pachctl auth login

Log in to Pachyderm

### 12.10.1 Synopsis

Login to Pachyderm. Any resources that have been restricted to the account you have with your ID provider (e.g. GitHub, Okta) account will subsequently be accessible.

```
pachctl auth login
```

### 12.10.2 Options

```
-o, --one-time-password  If set, authenticate with a Dash-provided One-Time
↪Password, rather than via GitHub
```

### 12.10.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.11 pachctl auth logout

Log out of Pachyderm by deleting your local credential

### 12.11.1 Synopsis

Log out of Pachyderm by deleting your local credential. Note that it's not necessary to log out before logging in with another account (simply run 'pachctl auth login' twice) but 'logout' can be useful on shared workstations.

```
pachctl auth logout
```

### 12.11.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.12 pachctl auth modify-admins

Modify the current cluster admins

### 12.12.1 Synopsis

Modify the current cluster admins. `--add` accepts a comma-separated list of users to grant admin status, and `--remove` accepts a comma-separated list of users to revoke admin status

```
pachctl auth modify-admins
```

### 12.12.2 Options

<code>--add strings</code>	Comma-separated <code>list</code> of users to grant admin status
<code>--remove strings</code>	Comma-separated <code>list</code> of users revoke admin status

### 12.12.3 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.13 pachctl auth set-config

Set Pachyderm's current auth configuration

### 12.13.1 Synopsis

Set Pachyderm's current auth configuration

```
pachctl auth set-config
```

### 12.13.2 Options

<code>-f, --file string</code>	<code>input</code> file (to use <code>as</code> the new config (default <code>"-"</code> ))
--------------------------------	---

### 12.13.3 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs



## 12.14 pachctl auth set

Set the scope of access that ‘username’ has to ‘repo’

### 12.14.1 Synopsis

Set the scope of access that ‘username’ has to ‘repo’. For example, ‘pachctl auth set github-alice none private-data’ prevents “github-alice” from interacting with the “private-data” repo in any way (the default). Similarly, ‘pachctl auth set github-alice reader private-data’ would let “github-alice” read from “private-data” but not create commits (writer) or modify the repo’s access permissions (owner). Currently all Pachyderm authentication uses GitHub OAuth, so ‘username’ must be a GitHub username

```
pachctl auth set <username> (none|reader|writer|owner) <repo>
```

### 12.14.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.15 pachctl auth use-auth-token

Read a Pachyderm auth token from stdin, and write it to the current user’s Pachyderm config file

### 12.15.1 Synopsis

Read a Pachyderm auth token from stdin, and write it to the current user’s Pachyderm config file

```
pachctl auth use-auth-token
```

### 12.15.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.16 pachctl auth whoami

Print your Pachyderm identity

### 12.16.1 Synopsis

Print your Pachyderm identity.

```
pachctl auth whoami
```

## 12.16.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.17 pachctl completion

Print or install the bash completion code.

### 12.17.1 Synopsis

Print or install the bash completion code. This should be placed as the file `pachctl` in the bash completion directory (by default this is `/etc/bash_completion.d`. If bash-completion was installed via homebrew, this would be `$(brew --prefix)/etc/bash_completion.d`.)

```
pachctl completion
```

### 12.17.2 Options

```
--install          Install the completion.  
--path /etc/bash_completion.d/ Path to install the completion to. This will_  
↪ default to /etc/bash_completion.d/ if unspecified. (default "/etc/bash_completion.d/  
↪ pachctl")
```

### 12.17.3 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.18 pachctl config

Manages the pachyderm config.

### 12.18.1 Synopsis

Gets/sets pachyderm config values.

### 12.18.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.19 pachctl config delete

Commands for deleting pachyderm config values

### 12.19.1 Synopsis

Commands for deleting pachyderm config values

### 12.19.2 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.20 pachctl config delete context

Deletes a context.

### 12.20.1 Synopsis

Deletes a context.

```
pachctl config delete context
```

### 12.20.2 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.21 pachctl config get

Commands for getting pachyderm config values

### 12.21.1 Synopsis

Commands for getting pachyderm config values

### 12.21.2 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.22 pachctl config get active-context

Gets the currently active context.

### 12.22.1 Synopsis

Gets the currently active context.

```
pachctl config get active-context
```

### 12.22.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.23 pachctl config get context

Gets a context.

### 12.23.1 Synopsis

Gets the config of a context by its name.

```
pachctl config get context
```

### 12.23.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.24 pachctl config get metrics

Gets whether metrics are enabled.

### 12.24.1 Synopsis

Gets whether metrics are enabled.

```
pachctl config get metrics
```

## 12.24.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.25 pachctl config list

Commands for listing pachyderm config values

### 12.25.1 Synopsis

Commands for listing pachyderm config values

## 12.25.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.26 pachctl config list context

Lists contexts.

### 12.26.1 Synopsis

Lists contexts.

```
pachctl config list context
```

## 12.26.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.27 pachctl config set

Commands for setting pachyderm config values

### 12.27.1 Synopsis

Commands for setting pachyderm config values

## 12.27.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose Output verbose logs
```

## 12.28 pachctl config set active-context

Sets the currently active context.

### 12.28.1 Synopsis

Sets the currently active context.

```
pachctl config set active-context
```

## 12.28.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose Output verbose logs
```

## 12.29 pachctl config set context

Set a context.

### 12.29.1 Synopsis

Set a context config from a given name and either JSON stdin, or a given kubernetes context.

```
pachctl config set context
```

### 12.29.2 Options

```
-k, --kubernetes string  Import a given kubernetes context's values into the  
↪Pachyderm context.  
--overwrite              Overwrite a context if it already exists.
```

## 12.29.3 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose Output verbose logs
```

## 12.30 pachctl config set metrics

Sets whether metrics are enabled.

### 12.30.1 Synopsis

Sets whether metrics are enabled.

```
pachctl config set metrics
```

### 12.30.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.31 pachctl config update

Commands for updating pachyderm config values

### 12.31.1 Synopsis

Commands for updating pachyderm config values

### 12.31.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.32 pachctl config update context

Updates a context.

### 12.32.1 Synopsis

Updates an existing context config from a given name (or the currently-active context, if no name is given).

```
pachctl config update context [context]
```

### 12.32.2 Options

```
--auth-info string      Set a new k8s auth info.
--cluster-name string   Set a new cluster name.
--namespace string      Set a new namespace.
--pachd-address string   Set a new name pachd address.
--server-cas string     Set new trusted CA certs.
```

### 12.32.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.33 pachctl copy

Copy a Pachyderm resource.

### 12.33.1 Synopsis

Copy a Pachyderm resource.

### 12.33.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.34 pachctl copy file

Copy files between pfs paths.

### 12.34.1 Synopsis

Copy files between pfs paths.

```
pachctl copy file <src-repo>@<src-branch-or-commit>:<src-path> <dst-repo>@<dst-branch-  
↪or-commit>:<dst-path>
```

### 12.34.2 Options

```
-o, --overwrite  Overwrite the existing content of the file, either from previous   
↪commits or previous calls to 'put file' within this commit.
```



### 12.34.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.35 pachctl create

Create a new instance of a Pachyderm resource.

### 12.35.1 Synopsis

Create a new instance of a Pachyderm resource.

### 12.35.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.36 pachctl create branch

Create a new branch, or update an existing branch, on a repo.

### 12.36.1 Synopsis

Create a new branch, or update an existing branch, on a repo, starting a commit on the branch will also create it, so there's often no need to call this.

```
pachctl create branch <repo>@<branch-or-commit>
```

### 12.36.2 Options

```
--head string      The head of the newly created branch.
-p, --provenance []string  The provenance for the branch. format: <repo>@<branch-
↳ or-commit> (default [])
```

### 12.36.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.37 pachctl create pipeline

Create a new pipeline.

### 12.37.1 Synopsis

Create a new pipeline from a pipeline specification. For details on the format, see [http://docs.pachyderm.io/en/latest/reference/pipeline\\_spec.html](http://docs.pachyderm.io/en/latest/reference/pipeline_spec.html).

```
pachctl create pipeline
```

### 12.37.2 Options

```
-b, --build                If true, build and push local docker images into the docker_
↪registry.
-f, --file string          The JSON file containing the pipeline, it can be a url or
↪local file. - reads from stdin. (default "-")
-p, --push-images          If true, push local docker images into the docker registry.
-r, --registry string      The registry to push images to. (default "docker.io")
-u, --username string      The username to push images as, defaults to your docker_
↪username.
```

### 12.37.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.38 pachctl create repo

Create a new repo.

### 12.38.1 Synopsis

Create a new repo.

```
pachctl create repo <repo>
```

### 12.38.2 Options

```
-d, --description string  A description of the repo.
```

### 12.38.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.39 pachctl debug

Debug commands for analyzing a running cluster.

### 12.39.1 Synopsis

Debug commands for analyzing a running cluster.

### 12.39.2 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.40 pachctl debug binary

Return the binary the server is running.

### 12.40.1 Synopsis

Return the binary the server is running.

```
pachctl debug binary
```

### 12.40.2 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.41 pachctl debug dump

Return a dump of running goroutines.

### 12.41.1 Synopsis

Return a dump of running goroutines.

```
pachctl debug dump
```

### 12.41.2 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.42 pachctl debug pprof

Analyze a profile of pachd in pprof.

### 12.42.1 Synopsis

Analyze a profile of pachd in pprof.

```
pachctl debug pprof <profile>
```

### 12.42.2 Options

<code>--binary-file string</code>	File to write the binary to. (default <code>"binary"</code> )
<code>-d, --duration duration</code>	Duration to run a CPU profile <b>for</b> . (default <code>1m0s</code> )
<code>--profile-file string</code>	File to write the profile to. (default <code>"profile"</code> )

### 12.42.3 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.43 pachctl debug profile

Return a profile from the server.

### 12.43.1 Synopsis

Return a profile from the server.

```
pachctl debug profile <profile>
```

### 12.43.2 Options

<code>-d, --duration duration</code>	Duration to run a CPU profile <b>for</b> . (default <code>1m0s</code> )
--------------------------------------	---

### 12.43.3 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.44 pachctl delete

Delete an existing Pachyderm resource.

### 12.44.1 Synopsis

Delete an existing Pachyderm resource.

### 12.44.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.45 pachctl delete all

Delete everything.

### 12.45.1 Synopsis

Delete all repos, commits, files, pipelines and jobs. This resets the cluster to its initial state.

```
pachctl delete all
```

### 12.45.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.46 pachctl delete branch

Delete a branch

### 12.46.1 Synopsis

Delete a branch, while leaving the commits intact

```
pachctl delete branch <repo>@<branch-or-commit>
```

### 12.46.2 Options

```
-f, --force  remove the branch regardless of errors; use with care
```

### 12.46.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.47 pachctl delete commit

Delete an input commit.

### 12.47.1 Synopsis

Delete an input commit. An input is a commit which is not the output of a pipeline.

```
pachctl delete commit <repo>@<branch-or-commit>
```

### 12.47.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.48 pachctl delete file

Delete a file.

### 12.48.1 Synopsis

Delete a file.

```
pachctl delete file <repo>@<branch-or-commit>:<path/in/pfs>
```

### 12.48.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.49 pachctl delete job

Delete a job.

### 12.49.1 Synopsis

Delete a job.

```
pachctl delete job <job>
```

## 12.49.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.50 pachctl delete pipeline

Delete a pipeline.

### 12.50.1 Synopsis

Delete a pipeline.

```
pachctl delete pipeline (<pipeline>|--all)
```

### 12.50.2 Options

```
--all      delete all pipelines
-f, --force delete the pipeline regardless of errors; use with care
```

### 12.50.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.51 pachctl delete repo

Delete a repo.

### 12.51.1 Synopsis

Delete a repo.

```
pachctl delete repo <repo>
```

### 12.51.2 Options

```
--all      remove all repos
-f, --force remove the repo regardless of errors; use with care
```

### 12.51.3 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.52 pachctl delete transaction

Cancel and delete an existing transaction.

### 12.52.1 Synopsis

Cancel and delete an existing transaction.

```
pachctl delete transaction [<transaction>]
```

### 12.52.2 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.53 pachctl deploy

Deploy a Pachyderm cluster.

### 12.53.1 Synopsis

Deploy a Pachyderm cluster.

### 12.53.2 Options

<code>--block-cache-size string</code>	Size of pachd's in-memory cache for PFS files. ↪ Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
<code>-c, --context string</code>	Name of the context to add to the pachyderm config.
<code>--dash-image string</code>	Image URL for pachyderm dashboard
<code>--dashboard-only</code>	Only deploy the Pachyderm UI (experimental), ↪ without the rest of pachyderm. This is for launching the UI adjacent to an existing ↪ Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
<code>--dry-run</code>	Don't actually deploy pachyderm to Kubernetes, ↪ instead just print the manifest.
<code>--dynamic-etcd-nodes int</code>	Deploy etcd as a StatefulSet with the given ↪ number of pods. The persistent volumes used by these pods are provisioned ↪ dynamically. Note that StatefulSet is currently a beta kubernetes feature, which ↪ might be unavailable in older versions of kubernetes.
<code>--etcd-cpu-request string</code>	(rarely set) The size of etcd's CPU request, ↪ which we give to Kubernetes. Size is in cores (with partial cores allowed and ↪ encouraged).



```

--etcd-memory-request string      (rarely set) The size of etcd's memory request.
↪Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--etcd-storage-class string       If set, the name of an existing StorageClass to
↪use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api               If set, instruct pachd to serve its object/
↪block API on its public port (not safe with auth enabled, do not set in production).
--image-pull-secret string        A secret in Kubernetes that's needed to pull
↪from your private registry.
--local-roles                     Use namespace-local roles instead of cluster
↪roles. Ignored if --no-rbac is set.
--log-level string               The level of log messages to print options are,
↪from least to most verbose: "error", "info", "debug". (default "info")
--namespace string               Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag              (feature flag) Do not set, used for testing
--no-dashboard                    Don't deploy the Pachyderm UI alongside
↪Pachyderm (experimental).
--no-expose-docker-socket         Don't expose the Docker socket to worker
↪containers. This limits the privileges of workers which prevents them from
↪automatically setting the container's working dir and user.
--no-guaranteed                  Don't use guaranteed QoS for etcd and pachd
↪deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
↪local clusters (such as a on Minikube), it should normally be used for production
↪clusters.
--no-rbac                        Don't deploy RBAC roles for Pachyderm. (for k8s
↪versions prior to 1.8)
-o, --output string              Output format. One of: json|yaml (default "json
↪")
--pachd-cpu-request string        (rarely set) The size of Pachd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--pachd-memory-request string     (rarely set) The size of Pachd's memory request
↪in addition to its block cache (set via --block-cache-size). Size is in bytes, with
↪SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string                The registry to pull images from.
--shards int                     (rarely set) The maximum number of pachd nodes
↪allowed in the cluster; increasing this number blindly can result in degraded
↪performance. (default 16)
--static-etcd-volume string       Deploy etcd as a ReplicationController with one
↪pod. The pod uses the given persistent volume.
--tls string                      string of the form "<cert path>,<key path>" of
↪the signed TLS certificate and private key that Pachd should use for TLS
↪authentication (enables TLS-encrypted communication with Pachd)

```

### 12.53.3 Options inherited from parent commands

```

--no-color    Turn off colors.
-v, --verbose Output verbose logs

```

## 12.54 pachctl deploy amazon

Deploy a Pachyderm cluster running on AWS.

## 12.54.1 Synopsis

Deploy a Pachyderm cluster running on AWS. <bucket-name>: An S3 bucket where Pachyderm will store PFS data.  
: The AWS region where Pachyderm is being deployed (e.g. us-west-1) <disk-size>: Size of EBS volumes, in GB (assumed to all be the same).

```
pachctl deploy amazon <bucket-name> <region> <disk-size>
```

## 12.54.2 Options

```
--cloudfront-distribution string    Deploying on AWS with cloudfront is
↳ currently an alpha feature. No security restrictions have been applied to
↳ cloudfront, making all data public (obscured but not secured)
--credentials string                Use the format "<id>,<secret>[,<token>]".
↳ You can get a token by running "aws sts get-session-token".
--iam-role string                   Use the given IAM role for authorization, as
↳ opposed to using static credentials. The given role will be applied as the
↳ annotation iam.amazonaws.com/role, this used with a Kubernetes IAM role management
↳ system such as kube2iam allows you to give pachd credentials in a more secure way.
--max-upload-parts int              (rarely set) Set a custom maximum number of
↳ upload parts. (default 10000)
--part-size int                     (rarely set) Set a custom part size for
↳ object storage uploads. (default 5242880)
--retries int                       (rarely set) Set a custom number of retries
↳ for object storage requests. (default 10)
--reverse                           (rarely set) Reverse object storage paths.
↳ (default true)
--timeout string                    (rarely set) Set a custom timeout for object
↳ storage requests. (default "5m")
--upload-acl string                  (rarely set) Set a custom upload ACL for
↳ object storage uploads. (default "bucket-owner-full-control")
--vault string                       Use the format "<address/hostport>,<role>,"
↳ <token>".
```

## 12.54.3 Options inherited from parent commands

```
--block-cache-size string           Size of pachd's in-memory cache for PFS files.
↳ Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
-c, --context string                 Name of the context to add to the pachyderm
↳ config.
--dash-image string                  Image URL for pachyderm dashboard
--dashboard-only                     Only deploy the Pachyderm UI (experimental),
↳ without the rest of pachyderm. This is for launching the UI adjacent to an existing
↳ Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
--dry-run                             Don't actually deploy pachyderm to Kubernetes,
↳ instead just print the manifest.
--dynamic-etcd-nodes int              Deploy etcd as a StatefulSet with the given
↳ number of pods. The persistent volumes used by these pods are provisioned
↳ dynamically. Note that StatefulSet is currently a beta kubernetes feature, which
↳ might be unavailable in older versions of kubernetes.
--etcd-cpu-request string             (rarely set) The size of etcd's CPU request,
↳ which we give to Kubernetes. Size is in cores (with partial cores allowed and
↳ encouraged).
--etcd-memory-request string          (rarely set) The size of etcd's memory request.
↳ Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
```

```

--etcd-storage-class string      If set, the name of an existing StorageClass to
↪ use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api             If set, instruct pachd to serve its object/
↪ block API on its public port (not safe with auth enabled, do not set in production).
--image-pull-secret string      A secret in Kubernetes that's needed to pull
↪ from your private registry.
--local-roles                   Use namespace-local roles instead of cluster
↪ roles. Ignored if --no-rbac is set.
--log-level string              The level of log messages to print options are,
↪ from least to most verbose: "error", "info", "debug". (default "info")
--namespace string              Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag            (feature flag) Do not set, used for testing
--no-color                      Turn off colors.
--no-dashboard                  Don't deploy the Pachyderm UI alongside
↪ Pachyderm (experimental).
--no-expose-docker-socket       Don't expose the Docker socket to worker
↪ containers. This limits the privileges of workers which prevents them from
↪ automatically setting the container's working dir and user.
--no-guaranteed                 Don't use guaranteed QoS for etcd and pachd
↪ deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
↪ local clusters (such as a on Minikube), it should normally be used for production
↪ clusters.
--no-rbac                       Don't deploy RBAC roles for Pachyderm. (for k8s
↪ versions prior to 1.8)
-o, --output string             Output format. One of: json|yaml (default "json
↪ ")
--pachd-cpu-request string      (rarely set) The size of Pachd's CPU request,
↪ which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪ encouraged).
--pachd-memory-request string   (rarely set) The size of Pachd's memory request,
↪ in addition to its block cache (set via --block-cache-size). Size is in bytes, with
↪ SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string               The registry to pull images from.
--shards int                    (rarely set) The maximum number of pachd nodes
↪ allowed in the cluster; increasing this number blindly can result in degraded
↪ performance. (default 16)
--static-etcd-volume string      Deploy etcd as a ReplicationController with one
↪ pod. The pod uses the given persistent volume.
--tls string                    string of the form "<cert path>,<key path>" of
↪ the signed TLS certificate and private key that Pachd should use for TLS
↪ authentication (enables TLS-encrypted communication with Pachd)
-v, --verbose                   Output verbose logs

```

## 12.55 pachctl deploy custom

Deploy a custom Pachyderm cluster configuration

### 12.55.1 Synopsis

Deploy a custom Pachyderm cluster configuration. If is “s3“, then the arguments are: <size of volumes (in GB)>

```

pachctl deploy custom --persistent-disk <persistent disk backend> --object-store
↪ <object store backend> <persistent disk args> <object store args>

```

## 12.55.2 Options

```

--isS3V2                Enable S3V2 client
--max-upload-parts int   (rarely set / S3V2 incompatible) Set a custom
↪maximum number of upload parts. (default 10000)
--object-store string    (required) Backend providing an object-storage API
↪to pachyderm. One of: s3, gcs, or azure-blob. (default "s3")
--part-size int          (rarely set / S3V2 incompatible) Set a custom part
↪size for object storage uploads. (default 5242880)
--persistent-disk string (required) Backend providing persistent local
↪volumes to stateful pods. One of: aws, google, or azure. (default "aws")
--retries int            (rarely set / S3V2 incompatible) Set a custom number
↪of retries for object storage requests. (default 10)
--reverse                (rarely set) Reverse object storage paths. (default
↪true)
-s, --secure             Enable secure access to a Minio server.
--timeout string         (rarely set / S3V2 incompatible) Set a custom
↪timeout for object storage requests. (default "5m")
--upload-acl string      (rarely set / S3V2 incompatible) Set a custom upload
↪ACL for object storage uploads. (default "bucket-owner-full-control")

```

## 12.55.3 Options inherited from parent commands

```

--block-cache-size string    Size of pachd's in-memory cache for PFS files.
↪Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
-c, --context string         Name of the context to add to the pachyderm
↪config.
--dash-image string          Image URL for pachyderm dashboard
--dashboard-only             Only deploy the Pachyderm UI (experimental),
↪without the rest of pachyderm. This is for launching the UI adjacent to an existing
↪Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
--dry-run                    Don't actually deploy pachyderm to Kubernetes,
↪instead just print the manifest.
--dynamic-etcd-nodes int     Deploy etcd as a StatefulSet with the given
↪number of pods. The persistent volumes used by these pods are provisioned
↪dynamically. Note that StatefulSet is currently a beta kubernetes feature, which
↪might be unavailable in older versions of kubernetes.
--etcd-cpu-request string    (rarely set) The size of etcd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--etcd-memory-request string (rarely set) The size of etcd's memory request.
↪Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--etcd-storage-class string  If set, the name of an existing StorageClass to
↪use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api          If set, instruct pachd to serve its object/
↪block API on its public port (not safe with auth enabled, do not set in production).
--image-pull-secret string   A secret in Kubernetes that's needed to pull
↪from your private registry.
--local-roles                Use namespace-local roles instead of cluster
↪roles. Ignored if --no-rbac is set.
--log-level string           The level of log messages to print options are,
↪from least to most verbose: "error", "info", "debug". (default "info")
--namespace string           Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag         (feature flag) Do not set, used for testing
--no-color                   Turn off colors.
--no-dashboard               Don't deploy the Pachyderm UI alongside
↪Pachyderm (experimental).

```

```

--no-expose-docker-socket      Don't expose the Docker socket to worker
↪containers. This limits the privileges of workers which prevents them from
↪automatically setting the container's working dir and user.
--no-guaranteed                Don't use guaranteed QoS for etcd and pachd
↪deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
↪local clusters (such as a on Minikube), it should normally be used for production
↪clusters.
--no-rbac                      Don't deploy RBAC roles for Pachyderm. (for k8s
↪versions prior to 1.8)
-o, --output string            Output format. One of: json|yaml (default "json
↪")
--pachd-cpu-request string      (rarely set) The size of Pachd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--pachd-memory-request string   (rarely set) The size of PachD's memory request
↪in addition to its block cache (set via --block-cache-size). Size is in bytes, with
↪SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string              The registry to pull images from.
--shards int                    (rarely set) The maximum number of pachd nodes
↪allowed in the cluster; increasing this number blindly can result in degraded
↪performance. (default 16)
--static-etcd-volume string      Deploy etcd as a ReplicationController with one
↪pod. The pod uses the given persistent volume.
--tls string                    string of the form "<cert path>,<key path>" of
↪the signed TLS certificate and private key that Pachd should use for TLS
↪authentication (enables TLS-encrypted communication with Pachd)
-v, --verbose                  Output verbose logs

```

## 12.56 pachctl deploy export-images

Export a tarball (to stdout) containing all of the images in a deployment.

### 12.56.1 Synopsis

Export a tarball (to stdout) containing all of the images in a deployment.

```
pachctl deploy export-images <output-file>
```

### 12.56.2 Options inherited from parent commands

```

--block-cache-size string      Size of pachd's in-memory cache for PFS files.
↪Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
-c, --context string            Name of the context to add to the pachyderm
↪config.
--dash-image string             Image URL for pachyderm dashboard
--dashboard-only                Only deploy the Pachyderm UI (experimental),
↪without the rest of pachyderm. This is for launching the UI adjacent to an existing
↪Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
--dry-run                       Don't actually deploy pachyderm to Kubernetes,
↪instead just print the manifest.
--dynamic-etcd-nodes int        Deploy etcd as a StatefulSet with the given
↪number of pods. The persistent volumes used by these pods are provisioned
↪dynamically. Note that StatefulSet is currently a beta kubernetes feature, which
↪might be unavailable in older versions of kubernetes.

```

```

--etcd-cpu-request string      (rarely set) The size of etcd's CPU request,
↳which we give to Kubernetes. Size is in cores (with partial cores allowed and
↳encouraged).
--etcd-memory-request string   (rarely set) The size of etcd's memory request.
↳Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--etcd-storage-class string    If set, the name of an existing StorageClass to
↳use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api            If set, instruct pachd to serve its object/
↳block API on its public port (not safe with auth enabled, do not set in production).
--image-pull-secret string      A secret in Kubernetes that's needed to pull
↳from your private registry.
--local-roles                  Use namespace-local roles instead of cluster
↳roles. Ignored if --no-rbac is set.
--log-level string             The level of log messages to print options are,
↳from least to most verbose: "error", "info", "debug". (default "info")
--namespace string             Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag           (feature flag) Do not set, used for testing
--no-color                     Turn off colors.
--no-dashboard                 Don't deploy the Pachyderm UI alongside
↳Pachyderm (experimental).
--no-expose-docker-socket      Don't expose the Docker socket to worker
↳containers. This limits the privileges of workers which prevents them from
↳automatically setting the container's working dir and user.
--no-guaranteed                Don't use guaranteed QoS for etcd and pachd
↳deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
↳local clusters (such as a on Minikube), it should normally be used for production
↳clusters.
--no-rbac                      Don't deploy RBAC roles for Pachyderm. (for k8s
↳versions prior to 1.8)
-o, --output string            Output format. One of: json|yaml (default "json
↳")
--pachd-cpu-request string      (rarely set) The size of Pachd's CPU request,
↳which we give to Kubernetes. Size is in cores (with partial cores allowed and
↳encouraged).
--pachd-memory-request string   (rarely set) The size of PachD's memory request
↳in addition to its block cache (set via --block-cache-size). Size is in bytes, with
↳SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string              The registry to pull images from.
--shards int                   (rarely set) The maximum number of pachd nodes
↳allowed in the cluster; increasing this number blindly can result in degraded
↳performance. (default 16)
--static-etcd-volume string      Deploy etcd as a ReplicationController with one
↳pod. The pod uses the given persistent volume.
--tls string                   string of the form "<cert path>,<key path>" of
↳the signed TLS certificate and private key that Pachd should use for TLS
↳authentication (enables TLS-encrypted communication with Pachd)
-v, --verbose                  Output verbose logs

```

## 12.57 pachctl deploy google

Deploy a Pachyderm cluster running on Google Cloud Platform.

## 12.57.1 Synopsis

Deploy a Pachyderm cluster running on Google Cloud Platform. <bucket-name>: A Google Cloud Storage bucket where Pachyderm will store PFS data. <disk-size>: Size of Google Compute Engine persistent disks in GB (assumed to all be the same). <credentials-file>: A file containing the private key for the account (downloaded from Google Compute Engine).

```
pachctl deploy google <bucket-name> <disk-size> [<credentials-file>]
```

## 12.57.2 Options inherited from parent commands

```
--block-cache-size string      Size of pachd's in-memory cache for PFS files.
↪Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
-C, --context string          Name of the context to add to the pachyderm
↪config.
--dash-image string           Image URL for pachyderm dashboard
--dashboard-only              Only deploy the Pachyderm UI (experimental),
↪without the rest of pachyderm. This is for launching the UI adjacent to an existing
↪Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
--dry-run                     Don't actually deploy pachyderm to Kubernetes,
↪instead just print the manifest.
--dynamic-etcd-nodes int      Deploy etcd as a StatefulSet with the given
↪number of pods. The persistent volumes used by these pods are provisioned
↪dynamically. Note that StatefulSet is currently a beta kubernetes feature, which
↪might be unavailable in older versions of kubernetes.
--etcd-cpu-request string      (rarely set) The size of etcd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--etcd-memory-request string   (rarely set) The size of etcd's memory request.
↪Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--etcd-storage-class string    If set, the name of an existing StorageClass to
↪use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api           If set, instruct pachd to serve its object/
↪block API on its public port (not safe with auth enabled, do not set in production).
--image-pull-secret string     A secret in Kubernetes that's needed to pull
↪from your private registry.
--local-roles                 Use namespace-local roles instead of cluster
↪roles. Ignored if --no-rbac is set.
--log-level string            The level of log messages to print options are,
↪from least to most verbose: "error", "info", "debug". (default "info")
--namespace string            Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag          (feature flag) Do not set, used for testing
--no-color                    Turn off colors.
--no-dashboard                Don't deploy the Pachyderm UI alongside
↪Pachyderm (experimental).
--no-expose-docker-socket     Don't expose the Docker socket to worker
↪containers. This limits the privileges of workers which prevents them from
↪automatically setting the container's working dir and user.
--no-guaranteed               Don't use guaranteed QoS for etcd and pachd
↪deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
↪local clusters (such as a on Minikube), it should normally be used for production
↪clusters.
--no-rbac                     Don't deploy RBAC roles for Pachyderm. (for k8s
↪versions prior to 1.8)
-o, --output string           Output format. One of: json|yaml (default "json")
↪")
```

```

--pachd-cpu-request string      (rarely set) The size of Pachd's CPU request,
↳which we give to Kubernetes. Size is in cores (with partial cores allowed and
↳encouraged).
--pachd-memory-request string   (rarely set) The size of PachD's memory request
↳in addition to its block cache (set via --block-cache-size). Size is in bytes, with
↳SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string               The registry to pull images from.
--shards int                    (rarely set) The maximum number of pachd nodes
↳allowed in the cluster; increasing this number blindly can result in degraded
↳performance. (default 16)
--static-etcd-volume string     Deploy etcd as a ReplicationController with one
↳pod. The pod uses the given persistent volume.
--tls string                    string of the form "<cert path>,<key path>" of
↳the signed TLS certificate and private key that Pachd should use for TLS
↳authentication (enables TLS-encrypted communication with Pachd)
-v, --verbose                   Output verbose logs

```

## 12.58 pachctl deploy import-images

Import a tarball (from stdin) containing all of the images in a deployment and push them to a private registry.

### 12.58.1 Synopsis

Import a tarball (from stdin) containing all of the images in a deployment and push them to a private registry.

```
pachctl deploy import-images <input-file>
```

### 12.58.2 Options inherited from parent commands

```

--block-cache-size string       Size of pachd's in-memory cache for PFS files.
↳Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
-c, --context string           Name of the context to add to the pachyderm
↳config.
--dash-image string             Image URL for pachyderm dashboard
--dashboard-only                Only deploy the Pachyderm UI (experimental),
↳without the rest of pachyderm. This is for launching the UI adjacent to an existing
↳Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
--dry-run                       Don't actually deploy pachyderm to Kubernetes,
↳instead just print the manifest.
--dynamic-etcd-nodes int        Deploy etcd as a StatefulSet with the given
↳number of pods. The persistent volumes used by these pods are provisioned
↳dynamically. Note that StatefulSet is currently a beta kubernetes feature, which
↳might be unavailable in older versions of kubernetes.
--etcd-cpu-request string       (rarely set) The size of etcd's CPU request,
↳which we give to Kubernetes. Size is in cores (with partial cores allowed and
↳encouraged).
--etcd-memory-request string    (rarely set) The size of etcd's memory request.
↳Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--etcd-storage-class string     If set, the name of an existing StorageClass to
↳use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api             If set, instruct pachd to serve its object/
↳block API on its public port (not safe with auth enabled, do not set in production).

```



```

--image-pull-secret string      A secret in Kubernetes that's needed to pull
↪from your private registry.
--local-roles                  Use namespace-local roles instead of cluster
↪roles. Ignored if --no-rbac is set.
--log-level string             The level of log messages to print options are,
↪from least to most verbose: "error", "info", "debug". (default "info")
--namespace string            Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag           (feature flag) Do not set, used for testing
--no-color                     Turn off colors.
--no-dashboard                 Don't deploy the Pachyderm UI alongside
↪Pachyderm (experimental).
--no-expose-docker-socket      Don't expose the Docker socket to worker
↪containers. This limits the privileges of workers which prevents them from
↪automatically setting the container's working dir and user.
--no-guaranteed                Don't use guaranteed QoS for etcd and pachd
↪deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
↪local clusters (such as a on Minikube), it should normally be used for production
↪clusters.
--no-rbac                      Don't deploy RBAC roles for Pachyderm. (for k8s
↪versions prior to 1.8)
-o, --output string            Output format. One of: json|yaml (default "json
↪")
--pachd-cpu-request string      (rarely set) The size of Pachd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--pachd-memory-request string   (rarely set) The size of PachD's memory request
↪in addition to its block cache (set via --block-cache-size). Size is in bytes, with
↪SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string              The registry to pull images from.
--shards int                  (rarely set) The maximum number of pachd nodes
↪allowed in the cluster; increasing this number blindly can result in degraded
↪performance. (default 16)
--static-etcd-volume string     Deploy etcd as a ReplicationController with one
↪pod. The pod uses the given persistent volume.
--tls string                    string of the form "<cert path>,<key path>" of
↪the signed TLS certificate and private key that Pachd should use for TLS
↪authentication (enables TLS-encrypted communication with Pachd)
-v, --verbose                  Output verbose logs

```

## 12.59 pachctl deploy list-images

Output the list of images in a deployment.

### 12.59.1 Synopsis

Output the list of images in a deployment.

```
pachctl deploy list-images
```

## 12.59.2 Options inherited from parent commands

```

--block-cache-size string      Size of pachd's in-memory cache for PFS files.
↪Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
-c, --context string          Name of the context to add to the pachyderm
↪config.
--dash-image string           Image URL for pachyderm dashboard
--dashboard-only              Only deploy the Pachyderm UI (experimental),
↪without the rest of pachyderm. This is for launching the UI adjacent to an existing
↪Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
--dry-run                     Don't actually deploy pachyderm to Kubernetes,
↪instead just print the manifest.
--dynamic-etcd-nodes int      Deploy etcd as a StatefulSet with the given
↪number of pods. The persistent volumes used by these pods are provisioned
↪dynamically. Note that StatefulSet is currently a beta kubernetes feature, which
↪might be unavailable in older versions of kubernetes.
--etcd-cpu-request string      (rarely set) The size of etcd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--etcd-memory-request string   (rarely set) The size of etcd's memory request.
↪Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--etcd-storage-class string    If set, the name of an existing StorageClass to
↪use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api           If set, instruct pachd to serve its object/
↪block API on its public port (not safe with auth enabled, do not set in production).
--image-pull-secret string     A secret in Kubernetes that's needed to pull
↪from your private registry.
--local-roles                 Use namespace-local roles instead of cluster
↪roles. Ignored if --no-rbac is set.
--log-level string            The level of log messages to print options are,
↪from least to most verbose: "error", "info", "debug". (default "info")
--namespace string            Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag          (feature flag) Do not set, used for testing
--no-color                    Turn off colors.
--no-dashboard                Don't deploy the Pachyderm UI alongside
↪Pachyderm (experimental).
--no-expose-docker-socket     Don't expose the Docker socket to worker
↪containers. This limits the privileges of workers which prevents them from
↪automatically setting the container's working dir and user.
--no-guaranteed               Don't use guaranteed QoS for etcd and pachd
↪deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
↪local clusters (such as a on Minikube), it should normally be used for production
↪clusters.
--no-rbac                     Don't deploy RBAC roles for Pachyderm. (for k8s
↪versions prior to 1.8)
-o, --output string           Output format. One of: json|yaml (default "json
↪")
--pachd-cpu-request string     (rarely set) The size of Pachd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--pachd-memory-request string  (rarely set) The size of Pachd's memory request
↪in addition to its block cache (set via --block-cache-size). Size is in bytes, with
↪SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string             The registry to pull images from.
--shards int                  (rarely set) The maximum number of pachd nodes
↪allowed in the cluster; increasing this number blindly can result in degraded
↪performance. (default 16)
--static-etcd-volume string    Deploy etcd as a ReplicationController with one
↪pod. The pod uses the given persistent volume.

```

```

--tls string          string of the form "<cert path>,<key path>" of
↳the signed TLS certificate and private key that Pachd should use for TLS
↳authentication (enables TLS-encrypted communication with Pachd)
-v, --verbose        Output verbose logs

```

## 12.60 pachctl deploy local

Deploy a single-node Pachyderm cluster with local metadata storage.

### 12.60.1 Synopsis

Deploy a single-node Pachyderm cluster with local metadata storage.

```
pachctl deploy local
```

### 12.60.2 Options

```

-d, --dev            Deploy pachd with local version tags, disable metrics,
↳expose Pachyderm's object/block API, and use an insecure authentication mechanism
↳(do not set on any cluster with sensitive data)
--host-path string  Location on the host machine where PFS metadata will be
↳stored. (default "/var/pachyderm")

```

### 12.60.3 Options inherited from parent commands

```

--block-cache-size string  Size of pachd's in-memory cache for PFS files.
↳Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
-c, --context string      Name of the context to add to the pachyderm
↳config.
--dash-image string       Image URL for pachyderm dashboard
--dashboard-only          Only deploy the Pachyderm UI (experimental),
↳without the rest of pachyderm. This is for launching the UI adjacent to an existing
↳Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
--dry-run                Don't actually deploy pachyderm to Kubernetes,
↳instead just print the manifest.
--dynamic-etcd-nodes int  Deploy etcd as a StatefulSet with the given
↳number of pods. The persistent volumes used by these pods are provisioned
↳dynamically. Note that StatefulSet is currently a beta kubernetes feature, which
↳might be unavailable in older versions of kubernetes.
--etcd-cpu-request string (rarely set) The size of etcd's CPU request,
↳which we give to Kubernetes. Size is in cores (with partial cores allowed and
↳encouraged).
--etcd-memory-request string (rarely set) The size of etcd's memory request.
↳Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--etcd-storage-class string If set, the name of an existing StorageClass to
↳use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api       If set, instruct pachd to serve its object/
↳block API on its public port (not safe with auth enabled, do not set in production).
--image-pull-secret string A secret in Kubernetes that's needed to pull
↳from your private registry.

```

```

--local-roles                Use namespace-local roles instead of cluster_
↪roles. Ignored if --no-rbac is set.
--log-level string           The level of log messages to print options are,
↪from least to most verbose: "error", "info", "debug". (default "info")
--namespace string          Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag         (feature flag) Do not set, used for testing
--no-color                  Turn off colors.
--no-dashboard              Don't deploy the Pachyderm UI alongside
↪Pachyderm (experimental).
--no-expose-docker-socket    Don't expose the Docker socket to worker
↪containers. This limits the privileges of workers which prevents them from
↪automatically setting the container's working dir and user.
--no-guaranteed              Don't use guaranteed QoS for etcd and pachd
↪deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
↪local clusters (such as a on Minikube), it should normally be used for production
↪clusters.
--no-rbac                   Don't deploy RBAC roles for Pachyderm. (for k8s
↪versions prior to 1.8)
-o, --output string          Output format. One of: json|yaml (default "json
↪")
--pachd-cpu-request string   (rarely set) The size of Pachd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--pachd-memory-request string (rarely set) The size of PachD's memory request
↪in addition to its block cache (set via --block-cache-size). Size is in bytes, with
↪SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string           The registry to pull images from.
--shards int                (rarely set) The maximum number of pachd nodes
↪allowed in the cluster; increasing this number blindly can result in degraded
↪performance. (default 16)
--static-etcd-volume string  Deploy etcd as a ReplicationController with one
↪pod. The pod uses the given persistent volume.
--tls string                string of the form "<cert path>,<key path>" of
↪the signed TLS certificate and private key that Pachd should use for TLS
↪authentication (enables TLS-encrypted communication with Pachd)
-v, --verbose               Output verbose logs

```

## 12.61 pachctl deploy microsoft

Deploy a Pachyderm cluster running on Microsoft Azure.

### 12.61.1 Synopsis

Deploy a Pachyderm cluster running on Microsoft Azure. : An Azure container where Pachyderm will store PFS data.  
 <disk-size>: Size of persistent volumes, in GB (assumed to all be the same).

```
pachctl deploy microsoft <container> <account-name> <account-key> <disk-size>
```

### 12.61.2 Options inherited from parent commands

```

--block-cache-size string      Size of pachd's in-memory cache for PFS files.
↪Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
-c, --context string          Name of the context to add to the pachyderm
↪config.
--dash-image string           Image URL for pachyderm dashboard
--dashboard-only              Only deploy the Pachyderm UI (experimental),
↪without the rest of pachyderm. This is for launching the UI adjacent to an existing
↪Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
--dry-run                     Don't actually deploy pachyderm to Kubernetes,
↪instead just print the manifest.
--dynamic-etcd-nodes int      Deploy etcd as a StatefulSet with the given
↪number of pods. The persistent volumes used by these pods are provisioned
↪dynamically. Note that StatefulSet is currently a beta kubernetes feature, which
↪might be unavailable in older versions of kubernetes.
--etcd-cpu-request string      (rarely set) The size of etcd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--etcd-memory-request string   (rarely set) The size of etcd's memory request.
↪Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--etcd-storage-class string    If set, the name of an existing StorageClass to
↪use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api           If set, instruct pachd to serve its object/
↪block API on its public port (not safe with auth enabled, do not set in production).
--image-pull-secret string     A secret in Kubernetes that's needed to pull
↪from your private registry.
--local-roles                 Use namespace-local roles instead of cluster
↪roles. Ignored if --no-rbac is set.
--log-level string            The level of log messages to print options are,
↪from least to most verbose: "error", "info", "debug". (default "info")
--namespace string            Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag          (feature flag) Do not set, used for testing
--no-color                    Turn off colors.
--no-dashboard                Don't deploy the Pachyderm UI alongside
↪Pachyderm (experimental).
--no-expose-docker-socket     Don't expose the Docker socket to worker
↪containers. This limits the privileges of workers which prevents them from
↪automatically setting the container's working dir and user.
--no-guaranteed               Don't use guaranteed QoS for etcd and pachd
↪deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
↪local clusters (such as a on Minikube), it should normally be used for production
↪clusters.
--no-rbac                     Don't deploy RBAC roles for Pachyderm. (for k8s
↪versions prior to 1.8)
-o, --output string           Output format. One of: json|yaml (default "json
↪")
--pachd-cpu-request string     (rarely set) The size of Pachd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--pachd-memory-request string  (rarely set) The size of Pachd's memory request
↪in addition to its block cache (set via --block-cache-size). Size is in bytes, with
↪SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string             The registry to pull images from.
--shards int                  (rarely set) The maximum number of pachd nodes
↪allowed in the cluster; increasing this number blindly can result in degraded
↪performance. (default 16)
--static-etcd-volume string    Deploy etcd as a ReplicationController with one
↪pod. The pod uses the given persistent volume.
--tls string                  string of the form "<cert path>,<key path>" of
↪the signed TLS certificate and private key that Pachd should use for TLS
↪authentication (enables TLS-encrypted communication with Pachd)

```

`-v, --verbose`

Output verbose logs

## 12.62 pachctl deploy storage

Deploy credentials for a particular storage provider.

### 12.62.1 Synopsis

Deploy credentials for a particular storage provider, so that Pachyderm can ingress data from and egress data to it.

### 12.62.2 Options inherited from parent commands

```

--block-cache-size string      Size of pachd's in-memory cache for PFS files.
↪Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
-C, --context string          Name of the context to add to the pachyderm
↪config.
--dash-image string           Image URL for pachyderm dashboard
--dashboard-only              Only deploy the Pachyderm UI (experimental),
↪without the rest of pachyderm. This is for launching the UI adjacent to an existing
↪Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
--dry-run                     Don't actually deploy pachyderm to Kubernetes,
↪instead just print the manifest.
--dynamic-etcd-nodes int      Deploy etcd as a StatefulSet with the given
↪number of pods. The persistent volumes used by these pods are provisioned
↪dynamically. Note that StatefulSet is currently a beta kubernetes feature, which
↪might be unavailable in older versions of kubernetes.
--etcd-cpu-request string     (rarely set) The size of etcd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--etcd-memory-request string  (rarely set) The size of etcd's memory request.
↪Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--etcd-storage-class string   If set, the name of an existing StorageClass to
↪use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api           If set, instruct pachd to serve its object/
↪block API on its public port (not safe with auth enabled, do not set in production).
--image-pull-secret string    A secret in Kubernetes that's needed to pull
↪from your private registry.
--local-roles                 Use namespace-local roles instead of cluster
↪roles. Ignored if --no-rbac is set.
--log-level string            The level of log messages to print options are,
↪from least to most verbose: "error", "info", "debug". (default "info")
--namespace string            Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag          (feature flag) Do not set, used for testing
--no-color                    Turn off colors.
--no-dashboard                Don't deploy the Pachyderm UI alongside
↪Pachyderm (experimental).
--no-expose-docker-socket     Don't expose the Docker socket to worker
↪containers. This limits the privileges of workers which prevents them from
↪automatically setting the container's working dir and user.
--no-guaranteed                Don't use guaranteed QoS for etcd and pachd
↪deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
↪local clusters (such as a on Minikube), it should normally be used for production
↪clusters.

```

```

--no-rbac                Don't deploy RBAC roles for Pachyderm. (for k8s
↪versions prior to 1.8)
-o, --output string      Output format. One of: json|yaml (default "json
↪")
--pachd-cpu-request string (rarely set) The size of Pachd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--pachd-memory-request string (rarely set) The size of Pachd's memory request,
↪in addition to its block cache (set via --block-cache-size). Size is in bytes, with
↪SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string        The registry to pull images from.
--shards int              (rarely set) The maximum number of pachd nodes
↪allowed in the cluster; increasing this number blindly can result in degraded
↪performance. (default 16)
--static-etcd-volume string Deploy etcd as a ReplicationController with one
↪pod. The pod uses the given persistent volume.
--tls string              string of the form "<cert path>,<key path>" of
↪the signed TLS certificate and private key that Pachd should use for TLS
↪authentication (enables TLS-encrypted communication with Pachd)
-v, --verbose             Output verbose logs

```

## 12.63 pachctl deploy storage amazon

Deploy credentials for the Amazon S3 storage provider.

### 12.63.1 Synopsis

Deploy credentials for the Amazon S3 storage provider, so that Pachyderm can ingress data from and egress data to it.

```

pachctl deploy storage amazon <region> <access-key-id> <secret-access-key> [<session-
↪token>]

```

### 12.63.2 Options

```

--max-upload-parts int    (rarely set) Set a custom maximum number of upload
↪parts. (default 10000)
--part-size int           (rarely set) Set a custom part size for object storage
↪uploads. (default 5242880)
--retries int             (rarely set) Set a custom number of retries for object
↪storage requests. (default 10)
--reverse                 (rarely set) Reverse object storage paths. (default
↪true)
--timeout string          (rarely set) Set a custom timeout for object storage
↪requests. (default "5m")
--upload-acl string       (rarely set) Set a custom upload ACL for object
↪storage uploads. (default "bucket-owner-full-control")

```

### 12.63.3 Options inherited from parent commands

```

--block-cache-size string      Size of pachd's in-memory cache for PFS files.
↪Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
-c, --context string          Name of the context to add to the pachyderm
↪config.
--dash-image string           Image URL for pachyderm dashboard
--dashboard-only              Only deploy the Pachyderm UI (experimental),
↪without the rest of pachyderm. This is for launching the UI adjacent to an existing
↪Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
--dry-run                     Don't actually deploy pachyderm to Kubernetes,
↪instead just print the manifest.
--dynamic-etcd-nodes int      Deploy etcd as a StatefulSet with the given
↪number of pods. The persistent volumes used by these pods are provisioned
↪dynamically. Note that StatefulSet is currently a beta kubernetes feature, which
↪might be unavailable in older versions of kubernetes.
--etcd-cpu-request string      (rarely set) The size of etcd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--etcd-memory-request string   (rarely set) The size of etcd's memory request.
↪Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--etcd-storage-class string    If set, the name of an existing StorageClass to
↪use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api           If set, instruct pachd to serve its object/
↪block API on its public port (not safe with auth enabled, do not set in production).
--image-pull-secret string     A secret in Kubernetes that's needed to pull
↪from your private registry.
--local-roles                 Use namespace-local roles instead of cluster
↪roles. Ignored if --no-rbac is set.
--log-level string            The level of log messages to print options are,
↪from least to most verbose: "error", "info", "debug". (default "info")
--namespace string            Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag          (feature flag) Do not set, used for testing
--no-color                    Turn off colors.
--no-dashboard                Don't deploy the Pachyderm UI alongside
↪Pachyderm (experimental).
--no-expose-docker-socket     Don't expose the Docker socket to worker
↪containers. This limits the privileges of workers which prevents them from
↪automatically setting the container's working dir and user.
--no-guaranteed               Don't use guaranteed QoS for etcd and pachd
↪deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
↪local clusters (such as a on Minikube), it should normally be used for production
↪clusters.
--no-rbac                     Don't deploy RBAC roles for Pachyderm. (for k8s
↪versions prior to 1.8)
-o, --output string           Output format. One of: json|yaml (default "json
↪")
--pachd-cpu-request string     (rarely set) The size of Pachd's CPU request,
↪which we give to Kubernetes. Size is in cores (with partial cores allowed and
↪encouraged).
--pachd-memory-request string  (rarely set) The size of Pachd's memory request,
↪in addition to its block cache (set via --block-cache-size). Size is in bytes, with
↪SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string             The registry to pull images from.
--shards int                  (rarely set) The maximum number of pachd nodes
↪allowed in the cluster; increasing this number blindly can result in degraded
↪performance. (default 16)
--static-etcd-volume string    Deploy etcd as a ReplicationController with one
↪pod. The pod uses the given persistent volume.

```



```

--tls string                string of the form "<cert path>,<key path>" of
↳the signed TLS certificate and private key that Pachd should use for TLS
↳authentication (enables TLS-encrypted communication with Pachd)
-v, --verbose                Output verbose logs

```

## 12.64 pachctl deploy storage google

Deploy credentials for the Google Cloud storage provider.

### 12.64.1 Synopsis

Deploy credentials for the Google Cloud storage provider, so that Pachyderm can ingress data from and egress data to it.

```
pachctl deploy storage google <credentials-file>
```

### 12.64.2 Options inherited from parent commands

```

--block-cache-size string    Size of pachd's in-memory cache for PFS files.
↳Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
-c, --context string         Name of the context to add to the pachyderm
↳config.
--dash-image string          Image URL for pachyderm dashboard
--dashboard-only             Only deploy the Pachyderm UI (experimental),
↳without the rest of pachyderm. This is for launching the UI adjacent to an existing
↳Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
--dry-run                    Don't actually deploy pachyderm to Kubernetes,
↳instead just print the manifest.
--dynamic-etcd-nodes int     Deploy etcd as a StatefulSet with the given
↳number of pods. The persistent volumes used by these pods are provisioned
↳dynamically. Note that StatefulSet is currently a beta kubernetes feature, which
↳might be unavailable in older versions of kubernetes.
--etcd-cpu-request string    (rarely set) The size of etcd's CPU request,
↳which we give to Kubernetes. Size is in cores (with partial cores allowed and
↳encouraged).
--etcd-memory-request string (rarely set) The size of etcd's memory request.
↳Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--etcd-storage-class string  If set, the name of an existing StorageClass to
↳use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api          If set, instruct pachd to serve its object/
↳block API on its public port (not safe with auth enabled, do not set in production).
--image-pull-secret string   A secret in Kubernetes that's needed to pull
↳from your private registry.
--local-roles                 Use namespace-local roles instead of cluster
↳roles. Ignored if --no-rbac is set.
--log-level string           The level of log messages to print options are,
↳from least to most verbose: "error", "info", "debug". (default "info")
--namespace string           Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag         (feature flag) Do not set, used for testing
--no-color                   Turn off colors.
--no-dashboard               Don't deploy the Pachyderm UI alongside
↳Pachyderm (experimental).

```

```

--no-expose-docker-socket    Don't expose the Docker socket to worker
containers. This limits the privileges of workers which prevents them from
automatically setting the container's working dir and user.
--no-guaranteed              Don't use guaranteed QoS for etcd and pachd
deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
local clusters (such as a on Minikube), it should normally be used for production
clusters.
--no-rbac                    Don't deploy RBAC roles for Pachyderm. (for k8s
versions prior to 1.8)
-o, --output string          Output format. One of: json|yaml (default "json")
--pachd-cpu-request string   (rarely set) The size of Pachd's CPU request,
which we give to Kubernetes. Size is in cores (with partial cores allowed and
encouraged).
--pachd-memory-request string (rarely set) The size of PachD's memory request
in addition to its block cache (set via --block-cache-size). Size is in bytes, with
SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string            The registry to pull images from.
--shards int                 (rarely set) The maximum number of pachd nodes
allowed in the cluster; increasing this number blindly can result in degraded
performance. (default 16)
--static-etcd-volume string   Deploy etcd as a ReplicationController with one
pod. The pod uses the given persistent volume.
--tls string                  string of the form "<cert path>,<key path>" of
the signed TLS certificate and private key that Pachd should use for TLS
authentication (enables TLS-encrypted communication with Pachd)
-v, --verbose                Output verbose logs

```

## 12.65 pachctl deploy storage microsoft

Deploy credentials for the Azure storage provider.

### 12.65.1 Synopsis

Deploy credentials for the Azure storage provider, so that Pachyderm can ingress data from and egress data to it.

```
pachctl deploy storage microsoft <account-name> <account-key>
```

### 12.65.2 Options inherited from parent commands

```

--block-cache-size string    Size of pachd's in-memory cache for PFS files.
Size is specified in bytes, with allowed SI suffixes (M, K, G, Mi, Ki, Gi, etc).
-c, --context string         Name of the context to add to the pachyderm
config.
--dash-image string          Image URL for pachyderm dashboard
--dashboard-only             Only deploy the Pachyderm UI (experimental),
without the rest of pachyderm. This is for launching the UI adjacent to an existing
Pachyderm cluster. After deployment, run "pachctl port-forward" to connect
--dry-run                    Don't actually deploy pachyderm to Kubernetes,
instead just print the manifest.
--dynamic-etcd-nodes int     Deploy etcd as a StatefulSet with the given
number of pods. The persistent volumes used by these pods are provisioned
dynamically. Note that StatefulSet is currently a beta kubernetes feature, which
might be unavailable in older versions of kubernetes.

```

```

--etcd-cpu-request string      (rarely set) The size of etcd's CPU request,
↳which we give to Kubernetes. Size is in cores (with partial cores allowed and
↳encouraged).
--etcd-memory-request string   (rarely set) The size of etcd's memory request.
↳Size is in bytes, with SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--etcd-storage-class string    If set, the name of an existing StorageClass to
↳use for etcd storage. Ignored if --static-etcd-volume is set.
--expose-object-api            If set, instruct pachd to serve its object/
↳block API on its public port (not safe with auth enabled, do not set in production).
--image-pull-secret string      A secret in Kubernetes that's needed to pull
↳from your private registry.
--local-roles                  Use namespace-local roles instead of cluster
↳roles. Ignored if --no-rbac is set.
--log-level string             The level of log messages to print options are,
↳from least to most verbose: "error", "info", "debug". (default "info")
--namespace string             Kubernetes namespace to deploy Pachyderm to.
--new-hash-tree-flag           (feature flag) Do not set, used for testing
--no-color                     Turn off colors.
--no-dashboard                 Don't deploy the Pachyderm UI alongside
↳Pachyderm (experimental).
--no-expose-docker-socket      Don't expose the Docker socket to worker
↳containers. This limits the privileges of workers which prevents them from
↳automatically setting the container's working dir and user.
--no-guaranteed                Don't use guaranteed QoS for etcd and pachd
↳deployments. Turning this on (turning guaranteed QoS off) can lead to more stable
↳local clusters (such as a on Minikube), it should normally be used for production
↳clusters.
--no-rbac                      Don't deploy RBAC roles for Pachyderm. (for k8s
↳versions prior to 1.8)
-o, --output string            Output format. One of: json|yaml (default "json
↳")
--pachd-cpu-request string      (rarely set) The size of Pachd's CPU request,
↳which we give to Kubernetes. Size is in cores (with partial cores allowed and
↳encouraged).
--pachd-memory-request string   (rarely set) The size of PachD's memory request
↳in addition to its block cache (set via --block-cache-size). Size is in bytes, with
↳SI suffixes (M, K, G, Mi, Ki, Gi, etc).
--registry string              The registry to pull images from.
--shards int                   (rarely set) The maximum number of pachd nodes
↳allowed in the cluster; increasing this number blindly can result in degraded
↳performance. (default 16)
--static-etcd-volume string     Deploy etcd as a ReplicationController with one
↳pod. The pod uses the given persistent volume.
--tls string                    string of the form "<cert path>,<key path>" of
↳the signed TLS certificate and private key that Pachd should use for TLS
↳authentication (enables TLS-encrypted communication with Pachd)
-v, --verbose                  Output verbose logs

```

## 12.66 pachctl diff

Show the differences between two Pachyderm resources.

## 12.66.1 Synopsis

Show the differences between two Pachyderm resources.

## 12.66.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.67 pachctl diff file

Return a diff of two file trees.

### 12.67.1 Synopsis

Return a diff of two file trees.

```
pachctl diff file <new-repo>@<new-branch-or-commit>:<new-path> [<old-repo>@<old-branch-or-commit>:<old-path>]
```

### 12.67.2 Examples

```
# Return the diff of the file "path" of the repo "foo" between the head of the
# "master" branch and its parent.
$ pachctl diff file foo@master:path

# Return the diff between the master branches of repos foo and bar at paths
# path1 and path2, respectively.
$ pachctl diff file foo@master:path1 bar@master:path2
```

### 12.67.3 Options

```
--diff-command string  Use a program other than git to diff files.
--full-timestamps      Return absolute timestamps (as opposed to the default,r
↳relative timestamps).
--name-only             Show only the names of changed files.
--no-pager              Don't pipe output into a pager (i.e. less).
-s, --shallow          Don't descend into sub directories.
```

### 12.67.4 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.68 pachctl edit

Edit the value of an existing Pachyderm resource.

### 12.68.1 Synopsis

Edit the value of an existing Pachyderm resource.

### 12.68.2 Options inherited from parent commands

```
--no-color  Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.69 pachctl edit pipeline

Edit the manifest for a pipeline in your text editor.

### 12.69.1 Synopsis

Edit the manifest for a pipeline in your text editor.

```
pachctl edit pipeline <pipeline>
```

### 12.69.2 Options

```
--editor string  Editor to use for modifying the manifest.
--reprocess      If true, reprocess datums that were already processed by ↳
↳previous version of the pipeline.
```

### 12.69.3 Options inherited from parent commands

```
--no-color  Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.70 pachctl enterprise

Enterprise commands enable Pachyderm Enterprise features

### 12.70.1 Synopsis

Enterprise commands enable Pachyderm Enterprise features

## 12.70.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.71 pachctl enterprise activate

Activate the enterprise features of Pachyderm with an activation code

### 12.71.1 Synopsis

Activate the enterprise features of Pachyderm with an activation code

```
pachctl enterprise activate <activation-code>
```

### 12.71.2 Options

```
--expires string  A timestamp indicating when the token provided above should  
→expire (formatted as an RFC 3339/ISO 8601 datetime). This is only applied if it's  
→earlier than the signed expiration time encoded in 'activation-code', and therefore  
→is only useful for testing.
```

### 12.71.3 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.72 pachctl enterprise get-state

Check whether the Pachyderm cluster has enterprise features activated

### 12.72.1 Synopsis

Check whether the Pachyderm cluster has enterprise features activated

```
pachctl enterprise get-state
```

### 12.72.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.73 pachctl extract

Extract Pachyderm state to stdout or an object store bucket.

### 12.73.1 Synopsis

Extract Pachyderm state to stdout or an object store bucket.

```
pachctl extract
```

### 12.73.2 Examples

```
# Extract into a local file:
$ pachctl extract > backup

# Extract to s3:
$ pachctl extract -u s3://bucket/backup
```

### 12.73.3 Options

```
--no-objects    don't extract from object storage, only extract data from etcd
-u, --url string An object storage url (i.e. s3://...) to extract to.
```

### 12.73.4 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.74 pachctl extract pipeline

Return the manifest used to create a pipeline.

### 12.74.1 Synopsis

Return the manifest used to create a pipeline.

```
pachctl extract pipeline <pipeline>
```

### 12.74.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.75 pachctl finish

Finish a Pachyderm resource.

### 12.75.1 Synopsis

Finish a Pachyderm resource.

### 12.75.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose Output verbose logs
```

## 12.76 pachctl finish commit

Finish a started commit.

### 12.76.1 Synopsis

Finish a started commit. Commit-id must be a writeable commit.

```
pachctl finish commit <repo>@<branch-or-commit>
```

### 12.76.2 Options

```
--description string  A description of this commit's contents (synonym for --  
↪message)  
-m, --message string  A description of this commit's contents (overwrites any_  
↪existing commit description)
```

### 12.76.3 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose Output verbose logs
```

## 12.77 pachctl finish transaction

Execute and clear the currently active transaction.



### 12.77.1 Synopsis

Execute and clear the currently active transaction.

```
pachctl finish transaction [<transaction>]
```

### 12.77.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.78 pachctl flush

Wait for the side-effects of a Pachyderm resource to propagate.

### 12.78.1 Synopsis

Wait for the side-effects of a Pachyderm resource to propagate.

### 12.78.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.79 pachctl flush commit

Wait for all commits caused by the specified commits to finish and return them.

### 12.79.1 Synopsis

Wait for all commits caused by the specified commits to finish and return them.

```
pachctl flush commit <repo>@<branch-or-commit> ...
```

### 12.79.2 Examples

```
# return commits caused by foo@XXX and bar@YYY
$ pachctl flush commit foo@XXX bar@YYY

# return commits caused by foo@XXX leading to repos bar and baz
$ pachctl flush commit foo@XXX -r bar -r baz
```

### 12.79.3 Options

```
--full-timestamps    Return absolute timestamps (as opposed to the default,  
↳relative timestamps).  
--raw                disable pretty printing, print raw json  
-r, --repos []string  Wait only for commits leading to a specific set of repos,  
↳(default [])
```

### 12.79.4 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.80 pachctl flush job

Wait for all jobs caused by the specified commits to finish and return them.

### 12.80.1 Synopsis

Wait for all jobs caused by the specified commits to finish and return them.

```
pachctl flush job <repo>@<branch-or-commit> ...
```

### 12.80.2 Examples

```
# Return jobs caused by foo@XXX and bar@YYY.  
$ pachctl flush job foo@XXX bar@YYY  
  
# Return jobs caused by foo@XXX leading to pipelines bar and baz.  
$ pachctl flush job foo@XXX -p bar -p baz
```

### 12.80.3 Options

```
--full-timestamps    Return absolute timestamps (as opposed to the default,  
↳relative timestamps).  
-p, --pipeline []string  Wait only for jobs leading to a specific set of pipelines,  
↳(default [])  
--raw                disable pretty printing, print raw json
```

### 12.80.4 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.81 pachctl fsck

Run a file system consistency check on pfs.

### 12.81.1 Synopsis

Run a file system consistency check on the pachyderm file system, ensuring the correct provenance relationships are satisfied.

```
pachctl fsck
```

### 12.81.2 Options

```
-f, --fix    Attempt to fix as many issues as possible.
```

### 12.81.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.82 pachctl garbage-collect

Garbage collect unused data.

### 12.82.1 Synopsis

Garbage collect unused data.

When a file/commit/repo is deleted, the data is not immediately removed from the underlying storage system (e.g. S3) for performance and architectural reasons. This is similar to how when you delete a file on your computer, the file is not necessarily wiped from disk immediately.

To actually remove the data, you will need to manually invoke garbage collection with “pachctl garbage-collect”.

Currently “pachctl garbage-collect” can only be started when there are no pipelines running. You also need to ensure that there’s no ongoing “put file”. Garbage collection puts the cluster into a readonly mode where no new jobs can be created and no data can be added.

Pachyderm’s garbage collection uses bloom filters to index live objects. This means that some dead objects may erroneously not be deleted during garbage collection. The probability of this happening depends on how many objects you have; at around 10M objects it starts to become likely with the default values. To lower Pachyderm’s error rate and make garbage-collection more comprehensive, you can increase the amount of memory used for the bloom filters with the `--memory` flag. The default value is 10MB.

```
pachctl garbage-collect
```

## 12.82.2 Options

```
-m, --memory string  The amount of memory to use during garbage collection.
↳Default is 10MB. (default "0")
```

## 12.82.3 Options inherited from parent commands

```
--no-color  Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.83 pachctl get

Get the raw data represented by a Pachyderm resource.

### 12.83.1 Synopsis

Get the raw data represented by a Pachyderm resource.

### 12.83.2 Options inherited from parent commands

```
--no-color  Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.84 pachctl get file

Return the contents of a file.

### 12.84.1 Synopsis

Return the contents of a file.

```
pachctl get file <repo>@<branch-or-commit>:<path/in/pfs>
```

### 12.84.2 Examples

```
# get file "XXX" on branch "master" in repo "foo"
$ pachctl get file foo@master:XXX

# get file "XXX" in the parent of the current head of branch "master"
# in repo "foo"
$ pachctl get file foo@master^:XXX

# get file "XXX" in the grandparent of the current head of branch "master"
```

```
# in repo "foo"
$ pachctl get file foo@master^2:XXX
```

### 12.84.3 Options

```
-o, --output string      The path where data will be downloaded.
-p, --parallelism int    The maximum number of files that can be downloaded in_
↪parallel (default 10)
-r, --recursive          Recursively download a directory.
```

### 12.84.4 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.85 pachctl get object

Print the contents of an object.

### 12.85.1 Synopsis

Print the contents of an object.

```
pachctl get object <hash>
```

### 12.85.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.86 pachctl get tag

Print the contents of a tag.

### 12.86.1 Synopsis

Print the contents of a tag.

```
pachctl get tag <tag>
```

## 12.86.2 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.87 pachctl glob

Print a list of Pachyderm resources matching a glob pattern.

### 12.87.1 Synopsis

Print a list of Pachyderm resources matching a glob pattern.

### 12.87.2 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.88 pachctl glob file

Return files that match a glob pattern in a commit.

### 12.88.1 Synopsis

Return files that match a glob pattern in a commit (that is, match a glob pattern in a repo at the state represented by a commit). Glob patterns are documented [here](#).

```
pachctl glob file <repo>@<branch-or-commit>:<pattern>
```

### 12.88.2 Examples

```
# Return files in repo "foo" on branch "master" that start
# with the character "A". Note how the double quotation marks around the
# parameter are necessary because otherwise your shell might interpret the "*".
$ pachctl glob file "foo@master:A*"

# Return files in repo "foo" on branch "master" under directory "data".
$ pachctl glob file "foo@master:data/*"
```

### 12.88.3 Options

<code>--full-timestamps</code>	Return absolute timestamps ( <b>as</b> opposed to the default, <b>relative</b> timestamps).
<code>--raw</code>	disable pretty printing, <b>print</b> raw json

## 12.88.4 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.89 pachctl inspect

Show detailed information about a Pachyderm resource.

### 12.89.1 Synopsis

Show detailed information about a Pachyderm resource.

### 12.89.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.90 pachctl inspect cluster

Returns info about the pachyderm cluster

### 12.90.1 Synopsis

Returns info about the pachyderm cluster

```
pachctl inspect cluster
```

### 12.90.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.91 pachctl inspect commit

Return info about a commit.

### 12.91.1 Synopsis

Return info about a commit.

```
pachctl inspect commit <repo>@<branch-or-commit>
```

## 12.91.2 Options

```
--full-timestamps  Return absolute timestamps (as opposed to the default, relative timestamps).  
--raw              disable pretty printing, print raw json
```

## 12.91.3 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

# 12.92 pachctl inspect datum

Display detailed info about a single datum.

## 12.92.1 Synopsis

Display detailed info about a single datum. Requires the pipeline to have stats enabled.

```
pachctl inspect datum <job> <datum>
```

## 12.92.2 Options

```
--raw  disable pretty printing, print raw json
```

## 12.92.3 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

# 12.93 pachctl inspect file

Return info about a file.

## 12.93.1 Synopsis

Return info about a file.

```
pachctl inspect file <repo>@<branch-or-commit>:<path/in/pfs>
```



## 12.93.2 Options

```
--raw    disable pretty printing, print raw json
```

## 12.93.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

# 12.94 pachctl inspect job

Return info about a job.

## 12.94.1 Synopsis

Return info about a job.

```
pachctl inspect job <job>
```

## 12.94.2 Options

```
-b, --block          block until the job has either succeeded or failed
--full-timestamps    Return absolute timestamps (as opposed to the default, relative timestamps).
--raw                disable pretty printing, print raw json
```

## 12.94.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

# 12.95 pachctl inspect pipeline

Return info about a pipeline.

## 12.95.1 Synopsis

Return info about a pipeline.

```
pachctl inspect pipeline <pipeline>
```

## 12.95.2 Options

```
--full-timestamps  Return absolute timestamps (as opposed to the default, relative timestamps).  
-v, --raw          disable pretty printing, print raw json
```

## 12.95.3 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

# 12.96 pachctl inspect repo

Return info about a repo.

## 12.96.1 Synopsis

Return info about a repo.

```
pachctl inspect repo <repo>
```

## 12.96.2 Options

```
--full-timestamps  Return absolute timestamps (as opposed to the default, relative timestamps).  
-v, --raw          disable pretty printing, print raw json
```

## 12.96.3 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

# 12.97 pachctl inspect transaction

Print information about an open transaction.

## 12.97.1 Synopsis

Print information about an open transaction.

```
pachctl inspect transaction [<transaction>]
```

## 12.97.2 Options

```
--full-timestamps  Return absolute timestamps (as opposed to the default, relative timestamps).
--raw              disable pretty printing, print raw json
```

## 12.97.3 Options inherited from parent commands

```
--no-color  Turn off colors.
-v, --verbose  Output verbose logs
```

# 12.98 pachctl list

Print a list of Pachyderm resources of a specific type.

## 12.98.1 Synopsis

Print a list of Pachyderm resources of a specific type.

## 12.98.2 Options inherited from parent commands

```
--no-color  Turn off colors.
-v, --verbose  Output verbose logs
```

# 12.99 pachctl list branch

Return all branches on a repo.

## 12.99.1 Synopsis

Return all branches on a repo.

```
pachctl list branch <repo>
```

## 12.99.2 Options

```
--raw  disable pretty printing, print raw json
```

## 12.99.3 Options inherited from parent commands

```
--no-color  Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.100 pachctl list commit

Return all commits on a repo.

### 12.100.1 Synopsis

Return all commits on a repo.

```
pachctl list commit <repo>[@<branch>]
```

### 12.100.2 Examples

```
# return commits in repo "foo"
$ pachctl list commit foo

# return commits in repo "foo" on branch "master"
$ pachctl list commit foo@master

# return the last 20 commits in repo "foo" on branch "master"
$ pachctl list commit foo@master -n 20

# return commits in repo "foo" since commit XXX
$ pachctl list commit foo@master --from XXX
```

### 12.100.3 Options

<code>-f, --from string</code>	list all commits since this commit
<code>--full-timestamps</code>	Return absolute timestamps ( <b>as</b> opposed to the default, <code>relative timestamps</code> ).
<code>-n, --number int</code>	list only this many commits; <b>if</b> set to zero, list all.
<code>--raw</code>	disable pretty printing, <b>print</b> raw json

### 12.100.4 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.101 pachctl list datum

Return the datums in a job.

### 12.101.1 Synopsis

Return the datums in a job.

```
pachctl list datum <job>
```

## 12.101.2 Options

<code>--page int</code>	Specify the page of results to send
<code>--pageSize int</code>	Specify the number of results sent back <b>in</b> a single page
<code>--raw</code>	disable pretty printing, <b>print</b> raw json

## 12.101.3 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.102 pachctl list file

Return the files in a directory.

### 12.102.1 Synopsis

Return the files in a directory.

```
pachctl list file <repo>@<branch-or-commit>[:<path/in/pfs>]
```

### 12.102.2 Examples

```
# list top-level files on branch "master" in repo "foo"
$ pachctl list file foo@master

# list files under directory "dir" on branch "master" in repo "foo"
$ pachctl list file foo@master:dir

# list top-level files in the parent commit of the current head of "master"
# in repo "foo"
$ pachctl list file foo@master^

# list top-level files in the grandparent of the current head of "master"
# in repo "foo"
$ pachctl list file foo@master^2

# list the last n versions of top-level files on branch "master" in repo "foo"
$ pachctl list file foo@master --history n

# list all versions of top-level files on branch "master" in repo "foo"
$ pachctl list file foo@master --history all
```

### 12.102.3 Options

<code>--full-timestamps</code>	Return absolute timestamps ( <b>as</b> opposed to the default, <code>↪relative timestamps</code> ).
<code>--history string</code>	Return revision history <b>for</b> files. (default <code>"none"</code> )
<code>--raw</code>	disable pretty printing, <b>print</b> raw json

### 12.102.4 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.103 pachctl list job

Return info about jobs.

### 12.103.1 Synopsis

Return info about jobs.

```
pachctl list job
```

### 12.103.2 Examples

```
# Return all jobs
$ pachctl list job

# Return all jobs from the most recent version of pipeline "foo"
$ pachctl list job -p foo

# Return all jobs from all versions of pipeline "foo"
$ pachctl list job -p foo --history all

# Return all jobs whose input commits include foo@XXX and bar@YYY
$ pachctl list job -i foo@XXX -i bar@YYY

# Return all jobs in pipeline foo and whose input commits include bar@YYY
$ pachctl list job -p foo -i bar@YYY
```

### 12.103.3 Options

<code>--full-timestamps</code>	Return absolute timestamps ( <b>as</b> opposed to the default, <code>↪relative timestamps</code> ).
<code>--history string</code>	Return jobs <b>from historical</b> versions of pipelines. (default <code>↪"none"</code> )
<code>-i, --input strings</code>	List jobs <b>with</b> a specific <b>set</b> of <b>input</b> commits. <b>format:</b> <code>↪&lt;repo&gt;@&lt;branch-or-commit&gt;</code>

<code>--no-pager</code>	Don't pipe output into a pager (i.e. less).
<code>-o, --output string</code>	List jobs <b>with</b> a specific output commit. <b>format:</b> <repo>@<branch-or-commit>
<code>-p, --pipeline string</code>	Limit to jobs made by pipeline.
<code>--raw</code>	disable pretty printing, <b>print</b> raw json

## 12.103.4 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.104 pachctl list pipeline

Return info about all pipelines.

### 12.104.1 Synopsis

Return info about all pipelines.

```
pachctl list pipeline [<pipeline>]
```

### 12.104.2 Options

<code>--full-timestamps</code>	Return absolute timestamps ( <b>as</b> opposed to the default, <b>relative</b> timestamps).
<code>--history string</code>	Return revision history <b>for</b> pipelines. (default " <b>none</b> ")
<code>--raw</code>	disable pretty printing, <b>print</b> raw json
<code>-s, --spec</code>	Output ' <b>create pipeline</b> ' compatibility specs.

## 12.104.3 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.105 pachctl list repo

Return all repos.

### 12.105.1 Synopsis

Return all repos.

```
pachctl list repo
```

## 12.105.2 Options

```
--full-timestamps  Return absolute timestamps (as opposed to the default, relative timestamps).  
-v, --raw          disable pretty printing, print raw json
```

## 12.105.3 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

# 12.106 pachctl list transaction

List transactions.

## 12.106.1 Synopsis

List transactions.

```
pachctl list transaction
```

## 12.106.2 Options

```
--full-timestamps  Return absolute timestamps (as opposed to the default, relative timestamps).  
-v, --raw          disable pretty printing, print raw json
```

## 12.106.3 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

# 12.107 pachctl logs

Return logs from a job.

## 12.107.1 Synopsis

Return logs from a job.

```
pachctl logs [--pipeline=<pipeline>|--job=<job>] [--datum=<datum>]
```



## 12.107.2 Examples

```
# Return logs emitted by recent jobs in the "filter" pipeline
$ pachctl logs --pipeline=filter

# Return logs emitted by the job aedfa12aedef
$ pachctl logs --job=aedfa12aedef

# Return logs emitted by the pipeline \"filter\" while processing /apple.txt and a
↪file with the hash 123aef
$ pachctl logs --pipeline=filter --inputs=/apple.txt,123aef
```

## 12.107.3 Options

```
--datum string      Filter for log lines for this datum (accepts datum ID)
-f, --follow         Follow logs as more are created.
--inputs string      Filter for log lines generated while processing these files,
↪(accepts PFS paths or file hashes)
--job string         Filter for log lines from this job (accepts job ID)
--master            Return log messages from the master process (pipeline must
↪be set).
-p, --pipeline string Filter the log for lines from this pipeline (accepts
↪pipeline name)
--raw              Return log messages verbatim from server.
-t, --tail int      Lines of recent logs to display.
```

## 12.107.4 Options inherited from parent commands

```
--no-color  Turn off colors.
-v, --verbose Output verbose logs
```

# 12.108 pachctl mount

Mount pfs locally. This command blocks.

## 12.108.1 Synopsis

Mount pfs locally. This command blocks.

```
pachctl mount <path/to/mount/point>
```

## 12.108.2 Options

```
-c, --commits []string Commits to mount for repos, arguments should be of the
↪form "repo@commit" (default [])
-d, --debug            Turn on debug messages.
```

### 12.108.3 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.109 pachctl port-forward

Forward a port on the local machine to pachd. This command blocks.

### 12.109.1 Synopsis

Forward a port on the local machine to pachd. This command blocks.

<code>pachctl port-forward</code>
-----------------------------------

### 12.109.2 Options

<code>--namespace string</code>	Kubernetes namespace Pachyderm <b>is</b> deployed <b>in</b> .
<code>-f, --pfs-port uint16</code> ↪30652)	The local port to bind PFS over HTTP to. (default ↪
<code>-p, --port uint16</code>	The local port to bind pachd to. (default 30650)
<code>-x, --proxy-port uint16</code> ↪to. (default 30081)	The local port to bind Pachyderm's dash proxy service ↪
<code>--remote-port uint16</code> ↪ (default 650)	The remote port that pachd <b>is</b> bound to <b>in</b> the cluster.
<code>-s, --s3gateway-port uint16</code> ↪30600)	The local port to bind the s3gateway to. (default ↪
<code>--saml-port uint16</code> ↪30654)	The local port to bind pachd's SAML ACS to. (default ↪
<code>-u, --ui-port uint16</code> ↪(default 30080)	The local port to bind Pachyderm's dash service to. ↪

### 12.109.3 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.110 pachctl put

Insert data into Pachyderm.

### 12.110.1 Synopsis

Insert data into Pachyderm.

## 12.110.2 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.111 pachctl put file

Put a file into the filesystem.

### 12.111.1 Synopsis

Put a file into the filesystem. This supports a number of ways to insert data into pfs.

```
pachctl put file <repo>@<branch-or-commit>[:<path/in/pfs>]
```

### 12.111.2 Examples

```
# Put data from stdin as repo/branch/path:
$ echo "data" | pachctl put file repo@branch:/path

# Put data from stdin as repo/branch/path and start / finish a new commit on the
↳branch.
$ echo "data" | pachctl put file -c repo@branch:/path

# Put a file from the local filesystem as repo/branch/path:
$ pachctl put file repo@branch:/path -f file

# Put a file from the local filesystem as repo/branch/file:
$ pachctl put file repo@branch -f file

# Put the contents of a directory as repo/branch/path/dir/file:
$ pachctl put file -r repo@branch:/path -f dir

# Put the contents of a directory as repo/branch/dir/file:
$ pachctl put file -r repo@branch -f dir

# Put the contents of a directory as repo/branch/file, i.e. put files at the top
↳level:
$ pachctl put file -r repo@branch:/ -f dir

# Put the data from a URL as repo/branch/path:
$ pachctl put file repo@branch:/path -f http://host/path

# Put the data from a URL as repo/branch/path:
$ pachctl put file repo@branch -f http://host/path

# Put the data from an S3 bucket as repo/branch/s3_object:
$ pachctl put file repo@branch -r -f s3://my_bucket

# Put several files or URLs that are listed in file.
# Files and URLs should be newline delimited.
```

```
$ pachctl put file repo@branch -i file

# Put several files or URLs that are listed at URL.
# NOTE this URL can reference local files, so it could cause you to put sensitive
# files into your Pachyderm cluster.
$ pachctl put file repo@branch -i http://host/path
```

### 12.111.3 Options

```
-c, --commit          DEPRECATED: Put file(s) in a new commit.
-f, --file strings    The file to be put, it can be a local file or a URL.
→ (default [-])
    --header-records uint    the number of records that will be converted to a
→ PFS 'header', and prepended to future retrievals of any subset of data from PFS;
→ needs to be used with --split=(json|line|csv)
-i, --input-file string    Read filepaths or URLs from a file. If - is used,
→ paths are read from the standard input.
-o, --overwrite        Overwrite the existing content of the file, either
→ from previous commits or previous calls to 'put file' within this commit.
-p, --parallelism int    The maximum number of files that can be uploaded in
→ parallel. (default 10)
-r, --recursive        Recursively put the files in a directory.
    --split line          Split the input file into smaller files, subject to
→ the constraints of --target-file-datums and --target-file-bytes. Permissible values
→ are line, `json`, `sql` and `csv`.
    --target-file-bytes uint    The target upper bound of the number of bytes that
→ each file contains; needs to be used with --split.
    --target-file-datums uint    The upper bound of the number of datums that each
→ file contains, the last file will contain fewer if the datums don't divide evenly;
→ needs to be used with --split.
```

### 12.111.4 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose    Output verbose logs
```

## 12.112 pachctl restart

Cancel and restart an ongoing task.

### 12.112.1 Synopsis

Cancel and restart an ongoing task.

### 12.112.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose    Output verbose logs
```

## 12.113 pachctl restart datum

Restart a datum.

### 12.113.1 Synopsis

Restart a datum.

```
pachctl restart datum <job> <datum-path1>,<datum-path2>,...
```

### 12.113.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.114 pachctl restore

Restore Pachyderm state from stdin or an object store.

### 12.114.1 Synopsis

Restore Pachyderm state from stdin or an object store.

```
pachctl restore
```

### 12.114.2 Examples

```
# Restore from a local file:
$ pachctl restore < backup

# Restore from s3:
$ pachctl restore -u s3://bucket/backup
```

### 12.114.3 Options

```
-u, --url string  An object storage url (i.e. s3://...) to restore from.
```

### 12.114.4 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.115 pachctl resume

Resume a stopped task.

### 12.115.1 Synopsis

Resume a stopped task.

### 12.115.2 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.116 pachctl resume transaction

Set an existing transaction as active.

### 12.116.1 Synopsis

Set an existing transaction as active.

```
pachctl resume transaction <transaction>
```

### 12.116.2 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.117 pachctl run

Manually run a Pachyderm resource.

### 12.117.1 Synopsis

Manually run a Pachyderm resource.

### 12.117.2 Options inherited from parent commands

```
--no-color  Turn off colors.  
-v, --verbose  Output verbose logs
```

## 12.118 pachctl run pipeline

Run an existing Pachyderm pipeline on the specified commits or branches.

### 12.118.1 Synopsis

Run a Pachyderm pipeline on the datums from specific commits. Note: pipelines run automatically when data is committed to them. This command is for the case where you want to run the pipeline on a specific set of data, or if you want to rerun the pipeline. If a commit or branch is not specified, it will default to using the HEAD of master.

```
pachctl run pipeline <pipeline> [<repo>@<commit or branch>...]
```

### 12.118.2 Examples

```
# Rerun the latest job for the "filter" pipeline
$ pachctl run pipeline filter

# Process the pipeline "filter" on the data from commits repo1@a23e4 and
↪repo2@bf363
$ pachctl run pipeline filter repo1@a23e4 repo2@bf363

# Run the pipeline "filter" on the data from the "staging" branch on repo
↪repo1
$ pachctl run pipeline filter repo1@staging
```

### 12.118.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.119 pachctl start

Start a Pachyderm resource.

### 12.119.1 Synopsis

Start a Pachyderm resource.

### 12.119.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.120 pachctl start commit

Start a new commit.

### 12.120.1 Synopsis

Start a new commit with parent-commit as the parent, or start a commit on the given branch; if the branch does not exist, it will be created.

```
pachctl start commit <repo>@<branch-or-commit>
```

### 12.120.2 Examples

```
# Start a new commit in repo "test" that's not on any branch
$ pachctl start commit test

# Start a commit in repo "test" on branch "master"
$ pachctl start commit test@master

# Start a commit with "master" as the parent in repo "test", on a new branch "patch"; ↵
↪essentially a fork.
$ pachctl start commit test@patch -p master

# Start a commit with XXX as the parent in repo "test", not on any branch
$ pachctl start commit test -p XXX
```

### 12.120.3 Options

```
--description string  A description of this commit's contents (synonym for --
↪message)
-m, --message string  A description of this commit's contents
-p, --parent string   The parent of the new commit, unneeded if branch is ↵
↪specified and you want to use the previous head of the branch as the parent.
```

### 12.120.4 Options inherited from parent commands

```
--no-color  Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.121 pachctl start pipeline

Restart a stopped pipeline.

### 12.121.1 Synopsis

Restart a stopped pipeline.



```
pachctl start pipeline <pipeline>
```

### 12.121.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose Output verbose logs
```

## 12.122 pachctl start transaction

Start a new transaction.

### 12.122.1 Synopsis

Start a new transaction.

```
pachctl start transaction
```

### 12.122.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose Output verbose logs
```

## 12.123 pachctl stop

Cancel an ongoing task.

### 12.123.1 Synopsis

Cancel an ongoing task.

### 12.123.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose Output verbose logs
```

## 12.124 pachctl stop job

Stop a job.

### 12.124.1 Synopsis

Stop a job. The job will be stopped immediately.

```
pachctl stop job <job>
```

### 12.124.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose Output verbose logs
```

## 12.125 pachctl stop pipeline

Stop a running pipeline.

### 12.125.1 Synopsis

Stop a running pipeline.

```
pachctl stop pipeline <pipeline>
```

### 12.125.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose Output verbose logs
```

## 12.126 pachctl stop transaction

Stop modifying the current transaction.

### 12.126.1 Synopsis

Stop modifying the current transaction.

```
pachctl stop transaction
```

### 12.126.2 Options inherited from parent commands

```
--no-color    Turn off colors.  
-v, --verbose Output verbose logs
```

## 12.127 pachctl subscribe

Wait for notifications of changes to a Pachyderm resource.

### 12.127.1 Synopsis

Wait for notifications of changes to a Pachyderm resource.

### 12.127.2 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.128 pachctl subscribe commit

Print commits as they are created (finished).

### 12.128.1 Synopsis

Print commits as they are created in the specified repo and branch. By default, all existing commits on the specified branch are returned first. A commit is only considered ‘created’ when it’s been finished.

```
pachctl subscribe commit <repo>@<branch>
```

### 12.128.2 Examples

```
# subscribe to commits in repo "test" on branch "master"
$ pachctl subscribe commit test@master

# subscribe to commits in repo "test" on branch "master", but only since commit XXX.
$ pachctl subscribe commit test@master --from XXX

# subscribe to commits in repo "test" on branch "master", but only for new commits
↳ created from now on.
$ pachctl subscribe commit test@master --new
```

### 12.128.3 Options

```
--from string    subscribe to all commits since this commit
--full-timestamps Return absolute timestamps (as opposed to the default,
↳ relative timestamps).
--new            subscribe to only new commits created from now on
--pipeline string subscribe to all commits created by this pipeline
--raw           disable pretty printing, print raw json
```

## 12.128.4 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.129 pachctl undeploy

Tear down a deployed Pachyderm cluster.

### 12.129.1 Synopsis

Tear down a deployed Pachyderm cluster.

<code>pachctl undeploy</code>
-------------------------------

### 12.129.2 Options

<code>-a, --all</code>	Delete everything, including the persistent volumes where metadata is stored. If your persistent volumes were dynamically provisioned (i.e. <b>if</b> you used the " <code>--dynamic-etcd-nodes</code> " flag), the underlying volumes will be removed, making metadata such repos, commits, pipelines, and jobs unrecoverable. If your persistent volume was manually provisioned (i.e. <b>if</b> you used the " <code>--static-etcd-volume</code> " flag), the underlying volume will <b>not</b> be removed.
<code>--namespace string</code>	Kubernetes namespace to undeploy Pachyderm from.

### 12.129.3 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.130 pachctl unmount

Unmount pfs.

### 12.130.1 Synopsis

Unmount pfs.

```
pachctl unmount <path/to/mount/point>
```

## 12.130.2 Options

```
-a, --all    unmount all pfs mounts
```

## 12.130.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.131 pachctl update-dash

Update and redeploy the Pachyderm Dashboard at the latest compatible version.

### 12.131.1 Synopsis

Update and redeploy the Pachyderm Dashboard at the latest compatible version.

```
pachctl update-dash
```

### 12.131.2 Options

```
--dry-run      Don't actually deploy Pachyderm Dash to Kubernetes, instead,
->just print the manifest.
-o, --output string  Output format. One of: json|yaml (default "json")
```

### 12.131.3 Options inherited from parent commands

```
--no-color    Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.132 pachctl update

Change the properties of an existing Pachyderm resource.

### 12.132.1 Synopsis

Change the properties of an existing Pachyderm resource.

## 12.132.2 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.133 pachctl update pipeline

Update an existing Pachyderm pipeline.

### 12.133.1 Synopsis

Update a Pachyderm pipeline with a new pipeline specification. For details on the format, see [http://docs.pachyderm.io/en/latest/reference/pipeline\\_spec.html](http://docs.pachyderm.io/en/latest/reference/pipeline_spec.html).

<code>pachctl update pipeline</code>
--------------------------------------

### 12.133.2 Options

<code>-b, --build</code>	If true, build <b>and</b> push local docker images into the docker_↵ ↵registry.
<code>-f, --file string</code>	The JSON file containing the pipeline, it can be a url <b>or</b> _↵ ↵local file. - reads <b>from</b> <b>stdin</b> . (default "-")
<code>-p, --push-images</code>	If true, push local docker images into the docker registry.
<code>-r, --registry string</code>	The registry to push images to. (default "docker.io")
<code>--reprocess</code>	If true, reprocess datums that were already processed by_↵ ↵previous version of the pipeline.
<code>-u, --username string</code>	The username to push images <b>as</b> , defaults to your OS_↵ ↵username.

### 12.133.3 Options inherited from parent commands

<code>--no-color</code>	Turn off colors.
<code>-v, --verbose</code>	Output verbose logs

## 12.134 pachctl update repo

Update a repo.

### 12.134.1 Synopsis

Update a repo.

<code>pachctl update repo &lt;repo&gt;</code>
---

## 12.134.2 Options

```
-d, --description string  A description of the repo.
```

## 12.134.3 Options inherited from parent commands

```
--no-color  Turn off colors.
-v, --verbose  Output verbose logs
```

## 12.135 pachctl version

Print Pachyderm version information.

### 12.135.1 Synopsis

Print Pachyderm version information.

```
pachctl version
```

### 12.135.2 Options

```
--client-only  If set, only print pachctl's version, but don't make any
↳RPCs to pachd. Useful if pachd is unavailable
--raw          disable pretty printing, print raw json
--timeout string  If set, 'pachctl version' will timeout after the given
↳duration (formatted as a go lang time duration--a number followed by ns, us, ms, s,
↳m, or h). If --client-only is set, this flag is ignored. If unset, pachctl will use
↳a default timeout; if set to 0s, the call will never time out. (default "default")
```

### 12.135.3 Options inherited from parent commands

```
--no-color  Turn off colors.
-v, --verbose  Output verbose logs
```





---

## Examples

---

### 13.1 OpenCV Edge Detection

This example does edge detection using OpenCV. This is our canonical starter demo. If you haven't used Pachyderm before, start here. We'll get you started running Pachyderm locally in just a few minutes and processing sample log lines.

[Open CV](#)

### 13.2 Word Count (Map/Reduce)

Word count is basically the “hello world” of distributed computation. This example is great for benchmarking in distributed deployments on large swaths of text data.

[Word Count](#)

### 13.3 Periodic Ingress from a Database

This example pipeline executes a query periodically against a MongoDB database outside of Pachyderm. The results of the query are stored in a corresponding output repository. This repository could be used to drive additional pipeline stages periodically based on the results of the query.

[Periodic Ingress from MongoDB](#)

### 13.4 Lazy Shuffle pipeline

This example demonstrates how lazy shuffle pipeline i.e. a pipeline that shuffles, combines files without downloading/uploading can be created. These types of pipelines are useful for intermediate processing step that aggregates or rearranges data from one or many sources. For more information [see](#)

[Lazy Shuffle pipeline](#)

## 13.5 Variant Calling and Joint Genotyping with GATK

This example illustrates the use of GATK in Pachyderm for Germline variant calling and joint genotyping. Each stage of this GATK best practice pipeline can be scaled individually and is automatically triggered as data flows into the top of the pipeline. The example follows [this tutorial](#) from GATK, which includes more details about the various stages.

[GATK - Variant Calling](#)

## 13.6 Machine Learning

### 13.6.1 Iris flower classification with R, Python, or Julia

The “hello world” of machine learning implemented in Pachyderm. You can deploy this pipeline using R, Python, or Julia components, where the pipeline includes the training of a SVM, LDA, Decision Tree, or Random Forest model and the subsequent utilization of that model to perform inferences.

[R, Python, or Julia - Iris flower classification](#)

### 13.6.2 Sentiment analysis with Neon

This example implements the machine learning template pipeline discussed in [this blog post](#). It trains and utilizes a neural network (implemented in Python using Nervana Neon) to infer the sentiment of movie reviews based on data from IMDB.

[Neon - Sentiment Analysis](#)

### 13.6.3 pix2pix with TensorFlow

If you haven’t seen pix2pix, check out [this great demo](#). In this example, we implement the training and image translation of the pix2pix model in Pachyderm, so you can generate cat images from edge drawings, day time photos from night time photos, etc.

[TensorFlow - pix2pix](#)

### 13.6.4 Recurrent Neural Network with Tensorflow

Based on [this Tensorflow example](#), this pipeline generates a new Game of Thrones script using a model trained on existing Game of Thrones scripts.

[Tensorflow - Recurrent Neural Network](#)

### 13.6.5 Distributed Hyperparameter Tuning

This example demonstrates how you can evaluate a model or function in a distributed manner on multiple sets of parameters. In this particular case, we will evaluate many machine learning models, each configured with different sets of parameters (aka hyperparameters), and we will output only the best performing model or models.

[Hyperparameter Tuning](#)

### 13.6.6 Spark Example

This example demonstrates integration of Spark with Pachyderm by launching a Spark job on an existing cluster from within a Pachyderm Job. The job uses configuration info that is versioned within Pachyderm, and stores its reduced result back into a Pachyderm output repo, maintaining full provenance and version history within Pachyderm, while taking advantage of Spark for computation.

[Spark Example](#)



---

## Setup for contributors

---

### 14.1 General requirements

First, go through the general local installation instructions [here](#). Additionally, make sure you have the following installed:

- go lang 1.12+
- docker
- jq
- pv

### 14.2 Bash helpers

To stay up to date, we recommend doing the following.

First clone the code: (Note, as of 07/11/19 pachyderm is using go modules and recommends cloning the code outside of the \$GOPATH, we use the location ~/workspace as an example, but the code can live anywhere)

```
cd ~/workspace
git clone git@github.com:pachyderm/pachyderm
```

Then update your ~/.bash\_profile by adding the line:

```
source ~/workspace/pachyderm/etc/contributing/bash_helpers
```

And you'll stay up to date!

### 14.3 Special macOS configuration

#### 14.3.1 File descriptor limit

If you're running tests locally, you'll need to up your file descriptor limit. To do this, first setup a LaunchDaemon to up the limit with sudo privileges:

```
sudo cp ~/workspace/pachyderm/etc/contributing/com.apple.launchd.limit.plist /Library/
↳ LaunchDaemons/
```

Once you restart, this will take effect. To see the limits, run:

```
launchctl limit maxfiles
```

Before the change is in place you'll see something like `256 unlimited`. After the change you'll see a much bigger number in the first field. This ups the system wide limit, but you'll also need to set a per-process limit.

Second, up the per process limit by adding something like this to your `~/ .bash_profile` :

```
ulimit -n 12288
```

Unfortunately, even after setting that limit it never seems to report the updated version. So if you try

```
ulimit
```

And just see `unlimited` , don't worry, it took effect.

To make sure all of these settings are working, you can test that you have the proper setup by running:

```
make test-pfs-server
```

If this fails with a timeout, you'll probably also see 'too many files' type of errors. If that test passes, you're all good!

### 14.3.2 Timeout helper

You'll need the `timeout` utility to run the `make launch` task. To install on mac, do:

```
brew install coreutils
```

And then make sure to prepend the following to your path:

```
PATH="/usr/local/opt/coreutils/libexec/gnubin:$PATH"
```

## 14.4 Dev cluster

Now launch the dev cluster: `make launch-dev-vm`.

And check it's status: `kubectl get all`.

## 14.5 pachctl

This will install the dev version of `pachctl` :

```
cd ~/workspace/pachyderm
make install
pachctl version
```

And make sure that `$GOPATH/bin` is on your `$PATH` somewhere

## 14.6 Fully resetting

Instead of running the makefile targets to re-compile `pachctl` and redeploy a dev cluster, we have a script that you can use to fully reset your pachyderm environment:

1. All existing cluster data is deleted
2. If possible, the virtual machine that the cluster is running on is wiped out
3. `pachctl` is recompiled
4. The dev cluster is re-deployed

This reset is a bit more time consuming than running one-off Makefile targets, but comprehensively ensures that the cluster is in its expected state, and is especially helpful when you're first getting started with contributions and don't yet have a complete intuition on the various ways a cluster may get in an unexpected state. It's been tested on docker for mac and minikube, but likely works in other kubernetes environments as well.

To run it, simply call `./etc/reset.py` from the pachyderm repo root.





---

## Gcloud cluster setup

---

In order to develop pachyderm against a gcloud-deployed cluster, follow these instructions.

### 15.1 First steps

First follow the [general setup instructions](#).

### 15.2 gcloud

#### Download Page

Setup Google Cloud Platform via the web

- login with your Gmail or G Suite account
  - click the silhouette in the upper right to make sure you're logged in with the right account
- get your owner/admin to setup a project for you (e.g. YOURNAME-dev)
- then they need to go into the project > settings > permissions and add you
  - hint to owner/admin: its the permissions button in one of the left hand popin menus (GKE UI can be confusing)
- you should have an email invite to accept
- click 'use google APIs' (or something along the lines of enable/manage APIs)
- click through to google compute engine API and enable it or click the 'get started' button to make it provision

Then, locally, run the following commands one at a time:

```
gcloud auth login
gcloud init

# This should have you logged in / w gcloud
# The following will only work after your GKE owner/admin adds you to the right
↪project on gcloud:

gcloud config set project YOURNAME-dev
gcloud compute instances list

# Now create instance using our bash helper
```

```
create_docker_machine

# And attach to the right docker daemon
eval "$(docker-machine env dev)"
```

Setup a project on gcloud

- go to [console.cloud.google.com/start](https://console.cloud.google.com/start)
- make sure you're logged in w your gmail account
- create project 'YOURNAME-dev'

## 15.3 kubectl

Now that you have gcloud, just do:

```
gcloud components update kubectl
# Now you need to start port forwarding to allow kubectl client talk to the
↪kubernetes service on GCE

portforwarding
# To see this alias, look at the bash_helpers

kubectl version
# should report a client version, not a server version yet

make launch-kube
# to deploy kubernetes service

kubectl version
# now you should see a client and server version

docker ps
# you should see a few processes
```

## 15.4 Pachyderm cluster deployment

```
make launch
```

---

## Repo layout

---

Following is a layout of the various directories that make up the pachyderm repo, and their purpose.

```

build
debian
doc - documentation used on readthedocs
-- pachctl - cobra auto-generated docs on command-line usage
etc - everything else
-- build - scripts for building releases
-- compatibility - contains mappings of pachyderm versions to the dash versions they
    ↳ 're compatible with
-- compile - scripts to facilitate compiling and building docker images
-- contributing - contains helper scripts/assets for contributors
-- deploy - scripts/assets for pachyderm deployments
|   -- cloudfront
|   -- gpu - scripts to help enable GPU resources on k8s/pachyderm
|   -- tracing - k8s manifests for enabling Jaeger tracing of pachyderm
-- initdev - scripts to stand up a vagrant environment for pachyderm
-- kube - internal scripts for working with k8s
-- kubernetes-kafka
-- kubernetes-prometheus
-- netcat
-- plugin
|   -- logging
|   -- monitoring
-- proto - scripts for compiling protobufs
-- testing - scripts/assets used for testing
|   -- artifacts - static assets used in testing/mocking
|   -- deploy - scripts to assist in deploying pachyderm on various cloud providers
|   -- entrypoint
|   -- migration - sample data used in testing pachyderm migrations
|   -- s3gateway - scripts for running conformance tests on the s3gateway
|   -- vault-s3-client
-- user-job
-- worker
examples - example projects; see readme for details of each one
src - source code
-- client - contains protobufs and the source code for pachyderm's go client
|   -- admin - admin-related functionality
|   |   -- 1_7 - old, v1.7-compatible protobufs
|   -- auth - auth-related functionality
|   -- debug - debug-related functionality
|   -- deploy - deployment-related functionality
|   -- enterprise - enterprise-related functionality

```

```
| -- health - health check-related functionality
| -- limit - limit-related functionality
| -- pfs - PFS-related functionality
| -- pkg - utility packages
| | -- config - pachyderm config file reading/writing
| | -- discovery
| | -- grpcutil - utilities for working with gRPC clients/servers
| | -- pbutil - utilities for working with protobufs
| | -- require - utilities for making unit tests terser
| | -- shard
| | -- tracing - facilitates pachyderm cluster Jaeger tracing
| -- pps - PPS-related functionality
| -- version - version check-related functionality
-- plugin
| -- vault
| | -- etc
| | -- pachyderm
| | -- pachyderm-plugin
| | -- vendor - vendored libraries for the vault plugin
-- server - contains server-side logic and CLI
| -- admin - cluster admin functionality
| | -- cmds - cluster admin CLI
| | -- server - cluster admin server
| -- auth - auth functionality
| | -- cmds - auth CLI
| | -- server - auth server
| | -- testing - a mock auth server used for testing
| -- cmd - contains the various pachyderm entrypoints
| | -- pachctl - the CLI entrypoint
| | -- pachctl-doc - helps generate docs for the CLI
| | -- pachd - the server entrypoint
| | -- worker - the worker entrypoint
| -- debug - debug functionality
| | -- cmds - debug CLI
| | -- server - debug server
| -- deploy - storage secret deployment server
| -- enterprise - enterprise functionality
| | -- cmds - enterprise CLI
| | -- server - enterprise server
| -- health - health check server
| -- http - PFS-over-HTTP server, used by the dash to serve PFS content
| -- pfs - PFS functionality
| | -- cmds - PFS CLI
| | -- fuse - support mounting PFS repos via FUSE
| | -- pretty - pretty-printing of PFS metadata in the CLI
| | -- s3 - the s3gateway, an s3-like HTTP API for serving PFS content
| | -- server - PFS server
| -- pkg - utility packages
| | -- ancestry - parses git ancestry reference strings
| | -- backoff - backoff algorithms for retrying operations
| | -- cache - a gRPC server for serving cached content
| | -- cert - functionality for generating x509 certificates
| | -- cmdutil - functionality for helping creating CLIs
| | -- collection - etcd collection management
| | -- dag - a simple in-memory directed acyclic graph data structure
| | -- deploy - functionality for deploying pachyderm
| | | -- assets - generates k8s manifests and other assets used in deployment
| | | -- cmds - deployment CLI
```

```

| | | -- images - handling of docker images
| | -- dlock - distributed lock on etcd
| | -- errutil - utility functions for error handling
| | -- exec - utilities for running external commands
| | -- hashtree - a Merkle tree library
| | -- lease - utility for managing resources with expirable leases
| | -- localcache - a concurrency-safe local disk cache
| | -- log - logging utilities
| | -- metrics - cluster metrics service using segment.io
| | -- migration
| | -- netutil - networking utilities
| | -- obj - tools for working with various object stores (e.g. S3)
| | -- pfsdb - the etcd database schema that PFS uses
| | -- pool - gRPC connection pooling
| | -- ppsconsts - PPS-related constants
| | -- ppsdb - the etcd database schema that PPS uses
| | -- ppsutil - PPS-related utility functions
| | -- pretty - function for pretty printing values
| | -- serviceenv - management of connections to pach services
| | -- sql - tools for working with postgres database dumps
| | -- sync - tools for syncing PFS content
| | -- tabwriter - tool for writing tab-delimited content
| | -- testutil - test-related utilities
| | -- uuid - UUID generation
| | -- watch - tool for watching etcd databases for changes
| | -- workload
| -- pps - PPS functionality
| | -- cmds - PPS CLI
| | -- example - example PPS requests
| | -- pretty - pretty printing of PPS output to the CLI
| | -- server - PPS server
| | -- githook - support for github PPS sources
| -- vendor - vendored packages
| -- worker - pachd master and sidecar
-- testing - testing tools
| -- loadtest - load tests for pachyderm
| | -- split - stress tests of PFS merge functionality
| -- match - a grep-like tool used in testing
| -- saml-idp
| -- vendor - vendored packages

```



---

## Coding Conventions

---

All code in this repo should be written in Go, Shell or Make. Exceptions are made for things under `examples` because we want to be able to give people examples of using the product in other languages. And for things like `doc/conf.py` which configures an outside tool we want to use for docs. However in choosing outside tooling we prefer tools that we can interface with entirely using Go. Go's new enough that it's not always possible to find such a tool so we expect to make compromises on this. In general you should operate under the assumption that code written in Go, Shell or Make is accessible to all developers of the project and code written in other languages is accessible to only a subset and thus represents a higher liability.

### 17.1 Shell

- <https://google.github.io/styleguide/shell.xml>
- Scripts should work on macOS as well as Linux.

### 17.2 Go

Go has pretty unified conventions for style, we vastly prefer embracing these standards to developing our own.

#### 17.2.1 Stylistic Conventions

- We have several Go checks that run as part of CI, those should pass. You can run them with `make pretest` and `make lint`.
- [Go Code Review Comments](#)
- [Effective Go](#)
- Command-line flags should use dashes, not underscores.
- Naming
  - Please consider package name when selecting an interface name, and avoid redundancy.
  - e.g.: `storage.Interface` is better than `storage.StorageInterface`.
  - Do not use uppercase characters, underscores, or dashes in package names.
  - Unless there's a good reason, the `package foo` line should match the name of the directory in which the `.go` file exists.

- Importers can use a different name if they need to disambiguate.
- Locks should be called `lock` and should never be embedded (always `lock sync.Mutex`). When multiple locks are present, give each lock a distinct name following Go conventions - `stateLock`, `mapLock` etc.

### 17.2.2 Testing Conventions

- All new packages and most new significant functionality must come with test coverage
- Avoid waiting for asynchronous things to happen (e.g. waiting 10 seconds and assuming that a service will be afterward). Instead you try, wait, retry, etc. with a limited number of tries. If possible use a method of waiting directly (e.g. ‘flush commit’ is much better than repeatedly trying to read from a commit).

### 17.2.3 Go Modules/Third-Party Code

- Go dependencies are managed with go modules (as of 07/11/2019).
- To add a new package or update a package. Do:
  - `go get foo` or for a more specific version `go get foo@v1.2.3`, `go get foo@master`, `go get foo@e3702bed2`
  - import `foo` package to you go code as needed.
  - Run `go mod vendor`
- Note: Go modules requires you clone the repo outside of the `$GOPATH` or you must pass the `GO111MODULE=on` flag to any go commands. See wiki page on [activating module support](#)
- See [The official go modules wiki](#) for more info.

### 17.2.4 Docs

- PRs for code must include documentation updates that reflect the changes that the code introduces.
- When writing documentation, follow the Style Guide conventions.
- PRs that have only documentation changes, such as typos, is a great place to start and we welcome your help!
- For most documentation PRs, you need to make `assets` and push the new `assets.go` file as well.