
Pachyderm Documentation

Release 1.7.11

Joe Doliner

Nov 14, 2018

1	Getting Started	3
2	Local Installation	5
3	Beginner Tutorial	9
4	Getting Your Data into Pachyderm	19
5	Creating Analysis Pipelines	25
6	Getting Data Out of Pachyderm	29
7	Deleting Data in Pachyderm	33
8	Updating Pipelines	35
9	Distributed Computing	37
10	Incremental Processing	41
11	Overview	45
12	Deploying Enterprise Edition	49
13	Access Controls	51
14	Advanced Statistics	59
15	Overview	65
16	Google Cloud Platform	67
17	Amazon Web Services	71
18	Azure	77
19	OpenShift	81
20	On Premises	85
21	Custom Object Stores	87

22 AWS CloudFront	91
23 Pachyderm Version Upgrades	93
24 Pachyderm Migrations	97
25 Non-Default Namespaces	99
26 RBAC	101
27 Autoscaling a Pachyderm Cluster	103
28 Data Management Best Practices	105
29 Sharing GPU Resources	107
30 Without configuration	109
31 Configuring your pipelines to share GPUs	111
32 General Troubleshooting	113
33 Deploy Specific Troubleshooting	119
34 Examples	123
35 Splitting Data for Distributed Processing	127
36 Combining/Merging/Joining Data	133
37 Creating Machine Learning Workflows	137
38 Processing Time-Windowed Data	139
39 Utilizing GPUs	145
40 Triggering Pipelines Periodically (cron)	151
41 Deferred Processing of Data	155
42 Ingressing From a Separate Object Store	159
43 Vault Secret Engine	161
44 Pipeline Specification	165
45 Environment Variables	177
46 Pachctl Command Line Tool	179
47 Pachyderm language clients	181

Welcome to the Pachyderm documentation portal! Below you'll find guides and information for beginners and experienced Pachyderm users. You'll also find API references docs.

If you can't find what you're looking for or have an issue not mentioned here, we'd love to hear from you either on [GitHub](#), our [Users Slack channel](#), or email us at support@pachyderm.io.

Note: if you are using a Pachyderm version < 1.4, you can find relevant docs [here](#).

Getting Started

Welcome to the documentation portal for first time Pachyderm users! We've organized information into two sections:

1. Local Installation: Get Pachyderm deployed locally on OSX or Linux.
2. Beginner Tutorial: Learn to use Pachyderm through a quick and simple tutorial.

If you'd like to read about the Pachyderm's open source and enterprise features before actually running it, check out the following:

- [The open source core](#)
- [Enterprise edition](#)
- [Use Cases](#)

Looking for more in-depth development docs? Check out the Pachyderm fundamentals:

- [Getting data into Pachyderm](#)
- [Creating analysis pipelines](#)
- [Distributed computing](#)
- [Incremental processing](#)
- [Getting data out of Pachyderm](#)
- [Updating pipelines](#)

Need to see different or more advanced Pachyderm examples? You can find a bunch of them [here](#).

Note - If you've already got a Kubernetes cluster running or would rather use AWS, GCE or Azure to deploy Pachyderm, check out our [deployment guides](#).

Local Installation

This guide will walk you through the recommended path to get Pachyderm running locally on OSX or Linux.

If you hit any errors not covered in this guide, check our [troubleshooting](#) docs for common errors, submit an issue on [GitHub](#), join our [users channel on Slack](#), or email us at support@pachyderm.io and we can help you right away.

Prerequisites

- *Minikube* (and VirtualBox)
- *Pachyderm Command Line Interface*

Minikube

Kubernetes offers a fantastic guide to [install minikube](#). Follow the Kubernetes installation guide to install Virtual Box, Minikube, and Kubectl. Then come back here to start Minikube:

```
minikube start
```

Note: Any time you want to stop and restart Pachyderm, you should start fresh with `minikube delete` and `minikube start`. Minikube isn't meant to be a production environment and doesn't handle being restarted well without a full wipe.

Pachctl

`pachctl` is a command-line utility used for interacting with a Pachyderm cluster.

```
# For OSX:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.7

# For Debian based linux (64 bit) or Window 10+ on WSL:
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
  ↳download/v1.7.11/pachctl_1.7.11_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/v1.7.11/pachctl_1.7.11_linux_amd64.tar.gz && tar -xvf /tmp/pachctl.tar.gz -
  ↳C /tmp && sudo cp /tmp/pachctl_1.7.11_linux_amd64/pachctl /usr/local/bin
```

Note: To install an older version of Pachyderm, navigate to that version using the menu in the bottom left.

To check that installation was successful, you can try running `pachctl help`, which should return a list of Pachyderm commands.

Deploy Pachyderm

Now that you have Minikube running, it's incredibly easy to deploy Pachyderm.

```
$ pachctl deploy local
```

This generates a Pachyderm manifest and deploys Pachyderm on Kubernetes. It may take a few minutes for the Pachyderm pods to be in a `Running` state, because the containers have to be pulled from DockerHub. You can see the status of the Pachyderm pods using `kubectl get pods`. When Pachyderm is ready for use, this should return something similar to:

```
$ 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's ok. That simply means that Kubernetes tried to bring up those pods before `etcd` was ready so it restarted them.

Port Forwarding

The last step is to set up port forwarding so commands you send can reach Pachyderm within the VM. We background this process since port forwarding blocks.

```
$ pachctl port-forward &
```

Once port forwarding is complete, `pachctl` should automatically be connected. Try `pachctl version` to make sure everything is working.

```
$ pachctl version
COMPONENT    VERSION
pachctl      1.7.0
pachd        1.7.0
```

We're good to go!

If for any reason `port-forward` doesn't work, you can connect directly by setting `ADDRESS` to the minikube IP with port 30650.

```
$ minikube ip
192.168.99.100
$ export ADDRESS=192.168.99.100:30650
```

Next Steps

Now that you have everything installed and working, check out our Beginner Tutorial to learn the basics of Pachyderm such as adding data and building pipelines for analysis.

The Pachyderm Enterprise dashboard is deployed by default with Pachyderm. You can get a FREE trial token and experiment with this interface to Pachyderm by visiting `localhost:30080` in your Internet browser (e.g., Google Chrome).

Beginner Tutorial

Welcome to the beginner tutorial for Pachyderm! If you've already got Pachyderm installed, this guide should take about 15 minutes, and it will introduce you to the basic concepts of Pachyderm.

Image processing with OpenCV

This guide will walk you through the deployment of a Pachyderm pipeline to do some 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. We are more than happy to help!

Prerequisites

This guide assumes that you already have Pachyderm running locally. Check out our [Local Installation](#) instructions if haven't done that yet and then come back here to continue.

Create a Repo

A `repo` is the highest level data primitive in Pachyderm. Like many things in Pachyderm, it shares its name with primitives 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

# See the repo we just created
$ pachctl list-repo
```

NAME	CREATED	SIZE
images	2 minutes ago	0 B

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 though, 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. In a situation where 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 what we want to name the file, “liberty.png”.

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

Finally, we 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
images        5 minutes ago   57.27 KiB

# We can view the commit we just created
$ pachctl list-commit images
REPO          ID                                PARENT          STARTED
↪            DURATION          SIZE
images        7162f5301e494ec8820012576476326c <none>          2 minutes
↪ago          38 seconds        57.27 KiB

# And view the file in that commit
$ pachctl list-file images master
NAME          TYPE          SIZE
liberty.png   file          57.27 KiB
```

We can 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 OSX
$ pachctl get-file images master liberty.png | open -f -a /Applications/Preview.app

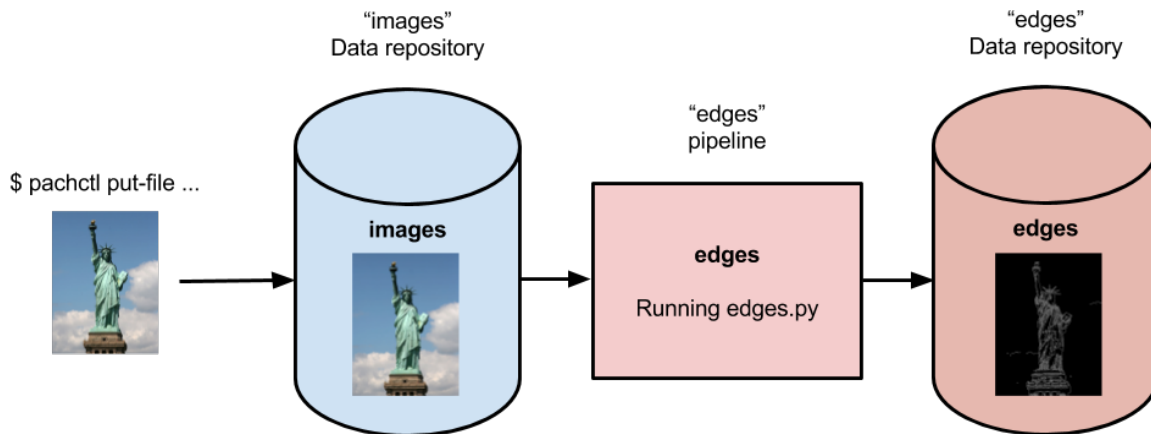
# on Linux
$ pachctl get-file images master liberty.png | display
```

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. This includes 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": {
    "atom": {
      "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 “atom” input, our images repo with a particular glob pattern.

The glob pattern defines how the input data can be broken up if we wanted 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))
```

Our python code is really straight forward. We're simply walking 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 for you. 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/doc/examples/opencv/edges.json
```

What Happens When You Create a Pipeline

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

This 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). As such, this first run this might take a minute or so, depending on your Internet connection. Subsequent runs will be much faster.

You can view the job with:

```
$ pachctl list-job
ID                                OUTPUT COMMIT                                STARTED
↳DURATION  RESTART PROGRESS  DL      UL      STATE
490a28be32de491e942372018cd42460 edges/bc2d20d0c23740f397622a62b0978c57 2 minutes ago
↳35 seconds 0          1 + 0 / 1 57.27KiB 22.22KiB success
```

Yay! Our pipeline succeeded! Notice, that there is an `OUTPUT COMMIT` column specified above. Pachyderm creates a corresponding output repo for every pipeline. This output repo will have the same name as the pipeline, and all the results of that pipeline will be versioned in this output repo. In our example, the “edges” pipeline created a repo called “edges” to store the results.

```
$ pachctl list-repo
NAME      CREATED      SIZE
edges     2 minutes ago 22.22 KiB
images    10 minutes ago 57.27 KiB
```

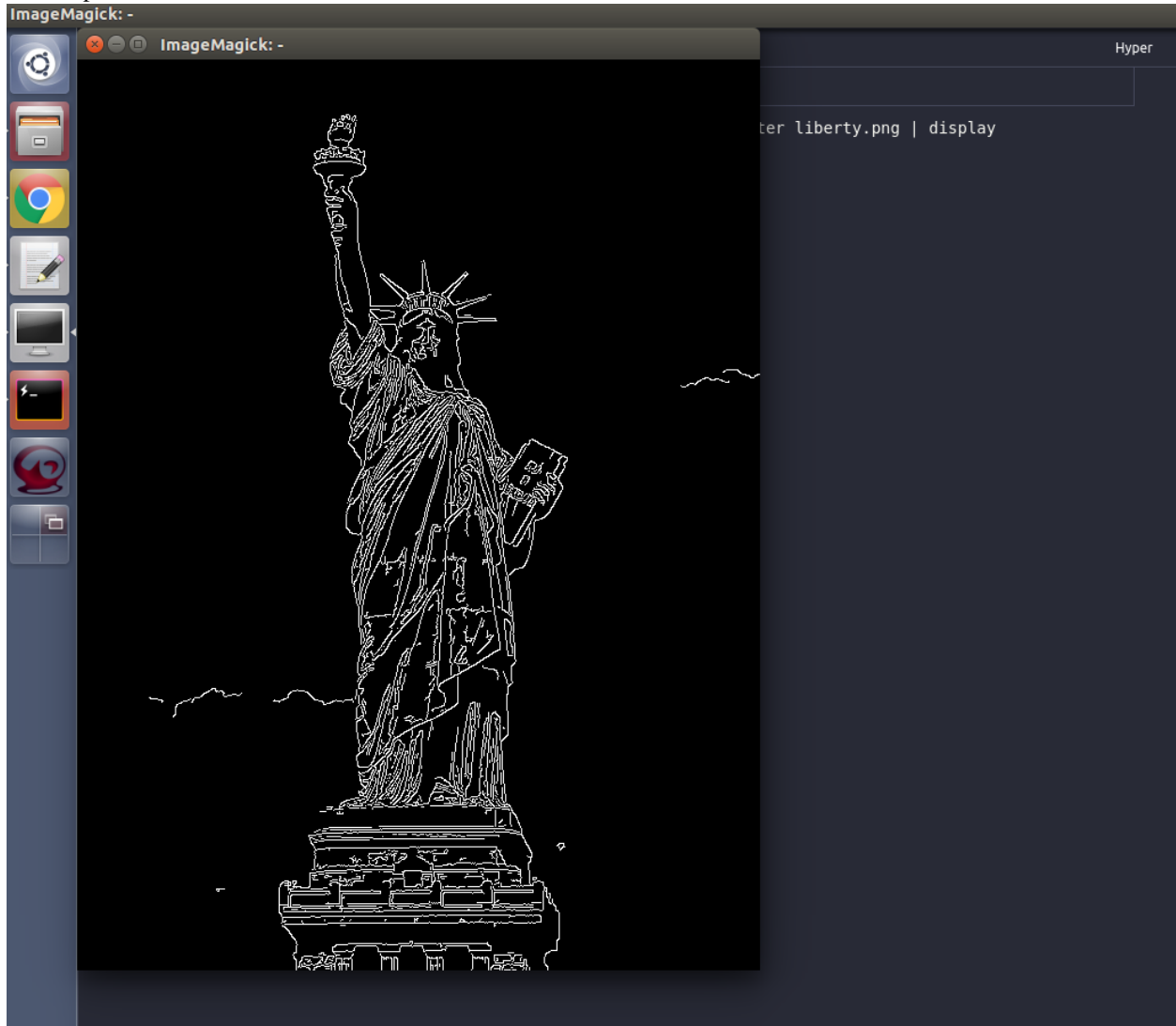
Reading the Output

We can view the output data from the “edges” repo in the same fashion that we viewed the input data.


```
# on OSX
$ pachctl get-file edges master liberty.png | open -f -a /Applications/Preview.app

# on Linux
$ pachctl get-file edges master liberty.png | display
```

The output should look similar to:



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                                OUTPUT COMMIT                                STARTED
↪ DURATION                      RESTART PROGRESS  DL      UL      STATE
81ae47a802f14038b95f8f248cddb2 edges/146a5e398f3f40a09f5151559fd4a6cb 7 seconds ago
↪ Less than a second 0          1 + 2 / 3 102.4KiB 74.21KiB success
ce448c12d0dd4410b3a5ae0c0f07e1f9 edges/c5d7ded9ba214d9aa4aa2c044625198c 16 seconds
↪ ago Less than a second 0      1 + 1 / 2 78.7KiB 37.15KiB success
490a28be32de491e942372018cd42460 edges/bc2d20d0c23740f397622a62b0978c57 9 minutes ago
↪ 35 seconds                   0          1 + 0 / 1 57.27KiB 22.22KiB success
```

```
# View the output data

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

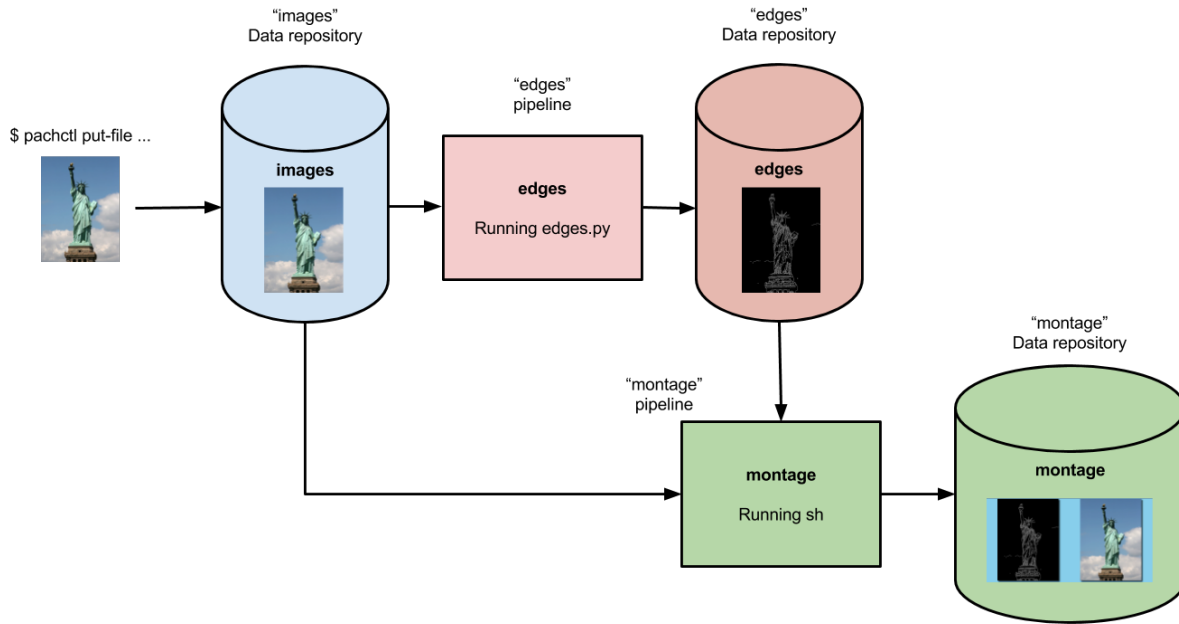
$ pachctl get-file edges master kitten.png | open -f -a /Applications/Preview.app

# 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 utilized a single stage Pachyderm pipeline, but 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": [ {
      "atom": {
        "glob": "/",
        "repo": "images"
      }
    },
    {
      "atom": {
        "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 pipeline spec is very similar to our `edges` pipeline except, for `montage` : (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 this case we are combining our multiple input data repositories using a `cross` pattern. There are multiple interesting ways to combine data in Pachyderm, which are further discussed [here](#) and [here](#). For the purposes of this example,

suffice it to say that this `cross` pattern creates a single pairing of our input images with our edge detected images.

We create this next pipeline as before, with `pachctl` :

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

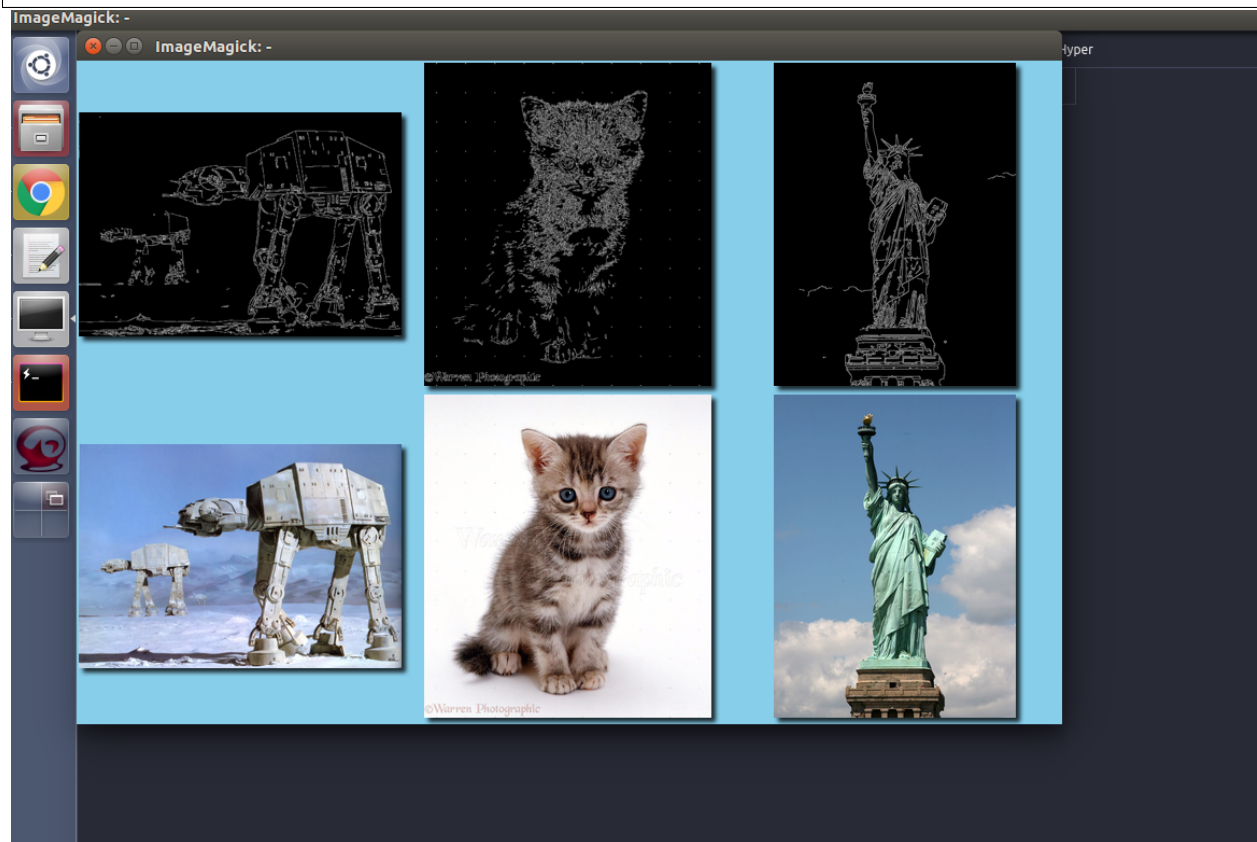
This will automatically trigger a job that generates a montage for all the current HEAD commits of the input repos:

```
$ pachctl list-job
ID                                OUTPUT COMMIT                                STARTED
↪                                ↪                                ↪
↪  DURATION                      RESTART PROGRESS  DL      UL      STATE
92cecc40c3144fd5b4e07603bb24b104 montage/1af4657db2404fcfba1c6cee6c71ae16 45 seconds
↪ago 6 seconds                  0        1 + 0 / 1 371.9KiB 1.284MiB success
81ae47a802f14038b95f8f248cddb2 edges/146a5e398f3f40a09f5151559fd4a6cb 2 minutes
↪ago Less than a second 0        1 + 2 / 3 102.4KiB 74.21KiB success
ce448c12d0dd4410b3a5ae0c0f07e1f9 edges/c5d7ded9ba214d9aa4aa2c044625198c 2 minutes
↪ago Less than a second 0        1 + 1 / 2 78.7KiB 37.15KiB success
490a28be32de491e942372018cd42460 edges/bc2d20d0c23740f397622a62b0978c57 11 minutes
↪ago 35 seconds                0        1 + 0 / 1 57.27KiB 22.22KiB success
```

And you can view the generated montage image via:

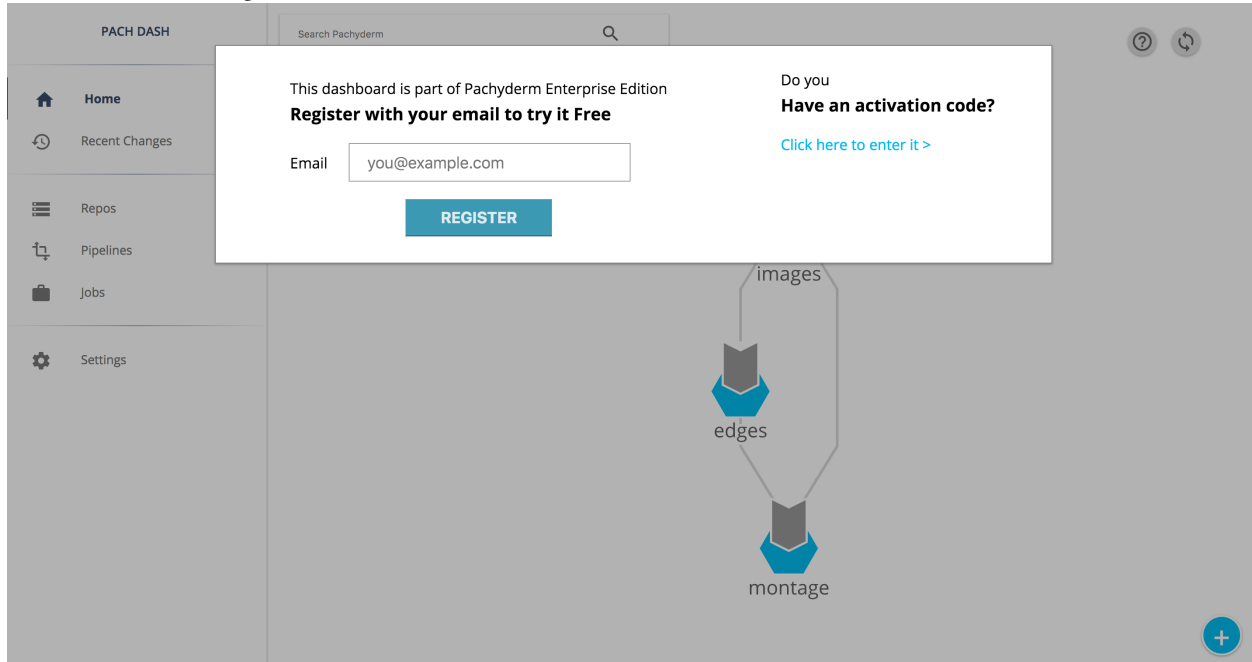
```
# on OSX
$ pachctl get-file montage master montage.png | open -f -a /Applications/Preview.app

# on Linux
$ pachctl get-file montage master montage.png | display
```

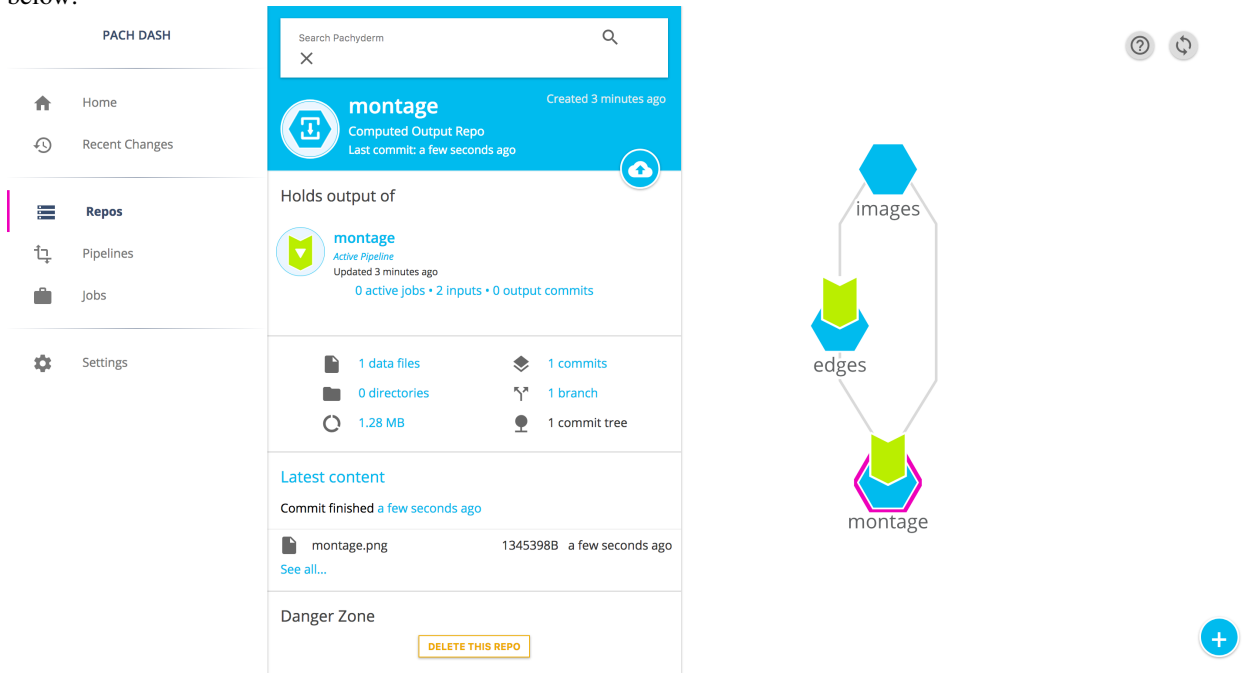


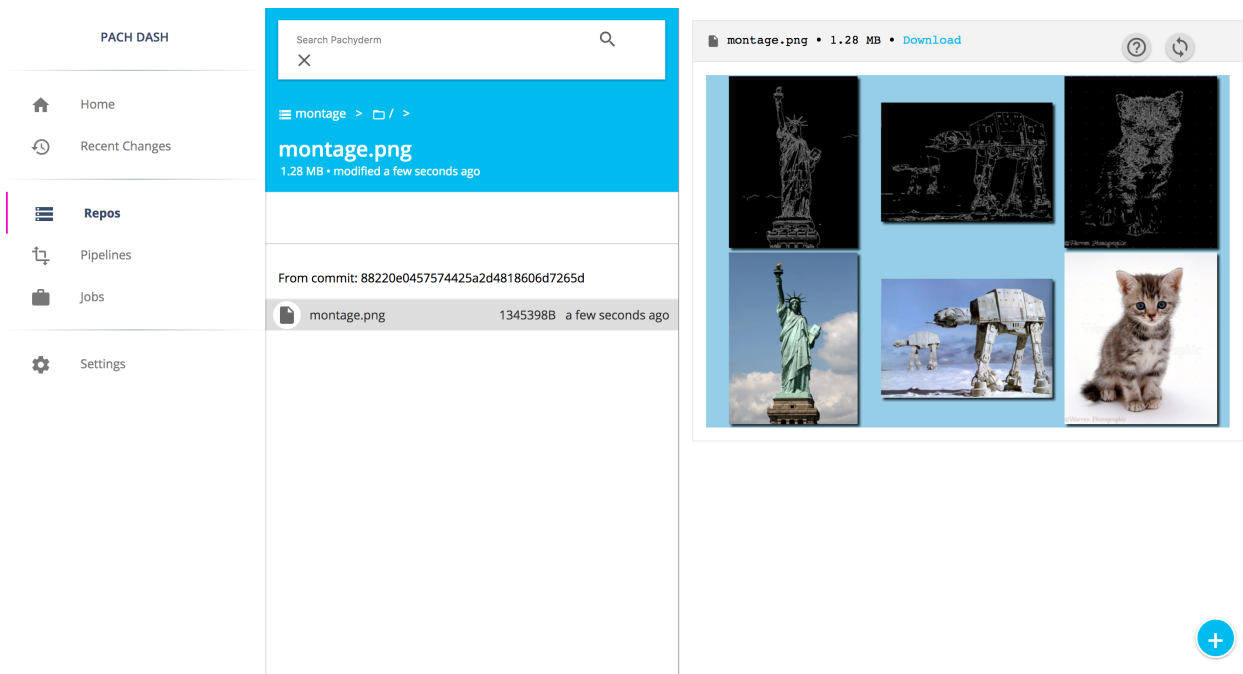
Exploring your DAG in the Pachyderm dashboard

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 will 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:





Next Steps

We've now got Pachyderm running locally with data and a pipeline! If you want to keep playing 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](#)
- [Getting Your Data into Pachyderm](#)
- [Creating Analysis 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 if you want to show off anything nifty you've created!

Getting Your Data into Pachyderm

Data that you put (or “commit”) into Pachyderm ultimately lives in an object store of your choice (S3, Minio, GCS, etc.). This data is content-addressed by Pachyderm to build our version control semantics for data and is therefore not “human-readable” directly in the object store. That being said, Pachyderm allows you and your pipeline stages to interact with versioned data like you would in a normal file system.

Jargon associated with putting data in Pachyderm

“Data Repositories”

Versioned data in Pachyderm lives in repositories (again think about something similar to “git for data”). Each data “repository” can contain one file, multiple files, multiple files arranged in directories, etc. Regardless of the structure, Pachyderm will version the state of each data repository as it changes over time.

“Commits”

Regardless of the method you use to get data into Pachyderm (CLI, language client, etc.), the mechanism that is used is a “commit” of data into a data repository. In order to put data into Pachyderm, a commit must be “started” (aka an “open commit”). Data can then be put into Pachyderm as part of that open commit and will be available once the commit is “finished” (aka a “closed commit”). Although you have to do this opening, putting, and closing for all data that is committed into Pachyderm, we have built in some convenient ways to do that with our CLI tool and clients (see below).

How to get data into Pachyderm

In terms of actually getting data into Pachyderm via “commits,” there are a few options:

- *Via the `pachctl` CLI tool*: This is the great option for testing, development, integration with CI/CD, and for users who prefer scripting.
- *Via one of the Pachyderm language clients*: This option is ideal for Go, Python, or Scala users who want to push data to Pachyderm from services or applications written in those languages. Actually, even if you don’t use Go, Python, or Scala, Pachyderm uses a protobuf API which supports many other languages, we just haven’t built the full clients yet.
- *Via the Pachyderm dashboard*: The Pachyderm Enterprise dashboard provides a very convenient way to upload data right from the GUI. You can find out more about Pachyderm Enterprise Edition [here](#).

pachctl

To get data into Pachyderm using `pachctl`, you first need to create one or more data repositories to hold your data:

```
$ pachctl create-repo <repo name>
```

Then, to put data into the created repo, you use the `put-file` command. Below are a few example uses of `put-file`, but you can see the complete documentation here. Note again, commits in Pachyderm must be explicitly started and finished, so `put-file` can only be called on an open commit (started, but not finished). The `-c` is a convenient option that allows you to start and finish a commit in addition to putting data as a one-line command.

Add a single file to a new branch:

```
# first start a commit
$ pachctl start-commit <repo> <branch>

# then put <file> at <path> in the <repo> on <branch>
$ pachctl put-file <repo> <branch> </path/to/file> -f <file>

# then finish the commit
$ pachctl finish-commit <repo> <branch>
```

Start and finish a commit while adding a file using `-c`:

```
$ pachctl put-file <repo> <branch> </path/to/file> -c -f <file>
```

Put data from a URL:

```
$ pachctl put-file <repo> <branch> </path/to/file> -c -f http://url_path
```

Put data directly from an object store:

```
# here you can use s3://, gcs://, or as://
$ pachctl put-file <repo> <branch> </path/to/file> -c -f s3://object_store_url
```

Put data directly from another location within Pachyderm:

```
$ pachctl put-file <repo> <branch> </path/to/file> -c -f pfs://pachyderm_location
```

Add multiple files at once by using the `-i` option or multiple `-f` flags. In the case of `-i`, the target file should be a list of files, paths, or URLs that you want to input all at once:

```
$ pachctl put-file <repo> <branch> -c -i <file containing list of files, paths, or_
↪URLs>
```

Pipe data from stdin into a data repository:

```
$ echo "data" | pachctl put-file <repo> <branch> </path/to/file> -c
```

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> -c -r <dir>
```

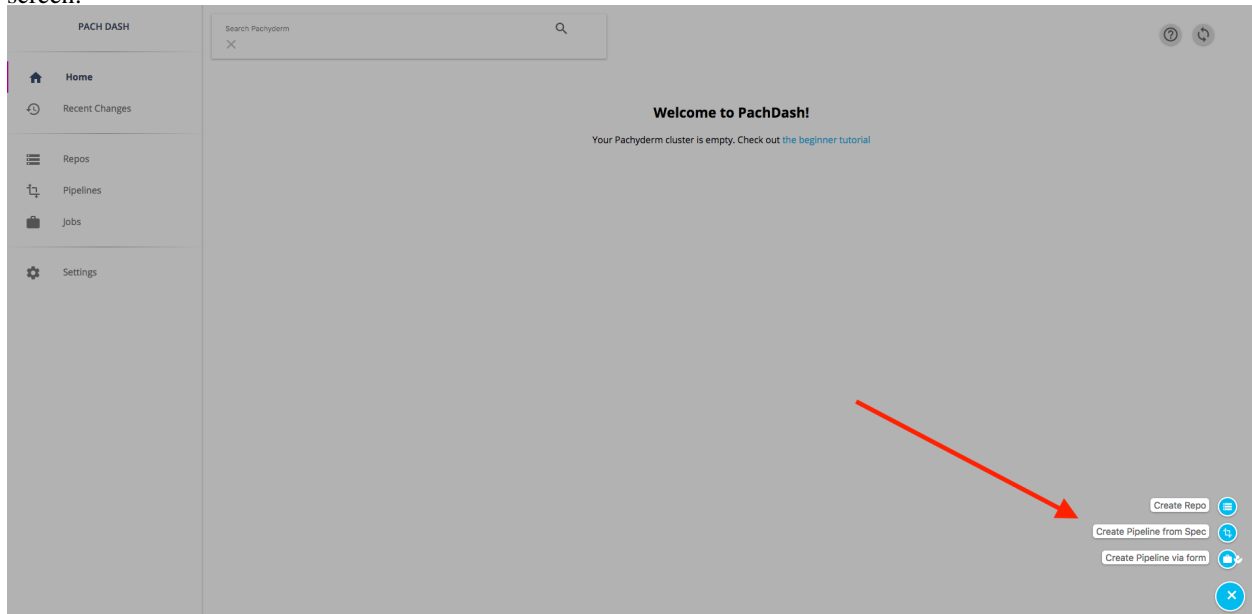

Pachyderm Language Clients

There are a number of Pachyderm language clients. These can be used to programmatically put data into Pachyderm, and much more. You can find out more about these clients [here](#).

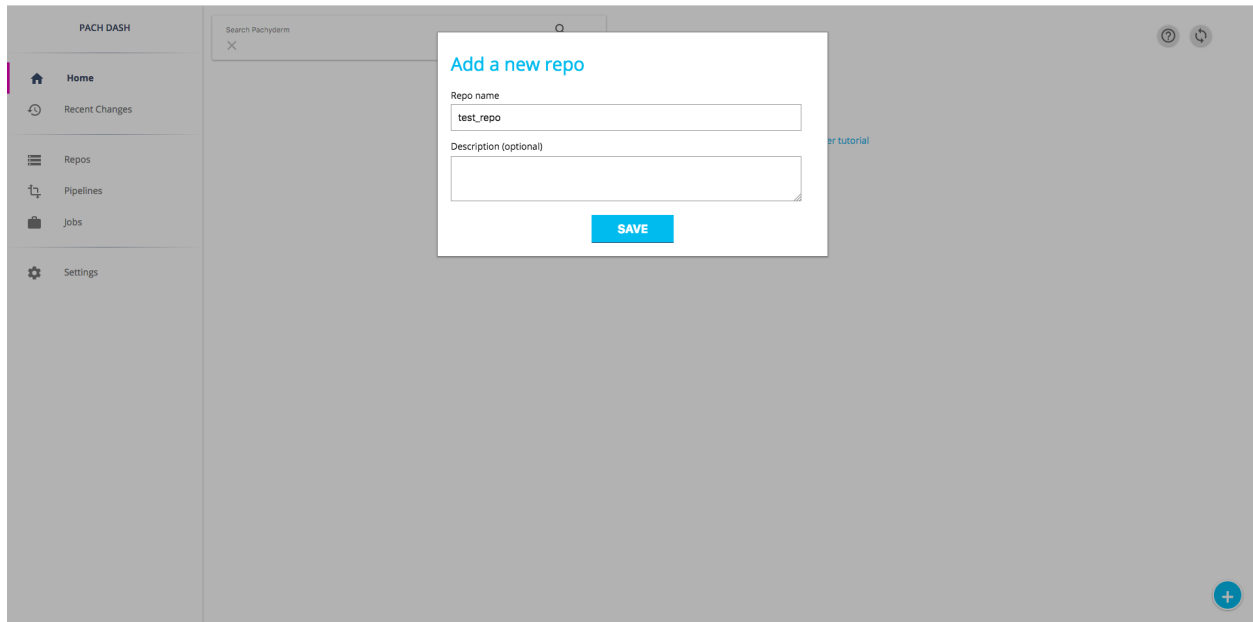
The Pachyderm Dashboard

When you deployed Pachyderm, the Pachyderm Enterprise dashboard was also deployed automatically (if you followed one of our [deploy guides](#) here). You can get a FREE trial token to experiment with this dashboard, which will let you create data repositories and add data to those repositories via a GUI. More information about getting your FREE trial token and activating the dashboard can be found [here](#).

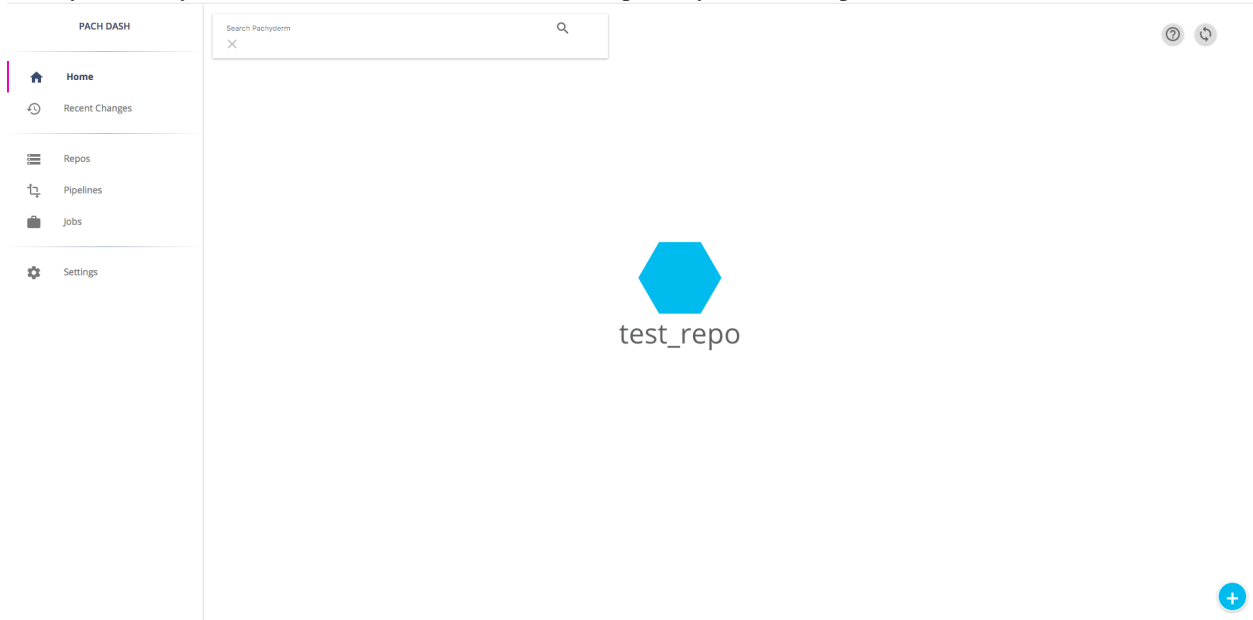
In the dashboard, you can create a data repository by clicking on the + sign icon in the lower right hand corner of the screen:



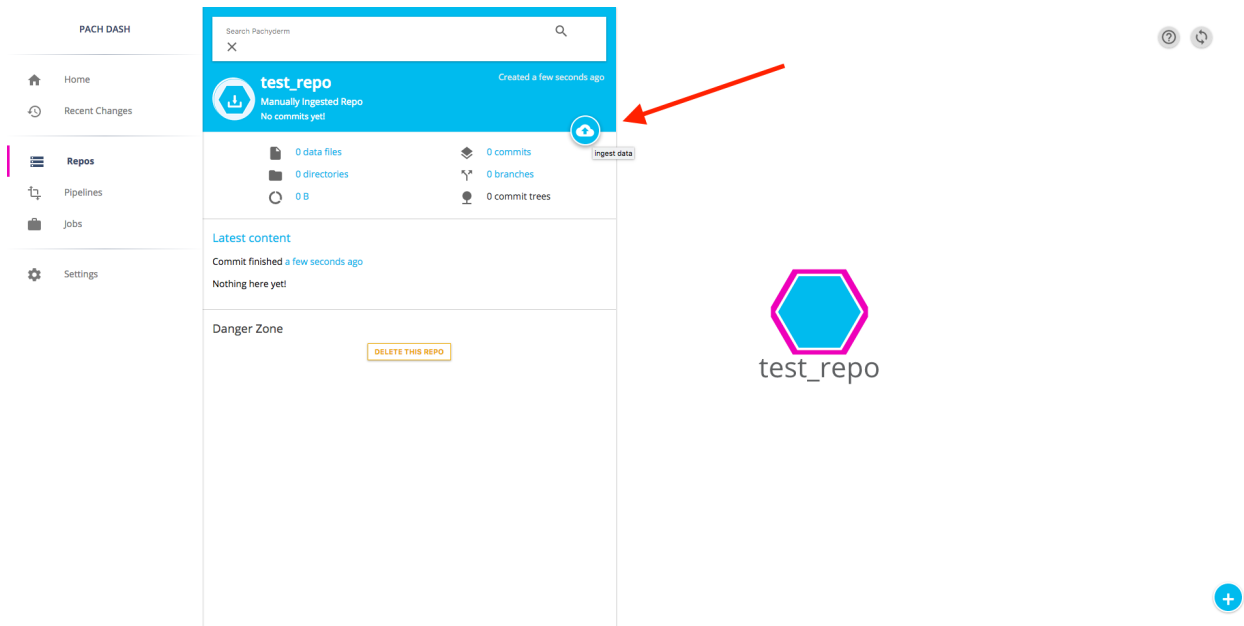
When you click “Create Repo,” a box will pop up prompting you for a name and optional description for the repo:



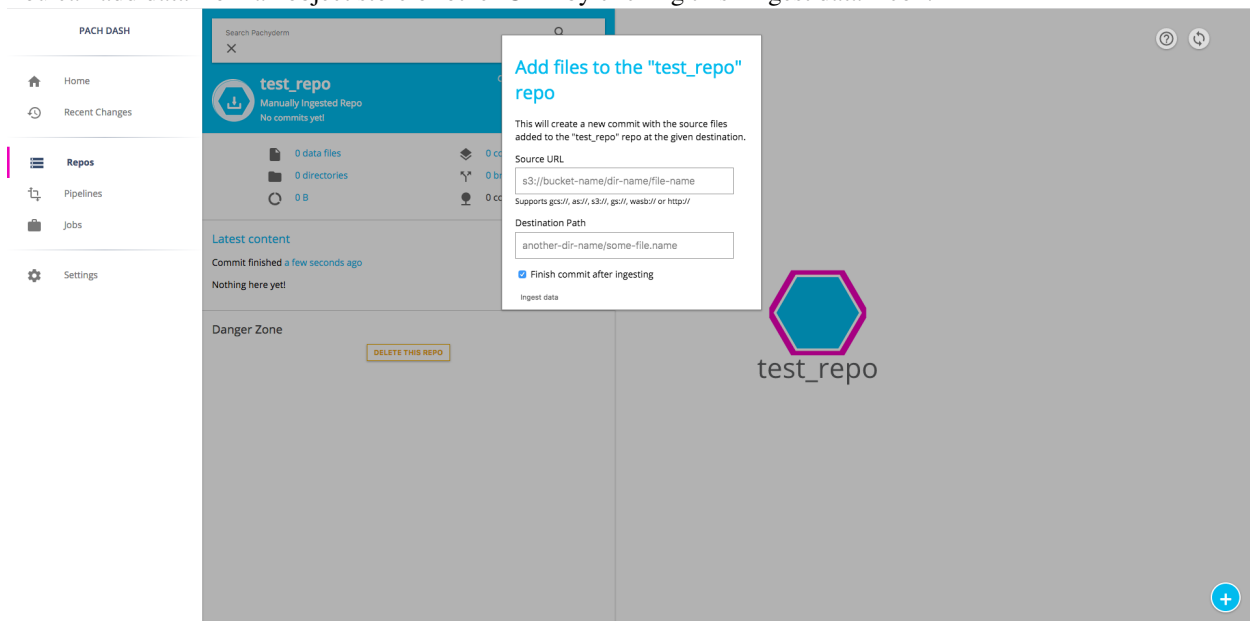
Once you fill in your name and click save, the new data repository will show up in the main dashboard screen:



To add data to this repository, you can click on the blue icon representing the repo. This will present you with some details about the repo along with an “ingest data” icon:



You can add data from an object store or other URL by clicking this “ingest data” icon:



Creating Analysis Pipelines

There are three steps to running an analysis in a Pachyderm “pipeline”:

1. Write your code.
2. Build a [Docker](#) image that includes your code and dependencies.
3. Create a Pachyderm “pipeline” referencing that Docker image.

Multi-stage pipelines (e.g., parsing -> modeling -> output) can be created by repeating these three steps to build up a graph of processing steps.

1. Writing your analysis code

Code used to process data in Pachyderm can be written using any languages or libraries you want. It can be as simple as a bash command or as complicated as a TensorFlow neural network. At the end of the day, all your code and dependencies will be built into a container that can run anywhere (including inside of Pachyderm). We’ve got demonstrative [examples on GitHub](#) using bash, Python, TensorFlow, and OpenCV and we’re constantly adding more.

As we touch on briefly in the beginner tutorial, your code itself only needs to read and write files from a local file system. It does NOT have to import any special Pachyderm functionality or libraries. You just need to be able to read files and write files.

For the reading files part, Pachyderm automatically mounts each input data repository as `/pfs/<repo_name>` in the running instances of your Docker image (called “containers”). The code that you write just needs to read input data from this directory, just like in any other file system. Your analysis code also does NOT have to deal with data sharding or parallelization as Pachyderm will automatically shard the input data across parallel containers. For example, if you’ve got four containers running your Python code, Pachyderm will automatically supply 1/4 of the input data to `/pfs/<repo_name>` in each running container. That being said, you also have a lot of control over how that input data is split across containers. Check out our [guide on parallelism and distributed computing](#) for more details on that subject.

For the writing files part (saving results, etc.), your code simply needs to write to `/pfs/out`. This is a special directory mounted by Pachyderm in all of your running containers. Similar to reading data, your code doesn’t have to manage parallelization or sharding, just write data to `/pfs/out` and Pachyderm will make sure it all ends up in the correct place.

2. Building a Docker Image

When you create a Pachyderm pipeline (which will be discussed next), you need to specify a Docker image including the code or binary you want to run. Please refer to the [official documentation](#) to learn how to build a Docker images.

Note: You specify what commands should run in the container in your pipeline specification (see **Creating a Pipeline** below) rather than the `CMD` field of your Dockerfile, and Pachyderm runs that command inside the container during jobs rather than relying on Docker to run it. The reason is that Pachyderm can't execute your code immediately when your container starts, so it runs a shim process in your container instead, and then calls your pipeline specification's `cmd` from there.

Unless Pachyderm is running on the same host that you used to build your image, you'll need to use a public or private registry to get your image into the Pachyderm cluster. One (free) option is to use Docker's DockerHub registry. You can refer to the [official documentation](#) to learn how to push your images to DockerHub. That being said, you are more than welcome to use any other public or private Docker registry.

Note, it is best practice to uniquely tag your Docker images with something other than `:latest`. This allows you to track which Docker images were used to process which data, and will help you as you update your pipelines. You can also utilize the `--push-images` flag on `update-pipeline` to help you tag your images as they are updated. See the updating pipelines docs for more information.

3. Creating a Pipeline

Now that you've got your code and image built, the final step is to tell Pachyderm to run the code in your image on certain input data. To do this, you need to supply Pachyderm with a JSON pipeline specification. There are four main components to a pipeline specification: name, transform, parallelism and input. Detailed explanations of the specification parameters and how they work can be found in the pipeline specification docs.

Here's an example pipeline spec:

```
{
  "pipeline": {
    "name": "wordcount"
  },
  "transform": {
    "image": "wordcount-image",
    "cmd": ["/binary", "/pfs/data", "/pfs/out"]
  },
  "input": {
    "atom": {
      "repo": "data",
      "glob": "/*"
    }
  }
}
```

After you create the JSON pipeline spec (and save it, e.g., as `your_pipeline.json`), you can create the pipeline in Pachyderm using `pachctl`:

```
$ pachctl create-pipeline -f your_pipeline.json
```

(`-f` can also take a URL if your JSON manifest is hosted on GitHub or elsewhere. Keeping pipeline specifications under version control is a great idea so you can track changes and seamlessly view or deploy older pipelines if needed.)

Creating a pipeline tells Pachyderm to run the `cmd` (i.e., your code) in your `image` on the data in the HEAD (most recent) commit of the input repo(s) as well as *all future commits* to the input repo(s). You can think of this pipeline as being “subscribed” to any new commits that are made on any of its input repos. It will automatically process the new data as it comes in.

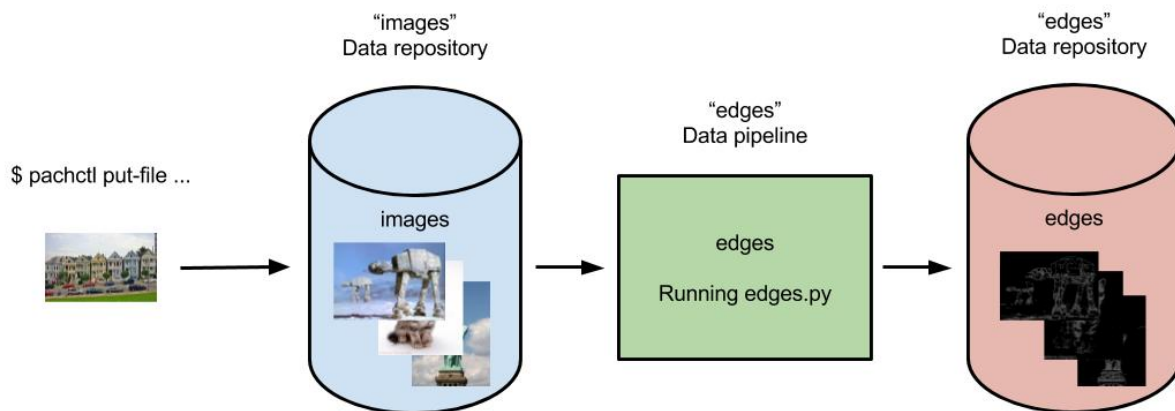
As soon as you create your pipeline, Pachyderm will launch worker pods on Kubernetes. These worker pods will remain up and running, such that they are ready to process any data committed to their input repos. This allows the

pipeline to immediately respond to new data when it's committed without having to wait for their pods to "spin up". However, this has the downside that pods will consume resources even while there's no data to process. You can trade-off the other way by setting the `standby` field to true in your pipeline spec. With this field set, the pipelines will "spin down" when there is no data to process, which means they will consume no resources. However, when new data does come in, the pipeline pods will need to spin back up, which introduces some extra latency. Generally speaking, you should default to not setting `standby` until cluster utilization becomes a concern. When it does, pipelines that run infrequently and are highly parallel are the best candidates for `standby`.

Getting Data Out of Pachyderm

Once you’ve got one or more pipelines built and have data flowing through Pachyderm, you need to be able to track that data flowing through your pipeline(s) and get results out of Pachyderm. Let’s use the OpenCV pipeline as an example.

Here’s what our pipeline and the corresponding data repositories look like:



Every commit of new images into the “images” data repository results in a corresponding output commit of results into the “edges” data repository. But how do we get our results out of Pachyderm? Moreover, how would we get the particular result corresponding to a particular input image? That’s what we will explore here.

Getting files with `pachctl`

The `pachctl` CLI tool command `get-file` can be used to get versioned data out of any data repository:

```
pachctl get-file <repo> <commit-id or branch> path/to/file
```

In the case of the OpenCV pipeline, we could get out an image named `example_pic.jpg`:

```
pachctl get-file edges master example_pic.jpg
```

But how do we know which files to get? Of course we can use the `pachctl list-file` command to see what files are available. But how do we know which results are the latest, came from certain input, etc.? In this case, we would like to know which edge detected images in the `edges` repo come from which input images in the `images` repo. This is where provenance and the `flush-commit` command come in handy.

Examining file provenance with flush-commit

Generally, `flush-commit` will let our process block on an input commit until all of the output results are ready to read. In other words, `flush-commit` lets you view a consistent global snapshot of all your data at a given commit. Note, we are just going to cover a few aspects of `flush-commit` here.

Let's demonstrate a typical workflow using `flush-commit`. First, we'll make a few commits of data into the `images` repo on the `master` branch. That will then trigger our `edges` pipeline and generate three output commits in our `edges` repo:

```
$ pachctl list-commit images
REPO      ID                                     PARENT
→ STARTED          DURATION          SIZE
images    c721c4bb9a8046f3a7319ed97d256bb9
→ a9678d2a439648c59636688945f3c6b5    About a minute ago    1 seconds    932.2 KiB
images    a9678d2a439648c59636688945f3c6b5
→ 87f5266ef44f4510a7c5e046d77984a6    About a minute ago    Less than a second    238.3 KiB
images    87f5266ef44f4510a7c5e046d77984a6    <none>
→ 10 minutes ago          Less than a second    57.27 KiB
$ pachctl list-commit edges
REPO      ID                                     PARENT
→ STARTED          DURATION          SIZE
edges     f716eabf95854be285c3ef23570bd836
→ 026536b547a44a8daa2db9d25bf88b79    About a minute ago    Less than a second    233.7 KiB
edges     026536b547a44a8daa2db9d25bf88b79
→ 754542b89c1c47a5b657e60381c06c71    About a minute ago    Less than a second    133.6 KiB
edges     754542b89c1c47a5b657e60381c06c71    <none>
→ 2 minutes ago          Less than a second    22.22 KiB
```

In this case, we have one output commit per input commit on `images`. However, this might get more complicated for pipelines with multiple branches, multiple input atoms, etc. To confirm which commits correspond to which outputs, we can use `flush-commit`. In particular, we can call `flush-commit` on any one of our commits into `images` to see which output came from this particular commit:

```
$ pachctl flush-commit images/a9678d2a439648c59636688945f3c6b5
REPO      ID                                     PARENT
→ STARTED          DURATION          SIZE
edges     026536b547a44a8daa2db9d25bf88b79
→ 754542b89c1c47a5b657e60381c06c71    3 minutes ago          Less than a second    133.6 KiB
```

Exporting data via egress

In addition to getting data out of Pachyderm with `pachctl get-file`, you can add an optional `egress` field to your pipeline specification. `egress` allows you to push the results of a Pipeline to an external data store such as S3, Google Cloud Storage or Azure Blob Storage. Data will be pushed after the user code has finished running but before the job is marked as successful.

Other ways to view, interact with, or export data in Pachyderm

Although `pachctl` and `egress` provide easy ways to interact with data in Pachyderm repos, they are by no means the only ways. For example, you can:

- Have one or more of your pipeline stages connect and export data to databases running outside of Pachyderm.
- Use a Pachyderm service to launch a long running service, like Jupyter, that has access to internal Pachyderm data and can be accessed externally via a specified port.
- Mount versioned data from the distributed file system via `pachctl mount ...` (a feature best suited for experimentation and testing).

Deleting Data in Pachyderm

Sometimes “bad” data gets committed to Pachyderm and you need a way to delete it. There are a couple of ways to address this, which depend on what exactly was “bad” about the data you committed and what’s happened in the system since you committed the “bad” data.

- *Deleting the HEAD of a branch* - You should follow this guide if you’ve just made a commit to a branch with some corrupt, incorrect, or otherwise bad changes to your data.
- *Deleting non-HEAD commits* - You should follow this guide if you’ve committed data to the branch after committing the data that needs to be deleted.
- *Deleting sensitive data* - You should follow these steps when you have committed sensitive data that you need to completely purge from Pachyderm, such that no trace remains.

Deleting The HEAD of a Branch

The simplest case is when you’ve just made a commit to a branch with some incorrect, corrupt, or otherwise bad data. In this scenario, the HEAD of your branch (i.e., the latest commit) is bad. Users who read from it are likely to be misled, and/or pipeline subscribed to it are likely to fail or produce bad downstream output.

To fix this you should use `delete-commit` as follows:

```
$ pachctl delete-commit <repo> <branch-or-commit-id>
```

When you delete the bad commit, several things will happen (all atomically):

- The commit metadata will be deleted.
- Any branch that the commit was the HEAD of will have its HEAD set to the commit’s parent. If the commit’s parent is `nil`, the branch’s HEAD will be set to `nil`.
- If the commit has children (commits which it is the parent of), those children’s parent will be set to the deleted commit’s parent. Again, if the deleted commit’s parent is `nil` then the children commit’s parent will be set to `nil`.
- Any jobs which were created due to this commit will be deleted (running jobs get killed). This includes jobs which don’t directly take the commit as input, but are farther downstream in your DAG.
- Output commits from deleted jobs will also be deleted, and all the above effects will apply to those commits as well.

Deleting Non-HEAD Commits

Recovering from commits of bad data is a little more complicated if you've committed more data to the branch after the bad data was added. You can still delete the commit as in the previous section, however, unless the subsequent commits overwrote or deleted the bad data, it will still be present in the children commits. *Deleting a commit does not modify its children.*

In git terms, `delete-commit` is equivalent to squashing a commit out of existence. It's not equivalent to reverting a commit. The reason for this behavior is that the semantics of revert can get ambiguous when the files being reverted have been otherwise modified. Git's revert can leave you with a merge conflict to solve, and merge conflicts don't make sense with Pachyderm due to the shared nature of the system and the size of the data being stored.

In these scenario, you can also delete the children commits, however those commits may also have good data that you don't want to delete. If so, you should:

1. Start a new commit on the branch with `pachctl start-commit`.
2. Delete all bad files from the newly opened commit with `pachctl delete-file`.
3. Finish the commit with `pachctl finish-commit`.
4. Delete the initial bad commits and all children up to the newly finished commit.

Depending on how you're using Pachyderm, the final step may be optional. After you finish the "fixed" commit, the HEADs of all your branches will converge to correct results as downstream jobs finish. However, deleting those commits allow you to clean up your commit history and makes sure that no one will ever access errant data when reading non-HEAD version of the data.

Deleting Sensitive Data

If the data you committed is bad because it's sensitive and you want to make sure that nobody ever accesses it, you should complete an extra step in addition to those above.

Pachyderm stores data in a content addressed way and when you delete a file or a commit, Pachyderm only deletes references to the underlying data, it doesn't delete the actual data until it performs garbage collection. To truly purge the data you must delete all references to it using the methods described above, and then you must run a garbage collect with `pachctl garbage-collect`.

Updating Pipelines

During development, it's very common to update pipelines, whether it's changing your code or just cranking up parallelism. For example, when developing a machine learning model you will likely need to try out a bunch of different versions of your model while your training data stays relatively constant. This is where `update-pipeline` comes in.

Updating your pipeline specification

In cases in which you are updating parallelism, adding another input repo, or otherwise modifying your pipeline specification, you just need to update your JSON file and call `update-pipeline`:

```
$ 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 elsewhere.

Updating the code used in a pipeline

You can also use `update-pipeline` to update the code you are using in one or more of your pipelines. To update the code in your pipeline:

1. Make the code changes.
2. Re-build your Docker image.
3. Call `update-pipeline` with the `--push-images` flag.

You need to call `update-pipeline` with the `--push-images` flag because, if you have already run your pipeline, Pachyderm has already pulled the specified images. It won't re-pull new versions of the images, unless we tell it to (which ensures that we don't waste time pulling images when we don't need to). When `--push-images` is specified, Pachyderm will do the following:

1. Tag your image with a new unique tag.
2. Push that tagged image to your registry (e.g., DockerHub).
3. Update the pipeline specification that you previously gave to Pachyderm with the new unique tag.

For example, you could update the Python code used in the OpenCV pipeline via:

```
pachctl update-pipeline -f edges.json --push-images --password <registry password> -u  
↔<registry user>
```

Re-processing data

As of 1.5.1, updating a pipeline will NOT reprocess previously processed data by default. New data that's committed to the inputs will be processed with the new code and “mixed” with the results of processing data with the previous code. Furthermore, data that Pachyderm tried and failed to process with the previous code due to code erroring will be processed with the new code.

`update-pipeline` (without flags) is designed for the situation where your code needs to be fixed because it encountered an unexpected new form of data.

If you'd like to update your pipeline and have that updated pipeline reprocess all the data that is currently in the HEAD commit of your input repos, you should use the `--reprocess` flag. This type of update will automatically trigger a job that reprocesses all of the input data in its current state (i.e., the HEAD commits) with the updated pipeline. Then from that point on, the updated pipeline will continue to be used to process any new input data. Previous results will still be available in via their corresponding commit IDs.

Distributed Computing

Distributing computation across multiple workers is a fundamental part of processing any big data or computationally intensive workload. There are two main questions to think about when trying to distribute computation:

1. *How many workers to spread computation across?*
2. *How to define which workers are responsible for which data?*

Pachyderm Workers

Before we dive into the above questions, there are a few details you should understand about Pachyderm workers.

Every worker for a given pipeline is an identical pod running the Docker image you specified in the pipeline spec. Your analysis code does not need do anything special to run in a distributed fashion. Instead, Pachyderm will spread out the data that needs to be processed across the various workers and make that data available for your code.

Pachyderm workers are spun up when you create the pipeline and are left running in the cluster waiting for new jobs (data) to be available for processing (committed). This saves having to recreate and schedule the worker for every new job.

Controlling the Number of Workers (Parallelism)

The number of workers that are used for a given pipeline is controlled by the `parallelism_spec` defined in the pipeline specification.

```
"parallelism_spec": {  
  // Exactly one of these two fields should be set  
  "constant": int  
  "coefficient": double
```

Pachyderm has two parallelism strategies: `constant` and `coefficient`. You should set one of the two corresponding fields in the `parallelism_spec`, and pachyderm chooses a parallelism strategy based on which field is set.

If you set the `constant` field, Pachyderm will start the number of workers that you specify. For example, set `"constant": 10` to use 10 workers.

If you set the `coefficient` field, Pachyderm will start 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 will start five workers. If you set it to 2.0, Pachyderm will start 20 workers (two per Kubernetes node).

NOTE: The `parallelism_spec` is optional and will default to `"coefficient": 1`, which means that it'll spawn one worker per Kubernetes node for this pipeline if left unset.

Spreading Data Across Workers (Glob Patterns)

Defining how your data is spread out among workers is arguably the most important aspect of distributed computation and is the fundamental idea around concepts like Map/Reduce.

Instead of confining users to just data-distribution patterns such as Map (split everything as much as possible) and Reduce (*all* the data must be grouped together), Pachyderm uses [Glob Patterns](#) to offer incredible flexibility in defining your data distribution.

Glob patterns are defined by the user for each `atom` within the `input` of a pipeline, and they tell Pachyderm how to divide the input data into individual “datums” that can be processed independently.

```
"input": {
  "atom": {
    "repo": "string",
    "glob": "string",
  }
}
```

That means you could easily define multiple “atoms”, one with the data highly distributed and another where it’s grouped together. You can then join the datums in these atoms via a cross product or union (as shown above) for combined, distributed processing.

```
"input": {
  "cross" or "union": [
    {
      "atom": {
        "repo": "string",
        "glob": "string",
      }
    },
    {
      "atom": {
        "repo": "string",
        "glob": "string",
      }
    },
    etc...
  ]
}
```

More information about “atoms,” unions, and crosses can be found in our [Pipeline Specification](#).

Datums

Pachyderm uses the glob pattern to determine how many “datums” an input atom consists of. Datums are the unit of parallelism in Pachyderm. That is, Pachyderm attempts to process datums in parallel whenever possible.

If you have two workers and define 2 datums, Pachyderm will send one datum to each worker. In a scenario where there are more datums than workers, Pachyderm will queue up extra datums and send them to workers as they finish processing previous datums.

Defining Datums via Glob Patterns

Intuitively, you should think of the input atom repo as a file system where the glob pattern is being applied to the root of the file system. The files and directories that match the glob pattern are considered datums.

For example, a glob pattern of just `/` would denote the entire input repo as a single datum. All of the input data would be given to a single worker similar to a typical reduce-style operation.

Another commonly used glob pattern is `/*`. `/*` would define each top level object (file or directory) in the input atom repo as its own datum. If you have a repo with just 10 files in it and no directory structure, every file would be a datum and could be processed independently. This is similar to a typical map-style operation.

But Pachyderm can do anything in between too. If you have a directory structure with each state as a directory and a file for each city such as:

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

and you need to process all the data for a given state together, `/*` would also be the desired glob pattern. You'd have one datum per state, meaning all the cities for a given state would be processed together by a single worker, but each state can be processed independently.

If we instead used the glob pattern `/*/` for the states example above, each `<city>.json` would be its own datum.

Glob patterns also let you take only a particular directory (or subset of directories) as an input atom instead of the whole input repo. If we create a pipeline that is specifically only for California, we can use a glob pattern of `/California/*` to only use the data in that directory as input to our pipeline.

Only Processing New Data

A datum defines the granularity at which Pachyderm decides what data is new and what data has already been processed. Pachyderm will never reprocess datums it's already seen with the same analysis code. But if any part of a datum changes, the entire datum will be reprocessed.

Note: If you change your code (or pipeline spec), Pachyderm will of course allow you to process all of the past data through the new analysis code.

Let's look at our states example with a few different glob patterns to demonstrate what gets processed and what doesn't. Suppose we have an input data layout such as:

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

If our glob pattern is `/`, then the entire input atom is a single datum, which means anytime any file or directory is changed in our input, all the the data will be processed from scratch. There are plenty of usecases where this is exactly what we need (e.g. some machine learning training algorithms).

If our glob pattern is `/*`, then each state directory is it's own datum and we'll only process the ones that have changed. So if we add a new city file, `Sacramento.json` to the `/California` directory, *only* the California datum, will be reprocessed.

If our glob pattern was `/*/*` then each `<city>.json` file would be it's own datum. That means if we added a `Sacramento.json` file, only that specific file would be processed by Pachyderm.

Incremental Processing

Pachyderm performs computations in an incremental fashion. That is, rather than computing a result all at once, it computes it in small pieces and then stitches those pieces together to form results. This allows Pachyderm to reuse results and compute things much more efficiently than traditional systems, which are forced to compute everything from scratch during every job.

Pachyderm supports two kinds of incremental processing:

1. *Inter-Datum Incrementality*
2. *Intra-Datum Incrementality*

If you are new to the idea of Pachyderm “datums,” you can learn more [here](#).

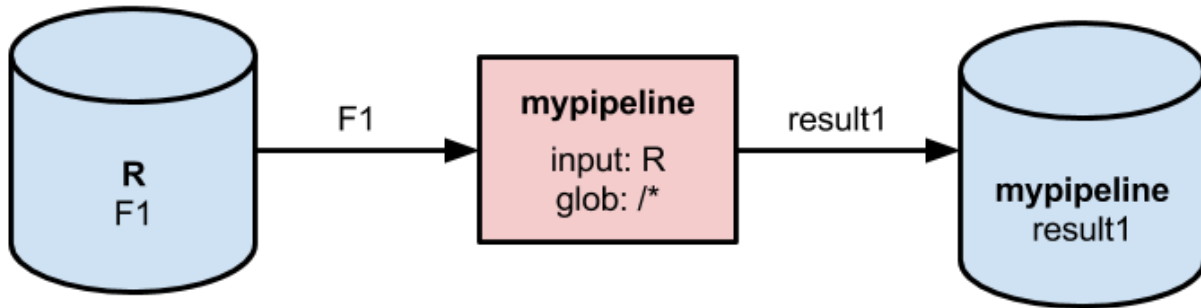
Inter-datum Incrementality

Each of the input datums in a Pachyderm pipeline is processed in isolation, and the results of these isolated computations are combined to create the final result. Pachyderm will never process the same datum twice (unless you update a pipeline with the `--reprocess` flag). If you commit new data in Pachyderm that leaves some of the previously existing datums intact, the results of processing those pre-existing datums in a previous job will also remain intact. That is, the previous results for those pre-existing datums won’t be recalculated.

This inter-datum incrementality is best illustrated with an example. Suppose we have a pipeline with a single input that looks like this:

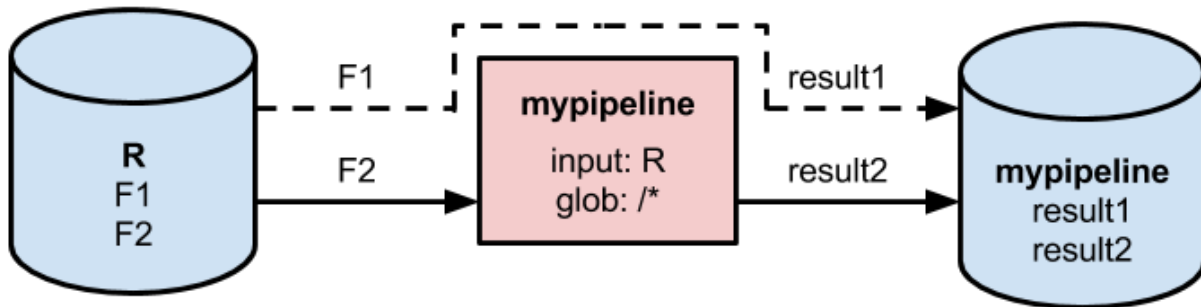
```
{
  "atom": {
    "repo": "R",
    "glob": "/*",
  }
}
```

Now, suppose you make a commit to `R` which adds a single file `F1`. Your pipeline will run a job, and that job will find a single datum to process (`F1`). This datum will be processed, because it’s the first time the pipeline has seen `F1`.

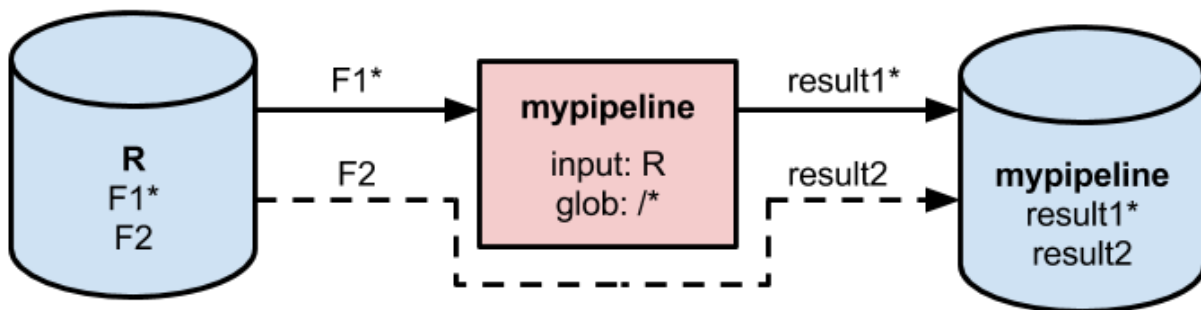


If you then make a second commit to `R` adding another file `F2`, the pipeline will run a second job. This job will find two datums to process (`F1` and `F2`). `F2` will be processed, because it hasn't been seen before. However `F1` will NOT be processed, because an output from processing it already exists in Pachyderm.

Instead, the output from the previous job for `F1` will be combined with the new result from processing `F2` to create the output of this second job. This reuse of the result for `F1` effectively halves the amount of work necessary to process the second commit.



Finally, suppose you make a third commit to `R`, which modifies `F1`. Again you'll have a job that sees two datums (the new `F1` and the already processed `F2`). This time `F2` won't get processed, but the new `F1` will be processed because it has different content as compared to the old `F1`.



Note, you as a user don't need to do anything to enable this inter-datum incrementality. It happens automatically, and it should be transparent from your perspective. In the above example, you get the same result you would have gotten if you committed the same data in a single commit.

As of Pachyderm v1.5.1, `list-job` and `inspect-job` will tell you how many datums the job processed and how many it skipped. Below is an example of a job that had 5 datums, 3 that were processed and 2 that were skipped.

ID	DURATION	OUTPUT COMMIT	DL	UL	STATE
↪ STARTED		RESTART PROGRESS			

```
54fbc366-3f11-41f6-9000-60fc8860fa55 pipeline/9c348deb64304d118101e5771e18c2af 13
↪seconds ago      10 seconds      0      3 + 2 / 5      0B      0B      success
```

Intra-datum Incrementality

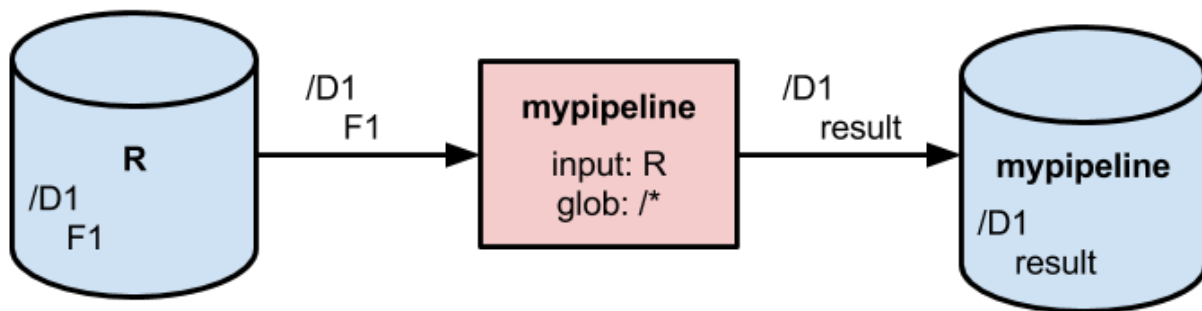
Pachyderm also supports intra-datum incrementality, which is useful when the processing you're doing can be done "online". For example, when you are performing online training of a model or when you are summing a set of numbers in an aggregation.

Not all computations can be done online. Thus, this intra-datum incrementality is optionally enabled for Pachyderm pipelines via the `incremental` field in the pipeline specification.

Again, an example is instructive. Suppose you have a pipeline like the one illustrated above in the *inter-datum* section. However, instead of each datum being a single file, it is now a directory (D1 , D2 , etc.) which contains multiple files (F1 , F2 , etc.).

Each of these files in the directories contain numbers, and our pipeline sums the numbers in all of the files to produce a `result` that includes the sum of the numbers per directory. The pipeline also enables incrementality via the `incremental` field.

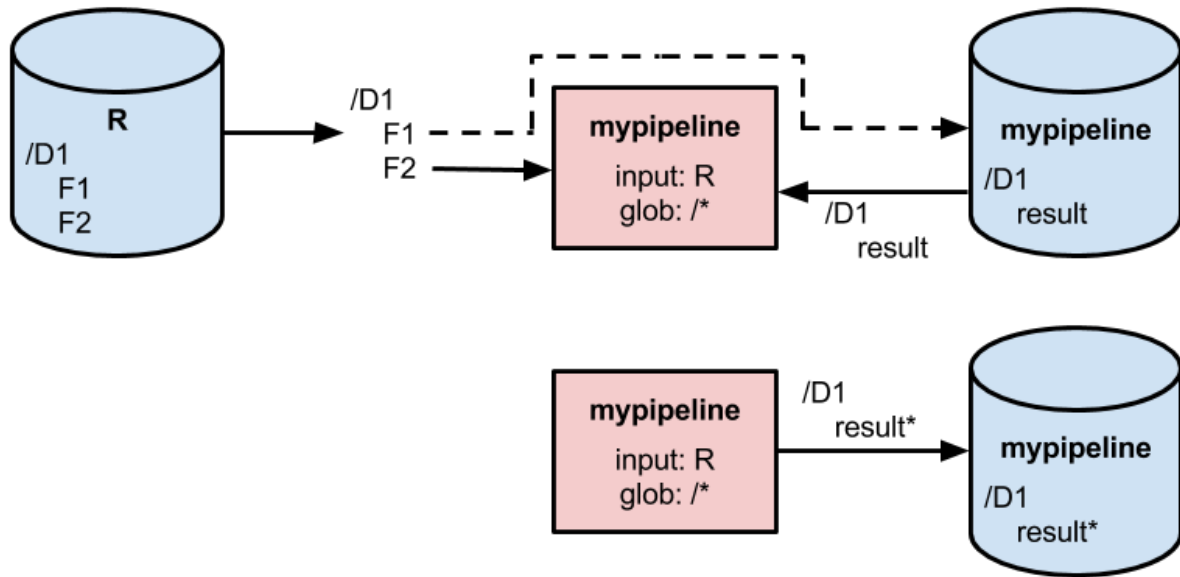
In a first commit, we add D1/F1 . Our pipeline will run a job which sums up the numbers in all of the files in D1 (in this case, just D1/F1). This is similar to what would happen if the pipeline did not enable intra-datum incrementality.



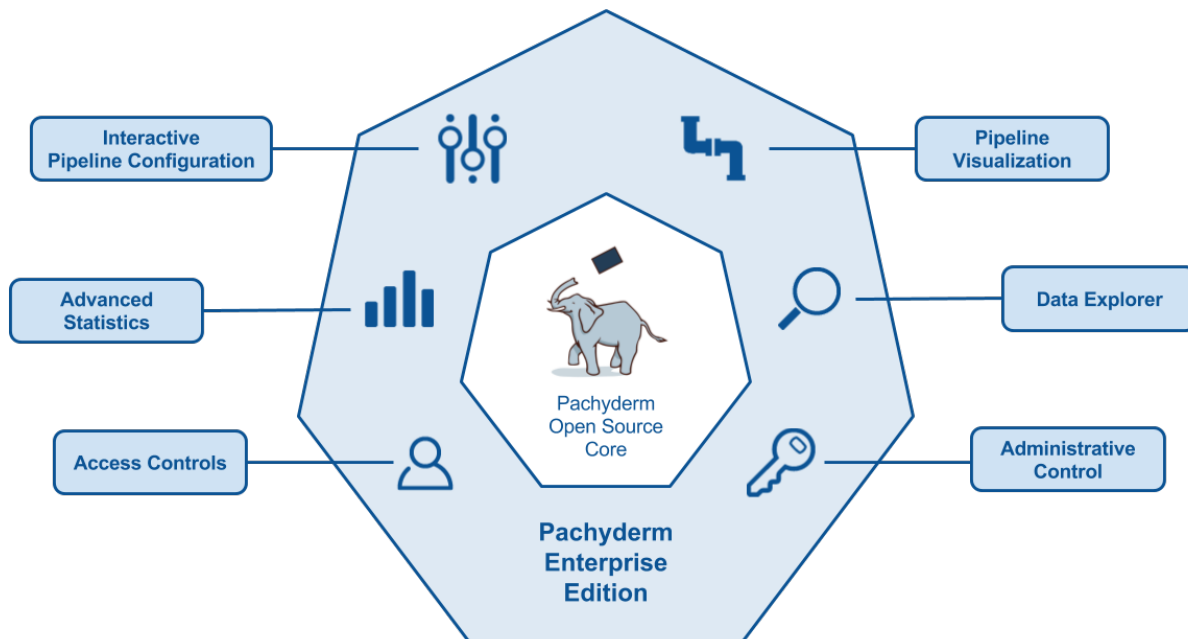
Then, in a second commit, we add D1/F2 . Another job will be triggered process D1 . However, the data it sees in D1 will be different from what it would be if the pipeline weren't incremental . Instead of seeing D1/F1 and D1/F2 , it will only see D1/F2 .

Moreover, the output directory, `/pfs/out` , won't be empty. `/pfs/out` will contain the results of last job that processed D1 . That is, it will contain the sum of all the numbers in D1/F1 .

As such, all our code needs to do is sum the numbers in D1/F2 and add them to the previous `result` , which we can access at `/pfs/out/D1/result` . We can then overwrite the previous `result` in `/pfs/out` with the new result.



Overview

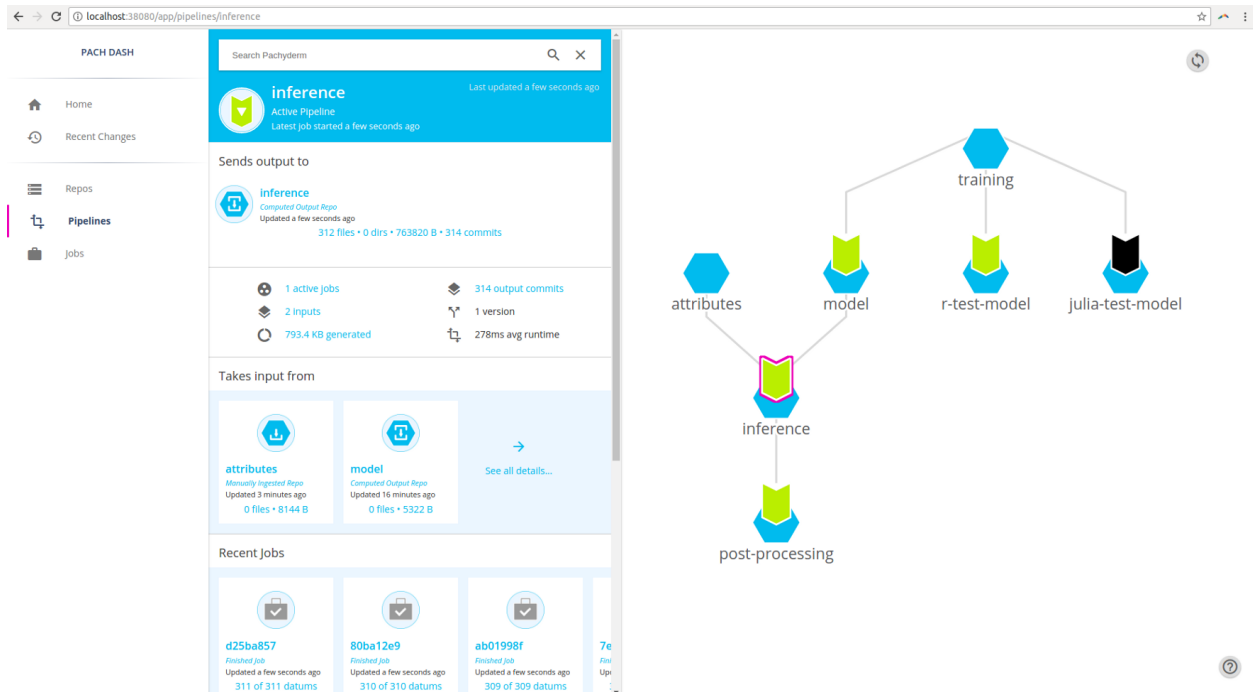


Pachyderm Enterprise Edition includes everything you need to scale and manage Pachyderm data pipelines in an enterprise setting. It delivers the most recent version of Pachyderm along with:

- 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

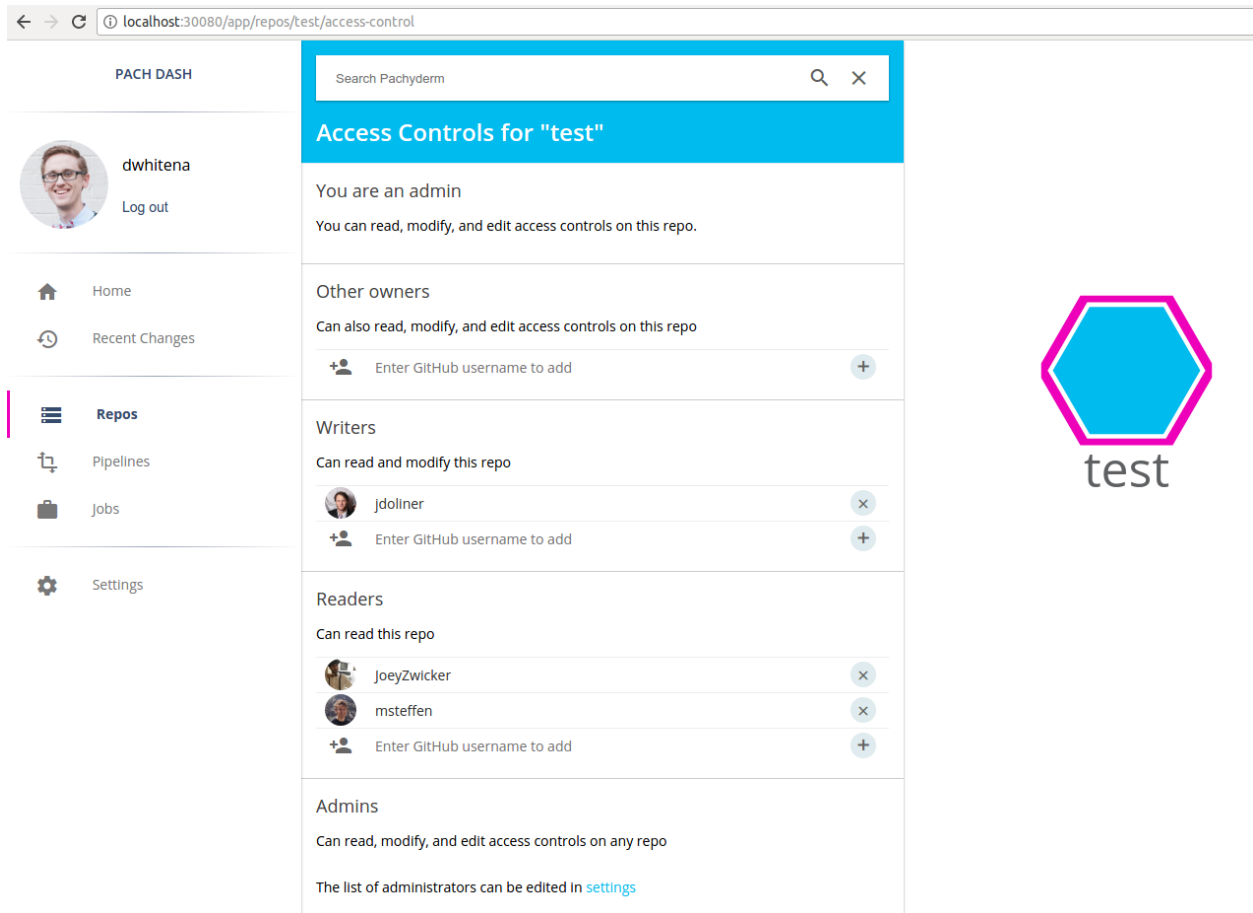
Pachyderm Enterprise Edition can be deployed easily on top of an existing or new deployment of Pachyderm, and we have engineers available to help enterprise customers get up and running very quickly. To get more information about Pachyderm Enterprise Edition, to ask questions, or to get access for evaluation,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

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.

Access Controls



The screenshot shows 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):

- Profile:** dwhitena, Log out
- Navigation:** Home, Recent Changes, **Repos** (active), Pipelines, Jobs, Settings

Main Content Area: Access Controls for "test"

Search: Search Pachyderm

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

+ Enter GitHub username to add +

Writers
Can read and modify this repo

	jdollner	x
+ Enter GitHub username to add		+

Readers
Can read this repo

	JoeyZwicker	x
	msteffen	x
+ Enter GitHub username to add		+

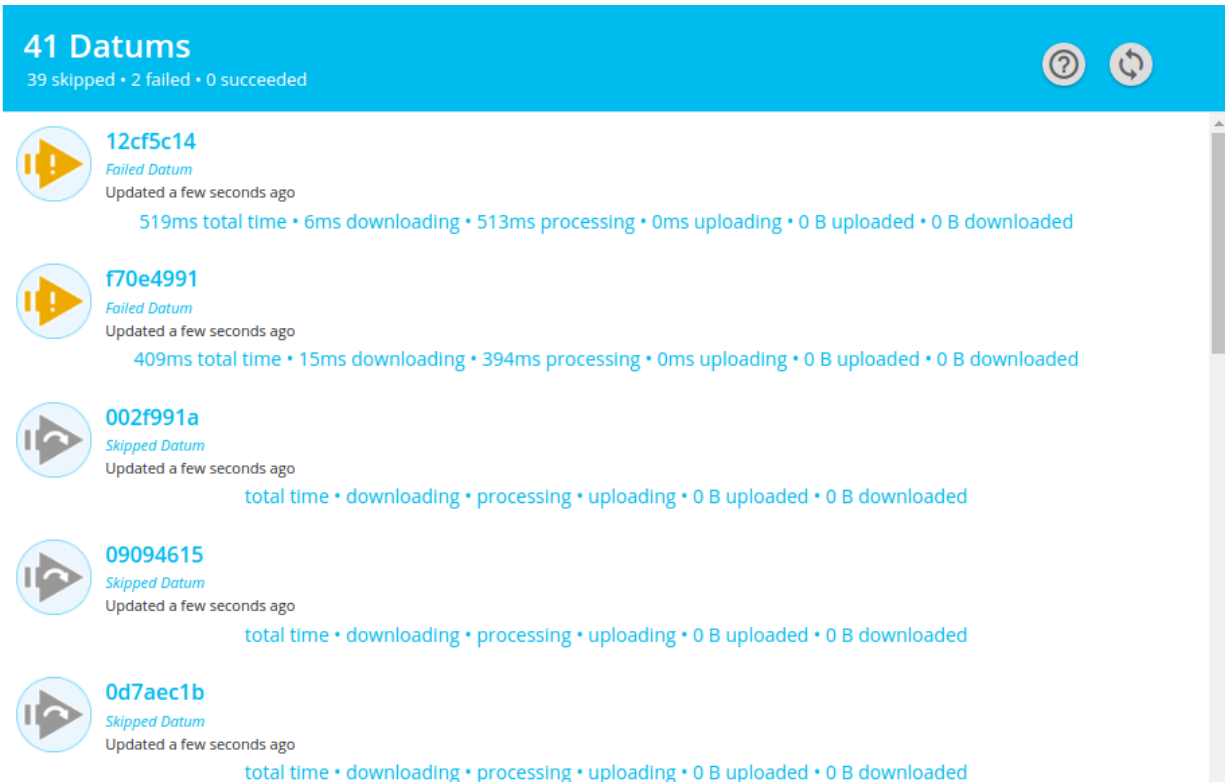
Admins
Can read, modify, and edit access controls on any repo

The list of administrators can be edited in [settings](#)

Repository Logo: A blue hexagon with a pink border and the text "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.

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.

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.

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.

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 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:

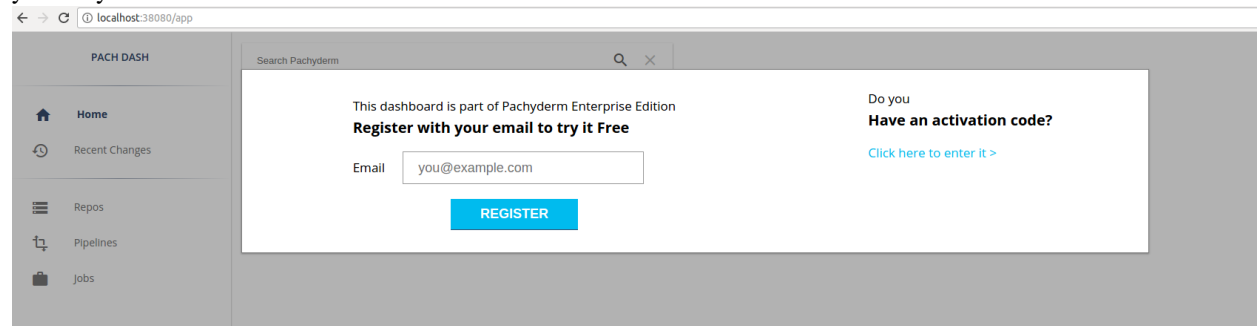
```
$ pachctl enterprise activate <activation-code>
```

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

Assuming that you have a running Pachyderm cluster and you have deployed the Pachyderm Enterprise dashboard using *this guide*, you should be able to visit `<pachyderm host IP>:30080` (e.g., `localhost:30080` when you are using `pachctl port-forward`) to see the dashboard. 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
```

Access Controls

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

These docs will guide you through:

1. *Understanding Pachyderm access controls.*
2. *Activating access control features (aka “auth” features).*
3. *Logging into Pachyderm.*
4. *Managing/updating user access to data repositories.*

We will also discuss:

- *The behavior of pipelines when using access control*
- *The behavior of a cluster when access control is de-activated or an enterprise token expires*

Understanding Pachyderm access controls

Assuming access controls are activated, each data repository (aka *repo*) in Pachyderm will have an Access Control List (ACL) associated with it. The ACL will include:

- **READERs** - users who can read the data versioned in the repo.
- **WRITERs** - users with **READER** access who can also commit additions, deletions, or modifications of data into the repo.
- **OWNERs** - users with **READER** and **WRITER** access who can also modify the repo’s ACL.

Currently, Pachyderm accounts correspond to GitHub users, who authenticate inside of Pachyderm using OAuth integration with GitHub. Pachyderm user accounts are identified within Pachyderm via their GitHub usernames.

There is a single, hardcoded “admin” group (and no other groups) in Pachyderm. Users in that admin group have the ability to perform any action in the cluster, including appointing other admins. Further, a repo with no ACL can only be managed by the cluster admins.

Activating access control

First, you will need to make sure that your cluster has Pachyderm Enterprise Edition activated (you can follow this guide to activate Enterprise Edition). The status of the Enterprise features can be verified by accessing the Pachyderm

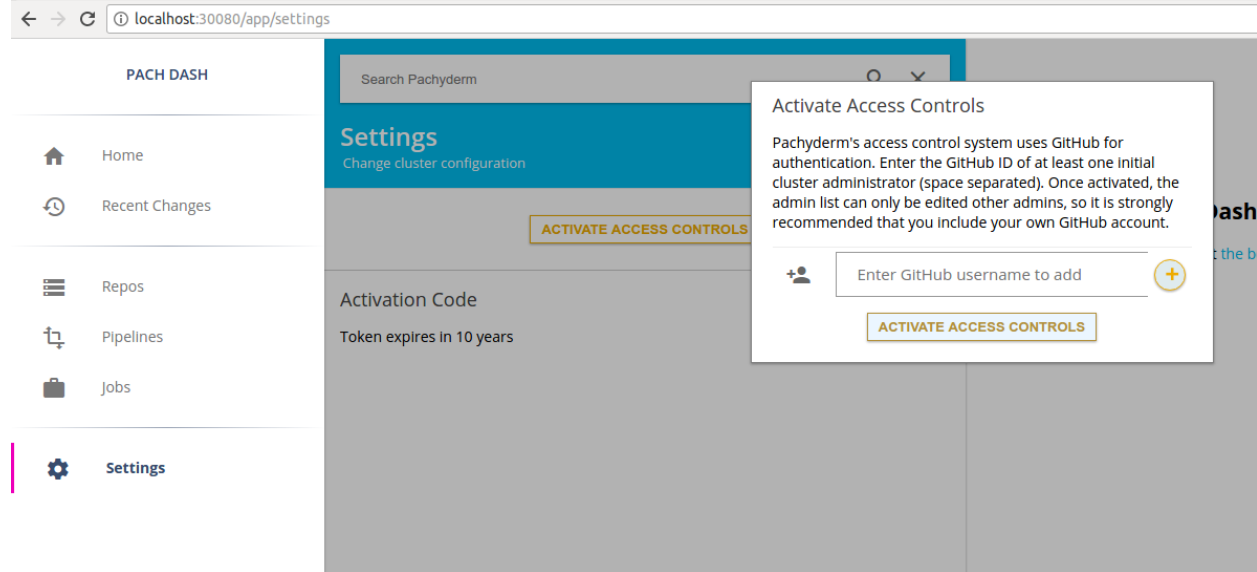
dashboard or with `pachctl` as follows:

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

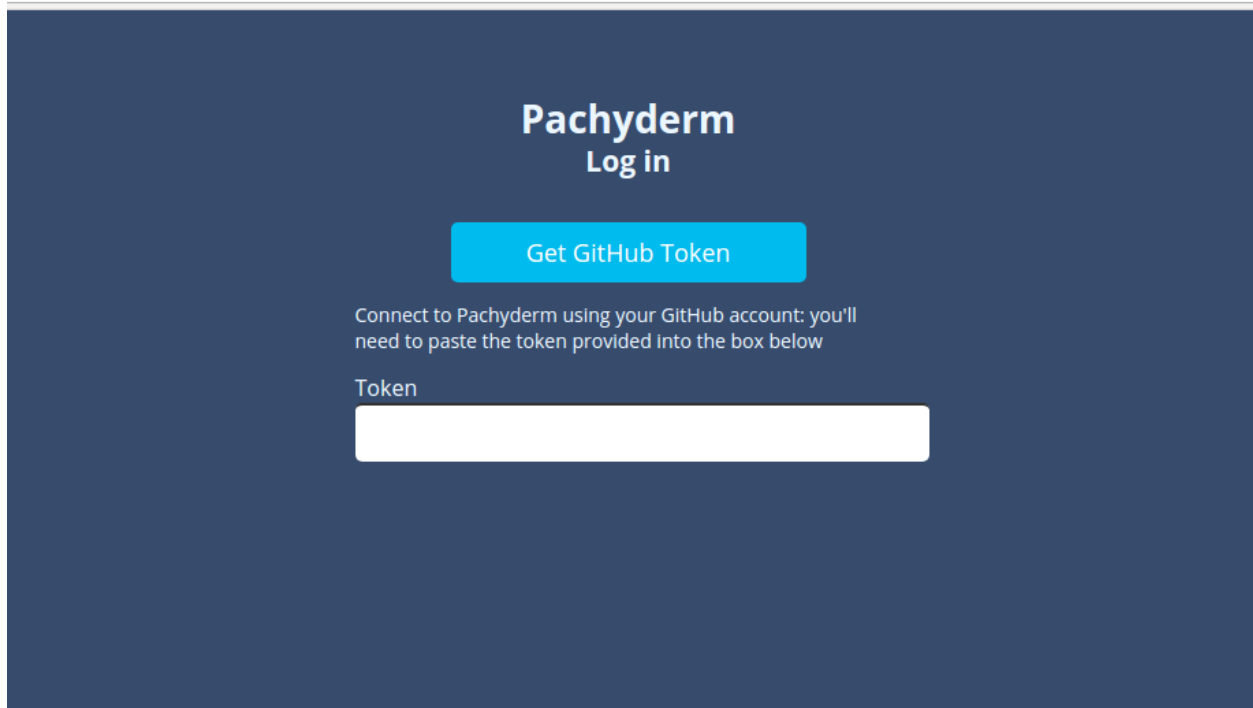
Next, we need to activate the Enterprise access control features. This can be done *in the dashboard* or with `pachctl auth activate`. However, before executing that command, we should decide on at least one user that will have admin privileges on the cluster. Pachyderm admins will be able to modify the scope of access for any non-admin users on the cluster. All users in Pachyderm are identified by their GitHub usernames.

Activating access controls with the dashboard

To activate access controls via the Pachyderm dashboard, go to the settings page where you should see a “Activate Access Controls” button. Click on that button. You will then be able to enter one or more Github users as cluster admins and activate access controls:



After activating access controls, you should see the following screen asking you to login to Pachyderm:



Activating access controls with pachctl

To activate access controls on a cluster and set the GitHub user `dwhitena` as an admin, we would execute the following `pachctl` command:

```
$ pachctl auth activate --admins=dwhitena
```

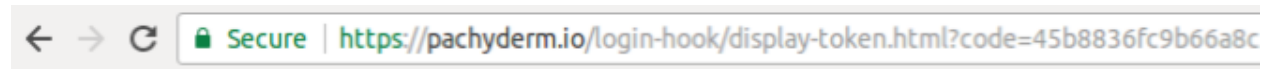
Your Pachyderm cluster can have more than one admin if you like, but you need to supply at least one with this command. To add multiple admins, You would just need to specify them here as a comma separated list.

Logging into Pachyderm

Now that we have activated access control, we can login to our cluster. When using the Pachyderm dashboard, you will need to [login on the dashboard](#), and, when using the `pachctl` CLI, you will need to [login via the CLI](#).

Login on the dashboard

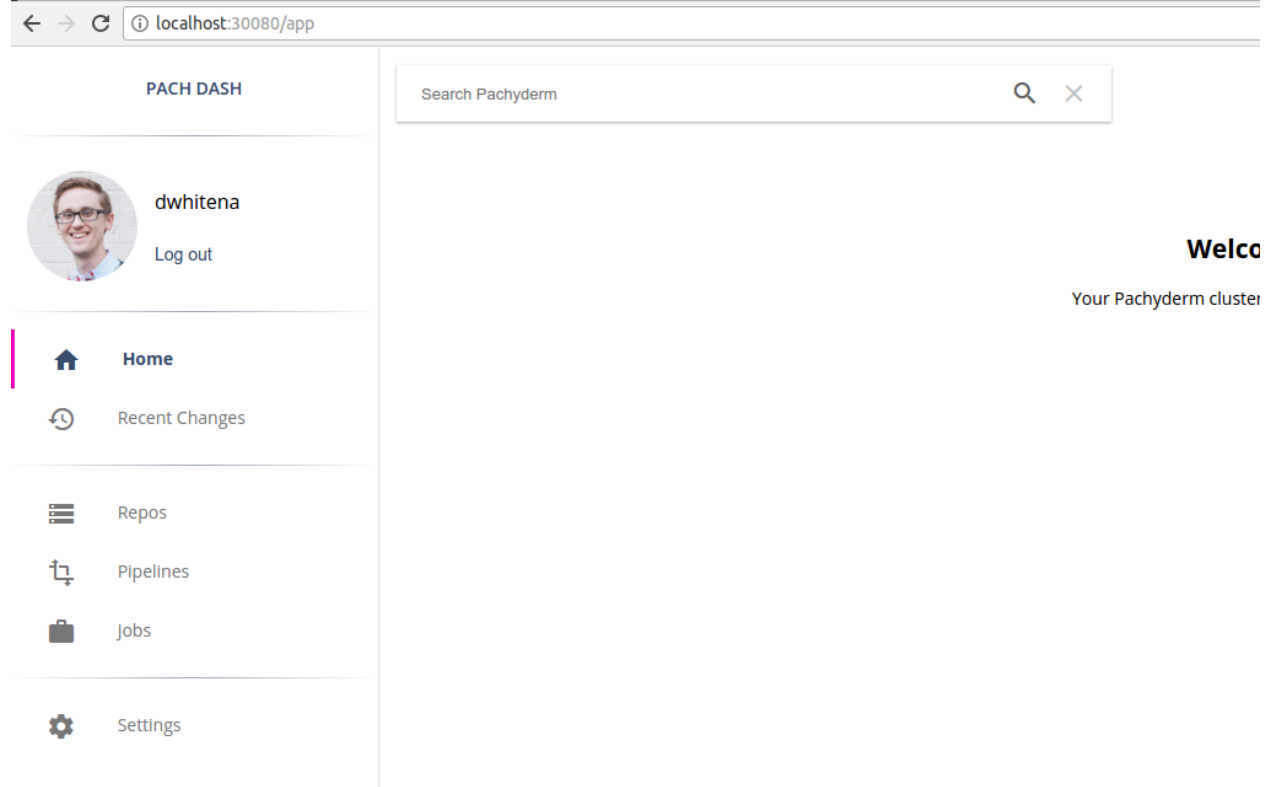
Once you have authorized access controls for Pachyderm, you will need to login to use the Pachyderm dashboard as shown above in [this section](#). To login, click the “Get GitHub token” button. You will then be presented with an option to “Authorize Pachyderm” (assuming that you haven’t authorized Pachyderm on GitHub previously). Once you authorize Pachyderm, you will be presented with a Pachyderm user token:



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

141cddf57f54

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:



Login using pachctl

You can use the `pachctl auth login <username>` to login via the CLI. When we execute this command, `pachctl` will provide us with a GitHub link to authenticate ourselves as the provided GitHub user, as shown below:

```
$ pachctl auth login dwhitena
(1) Please paste this link into a browser:

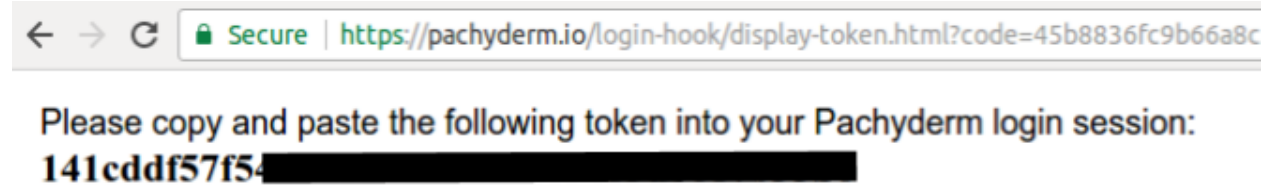
https://github.com/login/oauth/authorize?client_id=d3481e92b4f09ea74ff8&redirect_
  ↪uri=https%3A%2F%2Fpachyderm.io%2Flogin-hook%2Fdisplay-token.html

(You will be directed to GitHub and asked to authorize Pachyderm's login app on
  ↪Github. If you accept, you will be given a token to paste here, which will give you
  ↪an externally verified account in this Pachyderm cluster)

(2) Please paste the token you receive from GitHub here:
```

When visiting this link in a browser, you will be presented with an option to “Authorize Pachyderm” (assuming that you haven’t authorized Pachyderm via GitHub previously). Once you authorize Pachyderm, you will be presented

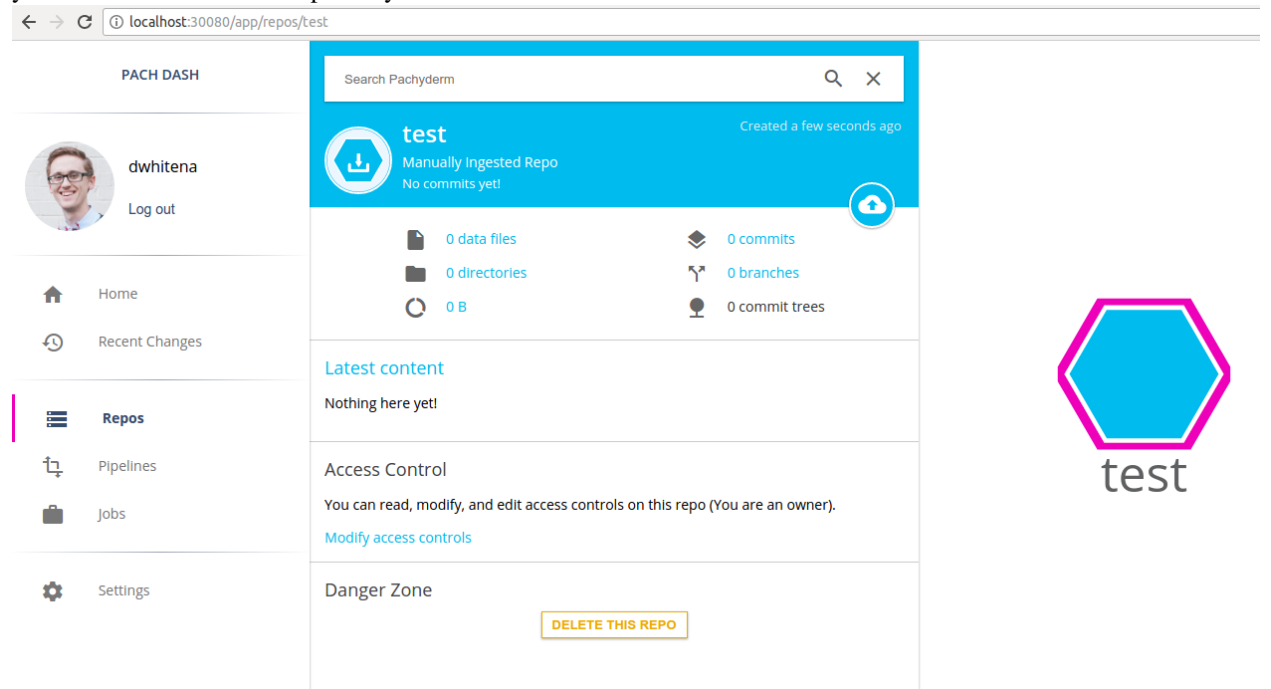
with a Pachyderm user token:



Copy and paste this token back into the terminal, as requested by `pachctl`, and press enter. You are now logged in to Pachyderm!

Managing and updating user access

Let's suppose that we create a repository call `test` when we are logged into Pachyderm as the user `dwhitena`. Because, the user `dwhitena` created this repository, `dwhitena` will have full read/write access to the repo. This can be confirmed on the dashboard by navigating to or clicking on the repo `test`. The results repo details will show your current access to the repository:



You can also confirm your access via the `pachctl auth get ...` command:

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

An OWNER of `test` or a cluster admin can then set other user's scope of access to the repo. This can be done via the `pachctl auth set ...` command or via 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, we can click on `Modify access controls` under the repo details in the dashboard. This will allow us to easily add the users one by one:

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

Left Sidebar (PACH DASH):

- Search Pachyderm (input field with magnifying glass and close icons)
- User profile: **dwhitena** (with a circular profile picture) and a **Log out** button.
- Navigation menu:
 - Home (house icon)
 - Recent Changes (clock icon)
 - Repos** (three horizontal bars icon)
 - Pipelines (funnel icon)
 - Jobs (briefcase icon)
 - Settings (gear icon)

Main Content Area: Access Controls for "test"

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

+ Enter GitHub username to add +

Writers
Can read and modify this repo

jdoliner ×
+ Enter GitHub username to add +

Readers
Can read this repo

joeyZwicker ×
msteffen ×
+ Enter GitHub username to add +

Admins
Can read, modify, and edit access controls on any repo

The list of administrators can be edited in [settings](#)

On the right side of the interface, there is a large blue hexagon with a pink border and the word **test** below it.

Behavior of pipelines as related to access control

In Pachyderm, you don't explicitly set the scope of access for users on pipelines. Rather, pipelines infer access from the repositories that are input to the pipeline, as follows:

- An OWNER, WRITER, or READER of a repo may subscribe a pipeline to that repo.
- When a user subscribes a pipeline to a repo, they will be set as an OWNER of that pipeline's output repo.
- The initial OWNER of a pipeline's output repo (or an admin) needs to set the scope of access for other users to that output repo.

Activation code expiration and de-activation

When an enterprise activation code expires, an auth-activated Pachyderm cluster goes into an "admin only" state. In this state, only admins will 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 (via the dashboard or via `pachctl enterprise activate ...`), the Pachyderm cluster will return to its previous state.

When access controls are de-activated on a Pachyderm cluster via `pachctl auth deactivate`, the cluster returns to being a non-access controlled Pachyderm cluster. That is,

- All ACLs are deleted.
- The cluster returns to being a blank slate in regards to access control. Everyone that can connect to Pachyderm will be able to access and modify the data in all repos.
- There will no longer be a concept of users (i.e., no one will be able to login to Pachyderm).

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.

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": {
    "atom": {
      "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.

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:

The screenshot shows the Pachyderm PACH DASH interface. On the left is a sidebar with navigation links: Home, Recent Changes, Repos, Pipelines, and Jobs. The main content area displays the 'edges' pipeline details. It shows the pipeline is active, last updated a few seconds ago, and sends output to 'edges'. It also shows statistics: 0 active jobs, 1 inputs, 42.02 MB generated, 4 output commits, 1 version, and 278ms avg runtime. Below this, it shows the pipeline takes input from 'images'. At the bottom, a 'Recent Jobs' section shows three jobs: a successful job (2a3b2408) and two failed jobs (f9b12d4-f9b1-4f6f-b1d2-aa408b4d1a8e and 777b12d4). Annotations with arrows point to the successful job (labeled '1 job success') and the failed jobs (labeled '2 job failures'). To the right of the main content area, a diagram shows the 'images' pipeline feeding into the 'edges' pipeline.

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.):

The screenshot shows the Pachyderm PACH DASH interface with the 'edges/777b12d4' job details. The job is marked as 'failed Job' and belongs to the 'edges' pipeline. It was created by the 'edges' pipeline, which is active and last updated 2 hours ago. The job statistics show: 41 datums total, 39 datums skipped, 3.68s runtime, 0 B input data downloaded, and 0 B output data uploaded. The transform details show the image 'pachyderm/opencv'.

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 dashboard interface. At the top, a blue header bar shows the path 'edges / edges/777b12d4 /' and the datum ID '12cf5c14' with a status of 'failed Datum'. Below this, a summary bar provides performance metrics: '519ms total runtime', '6ms download time', '513ms processing time', and '0ms upload time'. It also shows 'Input files', '0 B downloaded', 'output files', and '0 B uploaded'. The 'Input Files' section includes a 'View the Files' link, which is highlighted by a red arrow and the text 'View the input files corresponding to the datum'. Below this, the 'Created by Pipeline' section shows the pipeline 'edges/777b12d4' with a 'Failed job' icon and 'Updated 2 hours ago'. It also indicates '39 of 41 datums done • 0 inputs • 3.68s'. The 'As part of job' section shows the job 'edges' with an 'Active Pipeline' icon and 'Updated 2 hours ago', noting '0 active jobs • 1 inputs • 4 output commits'. The 'Logs from worker pipeline-edges-v1-q3x4q' section shows a list of logs with 31 lines and 7.17 KB. The logs include a traceback for a failure in the file '/edgesec.py' at line 18. A red bracket on the right side of the logs is labeled 'Logs corresponding to the failure'.

edges / edges/777b12d4 /
12cf5c14
failed Datum

519ms total runtime 6ms download time 513ms processing time 0ms upload time
Input files 0 B downloaded output files 0 B uploaded

Input Files
View the Files

Created by Pipeline
edges/777b12d4
Failed job
Updated 2 hours ago
39 of 41 datums done • 0 inputs • 3.68s

As part of job
edges
Active Pipeline
Updated 2 hours ago
0 active jobs • 1 inputs • 4 output commits

Logs from worker pipeline-edges-v1-q3x4q
logs • 31 lines • 7.17 KB • Download

```
1 process call started - request: job_id:"777b12d4-f9b1-4f6f-b1d2-aa400b4d1a0e" data:<file_info:<file:<
2 input has not been processed, downloading data
3 input data download took (6.389642ms)
4 beginning to run user code
5 Traceback (most recent call last):
6
7 File "/edgesec.py", line 18, in <module>
```

View the input files corresponding to the datum

Logs corresponding to the failure

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](#)

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.

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.

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
```

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}
```

Note that 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 1.8.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				
17-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}
```

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 OSX:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.7

# For Linux (64 bit) or Window 10+ on WSL:
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
→download/v1.7.11/pachctl_1.7.11_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
1.7.0
```

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. Also, RBAC can be enabled as further documented [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      1.7.0
pachd        1.7.0
```

Amazon Web Services

Advanced

- Deploy within an existing VPC
- Connect to your Pachyderm Cluster

Standard Deployment

We recommend one of the following two methods for deploying Pachyderm on AWS:

1. By manually deploying Kubernetes and Pachyderm.
 - This is appropriate if you (i) already have a kubernetes deployment, (ii) if you would like to customize the types of instances, size of volumes, etc. in your cluster, (iii) if you're setting up a production cluster, or (iv) if you are processing a lot of data or have computationally expensive workloads.
2. By executing a one shot deploy script that will both deploy Kubernetes and Pachyderm.
 - This option is appropriate if you are just experimenting with Pachyderm. The one-shot script will get you up and running in no time!

In addition, we recommend setting up AWS CloudFront for any production deployments. AWS puts S3 rate limits in place that can limit the data throughput for your cluster, and CloudFront helps mitigate this issue. Follow these instructions to deploy with CloudFront

- Deploy a Pachyderm cluster with CloudFront

Manual Pachyderm Deploy

Prerequisites

- [AWS CLI](#) - have it installed and have your [AWS credentials](#) configured.
- [kubectl](#)
- [kops](#)
- [pachctl](#)

Deploy Kubernetes

The easiest way to install Kubernetes on AWS (currently) is with kops. You can follow [this step-by-step guide from Kubernetes](#) for the deploy. Note, we recommend using at `r4.xlarge` or larger instances in the cluster.

Once, you have a Kubernetes cluster up and running in AWS, you should be able to see the following output from `kubectl`:

```
$ kubectl get all
NAME                TYPE          CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
svc/kubernetes      ClusterIP     100.64.0.1   <none>        443/TCP    7m
```

Deploy Pachyderm

To deploy Pachyderm on your k8s cluster you will need to:

1. Install the `pachctl` CLI tool,
2. Add some storage resources on AWS,
3. Deploy Pachyderm on top of the storage resources.

Install `pachctl`

To deploy and interact with Pachyderm, you will need `pachctl`, Pachyderm's command-line utility. To install `pachctl` run one of the following:

```
# For OSX:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.7

# For Linux (64 bit) or Window 10+ on WSL:
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
↳download/v1.7.11/pachctl_1.7.11_amd64.deb && sudo dpkg -i /tmp/pachctl.deb
```

You can try running `pachctl version --client-only` to verify that `pachctl` has been successfully installed.

```
$ pachctl version --client-only
1.7.0
```

Set up the Storage Resources

Pachyderm needs an [S3 bucket](#), and a [persistent disk](#) (EBS in AWS) to function correctly.

Here are the environmental variables you should set up to create and utilize these resources:

```
# BUCKET_NAME needs to be globally unique across the entire AWS region
$ BUCKET_NAME=<The name of the S3 bucket where your data will be stored>

# 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)>
```

Then to actually create the backing S3 bucket, you can run one of the following:

```
# If AWS_REGION is us-east-1.
$ aws s3api create-bucket --bucket ${BUCKET_NAME} --region ${AWS_REGION}

# If AWS_REGION is outside of us-east-1.
$ aws s3api create-bucket --bucket ${BUCKET_NAME} --region ${AWS_REGION} --create-
↪bucket-configuration LocationConstraint=${AWS_REGION}
```

As a sanity check, you should be able to see the bucket that you just created when you run the following:

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

Deploy Pachyderm

You can deploy Pachyderm on AWS using:

- *An IAM role*, or
- *Static credentials*

Deploying with an IAM role

Run the following command to deploy your Pachyderm cluster:

```
$ pachctl deploy amazon ${BUCKET_NAME} ${AWS_REGION} ${STORAGE_SIZE} --dynamic-etcd-
↪nodes=1 --iam-role <your-iam-role>
```

Note that for this to work, the following need to be true:

- The nodes on which Pachyderm is deployed need to be assigned with the utilized IAM role. If you created your cluster with `kops`, the nodes should have a dedicated IAM role. You can find this IAM role by going to the AWS console, clicking on one of the EC2 instance in the k8s cluster, and inspecting the “Description” of the instance.
- The IAM role needs to have access to the bucket you just created. To ensure that it has access, you can go to the `Permissions` tab of the IAM role and edit the policy to include the following segment (Make sure to replace `your-bucket` with your actual bucket name):

```
{
  "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/*"
]
```

- The IAM role needs to have the proper “trust relationships” set up. You can verify this by navigating to the Trust relationships tab of your IAM role, clicking Edit trust relationship , and ensuring that you see a statement with `sts:AssumeRole` . For instance, this would be a valid trust relationship:

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

Once you’ve run `pachctl deploy ...` and waited a few minutes, you should see the following running pods in Kubernetes:

```
$ 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, that’s totally ok. That simply means that Kubernetes tried to bring up those containers before etcd was ready so it restarted them.

If you see the above pods running, the last thing you need to do is forward a couple ports 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 verify that the cluster is working by executing `pachctl version` , which should return a version for both `pachctl` and `pachd` :

```
$ pachctl version
COMPONENT    VERSION
pachctl      1.7.0
pachd        1.7.0
```

Deploying with static credentials

When you installed kops, you should have created a dedicated IAM user (see [here](#) for details). You could deploy Pachyderm using the credentials of this IAM user directly, although that’s not recommended:

```
$ AWS_ACCESS_KEY_ID=<access key ID>

$ AWS_SECRET_ACCESS_KEY=<secret access key>
```

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},"
```

Note, the `,` at the end of the `credentials` flag in the `deploy` command is for an optional temporary AWS token. You might utilize this sort of temporary token if you are just experimenting with a deploy. However, such a token should NOT be used for a production deploy.

It may take a few minutes for Pachyderm to start running on the cluster, but you should eventually see the following running pods:

```
$ 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
```

If you see an output similar to the above, the last thing you need to do is forward a couple ports 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 verify that the cluster is working by running `pachctl version`, which should return a version for both `pachctl` and `pachd`:

```
$ pachctl version
COMPONENT    VERSION
pachctl      1.7.0
pachd        1.7.0
```

One Shot Script

Prerequisites

- [AWS CLI](#) - have it installed and have your [AWS credentials](#) configured.
- [kubectl](#)
- [kops](#)
- [pachctl](#)
- [jq](#)
- [uuid](#)

Run the deploy script

Once you have the prerequisites mentioned above, download and run our AWS deploy script by running:

```
$ curl -o aws.sh https://raw.githubusercontent.com/pachyderm/pachyderm/master/etc/
↪deploy/aws.sh
$ chmod +x aws.sh
$ sudo -E ./aws.sh
```

This script will use `kops` to deploy Kubernetes and Pachyderm in AWS. The script will ask you for your AWS credentials, region preference, etc. If you would like to customize the number of nodes in the cluster, node types, etc., you can open up the deploy script and modify the respective fields.

The script will take a few minutes, and Pachyderm will take an addition couple of minutes to spin up. Once it is up, `kubectl get pods` should return something like:

```
$ 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
```

Connect `pachctl`

You will then need to forward a couple ports 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 verify that the cluster is working by executing `pachctl version`, which should return a version for both `pachctl` and `pachd`:

```
$ pachctl version
COMPONENT    VERSION
pachctl      1.7.0
pachd        1.7.0
```

Remove

You can delete your Pachyderm cluster using `kops`:

```
$ kops delete cluster
```

In addition, there is the entry in `/etc/hosts` pointing to the cluster that will need to be manually removed. Similarly, kubernetes state s3 bucket and pachyderm storage bucket will need to be manually removed.

To deploy Pachyderm to Azure, you need to:

1. *Install Prerequisites*
2. *Deploy Kubernetes*
3. *Deploy Pachyderm on Kubernetes*

Prerequisites

Install the following prerequisites:

- Azure CLI `>= 2.0.1`
- `jq`
- `kubectrl`
- `pachctl`

Deploy Kubernetes

The easiest way to deploy a Kubernetes cluster is through the [Azure Container Service \(AKS\)](#). To create a new AKS Kubernetes cluster using the Azure CLI `az`, run:

```
$ RESOURCE_GROUP=<a unique name for the resource group where Pachyderm will be_
↳ deployed, e.g. "pach-resource-group">

$ LOCATION=<a Azure availability zone where AKS is available, e.g, "Central US">

$ NODE_SIZE=<size for the k8s instances, we recommend at least "Standard_DS4_v2">

$ CLUSTER_NAME=<unique name for the cluster, e.g., "pach-aks-cluster">

# Create the Azure resource group.
$ az group create --name=${RESOURCE_GROUP} --location=${LOCATION}

# Create the AKS cluster.
$ az aks create --resource-group ${RESOURCE_GROUP} --name ${CLUSTER_NAME} --generate-
↳ ssh-keys --node-vm-size ${NODE_SIZE}
```

Once Kubernetes is up and running you should be able to confirm the version of the Kubernetes server via:

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"9", GitVersion:"v1.9.3", GitCommit:
↳ "d2835416544f298c919e2ead3be3d0864b52323b", GitTreeState:"clean", BuildDate:"2018-
↳ 02-07T12:22:21Z", GoVersion:"go1.9.2", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"7", GitVersion:"v1.7.9", GitCommit:
↳ "19fe91923d584c30bd6db5c5a21e9f0d5f742de8", GitTreeState:"clean", BuildDate:"2017-
↳ 10-19T16:55:06Z", GoVersion:"go1.8.3", Compiler:"gc", Platform:"linux/amd64"}
```

Note - Azure AKS is still a relatively new managed service. As such, we have had some issues consistently deploying AKS clusters in certain availability zones. If you get timeouts or issues when provisioning an AKS cluster, we recommend trying in a fresh resource group and possibly trying a different zone.

Deploy Pachyderm

To deploy Pachyderm we will need to:

1. Add some storage resources on Azure,
2. Install the Pachyderm CLI tool, `pachctl`, and
3. Deploy Pachyderm on top of the storage resources.

Set up the Storage Resources

Pachyderm requires an object store and persistent volume ([Azure Storage](#)) to function correctly. To create these resources, you need to clone the [Pachyderm GitHub repo](#) and then run the following from the root of that repo:

```
$ STORAGE_ACCOUNT=<The name of the storage account where your data will be stored,
↳ unique in the Azure location>

$ CONTAINER_NAME=<The name of the Azure blob container where your data will be stored>

$ STORAGE_SIZE=<the size of the persistent volume that you are going to create in
↳ GBs, we recommend at least "10">

# Create an Azure storage account
az storage account create \
  --resource-group="${RESOURCE_GROUP}" \
  --location="${LOCATION}" \
  --sku=Standard_LRS \
  --name="${STORAGE_ACCOUNT}" \
  --kind=Storage

# Build a microsoft tool for creating Azure VMs from an image. Necessary to create
↳ the blank PV.
$ STORAGE_KEY="$(az storage account keys list \
  --account-name="${STORAGE_ACCOUNT}" \
  --resource-group="${RESOURCE_GROUP}" \
  --output=json \
  | jq '[0].value' -r
)"
```

Install pachctl

`pachctl` is a command-line utility used for interacting with a Pachyderm cluster.

```
# For OSX:
$ brew tap pachyderm/tap && brew install pachyderm/tap/pachctl@1.7

# For Linux (64 bit) or Window 10+ on WSL:
$ curl -o /tmp/pachctl.deb -L https://github.com/pachyderm/pachyderm/releases/
  ↪download/v1.7.11/pachctl_1.7.11_amd64.deb && sudo dpkg -i /tmp/pachctl.deb
```

You can try running `pachctl version` to check that this worked correctly:

```
$ pachctl version --client-only
COMPONENT      VERSION
pachctl        1.7.0
```

Deploy Pachyderm

Now we're ready to deploy Pachyderm:

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

It may take a few minutes for the `pachd` pods to be running because it's pulling containers from Docker Hub. When Pachyderm is up and running, you should see something similar to the following state:

```
$ 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: If you see a few restarts on the `pachd` nodes, that's totally ok. That simply means that Kubernetes tried to bring up those containers before `etcd` was ready so it restarted them.

Finally, assuming you want to connect to the cluster from your local machine (i.e., your laptop), 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 by creating a new repo.

```
$ pachctl version
COMPONENT      VERSION
pachctl        1.7.0
pachd          1.7.0
```


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.

Deploy Pachyderm

1. How you deploy Pachyderm on OpenShift is largely going to depend on where OpenShift is deployed.
 - OpenShift Deployed on [AWS](#)
 - OpenShift Deployed on [GCP](#)
 - OpenShift Deployed on [Azure](#)
 - OpenShift Deployed [on-premise](#)
1. Replace `hostPath` with `emptyDir` in your cluster manifest (Your manifest is generated by the `pachctl deploy ...` command or can be generated manually. To only generate the manifest, run `pachctl deploy ...` with the `--dry-run` flag).

```
"spec": {
  "volumes": [
    {
      "name": "pach-disk",
      "emptyDir": {}
    }
  ],
... <snip> ...

  "spec": {
    "volumes": [
      {
        "name": "etcd-storage",
        "emptyDir": {}
      }
    ]
  },
```

Please note that `emptyDir` does not persist your data. You need to configure persistent volume or `hostPath` to persist your data.

1. Deploy 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-65fd68d6d4-8vjg7              1/1      Running   0           4m
```

Configure your cluster to run pipelines

1. Add `cluster-reader` and edit `role` to pachyderm service account:

```
$ oadm policy add-cluster-role-to-user cluster-reader system:serviceaccount:
→<PROJECT_NAME>:pachyderm
$ oadm policy add-cluster-role-to-user edit system:serviceaccount:<PROJECT_NAME>:
→pachyderm
```

2. Add the pachyderm service account to the pipeline Pod (ReplicationController).

```
oc patch rc pipeline-edges-v1 -p 'spec:
  template:
    spec:
      serviceAccount: pachyderm
      serviceAccountName: pachyderm'
```

or manually edit rc `oc edit rc <RC_PIPELINE> -o json:`

```
...
  "dnsPolicy": "ClusterFirst",
  "serviceAccountName": "pachyderm",
  "serviceAccount": "pachyderm",
  "securityContext": {}
  ...
```

3. Replace `hostPath` with `emptyDir`. Again, please note that `emptyDir` does not persist your data. You need to configure persistent volume or `hostPath` to persist.
4. Redeploy the updated Pods.

```
$ oc scale rc pipeline-edges-v1 --replicas=0
$ oc scale rc pipeline-edges-v1 --replicas=4
```

You can see the pipeline pods are running and successful job.

```
$ oc get pods
NAME                                READY    STATUS    RESTARTS   AGE
etcd-kbi4n                          1/1      Running   0           1h
pachd-z3b7y                          1/1      Running   0           1h
pipeline-edges-v1-28vdj              1/1      Running   0           12s
pipeline-edges-v1-fpa8v              1/1      Running   0           12s
pipeline-edges-v1-mshi0              1/1      Running   0           12s
pipeline-edges-v1-yx2wa              1/1      Running   0           12s
```

```
$ pachctl list-job
ID                                OUTPUT COMMIT
→STARTED                        DURATION  RESTART PROGRESS STATE
1b2c1b49-f536-484f-b0e3-07b3906572be edges/006f0aecb2b048d5b5edee0cdb766879 55
→minutes ago 51 minutes 0        1 / 1    success
```

Problems related to OpenShift deployment are tracked in [this issue](#). If you have additional related questions, please ask them on Pachyderm's [slack channel](#) or via email support@pachyderm.io.

On Premises

Pachyderm is built on [Kubernetes](#) and can be backed by an object store of your choice. As such, Pachyderm can run on any on premise platforms/frameworks that support Kubernetes, a persistent disk/volume, and an object store.

Prerequisites

1. `kubectl`
2. `pachctl`

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.

Object Store

Once you have Kubernetes up and running, deploying Pachyderm is a matter of supplying Kubernetes with a JSON/YAML manifest to create the Pachyderm resources. This includes providing information that Pachyderm will use to connect to a backing object store.

For on premise deployments, we recommend using [Minio](#) as a backing object store. However, at this point, you could utilize any backing object store that has an S3 compatible API. To create a manifest template for your on premise deployment, run:

```
pachctl deploy custom --persistent-disk google --object-store s3 <persistent disk_  
↪name> <persistent disk size> <object store bucket> <object store id> <object store_  
↪secret> <object store endpoint> --static-etcd-volume=${STORAGE_NAME} --dry-run >_  
↪deployment.json
```

Then you can modify `deployment.json` to fit your environment and kubernetes deployment. Once, you have your manifest ready, deploying Pachyderm is as simple as:

```
kubectl create -f deployment.json
```

Need Help?

If you need help with your on premises deploy, please reach out to us on Pachyderm's [slack channel](#) or via email at support@pachyderm.io. We are happy to help!

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](#).

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.

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}
```

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]
```

Then 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}
```

Azure + Custom Object Store

Additional prerequisites:

- Install **Azure CLI** **>= 2.0.1**
- Install **jq**
- Clone 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 '.[].name'
```

Then 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}
```

AWS CloudFront

To deploy a production ready AWS cluster with CloudFront:

1. *Deploy a cloudfront enabled Pachyderm cluster in AWS*
2. *Obtain a cloudfront keypair*
3. *Apply the security credentials*
4. *Verify the setup*

Deploy a cloudfront enabled cluster in AWS

You'll need to use our “one shot” AWS deployment script to deploy this cluster as follows:

```
$ curl -o aws.sh https://raw.githubusercontent.com/pachyderm/pachyderm/master/etc/
→deploy/aws.sh
$ chmod +x aws.sh
$ sudo -E ./aws.sh --region=us-east-1 --zone=us-east-1b --use-cloudfront &> deploy.log
```

Here we've redirected the output to a file. Make sure you keep this file around for reference.

Note: You may see a few extra restarts on your pachd pod. Sometimes it takes a bit before your cloudfront distribution comes online

Obtain a cloudfront keypair

You will most likely need to Ask your IT department for a cloudfront keypair, because only a root AWS account can generate this keypair. You can pass along [this link with instructions](#).

When you get the keypair, you should receive:

- the private/public key (although you only need the private key)
- the keypair ID (which is usually in the filename)

For example,

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

Here we see that the Key Pair ID is `APKAXXXXXXXXXXXXXXXX` , and the second file is the private key, which should look similar to the following:

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

Apply the security credentials

You can now run the following script to apply these security credentials to your cloudfront distribution:

```
$ curl -o secure-cloudfront.sh https://raw.githubusercontent.com/pachyderm/pachyderm/
↪master/etc/deploy/cloudfront/secure-cloudfront.sh
$ chmod +x secure-cloudfront.sh
$ ./secure-cloudfront.sh --region us-west-2 --zone us-west-2c --bucket YYYY-pachyderm-
↪store --cloudfront-distribution-id E1BEBVLIDYTLV --cloudfront-keypair-id
↪APKAXXXXXXXXXXXXXXXX --cloudfront-private-key-file ~/Downloads/pk-APKAXXXXXXXXXXXXXXXX.pem
```

where the values for the `--bucket` and `--cloudfront-distribution-id` flags can be obtained from the `deploy.log` file containing your deployment logs.

You will then need to restart the `pachd` pod in kubernetes for the changes to take effect:

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

Verify the setup

To verify the setup, we can look at the `pachd` logs to confirm usage of 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
↪(APKAXXXXXXXXXXXXXXXX) - to sign cloudfront URLs
```

Pachyderm Version Upgrades

Pachyderm releases new major versions (1.4, 1.5, 1.6, etc.) roughly every 2-3 months and releases minor versions as needed/warranted. Upgrading the version of your Pachyderm cluster should be relatively painless, and you should try to upgrade to make sure that you benefit from the latest features, bug fixes, etc. This guide will walk you through that upgrading process.

Note - Occasionally, Pachyderm introduces changes that are backward-incompatible. For example, repos/commits/files created on an old version of Pachyderm may be unusable on a new version of Pachyderm. When that happens (which isn't very often), migrations of Pachyderm metadata will happen automatically upon upgrading. We try our best to make these type of changes transparent (in blog posts, changelogs, etc.), and you can read more about the migration process and best practices [here](#).

Before Upgrading

Pachyderm's state (the data you have/are processing and metadata associated with commits, jobs, etc.) is stored in the object store bucket and persistent volume(s) you specified at deploy time. As such, you may want to back up one or both of these storage resources before upgrading, just in case something unexpected happens. You should follow your cloud provider's recommendation for backing up these resources. For example, here are official guides on backing up persistent volumes on Google Cloud Platform and AWS, respectively:

- [Creating snapshots of GCE persistent volumes](#)
- [Creating snapshots of Elastic Block Store \(EBS\) volumes](#)

In addition or alternatively, you can utilize `pachctl extract` and `pachctl restore` to extract the state of a Pachyderm cluster and restore a Pachyderm cluster to an extracted state. This process is further described [here](#).

That being said, the upgrading steps detailed below should not effect these storage resources, and it's perfectly fine to upgrade to the new version of Pachyderm with the same storage resources.

It's also good idea to version or otherwise save the Pachyderm deploy commands (`pachctl deploy ...`) that you utilize when deploying, because you can re-use those exact same commands when re-deploying, as further detailed below.

Upgrading Pachyderm

Upgrading your Pachyderm version is as easy as:

1. *Upgrading `pachctl`*
2. *Re-deploying Pachyderm*

Upgrading pachctl

To deploy an upgraded Pachyderm, we need to retrieve the latest version of `pachctl`. Details on installing the latest version can be found [here](#). You should be able to upgrade via `brew` or `apt` depending on your environment.

Once you install the new version of `pachctl` (e.g., 1.7.0 in our example), you can confirm this via:

```
$ pachctl version --client-only
COMPONENT      VERSION
pachctl        1.7.0
```

Re-deploying Pachyderm

You can now re-deploy Pachyderm with the **same** deploy command that you originally used to deploy Pachyderm. That is, you should specify the same arguments, fields, and storage resources that you specified when deploying the previously utilized version of Pachyderm. The various deploy options/commands are further detailed [here](#). However, it should look something like:

```
$ pachctl deploy <args>
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"
```

After a few minutes, you should then see a healthy Pachyderm cluster running in Kubernetes:

```
$ 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
```

And you can confirm the new version of Pachyderm as follows:

```
pachctl version
COMPONENT      VERSION
pachctl        1.7.0
pachd          1.7.0
```

Common Issues, Questions

Dynamic/static volumes

It is recommended that you deploy Pachyderm using dynamically etcd volumes when possible. If you have deployed Pachyderm using dynamic volumes, you can still use the *same* deploy command to re-deploy Pachyderm (i.e., the

one specifying dynamic etcd volumes). Kubernetes is smart enough to see the previously utilized volumes and re-use them.

etcd re-deploy problems

Depending on the cloud you are deploying to and the previous deployment configuration, we have seen certain cases in which volumes don't get attached to the right nodes on re-deploy (especially when using AWS). In these scenarios, you may see the etcd pod stuck in a Pending, CrashLoopBackoff, or other failed state. Most often, deleting the corresponding etcd pod(s) or nodes (to redeploy them) or re-deploying all of Pachyderm again will fix the issue.

AlreadyExists errors on re-deploy

Occasionally, you might see errors similar to the following:

```
Error from server (AlreadyExists): error when creating "STDIN": secrets "pachyderm-  
↪storage-secret" already exists
```

This might happen when re-deploying the enterprise dashboard, for example. These warning are benign.

Reconnecting pachctl

When you upgrade Pachyderm versions, you may lose your local `port-forward` to connect `pachctl` to your cluster. Alternatively, if you are setting the `ADDRESS` environmental variable manually to connect `pachctl` to your cluster, the IP address for Pachyderm may have changed. To fix this, you can either:

- Re-run `pachctl port-forward &`, or
- Set the `ADDRESS` environmental variable to the update value, e.g., `export ADDRESS=<k8s master IP>:30650`.

Pachyderm Migrations

New versions of Pachyderm often require a migration for some or all of the on disk objects which persist Pachyderm’s metadata for commits, jobs, etc. This document describes how Pachyderm migration works and the best practices surrounding it.

How To Migrate

As of 1.7, Pachyderm’s migration works by extracting objects into a stream of API requests, and replaying those requests onto the newer version of pachd. This process happens automatically using Kubernetes’ “rolling update” functionality. All you need to do is upgrade Pachyderm (with `pachctl deploy`) as further described here. Generally, you will need to:

1. Have version 1.6.10 or later of Pachyderm up and running in Kubernetes.
2. (Optional, but recommended) Create a backup of your cluster state with `pachctl extract` (see [below](#)).
3. Upgrade `pachctl` (see [here](#) for more details).
4. Run `pachctl deploy ...` with whatever arguments you used to deploy Pachyderm previously.

While the migration is running, you will see 2 `pachd` pods running, the one that was already running and the new one. The original `pachd` pod (deployed with the previous version of Pachyderm) will still respond to requests. However, write operations will race with the migration and may not make it to the new cluster. Thus, **you should make sure that all external processes that write data to repos (i.e., calls to `put-file`) or create new pipelines are turned down before migration begins.** You don’t need to worry about pipelines running during the migration process.

Backups

It is highly recommended that you backup your cluster before you perform a migration. This is accomplished with the `pachctl extract` command. Running this command will generate a stream of API requests, similar to the stream used by migration above. This stream can then be used to reconstruct your cluster by running `pachctl restore`. See the docs for `pachctl extract` and `pachctl restore` for further usage.

Before You Migrate 1.6.x to 1.7.x+

1.7 is the first Pachyderm version to support `extract` and `restore` which are necessary for migration. 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. 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.

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
$ pachctl deploy <args> --namespace pachyderm
```

After the Pachyderm pods are up and running, you should see something similar to:

```
$ kubectl get pods --namespace pachyderm
NAME                                READY    STATUS    RESTARTS   AGE
dash-68578d4bb4-mmtbj              2/2      Running   0           3m
etcd-69fcfb5fcf-dgc8j              1/1      Running   0           3m
pachd-784bdf7cd7-7dzxr              1/1      Running   0           3m
```

Note - When using a non-default namespace for Pachyderm, you will have to use the `--namespace` flag for various other `pachctl` command for them to work as expected. These include port-forwarding and undeploy:

```
# forward Pachyderm ports when it was deployed to a non-default namespace
$ pachctl port-forward --namespace pachyderm &

# undeploying Pachyderm when it was deployed to a non-default namespace
$ pachctl undeploy --namespace pachyderm
```

Alternatively or additionally, you might want to set a context similar to the following at the Kubernetes level, such that you can get Pachyderm logs, pod statuses, etc. easily via `kubectl logs ...`, `kubectl get pods`, etc.:

```
$ kubectl config set-context pach --namespace=<pachyderm namespace> \
  --cluster=<cluster> \
  --user=<user>
```


RBAC

Pachyderm has support for Kubernetes Role-Based Access Controls (RBAC). This support is a default part of all Pachyderm deployments, there's nothing special for you to do as a user. You can see the ClusterRole which is created for Pachyderm's service account by doing:

```
kubectl get clusterrole/pachyderm -o json
```

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
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.

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).

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.

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.

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*

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 `lazy input`. That way, your shuffling pipeline can skip both the download and the upload. An example for this type of shuffle pipeline is [here](#)

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.

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](#).

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!

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!

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.

General Troubleshooting

Here are some common issues by symptom along with steps to resolve them. They are organized into the following categories:

- *Deploying a Pachyderm cluster*
- *Connecting to a Pachyderm cluster*
- *Problems running pipelines*

Deploying A Pachyderm Cluster

- *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-1333950811-0smlp	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-1333950811-0sm1p
```

If you see an error including `Error attaching EBS volume` or similar, see the recourse for that error here under the corresponding section below this one. 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-1333950811-0sm1p_default (a92b6665-506a-11e7-8e07-
→02e3d74c49ac) "
```

That 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-1333950811-0sm1p
```

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 likely reveal a misconfiguration in your deploy. For example, you might see, `BucketRegionError: incorrect region, the bucket is not in 'us-west-2' region`. In that case, you've deployed your bucket in a different region than your cluster.

If the error / recourse isn't obvious from the error message, you can now provide the content of the `pachd` logs when getting help in our Slack channel or by opening a GitHub Issue. Please 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
```

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, first find the node of which the pod is scheduled. In the output of the `kubectl describe po/pachd-xxx` command above, you should see the name of the node on which the pod is running. In the AWS web console, find that node.. Once you have the right node, look in 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.

Connecting to a Pachyderm Cluster

- *Cannot connect via `pachctl` - context deadline exceeded*
- *Certificate error when using `kubectl`*
- *Uploads/downloads are slow*

Cannot connect via `pachctl` - context deadline exceeded

Symptom

You may be using the environmental variable `ADDRESS` to specify how `pachctl` talks to your Pachyderm cluster, or you may be forwarding the pachyderm port via `pachctl port-forward`. In any event, you might see something similar to:

```
$ echo $ADDRESS
1.2.3.4:30650
$ pachctl version
COMPONENT      VERSION
pachctl        1.4.8
context deadline exceeded
```

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`), but it can also be seen when running `pachctl port-forward` because it uses `kubectl` under the hood. 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

Check if you're using port-forwarding. Port forwarding throttles traffic to ~1MB/s. If you need to do large downloads/uploads you should consider using the `ADDRESS` variable instead to connect directly to your k8s master node. See this note

You'll also want to make sure you've allowed ingress access through any firewalls to your k8s cluster.

Problems Running Pipelines

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:

```
$kubectl 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](#)

Deploy Specific Troubleshooting

Here are some common issues by symptom related to certain deploys. They are organized into the following categories:

- [AWS](#)
 - Google - coming soon...
 - Azure - coming soon...
-

AWS Deployment

- *Can't connect to to the Pachyderm cluster after a rolling update*
- *The one shot deploy script, `aws.sh`, never completes*
- *VPC limit exceeded*
- *GPU node never appears*

Can't connect to the Pachyderm cluster after a rolling update

Symptom

After running `kops rolling-update`, `kubectl` (and/or `pachctl`) cannot connect to the cluster. All `kubectl` requests hang.

Recourse

First get your cluster name. This will be in the deploy logs you saved from running `aws.sh` (if you utilized the [one shot deployment](#)), or can be retrieved via `kops get clusters`.

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 navigate there, open the AWS web console for EC2. Check to see if you have any instances with names like::

```
master-us-west-2a.masters.tfgpu.kubernetes.com
nodes.tfgpu.kubernetes.com
```

If not, 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.
```

Examples

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](#)

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](#)

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](#)

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](#)

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

Machine Learning

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

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

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

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

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

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](#)

Splitting Data for Distributed Processing

As described in the [distributed computing with Pachyderm docs](#), Pachyderm allows 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 data set that you want or need to commit into Pachyderm as a single file, rather than a bunch of smaller files (e.g., one per record) that are easily mapped to datums. In these cases, Pachyderm provides an easy way to automatically split your data set for subsequent distributed computing.

Let’s say that we have a data set consisting of information about our users. This data is in CSV format in a single file, `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 we just put this into Pachyderm as a single file, we could not subsequently process each of these user records in parallel as separate “datums” (see [this guide](#) for more information on datums and distributed computing). Of course, you could manually separate out each of these user records into separate files before you commit them into the `users` repo or via a pipeline stage dedicated to this splitting task. This would work, but Pachyderm actually makes it much easier for you.

The `put-file` API includes an option for splitting up the file into separate datums automatically. You can do this with the `pachctl` CLI tool via the `--split` flag on `put-file`. For example, to automatically split the `user_data.csv` file up into separate datums for each line, you could execute the following:

```
$ pachctl put-file users master -c -f user_data.csv --split line --target-file-datums_
↪ 1
```

The `--split line` argument specifies that Pachyderm should split this file on lines, and the `--target-file-datums 1` arguments specifies that each resulting file should include at most one “datum” (or one line). Note, that Pachyderm will still show the `user_data.csv` entity to you as one entity in the repo:

```
$ pachctl list-file users master
NAME                TYPE      SIZE
user_data.csv      dir      5.346 KiB
```

But, this entity is now a directory containing all of the split records:

```
$ pachctl list-file users master user_data.csv
NAME                                     TYPE      SIZE
user_data.csv/0000000000000000000    file      43 B
user_data.csv/00000000000000000001    file      39 B
user_data.csv/00000000000000000002    file      37 B
user_data.csv/00000000000000000003    file      34 B
user_data.csv/00000000000000000004    file      35 B
user_data.csv/00000000000000000005    file      41 B
user_data.csv/00000000000000000006    file      32 B
etc...
```

A pipeline that then takes the repo `users` as input with a glob pattern of `/user_data.csv/*` would process each user record (i.e., each line of the CSV) in parallel.

This is, of course, just one example. Right now, Pachyderm supports this type of splitting on lines or on JSON blobs. Here are a few more examples:

```
# Split a json file on json blobs, putting
# each json blob into it's own file.
$ pachctl put-file users master -c -f user_data.json --split json --target-file-
↳ datums 1

# Split a json file on json blobs, putting
# 3 json blobs into each split file.
$ pachctl put-file users master -c -f user_data.json --split json --target-file-
↳ datums 3

# Split a file on lines, putting each 100
# bytes chunk into the split files.
$ pachctl put-file users master -c -f user_data.txt --split line --target-file-bytes-
↳ 100
```

Specifying Header/Footer

Additionally, if your data has a common header or footer, you can specify these manually via `pachctl put-header` or `pachctl put-footer`. This is helpful for CSV data.

To do this, you'll need to specify the header/footer on the *parent directory* of your data. It's a little “magical”, but you're essentially embedding the header/footer into the directory and then Pachyderm will apply that header/footer to all the files in that directory. Below we have an example of splitting a CSV with a header, then setting the header explicitly. Notice that once we've set the header, whenever we get a file under that directory, the header is applied. You can still use glob patterns to get all the data under the directory, and in that case the header is still applied.

```
# Raw CSV
$ cat users.csv
id,name,email
4,alice,aaa@place.com
7,bob,bbb@place.com

# Take the raw CSV data minus the header and split it into multiple files:
$ cat users.csv | tail -n +2 | pachctl put-file bar master users --split line
Reading from stdin.
$ pachctl list-file bar master
NAME  TYPE  SIZE
```

```

users dir 42B
$ pachctl list-file bar master /users/
NAME                                TYPE SIZE
/users/000000000000000000 file 22B
/users/000000000000000001 file 20B
# Before we set the header, we just see the raw data when we issue a get-file
$ pachctl get-file bar master /users/000000000000000000
4,alice,aaa@place.com

# Now we take the CSV header and apply it to the directory:
$ cat users.csv | head -n 1 | pachctl put-header bar master users
# Now when we read an individual file, we see the header plus the contents
$ pachctl get-file bar master /users/000000000000000000
id,name,email
4,alice,aaa@place.com

# If you issue a get-file on the directory, it returns just the header/footer
$ pachctl get-file bar master /users
id,name,email
# We can get the entire CSV file back with:
$ pachctl get-file bar master /users/*
id,name,email
4,alice,aaa@place.com
7,bob,bbb@place.com

# Delete the existing header:
$ echo "" | pachctl put-header repo branch path -f -
# We've now deleted the header
pachctl get-file bar master /users/*
4,alice,aaa@place.com
7,bob,bbb@place.com

```

For more info, such as how to delete a header/footer, see `pachctl put-header --help`.

PG Dump / SQL Support

You can also ingest data from postgres using split file.

1. Generate your PG Dump file

```

$ pg_dump -t users -f users.pgdump
$ 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
--
```

1. Ingest SQL data using split file

When you use `pachctl put-file --split sql ...` your pg dump file is split into three parts - the header, rows, and the footer. The header contains all the SQL statements in the pg dump that setup 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 on the directory containing the rows. This way, if you request a `get-file` on the directory, you'll get just the header and footer. If you request an individual file, you'll see the header plus the row(s) plus the footer. If you request all the files with a glob pattern, e.g. `/directoryname/*`, you'll receive the header plus all the rows plus the footer, recreating the full pg dump. In this way, you can construct full or partial pg dump files so that you can load full or partial data sets.

```
$ pachctl put-file data master -f users.pgdump --split sql
$ pachctl put-file data master users --split sql -f users.pgdump
$ pachctl list-file data master
NAME                TYPE SIZE
users               dir  914B
$ pachctl list-file data master /users
NAME                TYPE SIZE
/users/0000000000000000    file 20B
/users/0000000000000001    file 18B
```

Then in your pipeline (where you've started and forked postgres), you can load the data by doing something like:

```
$ cat /pfs/data/users/* | sudo -u postgres psql
```

And with a glob pattern `/*` this code would load each raw postgres chunk into your postgres instance for processing by your pipeline.

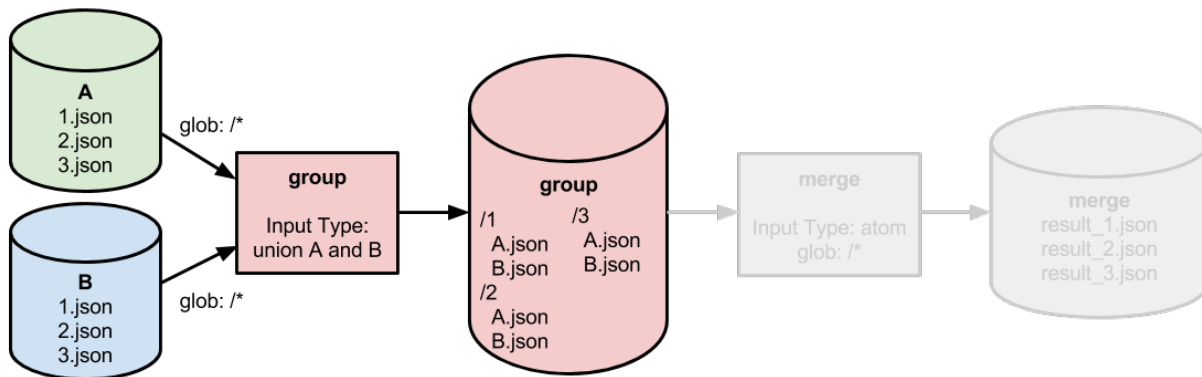
For this use case, you'll likely want to use `--target-file-datums` or `--target-file-bytes` since it's likely that you'll want to run your queries against many rows at a time.

Combining/Merging/Joining Data

There are a variety of use cases in which you would want to match datums from multiple data repositories to do some combined processing, joining, or aggregation. For example, you may need to process multiple records corresponding to a certain user, a certain experiment, or a certain device together. In these scenarios, we recommend a 2-stage method of merging your data:

1. *A first pipeline* that groups all of the records for a specific key/index.
2. *A second pipeline* that takes that grouped output and performs the merging, joining, or other processing for the group.

1. Grouping records that need to be processed together



Let's say that we have two repositories containing JSON records, A and B. These repositories may correspond to two experiments, two geographic regions, two different devices generating data, etc. In any event, the repositories look similar to:

```
$ pachctl list-file A master
NAME                TYPE      SIZE
1.json              file      39 B
2.json              file      39 B
3.json              file      39 B
$ pachctl list-file B master
NAME                TYPE      SIZE
1.json              file      39 B
2.json              file      39 B
3.json              file      39 B
```

We need to process A/1.json with B/1.json to merge their contents or otherwise process them together. Thus, we need to group each set of JSON records into respective “datums” that can each be processed together by our *second pipeline* (read more about datums and distributed processing [here](#)).

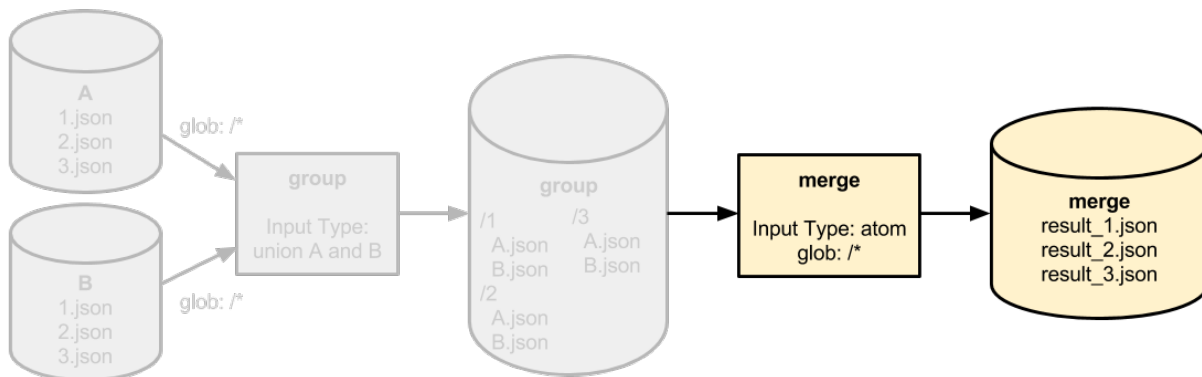
The first pipeline takes a union of A and B as inputs, each with glob pattern /*. As each JSON file is processed, it is copied to a folder in the output corresponding to the key/index for that record (in this case, just the number in the file name). It is also re-named to a unique name corresponding to its source:

```
/1
  A.json
  B.json
/2
  A.json
  B.json
/3
  A.json
  B.json
```

Note, that when performing this grouping:

- You should use "lazy": true to avoid unnecessary downloads of data.
- You should use sym-links to avoid unnecessary uploads of data and unnecessary duplication of data (see more information on “copy elision” [here](#)).

2. Processing the grouped records



Once the records that need to be processed together are grouped by the first pipeline, our second pipeline can take the `group` repository as input with a glob pattern of /*. This will let the second pipeline process each grouping of records in parallel.

The second pipeline will perform any merging, aggregation, or other processing on the respective grouping of records and could, for example, 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
```

Implications and Notes

- This 2-stage pattern of combining data could be used for merging or grouped processing of data from various experiments, devices, etc. However, the same pattern can be applied to perform distributed joins of tabular data or data from database tables. For example, you could join user email records together with user IP records on the key/index of a user ID.
- Each of the 2 stages can be parallelized across workers to scaled with the size of your data and the number of data sources that you are merging.
- In some cases, your data may not be split into separate files for each record. In these cases, you can utilize Pachyderm splitting functionality to prepare your data for this sort of distributed merging/joining.

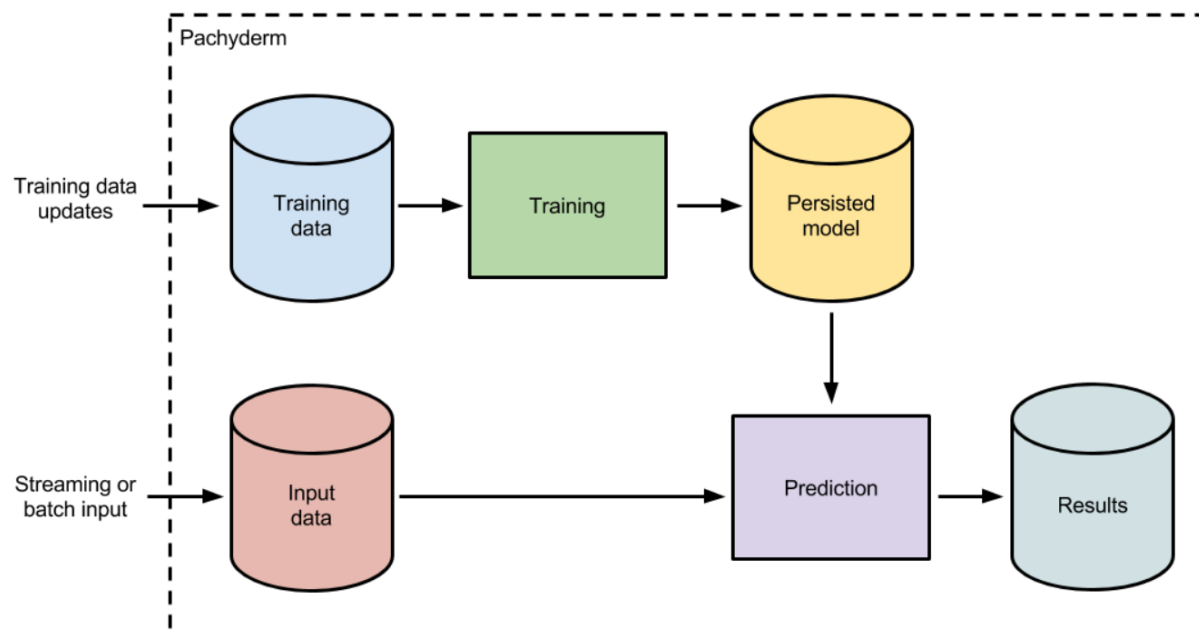
Creating Machine Learning Workflows

Because Pachyderm is language/framework agnostic and because it easily distributes analyses over large data sets, data scientists can use whatever tooling they like for ML. Even if that tooling isn't familiar to the rest of an engineering organization, data scientists can autonomously develop and deploy scalable solutions via containers. Moreover, Pachyderm's pipelining logic paired with data versioning, allows any results to be exactly reproduced (e.g., for debugging or during the development of improvements to a model).

We recommend combining model training processes, persisted models, and a model utilization processes (e.g., making inferences or generating results) into a single Pachyderm pipeline DAG (Directed Acyclic Graph). Such a pipeline allows us to:

- Keep a rigorous historical record of exactly what 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 is not performing or when "bad data" is introduced into a training data set.

This sort of sustainable ML pipeline looks like this:



A data scientist can update the training dataset at any time to automatically train a new persisted model. This training could utilize any language or framework (Spark, Tensorflow, scikit-learn, etc.) and output any format of persisted

model (pickle, XML, POJO, etc.). Regardless of framework, the model will be versioned by Pachyderm, and you will be able to track what “Input data” was input into which model AND exactly what “Training data” was used to train that model.

Any new input data coming into the “Input data” repository will be processed with the updated model. Old predictions can be re-computed with the updated model, or new models could be backtested on previously input and versioned data. This will allow you to avoid manual updates to historical results or having to worry about how to swap out ML models in production!

Examples

We have implemented this machine learning workflow in [some example pipelines](#) using a couple of different frameworks. These examples are a great starting point if you are trying to implement ML in Pachyderm.

Processing Time-Windowed Data

If you are analyzing data that is changing over time, chances are that you will want to perform some sort of analysis on “the last two weeks of data,” “January’s data,” or some other moving or static time window of data. There are a few different ways of doing these types of analyses in Pachyderm, depending on your use case. We recommend one of the following patterns for:

1. *Fixed time windows* - for rigid, fixed time windows, such as months (Jan, Feb, etc.) or days (01-01-17, 01-02-17, etc.).
2. *Moving or rolling time windows* - for rolling time windows of data, such as three day windows or two week windows.

Fixed time windows

As further discussed in [Creating Analysis Pipelines](#) and [Distributed Computing](#), the basic unit of data partitioning in Pachyderm is a “datum” which is defined by a glob pattern. When analyzing data within fixed time windows (e.g., corresponding to fixed calendar times/dates), we recommend organizing your data repositories such that each of the time windows that you are going to analyze corresponds to a separate files or directories in your repository. By doing this, you will be able to:

- 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 JSON sales data that need to be analyzed, you could create a `sales` data repository and structure it like:

```
sales
-- January
|  -- 01-01-17.json
|  -- 01-02-17.json
|  -- etc...
-- February
|  -- 01-01-17.json
|  -- 01-02-17.json
|  -- etc...
-- March
|  -- 01-01-17.json
|  -- 01-02-17.json
|  -- etc...
```

When you run a pipeline with an input repo of `sales` having a glob pattern of `/*`, each month's worth of sales data is processed in parallel (if possible). Further, when you add new data into a subset of the months or add data into a new month (e.g., May), only those updated datums will be re-processed.

More generally, this structure allows you to create:

- Pipelines that aggregate, or otherwise process, daily data on a monthly basis via a `/*` glob pattern.
- Pipelines that only analyze a certain month's data via, e.g., a `/January/*` or `/January/` glob pattern.
- Pipelines that process data on a daily basis via a `/*/*` glob pattern.
- Any combination of the above.

Moving or rolling time windows

In certain use cases, you need to run analyses for moving or rolling time windows, even when those don't correspond to certain calendar months, days, etc. For example, you may need to analyze the last three days of data, the three days of data prior to that, the three days of data prior to that, etc. 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 rolling/moving time windows.
2. Maintain a time windowed set of data corresponding to the latest of the rolling/moving time windows.

Binning data into rolling/moving time windows

In this method of processing rolling time windows, we'll use a two-pipeline [DAG](#) to analyze time windows efficiently:

- *Pipeline 1* - Read in data, determine which bins the data corresponds to, and write the data into those bins
- *Pipeline 2* - Read in and analyze the binned data.

By splitting this analysis into two pipelines we can benefit from 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. Now we can scale the pipelines easily as the number of files increases.

Let's take the three day rolling time windows as an example, and let's say that we want to analyze three day rolling windows of sales data. In a first repo, called `sales`, a first day's worth of sales data is committed:

```
sales
-- 01-01-17.json
```

We then create a first pipeline to bin this into a repository directory corresponding to our 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 our next day's worth of sales is committed,

```
sales
-- 01-01-17.json
-- 01-02-17.json
```


the first pipeline executes again to bin the 01-02-17 data into any relevant bins. In this case, we would put it in the previously created bin for 01-01-17 to 01-03-17, but we would also put it into a bin 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

```

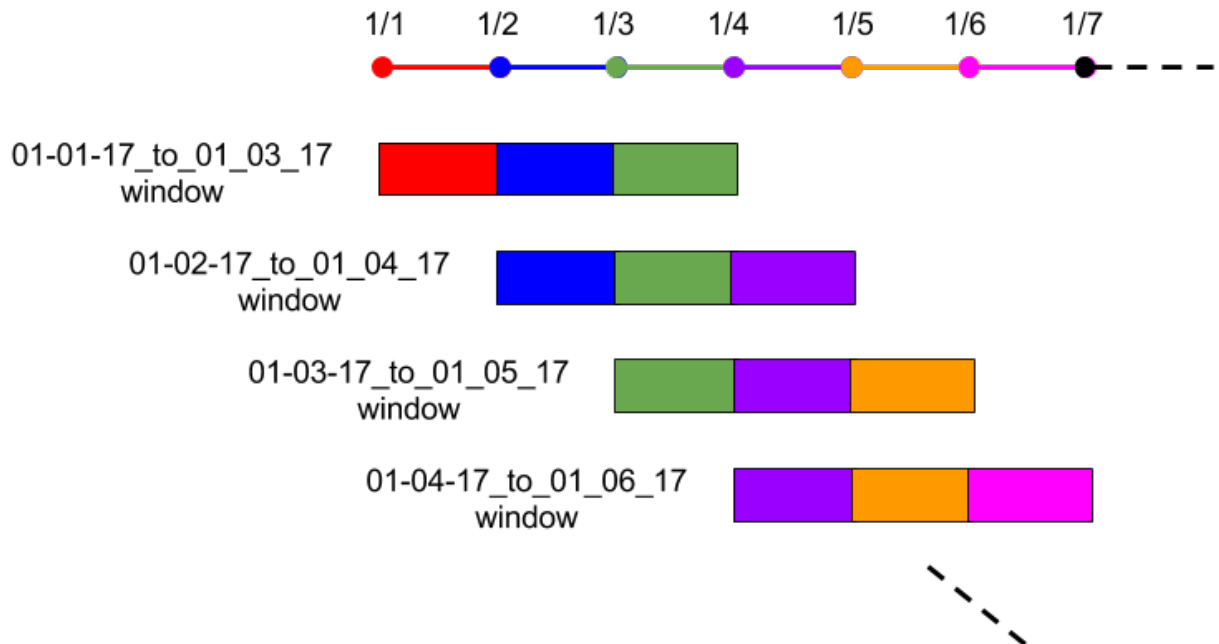
As more and more daily data is added, you will end up with a directory structure that looks like:

```

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
-- etc...

```

and is maintained over time as new data is committed:



Your second pipeline can then process these bins in parallel, via a glob pattern of `/*`, or in any other relevant way as discussed further in the “*Fixed time windows*” section. Both your first and second pipelines can be easily parallelized.

Note - When looking at the above directory structure, it may seem like there is an unnecessary duplication of the data. 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 allow you to process these types of rolling time windows.

Note - It might also seem as if there is unnecessary data transfers over the network to perform the above binning. Pachyderm can ensure that performing these types of “shuffles” doesn’t actually require transferring data over the network. Read more about that [here](#).

Maintaining a single time-windowed data set

The advantage of the binning pattern above is that any of the rolling time windows are available for processing. They can be compared, aggregated, combined, etc. in any way, and any results or aggregations are kept in sync with updates to the bins. However, you do need to put in some logic 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. It also involves two pipelines:

- *Pipeline 1* - Read in data, determine which files belong in your moving time window, and write the relevant files into an updated version of the moving time-windowed data set.
- *Pipeline 2* - Read in and analyze the moving time-windowed data set.

Let’s utilize our sales example again to see how this would work. In the example, we want to keep a moving time window of the last three days worth of data. Now say that our daily `sales` repo looks like the following:

```
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, our first pipeline pulls out the last three days of data and arranges it like so:

```
last_three_days
-- 01-02-17.json
-- 01-03-17.json
-- 01-04-17.json
```

Think of this as a “shuffle” step. Then, when the January 5th file, `01-05-17.json`, is committed,

```
sales
-- 01-01-17.json
-- 01-02-17.json
-- 01-03-17.json
-- 01-04-17.json
-- 01-05-17.json
```

the first pipeline would again update the moving window:

```
last_three_days
-- 01-03-17.json
-- 01-04-17.json
-- 01-05-17.json
```

Whatever analysis we need to run on the moving windowed data set in `moving_sales_window` can use a glob pattern of `/` or `/*` (depending on whether we need to process all of the time windowed files together or they can be processed in parallel).

Warning - When creating this type of moving time-windowed data set, the concept of “now” or “today” is relative. It is important that you make a sound choice for how to define time based on your use case (e.g., by defaulting to UTC). You should not use a function such as `time.now()` to figure out a current day. The actual time at which this

analysis is run may vary. If you have further questions about this issue, please do not hesitate to reach out to us via [Slack](#) or at support@pachyderm.io.

Utilizing GPUs

Pachyderm has support for utilizing GPUs within Pachyderm pipelines (e.g., for training machine learning models). To do this you will need to:

1. Create a Docker image that is able to utilize GPUs
2. Write a pipeline spec that specifies GPU nodes
3. Deploy a GPU enabled pachyderm cluster

For a concrete example, see our [example Tensorflow pipeline](#) for image-to-image translation, which includes a pipeline specification for running model training on a GPU node.

Creating a GPU Enabled Docker Image

For your Docker image, you'll want to use or build an image that can utilize GPU resources. If you are using Tensorflow, for example, you could build your Docker image FROM the public GPU enabled Tensorflow image:

```
FROM tensorflow/tensorflow:0.12.0-gpu
...
```

Or you might follow [this guide](#) for working with Docker and NVIDIA (although we haven't full tested this guide).

Writing Your Pipeline Specification

Ensuring that your environment can access GPU drivers

You can bake this into your Docker image in some cases, but other images, such as the TensorFlow base image, may require that you explicitly tell your application about shared libraries (e.g., CUDA). To do that, you may need to set one or more environmental variables. This will be application/framework dependent. For example, if we were using the Tensorflow base image, we would need to make sure that we set `LD_LIBRARY_PATH`, such that TensorFlow knows about CUDA:

```
LD_LIBRARY_PATH="/usr/lib/nvidia:/usr/local/cuda/lib64:/rootfs/usr/lib/x86_64-linux-
↳gnu"
```

Again, this can be baked into your Docker image via an `ENV` statement in your Dockerfile:

```
ENV LD_LIBRARY_PATH /usr/lib/nvidia:/usr/local/cuda/lib64:/rootfs/usr/lib/x86_64-
↳linux-gnu
```

or it can be defined in your [pipeline specification](#) via the `env` field (as shown below).

Creating your pipeline specification with access to GPU resources

In addition to properly setting up the environment, we need to tell the Pachyderm cluster that our pipeline needs a GPU resource. To do that we'll add a `gpu` entry to the `resources` field in the [pipeline specification](#).

An example pipeline definition for a GPU enabled Pachyderm Pipeline is as follows:

```
{
  "pipeline": {
    "name": "train"
  },
  "transform": {
    "image": "acme/your-gpu-image",
    "cmd": [
      "python",
      "train.py"
    ],
    "env": {
      "LD_LIBRARY_PATH": "/usr/lib/nvidia:/usr/local/cuda/lib64:/rootfs/usr/lib/x86_
↪64-linux-gnu"
    }
  },
  "resource_requests": {
    "gpu": 1
  },
  "inputs": {
    "atom": {
      "repo": "data",
      "glob": "/*"
    }
  }
}
```

Deploy a GPU Enabled Pachyderm Cluster

NOTE: You can also *test Pachyderm + GPUs locally*

To deploy a Pachyderm cluster with GPU support we assume:

- You're using `kops` for your deployment (which means you're using AWS or GCE, not GKE because k8s is done for you). Other deploy methods are available, but these are the ones we've tested most thoroughly.
- You have a working pachyderm cluster already up and running that you can connect to with `kubectl`.

Add GPU nodes to your k8s cluster

You can create GPU nodes by first using `kops` to create a new instance group:

```
$ kops create ig gpunodes --name XXXXXX-pachydermcluster.kubernetes.com --state s3://
↪k8scom-state-store-pachyderm-YYYYY --subnet us-west-2c
```

where your specification will look something like:

```

1 apiVersion: kops/v1alpha2
2 kind: InstanceGroup
3 metadata:
4   creationTimestamp: null
5   name: gpunodes
6 spec:
7   image: kope.io/k8s-1.5-debian-jessie-amd64-hvm-ebs-2017-01-09
8   machineType: p2.xlarge
9   maxSize: 1
10  minSize: 1
11  role: Node
12  subnets:
13  - us-west-2c

```

In this example we used Amazon’s p2.xlarge instance which contains a single GPU node.

Node - If you upped your `rootVolumeSize` (and set the `rootVolumeType` in your other instance group), you should do the same here. In the absence of GPU jobs, normal jobs could get scheduled on this node, in which case you’ll have the same disk requirements as the rest of your cluster. There is currently no way of setting “disk” resource requests, so we have to use a convention instead.

Enable GPUs only on the GPU nodes

Again, you can use `kops` to edit your new instance group:

```

$ kops edit ig gpunodes --name XXXXXXXX-pachydermcluster.kubernetes.com --state s3://
↪k8scom-state-store-pachyderm-YYYY

```

and add the fields:

```

spec:
...
  hooks:
  - execContainer:
      image: pachyderm/nvidia_driver_install:dcde76f919475a6585c9959b8ec41334b05103bb
  kubelet:
    featureGates:
      Accelerators: "true"

```

Note: It’s YAML and spaces are very important. Also, if you see “fields were not recognized,” you likely need to update the version of `kops`.

These lines provide an image that gets run on every GPU node’s startup. This image will install the NVIDIA drivers on the host machine, update the host machine to mount the device at startup, and restart the host machine.

The feature gate enables k8s GPU detection. That’s what gives us the `alpha.kubernetes.io/nvidia-gpu: "1"` resources.

Update your cluster

Finally, we “update” our cluster to actually make the above changes:

```

$ kops update cluster --name XXXXXXXX-pachydermcluster.kubernetes.com --state s3://
↪k8scom-state-store-pachyderm-YYYY --yes

```

This will spin up the new `gpunodes` instance group, and apply the changes to your kops cluster.

Sanity check

You'll know the cluster is ready to schedule GPU resources when:


- you see the new node in the output of `kubectl get nodes` and the state is `Ready`, and
- the node has the `alpha.kubernetes.io/nvidia-gpu: "1"` field set (and the value is 1 not 0)

```
$ kubectl get nodes/ip-172-20-38-179.us-west-2.compute.internal -o yaml | grep nvidia
alpha.kubernetes.io/nvidia-gpu: "1"
alpha.kubernetes.io/nvidia-gpu-name: Tesla-K80
alpha.kubernetes.io/nvidia-gpu: "1"
alpha.kubernetes.io/nvidia-gpu: "1"
```

Deal with known issues (if necessary)

If you're not seeing the node, its possible that your resource limits (from your cloud provider) are preventing you from creating the GPU node(s). You should check your resource limits and ensure that GPU nodes are available in your region/zone (as further discussed here).

If you have checked your resource limits and everything seems ok, its very possible that you're hitting a [known k8s bug](#). In this case, you can try to overcome the issue by restarting the k8s api server. To do that, run:

```
$ kubectl --namespace=kube-system get pod | grep kube-apiserver | cut -f 1 -d " " |  while read pod; do kubectl --namespace=kube-system delete po/$pod; done
```

It can take a few minutes for the node to get recognized by the k8s cluster again.

Test Locally

NOTE - This has only been tested on a linux machine.

If you want to test that your pipeline is working on a local cluster (you're Pachyderm in a local cluster), you can do so, but you'll need to attach the NVIDIA drivers correctly. There are two methods for this:

1. Fresh install

Install the NVIDIA drivers locally if you haven't already. If you're not sure, run `which nvidia-smi`. If it returns no result, you probably don't have them installed. To install them, you can run the following command. Warning! This command will restart your system and will modify your `/etc/rc.local` file, which you may want to backup.

```
$ sudo /usr/bin/docker run -v /:/rootfs/ -v /var/run/dbus:/var/run/dbus -v /run/
↳ systemd:/run/systemd --net=host --privileged pachyderm/nvidia_driver_install:
↳ dcde76f919475a6585c9959b8ec41334b05103bb
```

After the restart, you should see the nvidia devices mounted:

```
$ ls /dev | grep nvidia
nvidia0
nvidiactl
```



```
nvidia-modeset
nvidia-uvm
```

At this point your local machine should be recognized by kubernetes. To check you'll do something like:

```
$ kubectl get nodes
NAME          STATUS    AGE           VERSION
127.0.0.1     Ready    13d          v1.7.10
$ kubectl get nodes/127.0.0.1 -o yaml | grep nvidia
  alpha.kubernetes.io/nvidia-gpu: "1"
  alpha.kubernetes.io/nvidia-gpu-name: Quadro-M2000M
  alpha.kubernetes.io/nvidia-gpu: "1"
  alpha.kubernetes.io/nvidia-gpu: "1"
```

If you don't see any `alpha.kubernetes.io/nvidia-gpu` fields it's likely that you didn't deploy k8s locally with the correct flags. An example of the right flags can be found [here](#). You can clone `git@github.com:pachyderm/pachyderm` and run `make launch-kube` locally if you're already running docker on your local machine.

2. Hook in existing drivers

Pachyderm expects to find the shared libraries it needs under `/usr/lib`. It mounts in `/usr/lib` into the container as `/rootfs/usr/lib` (only when you've specified a GPU resource). In this case, if your drivers are not found, you can update the `LD_LIBRARY_PATH` in your container as appropriate.

Triggering Pipelines Periodically (cron)

Pachyderm pipelines are triggered by changes to their input data repositories (as further discussed in [What Happens When You Create a Pipeline](#)). However, if a pipeline consumes data from sources outside of Pachyderm, it can't use Pachyderm's triggering mechanism to process updates from those sources. For example, you might need to:

- Scrape websites
- Make API calls
- Query a database
- Retrieve a file from S3 or FTP

You can schedule pipelines like these to run regularly with Pachyderm's built-in `cron` input type. You can find an example pipeline that queries MongoDB periodically [here](#).

There are two types of cron inputs:

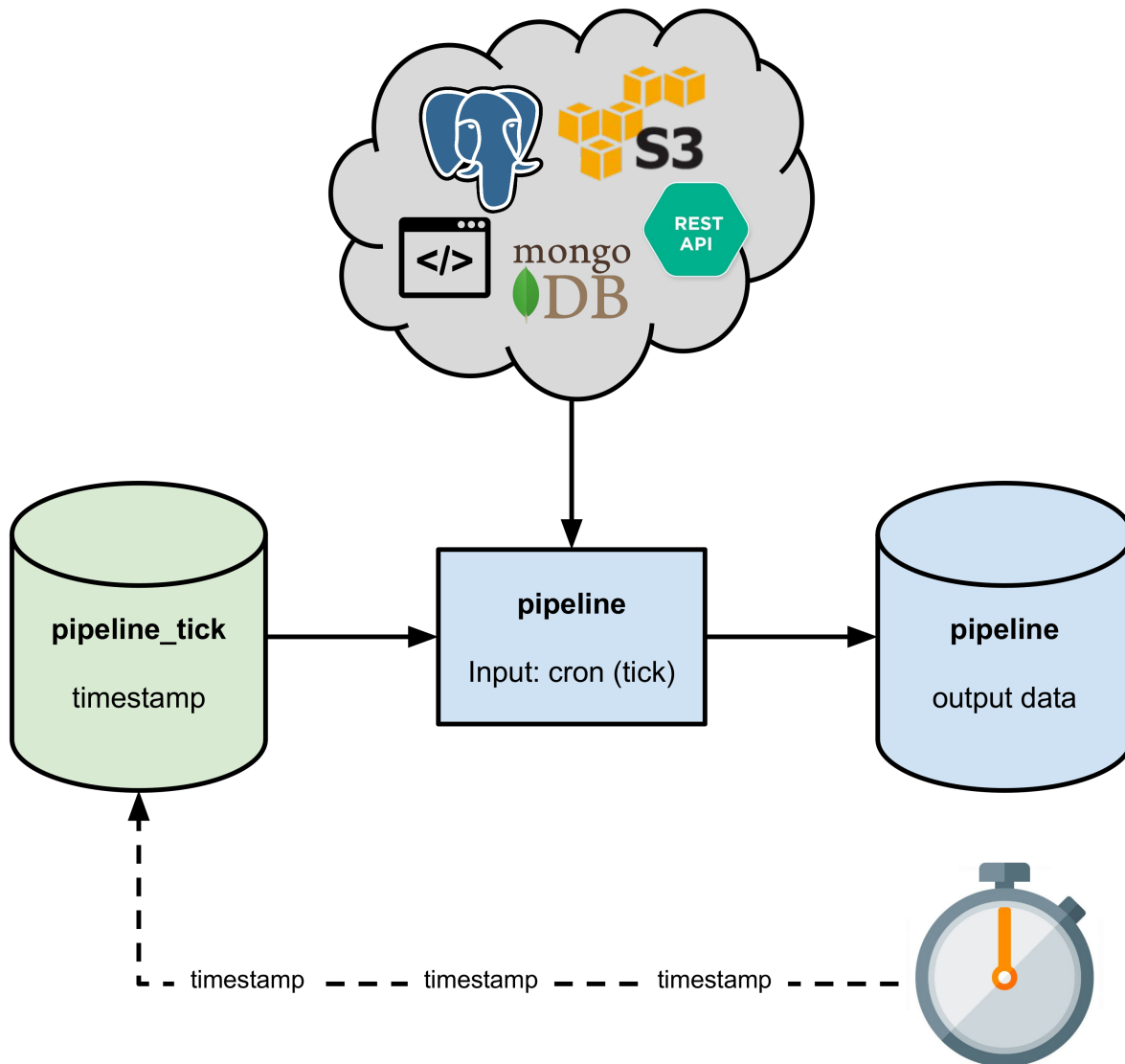
- *Non-incremental cron input* - For when you want to overwrite a single dataset regularly
- *Incremental cron input* - For when you want to store new datasets alongside previous results.

Non-Incremental Cron

Let's say that we want to query a database every 10 seconds and update our dataset every time the pipeline is triggered. We could do this with a non-incremental `cron` input as follows:

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

When we create this pipeline, Pachyderm will create a new input data repository corresponding to the `cron` input. It will then automatically commit an updated timestamp file every 10 seconds to the `cron` input repository, which will automatically trigger our pipeline.



The pipeline will run every 10 seconds, querying our database and updating its output.

We have used the `@every 10s` cron spec here, but you can use any cron spec formatted according to [RFC 3339](#). For example, `*/10 * * * *` would indicate that the pipeline should run every 10 minutes (these time formats should be familiar to those who have used cron in the past, and you can find more examples [here](#))

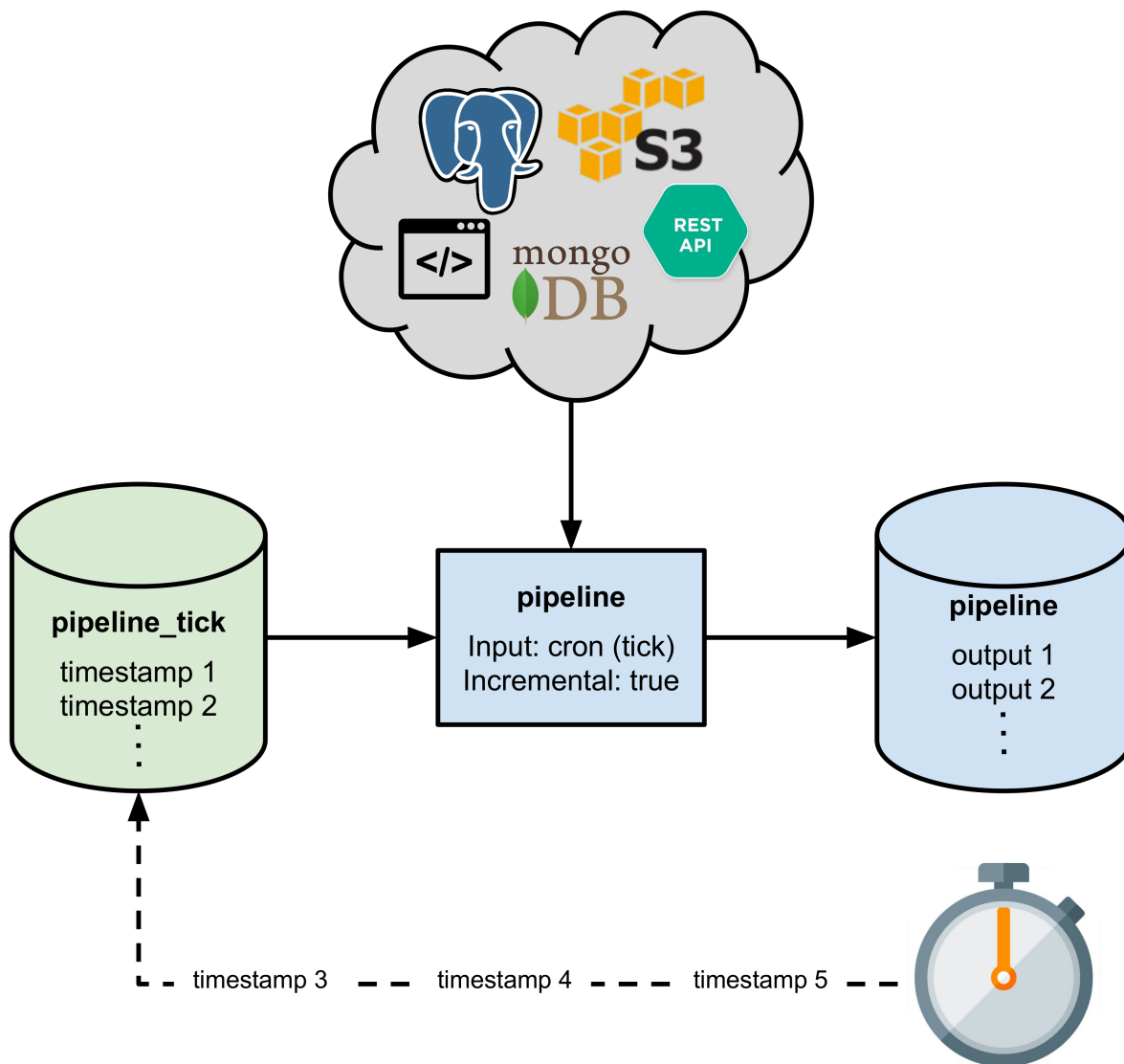
Incremental Cron

In the *above example*, Pachyderm will overwrite the output data from our `cron` triggered pipeline each time it runs. This happens because Pachyderm is updating the same input datum (the timestamp) after every period (see our incremental processing docs for more information on datums and incrementality).

If we don't want to replace our previous datasets during every run, we must enable incrementality in the pipeline specification:

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

When we do this, Pachyderm won't update the same timestamp in the `cron` data repository, and we can accumulate results periodically over time:



Note: even with `"incremental": true` you can still overwrite data in the output data repository (e.g. by replacing a file with a new file having the same name). The point is that the pipeline controls this process—it's not

automatic.

Deferred Processing of Data

While they're running, Pachyderm Pipelines will process any new data you commit to their input branches. This can be annoying in cases where you want to commit data more frequently than you want to process.

This is generally not an issue because Pachyderm pipelines are smart about not reprocessing things they've already processed, but some pipelines need to process everything from scratch. For example, you may want to commit data every hour, but only want to retrain a machine learning model on that data daily since it needs to train on all the data from scratch.

In these cases there's a massive performance benefit to deferred processing. This document covers how to achieve that and control exactly what gets processed when using the Pachyderm system.

The key thing to understand about controlling when data is processed in Pachyderm is that you control this using the *filesystem*, rather than at the pipeline level. Pipelines are inflexible but simple, they always try to process the data at the heads of their input branches. The filesystem, on the other hand, is much more 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. The examples below describe how specifically this should work for common cases.

Using a staging branch

The simplest and most common pattern for deferred processing is using a `staging` branch in addition to the usual `master` branch that the pipeline takes as input. To begin, create your input repo and your pipeline (which by default will read from the `master` branch). This will automatically create a branch on your input repo called `master`. You can check this with `list-branch`:

```
$ pachctl list-branch data
BRANCH HEAD
master -
```

Notice that the head commit is empty. This is why the pipeline has no jobs as pipelines process the `HEAD` commit of their input branches. No `HEAD` commit means no processing. If you were to commit data to the `master` branch, the pipeline would immediately kick off a job to process what you committed. However, if you want to commit something without immediately processing it you need to commit it to a different branch. That's where a `staging` branch comes in – you're essentially adding your data into a staging area to then process later.

Commit a file to the staging branch:

```
$ pachctl put-file data staging -f <file>
```

Your repo now has 2 branches, `staging` and `master` (`put-file` automatically creates branches if they don't exist). If you do `list-branch` again you should see:

```
$ pachctl list-branch data
BRANCH HEAD
staging f3506f0fab6e483e8338754081109e69
master -
```

Notice that `master` still doesn't have a head commit, but the new branch, `staging`, does. There still have been no jobs, because there are no pipelines taking `staging` as inputs. You can continue to commit to `staging` to add new data to the branch and it still won't process anything. True to its name, it's acting as a staging ground for data.

When you're ready to actually process the data all you need to do is update the master branch to point to the head of the staging branch:

```
$ pachctl create-branch data master --head staging
$ pachctl list-branch
staging f3506f0fab6e483e8338754081109e69
master f3506f0fab6e483e8338754081109e69
```

Notice that `master` and `staging` now have the same head commit. This means that your pipeline finally has something to process. If you do `list-job` you should see a new job. Notice that even if you created multiple commits on `staging` before updating `master` you still only get 1 job. Despite the fact that those other commits are ancestors of the current HEAD of `master`, they were never the actual HEAD of `master` themselves, so they don't get processed. This is often fine because commits in Pachyderm are generally additive, so processing the HEAD commit also processes data from previous commits.

However, sometimes you want to process specific intermediary commits. To do this, all you need to do is set `master` to have them as HEAD. For example if you had 10 commits on `staging` and you wanted to process the 7th, 3rd, and most recent commits, you would do:

```
$ pachctl create-branch data master --head staging^7
$ pachctl create-branch data master --head staging^3
$ pachctl create-branch data master --head staging
```

If you do `list-job` while running the above commands, you will see between 1 and 3 new jobs. Eventually there will be a job for each of the HEAD commits, however Pachyderm won't create a new job until the previous job has completed.

What to do if you accidentally process something you didn't want to

In all of the examples above we've been *advancing* the `master` branch to later commits. However, this isn't a requirement of the system, you can move backward to previous commits just as easily. For example, if after the above commands you realized that actually want your final output to be the result of processing `staging^1`, you can "roll back" your HEAD commit the same way we did before.

```
$ pachctl create-branch data master --head staging^1
```

This will kick off 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`.

More complicated staging patterns

Using a `staging` branch allows you to defer processing, but it's inflexible, in that you need to know ahead of time what you want your input commits to look like. Sometimes you want to be able to commit data in an ad-hoc, disorganized way and then organize it later. For this, instead of updating your `master` branch to point at commits

from `staging` , you can copy files directly from `staging` to `master` . With `copy-file` , this only copies references, it doesn't move the actual data for the files around.

This would look like:

```
$ pachctl start-commit data master
$ pachctl copy-file data staging file1 data master
$ pachctl copy-file data staging file2 data master
...
$ pachctl finish-commit data master
```

You can also, of course, issue `delete-file` and `put-file` while the commit is open if you want to remove something from the parent commit or add something that isn't stored anywhere else.

Ingressing From a Separate Object Store

Occasionally, you might find yourself needing to ingress data from or egress data (with the `put-file` command or `egress` field in the pipeline spec) to/from an object store that runs in a different cloud. For instance, you might be running a Pachyderm cluster in Azure, but you need to ingress files from a S3 bucket.

Fortunately, Pachyderm can be configured to ingress/egress from any number of supported cloud object stores, which currently include S3, Azure, and GCS. In general, all you need to do is to provide Pachyderm with the credentials it needs to communicate with the cloud provider.

To provide Pachyderm with the credentials, you use the `pachctl deploy storage` command:

```
$ pachctl deploy storage <backend> ...
```

Here, `<backend>` can be one of `aws`, `google`, and `azure`, and the different backends take different parameters. Execute `pachctl deploy storage <backend>` to view detailed usage information.

For example, here's how you would deploy credentials for a S3 bucket:

```
$ pachctl deploy storage aws <bucket-name> <access key id> <secret access key>
```

Credentials are stored in a [Kubernetes secret](#) and therefore share the same security properties.

Vault Secret Engine

Pachyderm supports Vault integration by providing a Vault Secret Engine.

Deployment

Vault instructions for the admin deploying/configuring/managing vault

1. Get plugin binary
 - Navigate to the Pachyderm Repo on github
 - Go to the latest release page
 - Download the `vault` asset
1. Download / Install that binary on your vault server instance

On your vault server:

```
# Assuming the binary was downloaded to /tmp/vault-plugins/pachyderm
export SHASUM=$(shasum -a 256 "/tmp/vault-plugins/pachyderm" | cut -d " " -f1)
echo $SHASUM
vault write sys/plugins/catalog/pachyderm sha_256="$SHASUM" command="pachyderm"
vault secrets enable -path=pachyderm -plugin-name=pachyderm plugin
```

Note: You may need to enable memory locking on the pachyderm plugin (see https://www.vaultproject.io/docs/configuration/#disable_mlock). That will look like:

```
sudo setcap cap_ipc_lock=+ep $(readlink -f /tmp/vault-plugins/pachyderm)
```

1. Configure the plugin

We'll need to gather and provide this information to the plugin for it to work:

- `admin_token` : is the (machine user) pachyderm token the plugin will use to cut new credentials on behalf of users
- `pachd_address` : is the URL where the pachyderm cluster can be accessed
- `ttl` : is the max TTL a token can be issued

Admin Token

To get a machine user `admin_token` from pachyderm:

If auth is not activated

(this activates auth with a robot user. It's also possible to activate auth with a github user. Also, the choice of `robot:admin` is arbitrary. You could name this admin `robot:<any string>`)

```
$ pachctl auth activate --initial-admin=robot:admin
Retrieving Pachyderm token...
WARNING: DO NOT LOSE THE ROBOT TOKEN BELOW WITHOUT ADDING OTHER ADMINS.
IF YOU DO, YOU WILL BE PERMANENTLY LOCKED OUT OF YOUR CLUSTER!
Pachyderm token for "robot:admin":
34cffc9254df40f0a277ee23e9fb005d

$ ADMIN_TOKEN=34cffc9254df40f0a277ee23e9fb005d
$ echo "${ADMIN_TOKEN}" | pachctl auth use-auth-token # authenticates you as the
↪cluster admin
```

If auth *is* already activated

```
# Login as a cluster admin
$ pachctl auth login
... login as cluster admin ...

# Appoint a new robot user as the cluster admin (if needed)
$ pachctl auth modify-admins --add=robot:admin

# Get a token for that robot user admin
$ pachctl auth get-auth-token robot:admin
New credentials:
  Subject: robot:admin
  Token: 3090e53de6cb4108a2c6591f3cbd4680

$ ADMIN_TOKEN=3090e53de6cb4108a2c6591f3cbd4680
```

Pass the new admin token to Pachyderm:

```
vault write pachyderm/config \
  admin_token="${ADMIN_TOKEN}" \
  pachd_address="${ADDRESS:-127.0.0.1:30650}" \
  ttl=5m # optional
```

1. Test the plugin

```
vault read pachyderm/version

# If this fails, check if the problem is in the client (rather than the server):
vault read pachyderm/version/client-only
```

1. Manage user tokens with revoke

```
$ vault token revoke d2f1f95c-2445-65ab-6a8b-546825e4997a
Success! Revoked token (if it existed)
```

Which will revoke the vault token. But if you also want to manually revoke a pachyderm token, you can do so by issuing:

```
$vault write pachyderm/revoke user_token=xxx
```

Usage

When your application needs to access pachyderm, you will first do the following:

1. Connect / login to vault

Depending on your language / deployment this can vary. see the vault documentation for more details.

1. Anytime you are going to issue a request to a pachyderm cluster first:
 - check to see if you have a valid pachyderm token
 - if you do not have a token, hit the `login` path as described below
 - if you have a token but it's TTL will expire soon (latter half of TTL is what's recommended), hit the `renew` path as described below
 - then use the response token when constructing your client to talk to the pachyderm cluster

Login

Again, your client could be in any language. But as an example using the vault CLI:

```
$ vault write -f pachyderm/login/robot:test
Key          Value
---          -
lease_id     pachyderm/login/robot:test/e93d9420-7788-4846-7d1a-8ac4815e4274
lease_duration 768h
lease_renewable true
pachd_address 192.168.99.100:30650
user_token    aa425375f03d4a5bb0f529379d82aa39
```

The response metadata contains the `user_token` that you need to use to connect to the pachyderm cluster, as well as the `pachd_address`. Again, if you wanted to use this Pachyderm token on the command line:

```
$ echo "aa425375f03d4a5bb0f529379d82aa39" | pachctl auth use-auth-token
$ ADDRESS=127.0.0.1:30650 pachctl list-repo
```

The TTL is tied to the vault lease in `lease_id`, which can be inspected or revoked using the vault lease API (documented here: <https://www.vaultproject.io/api/system/leases.html>):

```
$ vault write /sys/leases/lookup lease_id=pachyderm/login/robot:test/e93d9420-7788-4846-7d1a-8ac4815e4274
Key          Value
---          -
expire_time  2018-06-17T23:32:23.317795215-07:00
id           pachyderm/login/robot:test/e93d9420-7788-4846-7d1a-8ac4815e4274
issue_time   2018-05-16T23:32:23.317794929-07:00
last_renewal <nil>
renewable    true
ttl          2764665
```

Renew

You should issue a `renew` request once the halfway mark of the TTL has elapsed. Like revocation, renewal is handled using the vault lease API:

```
$ vault write /sys/leases/renew lease_id=pachyderm/login/robot:test/e93d9420-7788-
↪4846-7d1a-8ac4815e4274 increment=3600
Key          Value
---          -
lease_id     pachyderm/login/robot:test/e93d9420-7788-4846-7d1a-8ac4815e4274
lease_duration 2h
lease_renewable true
```

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.

JSON Manifest Format

```
{
  "pipeline": {
    "name": string
  },
  "description": string,
  "transform": {
    "image": string,
    "cmd": [ string ],
    "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
  },
  "resource_requests": {
    "memory": string,
    "cpu": number,
```

```
    "disk": string,
  },
  "resource_limits": {
    "memory": string,
    "cpu": number,
    "gpu": number,
    "disk": string,
  },
  "datum_timeout": string,
  "datum_tries": int,
  "job_timeout": string,
  "input": {
    <"atom", "cross", "union", "cron", or "git" see below>
  },
  "output_branch": string,
  "egress": {
    "URL": "s3://bucket/dir"
  },
  "standby": bool,
  "incremental": bool,
  "cache_size": string,
  "enable_stats": bool,
  "service": {
    "internal_port": int,
    "external_port": int
  },
  "max_queue_size": int,
  "chunk_spec": {
    "number": int,
    "size_bytes": int
  },
  "scheduling_spec": {
    "node_selector": {string: string},
    "priority_class_name": string
  },
  "pod_spec": string
}
```

"atom" input

```
"atom": {
  "name": string,
  "repo": string,
  "branch": string,
  "glob": string,
  "lazy" bool,
  "empty_files": bool
}
```

"cross" or "union" input

```
"cross" or "union": [
  {
    "atom": {
```

```

    "name": string,
    "repo": string,
    "branch": string,
    "glob": string,
    "lazy" bool,
    "empty_files": bool
  }
},
{
  "atom": {
    "name": string,
    "repo": string,
    "branch": string,
    "glob": string,
    "lazy" bool,
    "empty_files": bool
  }
}
etc...
]

```

```
-----
"cron" input
-----
```

```

"cron": {
  "name": string,
  "spec": string,
  "repo": string,
  "start": time
}

```

```
-----
"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 nil. Following is an example of a minimal spec:

```

{
  "pipeline": {
    "name": "wordcount"
  },
  "transform": {
    "image": "wordcount-image",
    "cmd": ["/binary", "/pfs/data", "/pfs/out"]
  },
  "input": {
    "atom": {
      "repo": "data",
      "glob": "/*"
    }
  }
}

```

```
}  
  }  
}
```

Following is a walk-through of all the fields.

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:

- contain only alphanumeric characters, `_` and `-`
- begin or end with only alphanumeric characters (not `_` or `-`)
- be no more than 50 characters in length

Description (optional)

`description` is an optional text field where you can put documentation about the pipeline.

Transform (required)

`transform.image` is the name of the Docker image that your jobs run in.

`transform.cmd` is the command passed to the Docker run invocation. Note that as with Docker, `cmd` is not run inside a shell which means that things like wildcard globbing (`*`), pipes (`|`) and file redirects (`>` and `>>`) will not work. To get that behavior, you can set `cmd` to be a shell of your choice (e.g. `sh`) and pass a shell script to `stdin`.

`transform.stdin` is an array of lines that are sent to your command on `stdin`. Lines need not end in newline characters.

`transform.env` is a map from key to value of environment variables that will be injected 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, they are useful for embedding sensitive data such as credentials. Secrets reference Kubernetes secrets by name and specify a path that the secrets should be mounted to, 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`.

[here](#).

`transform.image_pull_secrets` is an array of image pull secrets, image pull secrets are similar to secrets except that they're 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 via:

```
$ 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 tell your pipeline about it via `"image_pull_secrets": ["myregistrykey"]`. Read more about image pull secrets [here](#).

`transform.accept_return_code` is an array of return codes (i.e. exit codes) from your docker command that are considered acceptable, which means that if your docker command exits with one of the codes in this array, it will be considered a successful run for the purpose of setting 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 will be run from, this can also be accomplished with a `WORKDIR` directive in your Dockerfile.

Parallelism Spec (optional)

`parallelism_spec` describes how Pachyderm should parallelize your pipeline. Currently, Pachyderm has two parallelism strategies: `constant` and `coefficient`.

If you set the `constant` field, Pachyderm will start the number of workers that you specify. For example, set `"constant": 10` to use 10 workers.

If you set the `coefficient` field, Pachyderm will start 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 will start five workers. If you set it to 2.0, Pachyderm will start 20 workers (two per Kubernetes node).

By default, we use the parallelism spec "coefficient=1", which means that we spawn one worker per node for this pipeline.

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 us schedule big jobs on separate machines, so that they don't 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, etc). 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 only 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) 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 should get 2000ms of CPU time per second (i.e. it's using 2 CPUs, essentially, though worker threads might spend e.g. 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, etc).

In both cases, the resource requests are not upper bounds. If the worker uses more memory than it's requested, it will not (necessarily) be killed. However, if the whole node runs out of memory, Kubernetes will start killing pods that have been placed on it and exceeded their memory request, to reclaim memory. To prevent your worker getting killed, 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 will be unable to schedule new workers (because no machine will have 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 killed.

For more information about resource requests and limits see the [Kubernetes docs](#) on the subject.

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.

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.

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.

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.

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.

```
{
  "atom": atom_input,
  "union": [input],
  "cross": [input],
  "cron": cron_input
}
```

Atom Input

Atom 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.atom.name` is the name of the input. An input with name `XXX` will be 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 `AtomInput` s that are unioned. This is because when `AtomInput` s are unioned, you'll only ever see a datum from one input at a time. Overlapping the names of unioned inputs allows you to write simpler code since you no longer need to consider which input directory a particular datum come 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'll be required to give at least one of them a unique name.

`input.atom.repo` is the repo to be used for the input.

`input.atom.branch` is the branch to watch for commits on, it may be left blank in which case "master" will be used.

`input.atom.glob` is a glob pattern that's used to determine how the input data is partitioned. It's explained in detail in the next section.

`input.atom.lazy` controls how the data is exposed to jobs. The default is `false` which means the job will eagerly download the data it needs to process and it will be exposed as normal files on disk. If `lazy` is set to `true`, data will be exposed as named pipes instead and no data will be downloaded until the job opens the pipe and reads it, if the pipe is never opened then no data will be downloaded. Some applications won't work with pipes, for example if they make syscalls such as `Seek` which pipes don't support. Applications that can work with pipes should use them since they're more performant, the difference will be especially notable if the job only reads a subset of the files that are available to it. Note that `lazy` currently doesn't support datums that contain more than 10000 files.

`input.atom.empty_files` controls how files are exposed to jobs. If true, it will cause files from this atom to be presented as empty files. This is useful in shuffle pipelines where you want to read the names of files and reorganize them using symlinks.

Union Input

Union inputs take the union of other inputs. For example:

inputA	inputB	inputA inputB
foo	fizz	foo
bar	buzz	fizz
		bar
		buzz

Notice that union inputs, do not take a name and maintain the names of the sub-inputs. In the above example you would see files under `/pfs/inputA/...` or `/pfs/inputB/...`, but never both at the same time. This can be annoying to write code for since the first thing your code needs to do is figure out which input directory is present. As of 1.5.3 the recommended way to fix this is to give your inputs the same `Name`, that way your code only needs to handle data being present in that directory. This, of course, only works if your code doesn't need to be aware of which of the underlying inputs the data comes from.

`input.union` is an array of inputs to union, note that these need not be `atom` inputs, they can also be `union` and `cross` inputs. Although there's no reason to take a union of unions since union is associative.

Cross Input

Cross inputs take the cross product of other inputs, in other words it creates tuples of the datums in the inputs. For example:

inputA	inputB	inputA inputB
foo	fizz	(foo, fizz)
bar	buzz	(foo, buzz)
		(bar, fizz)
		(bar, buzz)

Notice that cross inputs, do not take a name and maintain the names of the sub-inputs. In the above example you would see files under `/pfs/inputA/...` and `/pfs/inputB/...`.

`input.cross` is an array of inputs to cross, note that these need not be `atom` inputs, they can also be `union` and `cross` inputs. Although there's no reason to take a cross of crosses since cross products are associative.

Cron Input

Cron inputs allow you to trigger pipelines based on time. It's based on the unix utility `cron`. When you create a pipeline with one or more Cron Inputs pachd will create a repo for each of them. When a cron input triggers, that is when the present time satisfies its spec, pachd will commit a single file, called "time" to the repo which contains the time which satisfied the spec. The time is formatted according to [RFC 3339](#).

```
{
  "name": string,
  "spec": string,
  "repo": string,
  "start": time,
}
```

`input.cron.name` is the name for the input, its semantics are similar to those of `input.atom.name`. Except that it's 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 Nonstandard schedules such as `"@daily"`.

`input.cron.repo` is the repo which will be created for the input. It is optional, if it's not specified then `"<pipeline-name>_<input-name>"` will be used.

`input.cron.start` is the time to start counting from for the input. It is optional, if it's not specified then the present time (when the pipeline is created) will be used. Specifying a time allows 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. Times should be formatted according to [RFC 3339](#).

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.atom.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 `kubect1 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.

Output Branch (optional)

This is the branch where the pipeline outputs new commits. By default, it's "master".

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.

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.

Incremental (optional)

Incremental, if set will cause the pipeline to be run "incrementally". This means that when a datum changes it won't be reprocessed from scratch, instead `/pfs/out` will be populated with the previous results of processing that datum and instead of seeing the full datum under `/pfs/repo` you will see only new/modified values. Incremental pipelines are discussed in more detail here.

Incremental processing is useful for [online algorithms](#), a canonical example is summing a set of numbers since the new numbers can be added to the old total without having to reconsider the numbers which went into that old total. Incremental is designed to work nicely with the `--split` flag to `put-file` because it will cause only the new chunks of the file to be displayed to each step of the pipeline.

Cache Size (optional)

`cache_size` controls how much cache a pipeline worker uses. In general, your pipeline's performance will increase with the cache size, but only up to a certain point depending on your workload.

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.

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 will be 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, via 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>`.

Max Queue Size (optional)

`max_queue_size` specifies that maximum number of elements that a worker should hold in its processing queue at a given time. 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 it allows workers to simultaneously download, process and upload different datums at the same time. Setting this value too high can 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.

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.

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.

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:

```
kubect1 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. This means that you can modify things such as the storage and user containers.

The Input Glob Pattern

Each atom 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.

PPS Mounts and File Access

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.

Pachctl Command Line Tool

Pachctl is the command line interface for Pachyderm. To install Pachctl, follow the [Local Installation](#) instructions

Synopsis

Access the Pachyderm API.

Environment variables:

ADDRESS=<host>:<port>, the pachd server to connect to (e.g. 127.0.0.1:30650).

Options

--no-metrics	Don't report user metrics for this command
-v, --verbose	Output verbose logs

Pachyderm language clients

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
```

Python Client -

The Python client is a user contributed client that has just recently been brought under the Pachyderm umbrella and made into an official client. We're working on getting it fully up to date and will be supporting it full going forward.

For more info, check out [pypachy](#) on [GitHub](#).

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.

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.