Advanced Data Science in Domino

Dr. Jennifer Davis
Rev3 May 4, 2022

Field Data Science Practice

- Assists at all phases of the project
  - Ideation
  - Research and development
  - Execution
- Expert data scientists and field engineers help you leverage Domino for success
  - Dedicated team of field data scientists and data engineers focus on your success
- We provide consultation services:
  - Best practices for data science on Domino
  - Developing Proof of Concepts and Minimum Viable Products
  - Execution of complex projects to solve business problems
  - Knowledge transfer upon project completion

Agenda

- Setting Up Your Project for Success
- Project Management Tools in Domino
- Environments in Domino
- On-Demand Distributed Compute
- Spark on-demand
- Ray on-demand
- Dask on-demand
- MPI on-demand
- Optimizing GPU usage

Advanced Course Objectives

- Upon completions of this course learners will be able to:
  - Discuss Domino Environments and Best Practices
  - Describe benefits and use cases for on-demand distributed computing on Domino
  - Discuss Spark, Ray, Dask and MPI integration with Domino
  - Demonstrate a ‘calculate pi’ example on distributed compute
  - Explain a Deep Learning (Ray) and a Machine Learning (Dask) example
  - Explain how GPUs provide performance and how to optimize GPU use in Domino

Setting Up Your Project for Success

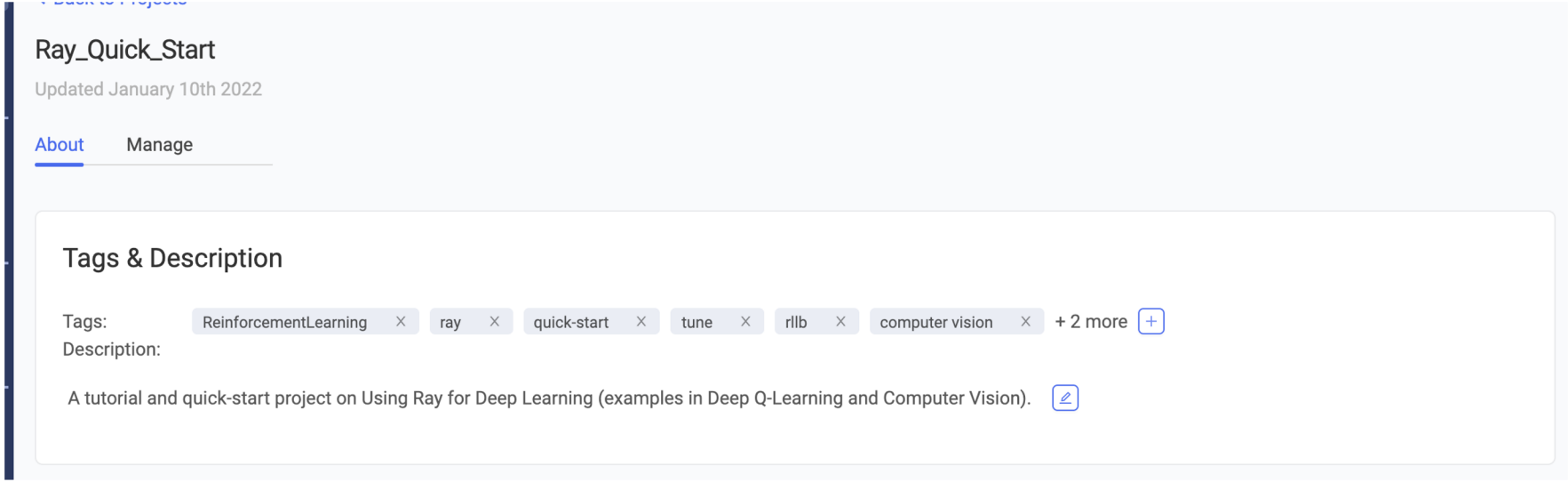
Setting Up Your Project for Success

- Project Management Tools in Domino
- Git in Domino
  - Code repositories (Git based projects)
- Environments in Domino
  - Naming Conventions
  - Reproducibility
  - More to come

Project Management Tools in Domino

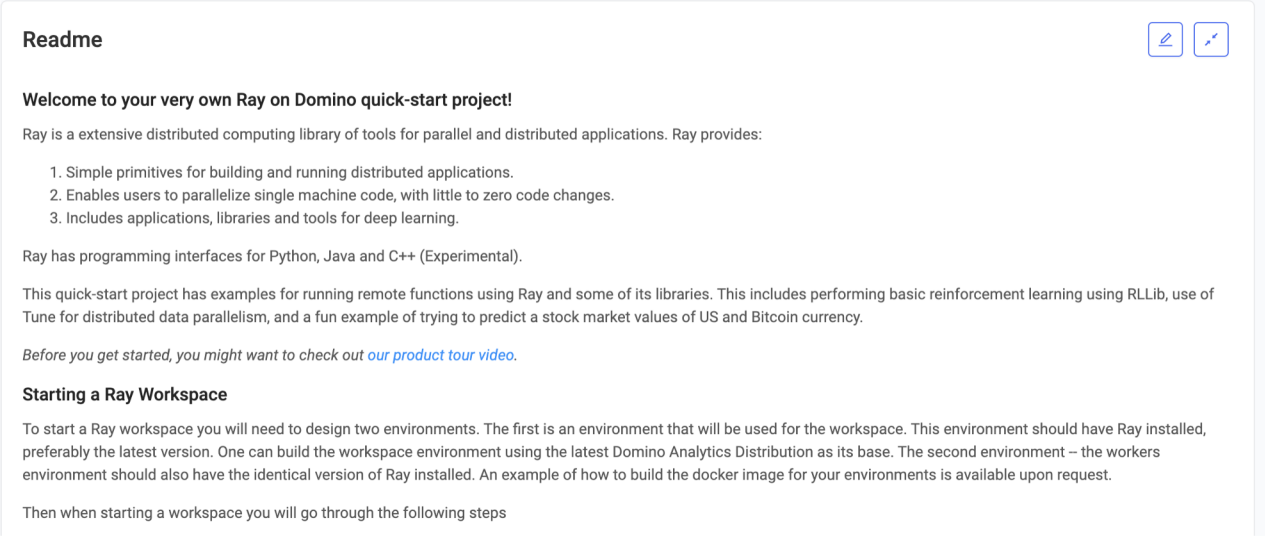
Setting Up Your Project for Success

- All projects can have tags attached to them which make it easier to track projects by topic
- A clear description will let users know more about the project



Setting Up Your Project for Success

- To create a README file simply add one to your repository. The readme will automatically be populated to your project
- If you use a git-based project your README file will also show on your github/bitbucket project



Readme

Welcome to your very own Ray on Domino quick-start project!

Ray is a extensive distributed computing library of tools for parallel and distributed applications. Ray provides:

1. Simple primitives for building and running distributed applications.
2. Enables users to parallelize single machine code, with little to zero code changes.
3. Includes applications, libraries and tools for deep learning.

Ray has programming interfaces for Python, Java and C++ (Experimental).

This quick-start project has examples for running remote functions using Ray and some of its libraries. This includes performing basic reinforcement learning using RLLib, use of Tune for distributed data parallelism, and a fun example of trying to predict a stock market values of US and Bitcoin currency.

Before you get started, you might want to check out our product tour video.

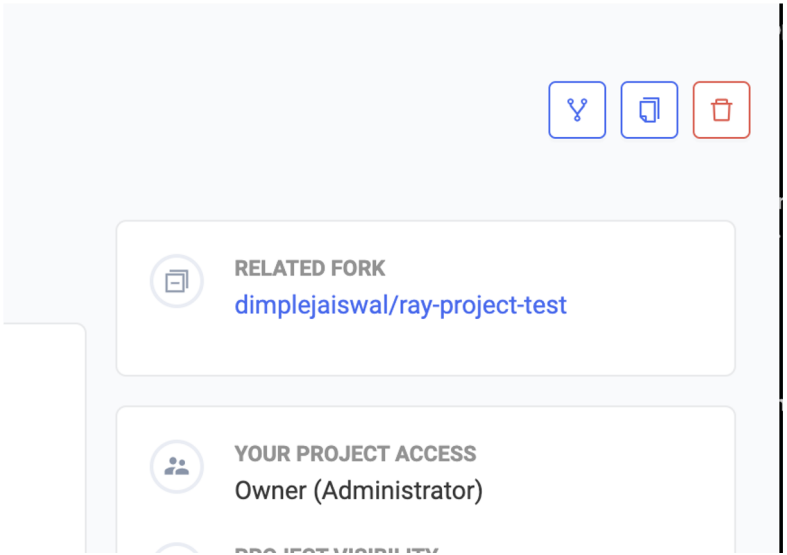
Starting a Ray Workspace

To start a Ray workspace you will need to design two environments. The first is an environment that will be used for the workspace. This environment should have Ray installed, preferably the latest version. One can build the workspace environment using the latest Domino Analytics Distribution as its base. The second environment -- the workers environment should also have the identical version of Ray installed. An example of how to build the docker image for your environments is available upon request.

Then when starting a workspace you will go through the following steps

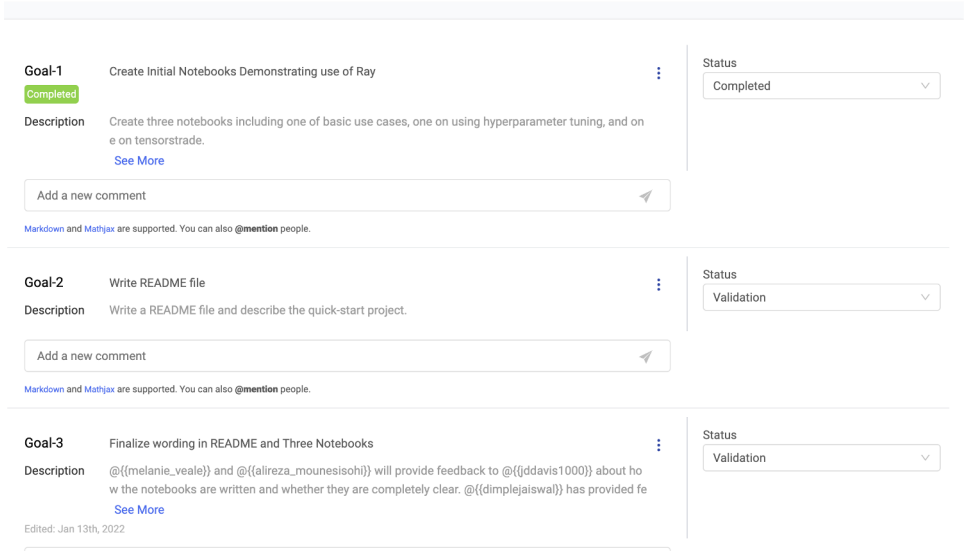
Forking Projects and QuickStarts

- Users can fork or clone projects they are interested in copying or learning from
- Users can use a particular project as “starter” project for other work they may want to do



Setting Up Project Goals, Tags and Assignments

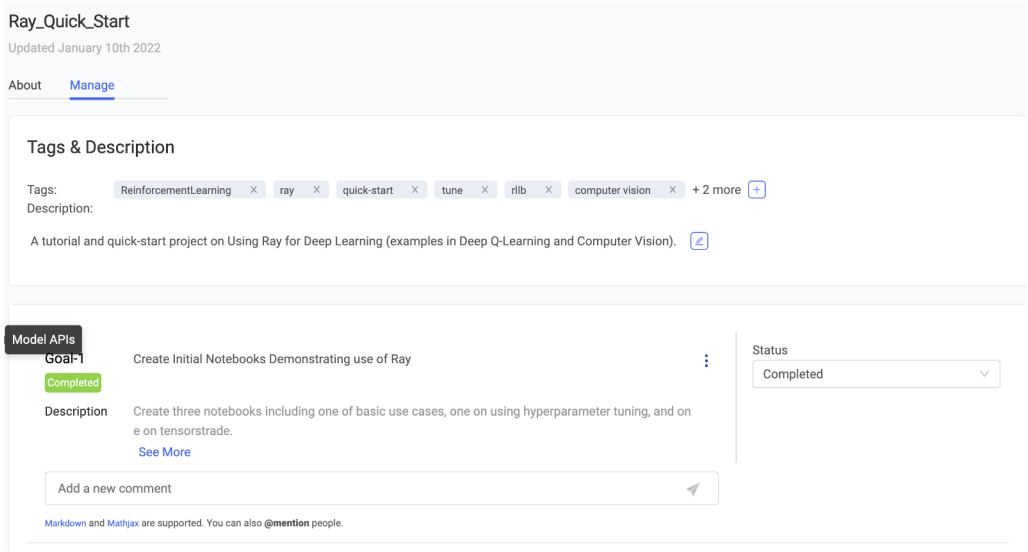
- Setting goals for your project can help when forming collaborations
- Goals also let reviewers know clearly where in the development process the project is and next steps



DOMINO

Using Goals, comments and tagging collaborators

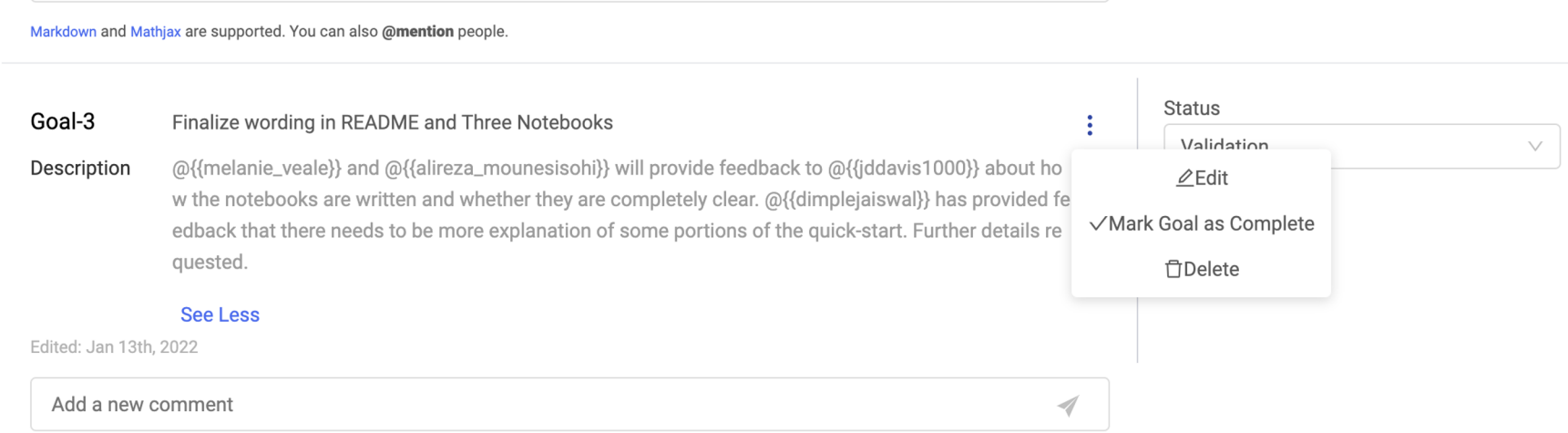
- Once project goals are set, collaborators can be tagged for specific goals
- Comments can be made for each goal and each goal is tracked for stage. For example this goal is marked completed.



DOMINO

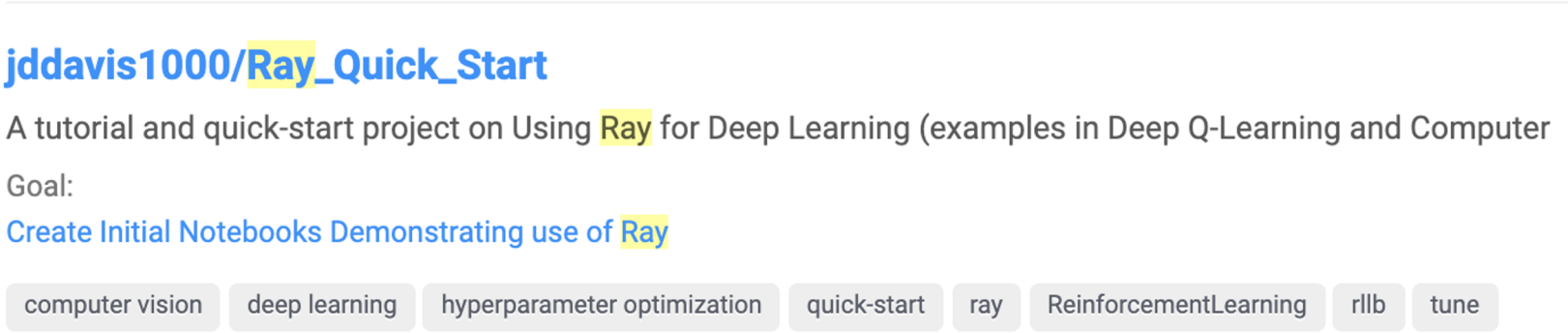
Commenting on Project Goals, Marking Completion

- Here's an example of a goal, its description, and how to mark the status
- This goal is in the validation stage, but there are other available stages



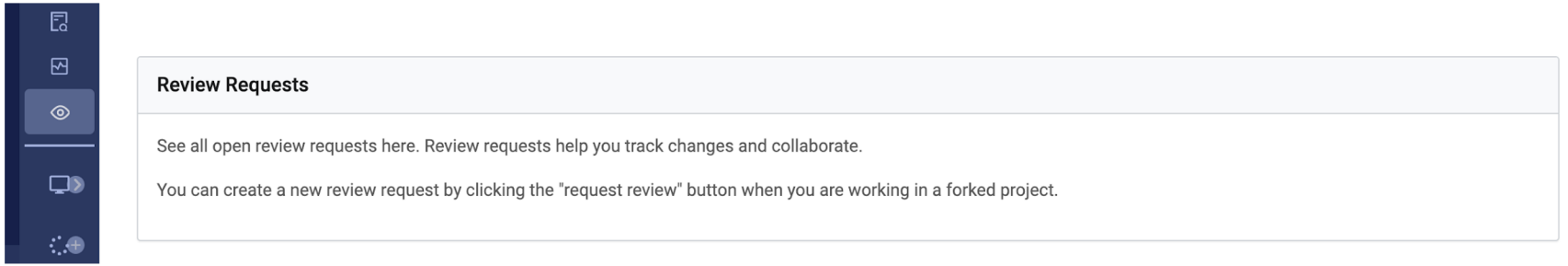
Users can search for project by keywords

- When using tagged and key words, one can search for projects by tag
- For example I added a project tag ‘Ray’. Users can find my project when searching for ‘Ray’



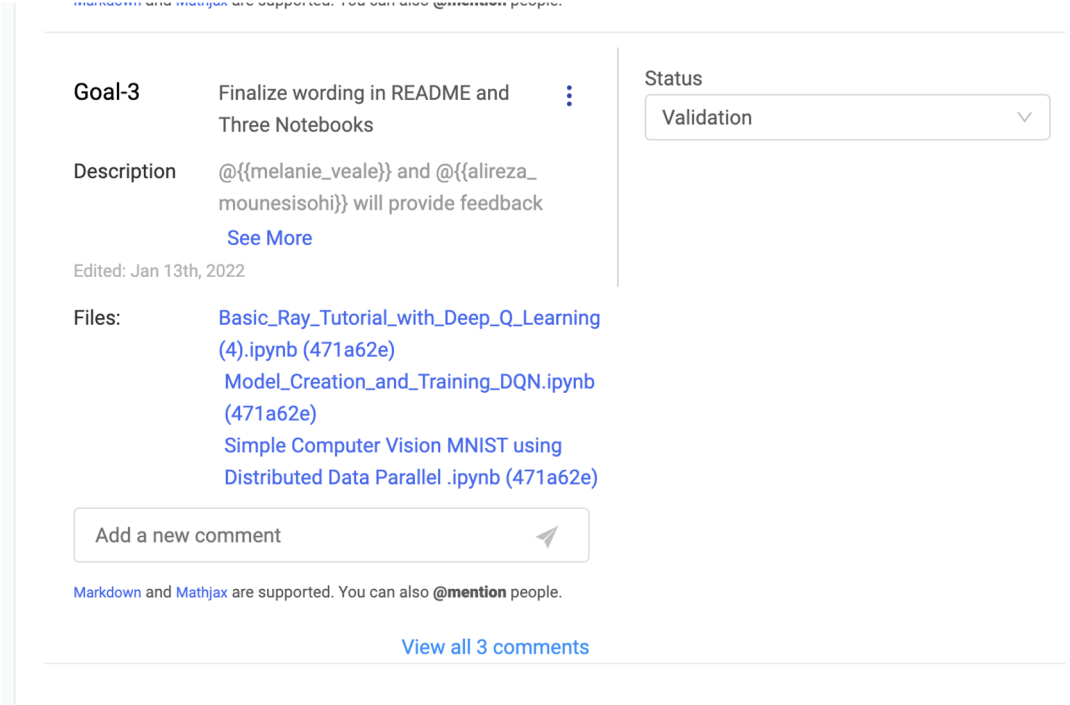
Requesting Reviews

- When working in Domino Data Lab, one can ask collaborators in the project to review specific files related to a particular goal



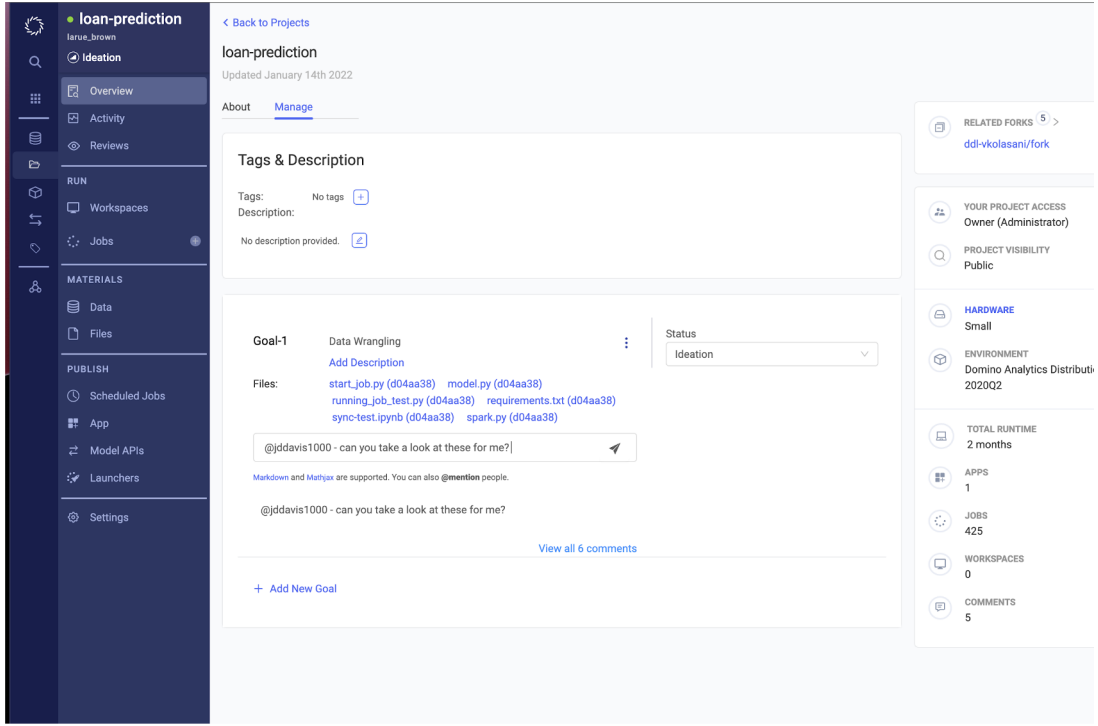
Data Access and Sharing, Tagging within a Project

When sharing Data one can tag files and datasets to ask collaborators for review



Linking files to a particular goal

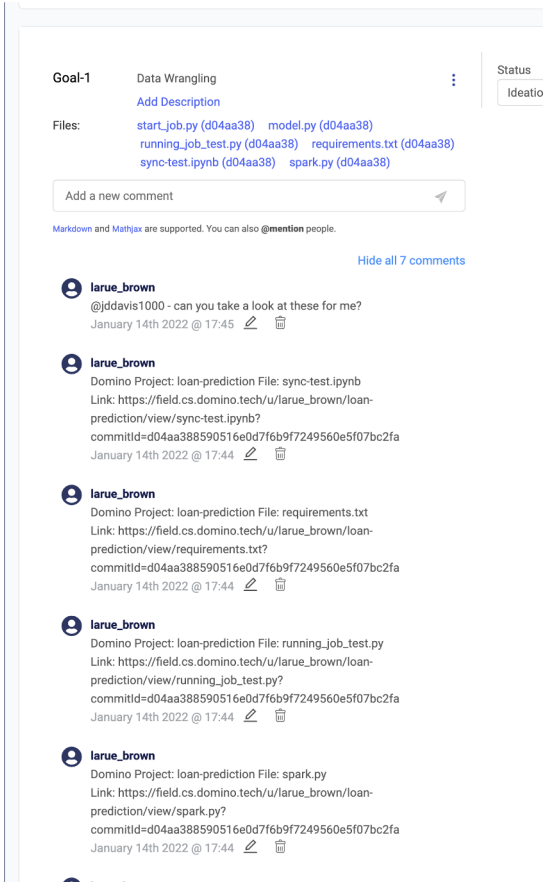
- In this example a goal is associated with files
- This allows collaborators or reviewers to quickly know which code or files to review



DOMINO

Example of notes in a particular goal

- This is a real team example of how comments were used for a particular goal in a group project
- Notice that comments can be used to tag people



DOMINO

Summary of Project Tools

- Tags for searchability
- Project README file
- Git Repository connectivity
- Project management and versioning
  - Assign project goals to individuals
  - Make comments and tag individuals under project goals
  - Set stage of project (e.g. completed, incomplete, validation, production)
  - Link a file to a particular goal

Summary of Versioning and Project Set-up

Versioning

- Fork projects
- Ask for a review of work on a particular forked project
- Versioning of Datasets (can revert to former versions of data)

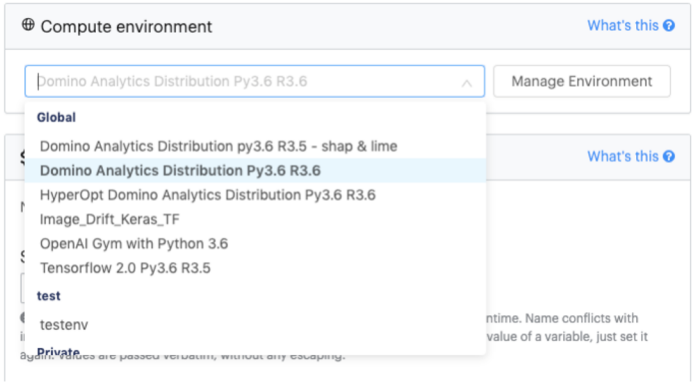
Project Set-up

- Set default environment
- Set default hardware tier
- Set permissions, e.g. contributor, owner, etc.

Environments in Domino

Environments

- An environment is a Domino abstraction on top of a Docker image, that provides additional flexibility and versioning
- When Domino starts your run, it creates a Docker container based on the environment associated with your project - Each run takes place in an isolated Docker container
- See docs for detailed instructions and info on advanced options



DOMINO

Example Docker file Instructions

`quay.io/domino/base:Ubuntu18_DAD_Py3.8_R4.0-20210503_CUDA11`

Supported Clusters

- ○ None
- ○ Domino managed Spark ⓘ
- ◉ Domino managed Ray ⓘ
- ○ Domino managed Dask ⓘ

Dockerfile Instructions

```
### Change Ray version as needed.
USER root

ENV RAY_VERSION=1.9.0

### If you want install Ray RLlib or "all", which includes it, you must
### install "cmake" first.
RUN apt-get update && apt-get install -y apt-transport-https
RUN apt-get install -y cmake
RUN apt-get install -y rsync

#python libraries
RUN pip install GPUtil
RUN pip install torch torchvision tensortrade

### Change this depending on which Ray extras you want to install:
### All options are ray, ray[tune], ray[rllib], ray[serve]
### If you want everything you can just use ray[all].
### See note above the "cmake" is required for ray[rllib] or ray[all].
RUN pip install ray[all]==$RAY_VERSION
RUN pip install pytorch-lightning git+https://github.com/ray-project/ray_lightning#ray_lightning
RUN pip install aioredis==1.3.0
RUN pip install torch torchvision

### Add any additional packages that you may need which are not included
### in the base image you are using for the compute environment. You would
### want the versions of these to match the versions of these packages on
### the base Ray cluster image.
### For example, for Torch you may include
### RUN pip install torch==1.8.0 torchvision==0.9.0

USER 1001
```

DOMINO

Environments Best Practices

- Docker optimizes its build process by keeping track of commands it has run and caching the results
- This means that if it sees the same set of commands as a previous build, it will assume it can use the cached version
- A single new command will invalidate the caching of all subsequent commands

Environments Best Practices

- There is a limit to the number of layers (that is, commands) a docker image can have. Currently, this limit is 127
- Keep in mind that the image upon which you are building may have already used many layers
- One way to work around this limit is to combine several commands into one via `&&`

Environments Best Practices

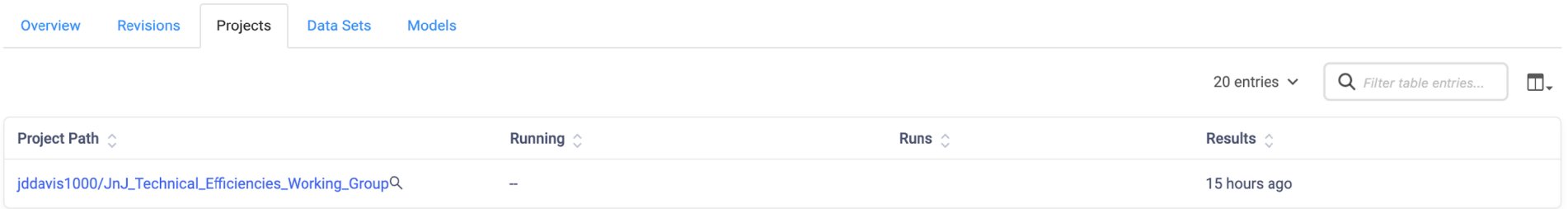
- If you are installing multiple python packages via pip, it's almost always best to use a single pip install command
- This ensures that dependencies and package versions are properly resolved.
- If you install via separate commands, you may end up inadvertently overriding a package with the wrong version, due a dependency specified by a later installation

Naming Conventions

- Name of the environment should be informative and include:
  - Version or Python and R
  - Name or abbreviation of Department
  - Key information on how the Environment will be used

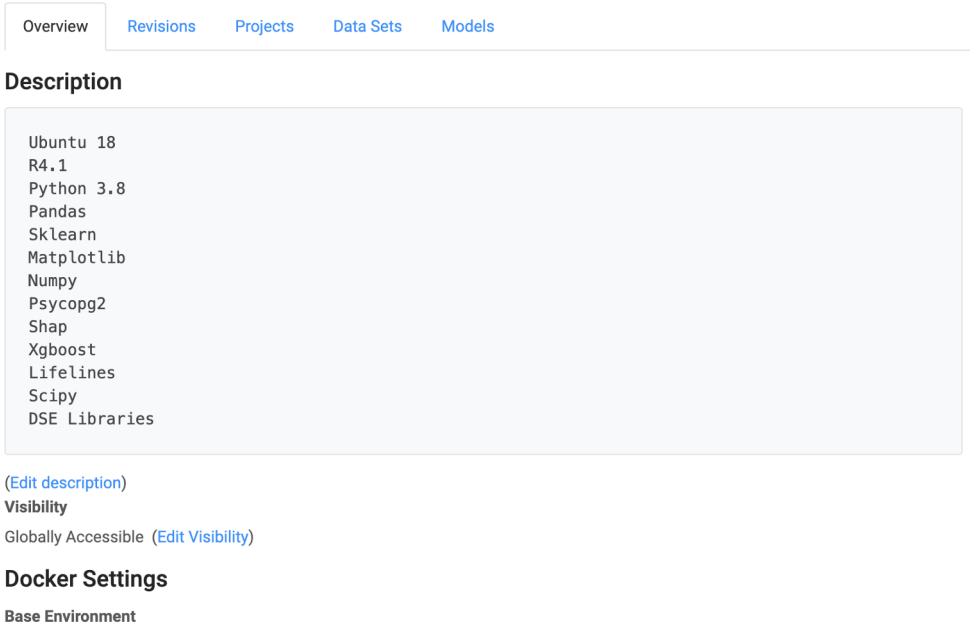
Information about Associated Projects

- Any projects that use the environment should have their default environment set to that specific environment
- The projects should show up in the ‘projects’ tab
- Before making any changes to the environment, stakeholders whose projects use the environment should be consulted
- If the environment is specific to an organization, a ‘gate-keeper’ should be responsible for updating or changing the environment



List Key Tools / Features

- At the top of the environment, the version of Ubuntu, key tools and features should be listed so that they are easily available for people considering use of the environment



DOMINO

Install Packages in one line of code when possible

- Keep Docker file code clean and remove anything that is unnecessary (such as commented out code or instructions)

Dockerfile Instructions

```
1  RUN pip install pandas sklearn matplotlib numpy psycopg2 shap xgboost lifelines scipy
```

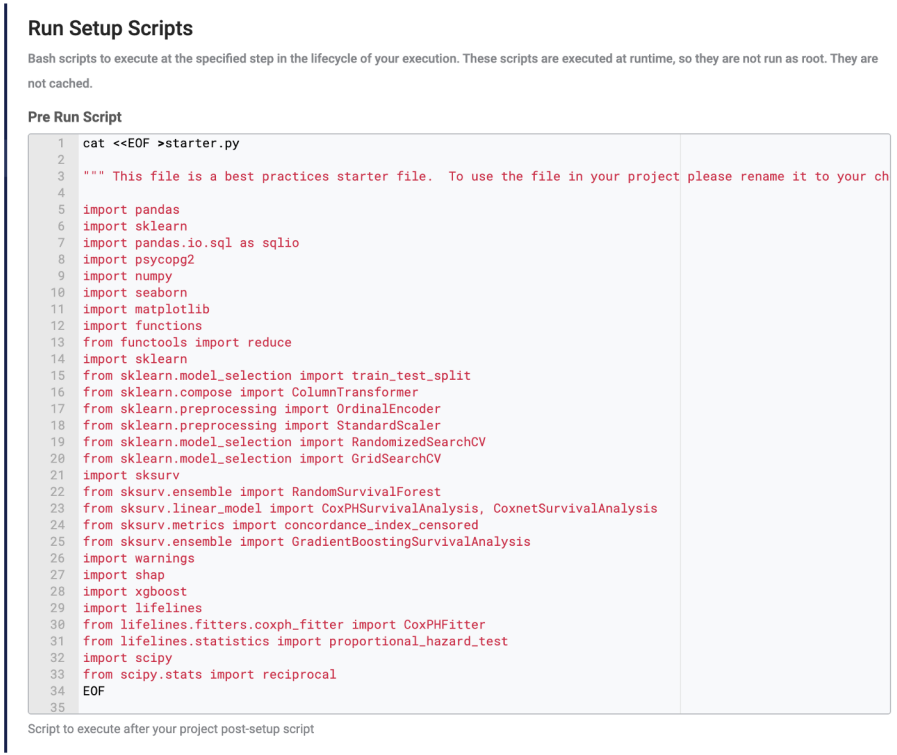
Your Dockerfile layers.

Edit Dockerfile

Pluggable Workspace Tools

Consider creating ‘Starter’ Files for Best Coding Practices

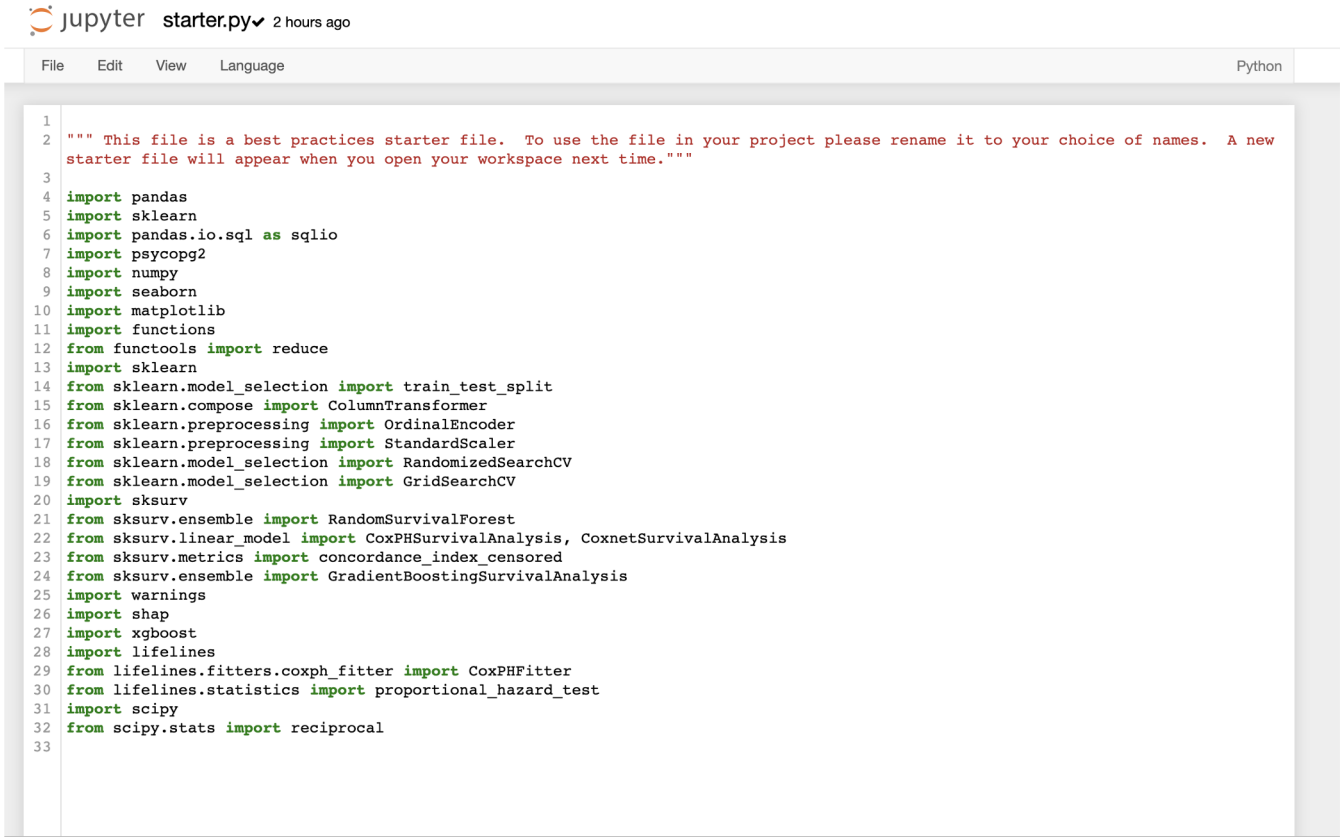
- A starter file can be created using a Run Setup Script
- The file should include instructions on how to use it
- The file should demonstrate best practices or how to import key libraries
- Files can be written in Python, R or both



Where the Starter File Appears

- This can be set up in the pre-run script, but the suggestion is to use the /mnt/ directory as its easy for users to find
- The file will appear each time a new workspace is started, so users should rename any starter file they are working on before closing the workspace.

What the Starter File Looks Like



Spinning Up Your Workspace

On-Demand Distributed Computing

What is Parallel Computing?

- Type of computing in which many calculations or processes are carried out simultaneously on several nodes and threads within a CPU or GPU

- Limits
  - Available Parallelism
  - Load Balancing
  - Managing communication between processes

- Pros
  - Faster speeds for data processing and computations
  - Enabling Big Data and Machine Learning

What is Distributed Computing?

- When multiple computers, CPUs or GPUs are used, tasks can be distributed among the cluster components unevenly and in parallel
- First rule: Don’t distribute your workload on clusters unless you see a need to
- Most deep learning and big data machine learning will be more performant on a distributed system
- Reinforcement learning is computationally intense, so distributed computing is helpful

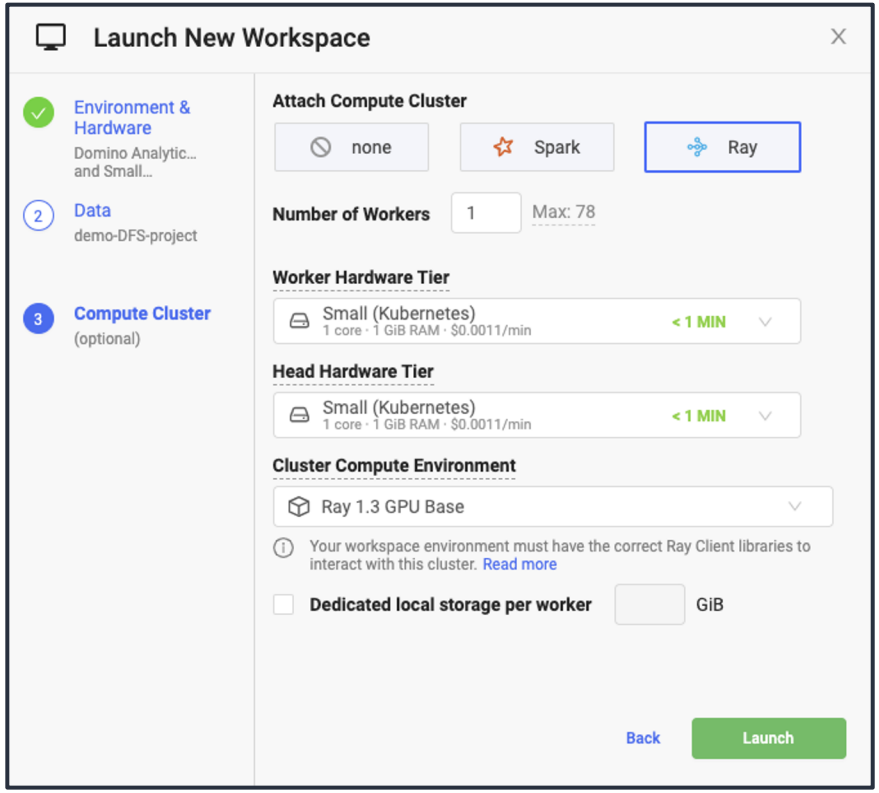
Cluster Environments on Domino

When using on-demand Clusters in Domino you will have two separate environments:

1. One for the workspace/job
    - See dockerfile instructions for Spark, Ray, Dask and MPI
2. One for the cluster workers/executors
    - Developed with a different base image - see docs for Spark, Ray, Dask and MPI

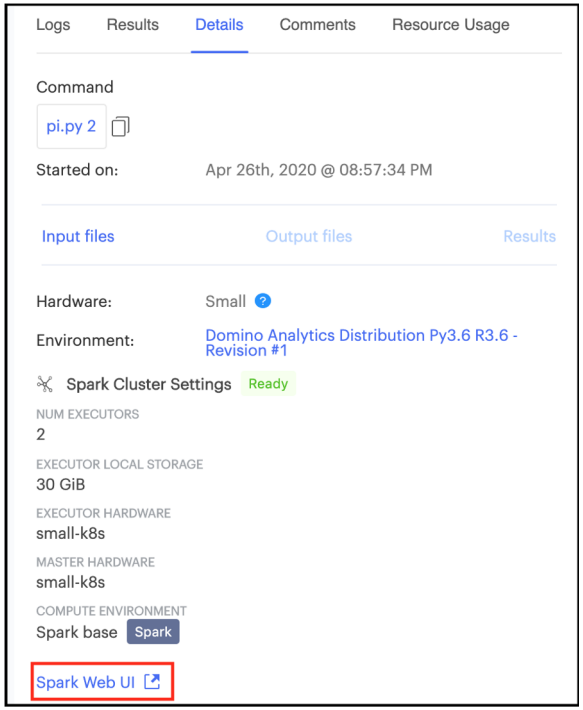
Cluster Settings

- Select the Attach Cluster option
- Set the number of Workers/Executors, and hardware tiers for each
- Dedicated local storage will be mounted to /tmp and de-provisioned when the cluster is shut down



Distributed Compute: Batch Jobs

- Set up the cluster when running a Job from the quick-action, files page, or Job dashboard
- The Spark or Ray UI is available for running Jobs from the details tab



DOMINO

On-Demand Spark

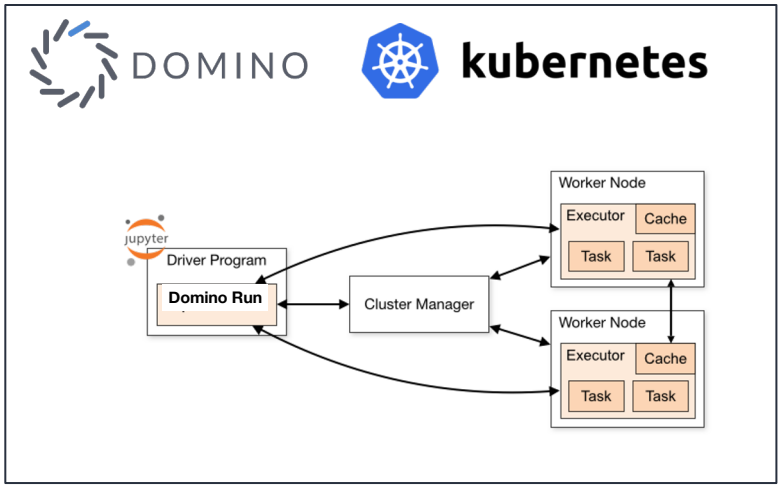
Agenda

- Setting up a Distributed Compute workspace for Spark
- Review of what Spark can do
- Spark Use Cases
- Calculating Pi – examples of non-parallel, parallel and distributed with Ray

Distributed Compute in Domino

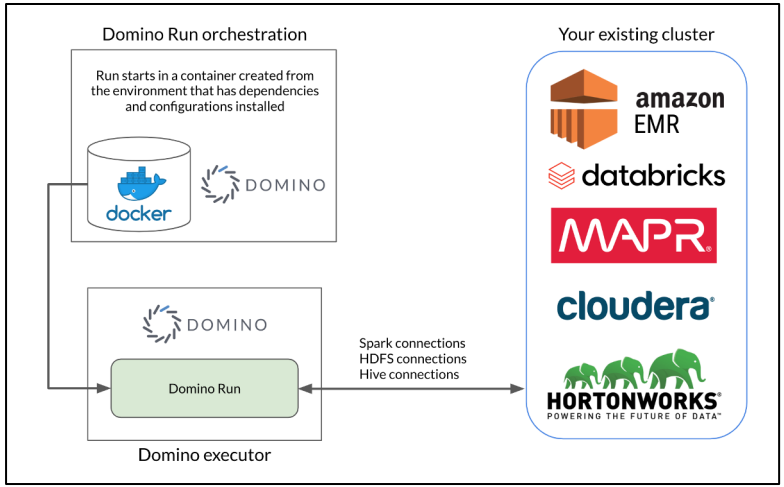
On-Demand Clusters

Natively orchestrated directly on Domino controlled infrastructure



External Clusters

Leverage existing long-lived Spark deployments in the enterprise



Spark on Demand in Domino: Use Cases

Distributed Machine Learning

Parallelize compute heavy workloads such as distributed training or hyper-parameter tuning

Take advantage of powerful machine learning algorithms from Spark MLlib

Interactive Exploratory Analysis

Efficiently load large data sets in distributed manner

Explore and understand the data using a familiar interface with Spark SQL

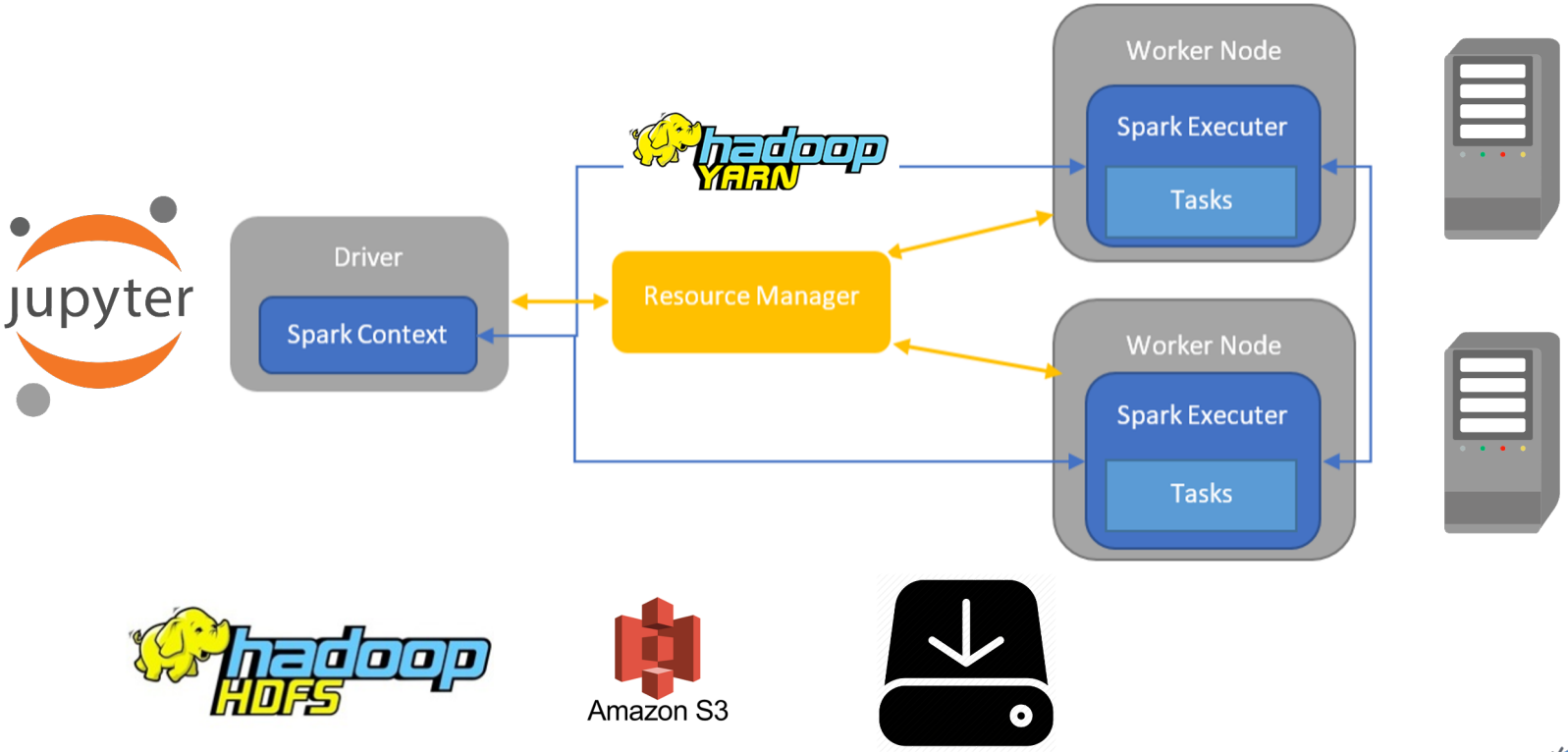
Featurization and Transformation

For experienced Spark practitioners

Sample, aggregate, and re-label large data sets

Optimal performance may require a practitioner who is skilled in tuning Spark

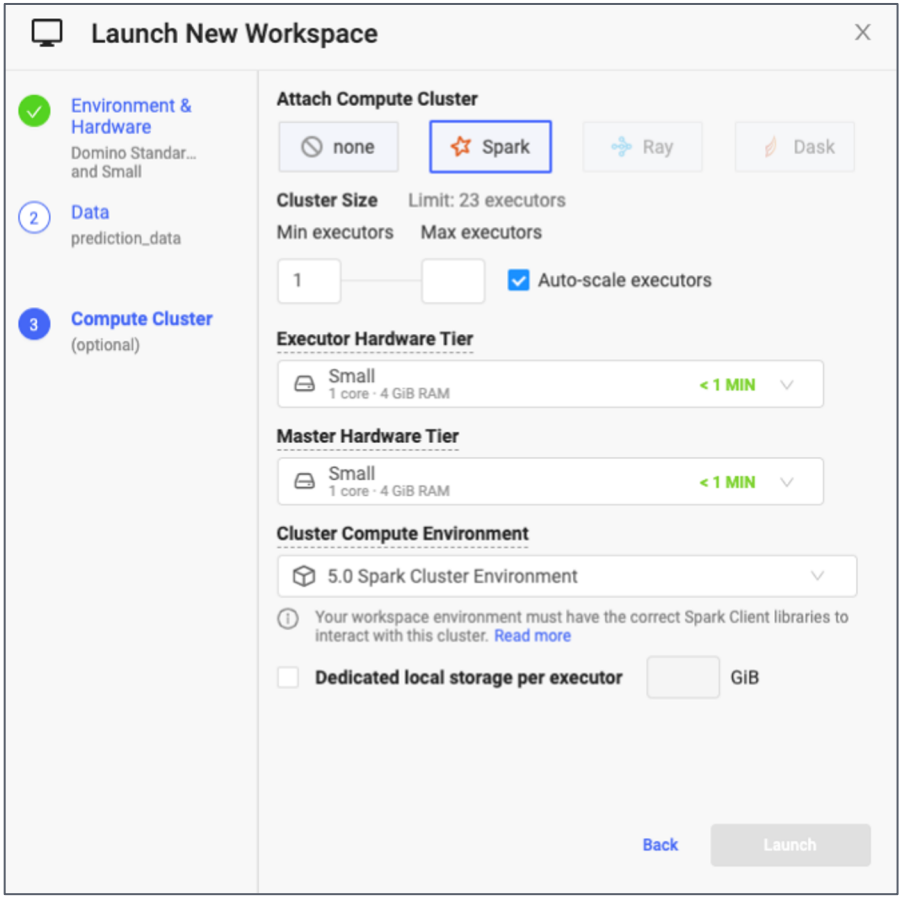
Spark Overview



DOMINO

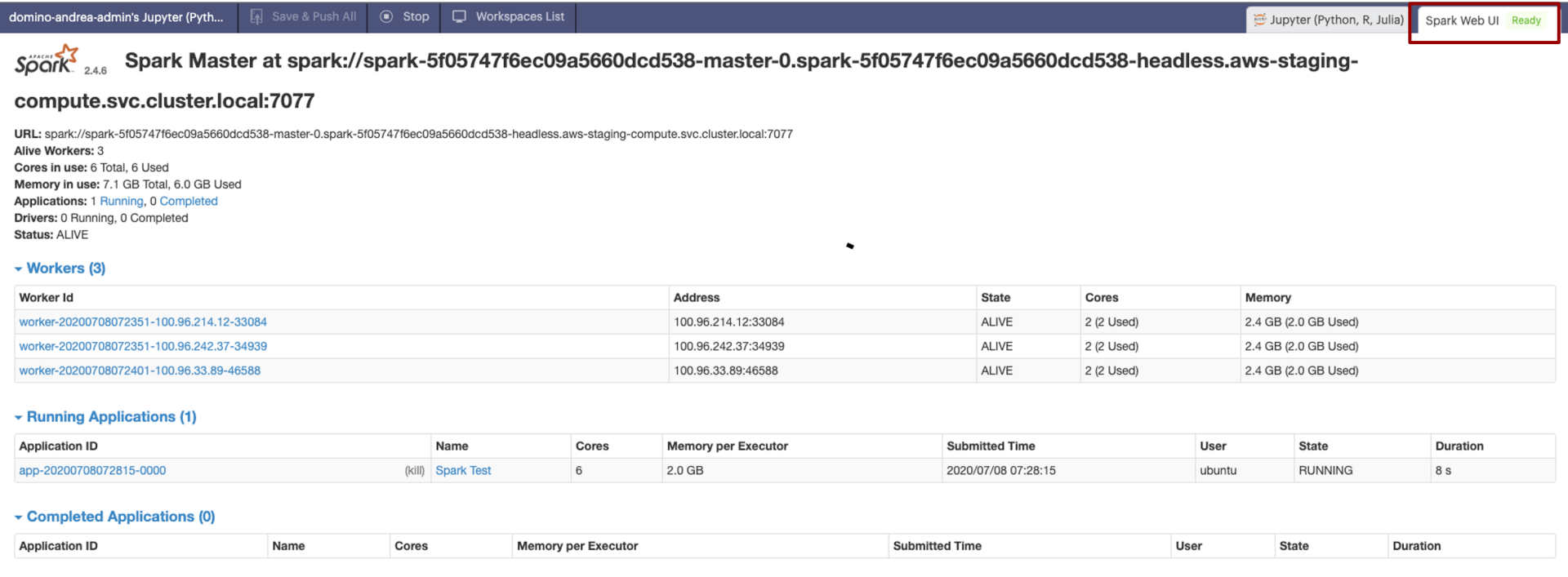
On-Demand Distributed Computing Overview

- Available only to the current execution
- Use with data residing outside the cluster - i.e. in Domino datasets or an S3 buckets
- Web UIs for the associated cluster are available in the same workspace
- Learn more in Domino 201
- Clusters can be auto-scaled



Accessing the Distributed Cluster UI

Available as a dedicated tab within your workspace



DOMINO

Tools and APIs for Spark

- Python API, PySpark
  - Easily use Python to code Spark applications
  - Syntax is a bit different than regular Python
  - Syntax is functional rather than object oriented

- R API, SparklyR
  - Syntax is the same as R code
  - Many R libraries are available natively
  - Many machine learning libraries are easily available
  - Hyper parameter tuning is available

Demo: ‘Hello World’ Example, Calculate Pi

On-Demand Ray

Agenda

- Setting up a Distributed Compute workspace for Ray
- Review of what Ray can do
- Ray Use Cases
- What Process Identifiers are and how to use them
- Calculating Pi – examples of non-parallel, parallel and distributed with Ray

Learning Goals

- Starting up your Workspace
- Understanding what Ray can do
- Understanding when Ray is appropriate to use
- Understanding the difference between regular calculations and distributed with Ray
- Know what a Process Identifier is and how to use it

Ray Overview

Ray.io provides a set of core low-level primitives (Ray Core) as well as pre-packaged libraries that take advantage of these primitives to enable solving powerful machine learning problems.

The following libraries come packaged with Ray:

- Tune: Scalable Hyperparameter Tuning
- Ray Train (formerly RaySGD): Distributed Training Wrappers
- RLlib: Scalable Reinforcement Learning
- Ray Serve: Scalable and Programmable Serving
- Ray Lightning - Distributed Pytorch Lightning

Ray on Demand in Domino: Use Cases

Distributed Multi-Node Training

Take existing PyTorch and Tensorflow models and scale them across multiple machines to dramatically reduce training times

Ray is suitable for both distributed CPU and GPU training

Hyperparameter Optimization

Launch a distributed hyperparameter sweep with just a few lines of code.

Take advantage of a large number of advanced parameter search algorithms.

Reinforcement Learning

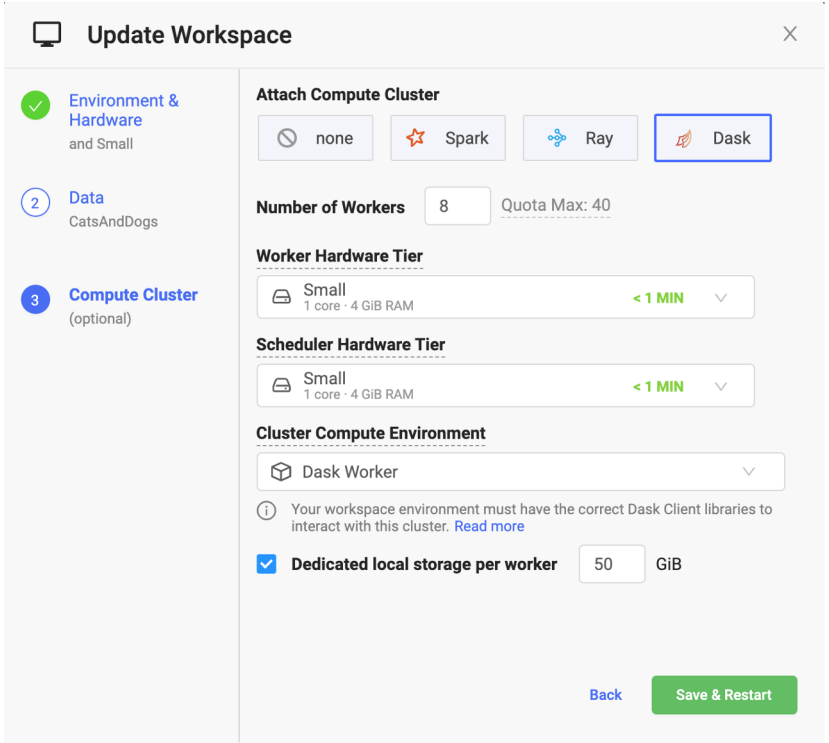
Take advantage of a number of built-in reinforcement learning algorithms, along with a general framework for developing your own.

Steps to setting up your workspace

- Name your workspace
  - Use a name you will remember that indicates whether it is distributed or not and the type of tool used (i.e. in this case Ray)
- Choose your workspace environment
- Ensure your datasets are available
- Choose your scheduler and compute hardware tier
- Choose the number of workers

Distributed Compute | Setting up your workspace

- With only a handful of specified parameters, such as the number of workers and the capacity of each worker, Domino will set up the cluster and seamlessly manage its lifecycle.
- Administrators retain control since the cluster configuration is based on the established Domino concepts of Hardware Tiers and Compute Environments.



Accessing the Ray Cluster

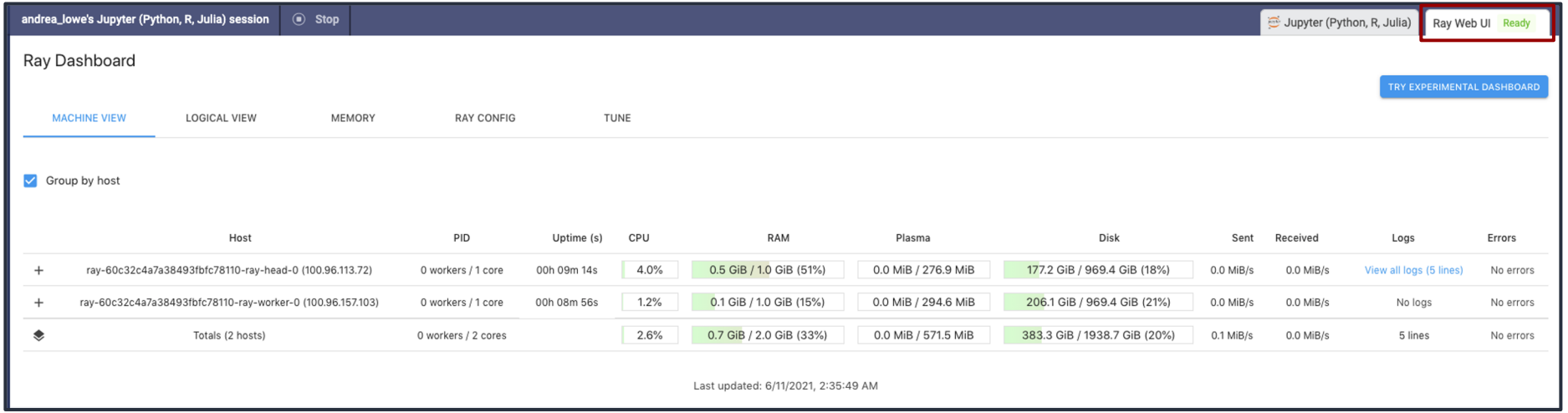
Domino sets up key environment variables

Use Ray Client instead of ray.init()

```
import ray
import ray.util
import os

if ray.is_initialized() == False:
   service_host = os.environ["RAY_HEAD_SERVICE_HOST"]
   service_port = os.environ["RAY_HEAD_SERVICE_PORT"]
    ray.util.connect(f"{service_host}:{service_port}")
```

Accessing Ray UI

The Ray dashboard is available as a dedicated tab within your workspace



Demo: Workspace Set-up

Ray Remote Functions

- Parallelism or distribution of regular python functions
- The decorator `@ray.remote` is used to tell ray to set up the function as a series of objects, and the `ray.get` function will tell ray to execute the process and return the object.
- The `ray.put` function will put an object into memory for ray to use later to execute ray functions upon
- Must be written correctly or ray will not work

Turning a Regular Function into a Ray / Distributed Functior

A few more examples below show how to appropriately use Ray to speed up calculations.

In [12]:
```
def return_multiple():
    time.sleep(5)
    return 1, 2, 3

st = time.time()
a, b, c = return_multiple()
end = time.time()
print(a,b,c)
print('The wall-clock time is ', str(end-st))
```

```
1 2 3
The wall-clock time is  5.00536847114563
```

In [13]:
```
@ray.remote(num_returns=3)
def return_multiple():
    time.sleep(5)
    return 1, 2, 3


st = time.time()
a, b, c = return_multiple.remote()
end = time.time()
print(ray.get(a), ray.get(b), ray.get(c))
print('The wall-clock time is ', str(end-st))
```

```
1 2 3
The wall-clock time is  0.0034894943237304688
```

How Ray Works: Process Identifier (PID)

- Each Ray session has a unique name
  - By default the name is `session_{timestamp}_{pid}` and `pid` belongs to the start-up process as it's the process that calls `ray.init()`
  - The process identifier can be used to trouble-shoot or trace errors during runs (i.e., is the error emanating from a head node or worker nodes).
- Ray can be run in parallel where all processes are run on a single node
- Ray can be run in a distributed fashion where some processes are on a head node and some are on a worker node

Example Code: Function without Ray

computing. Let's take a closer look below.

In [4]:
```
import time
import os
import ray


y = 1
object_ref = y

def add(x, a=1):
    if x == 'add':
        answer = a + 1
    else:
        answer = a
    time.sleep(5)
    print(answer)

number_add =add('add')
number_none =add('hello')

object_ids = []
st = time.time()
for x in range(2):
    x = x
    y_id = add('add')
    object_ids.append(y_id) # the object ids will print out

## getting the results to pass to another function
objects = object_ids
end = time.time()
print("Time taken: ", str(end-st))
```

```
2
1
2
2
Time taken:  10.007688760757446
```

Example Code: Function with Ray

In [5]:
```
import ray
import time

y = 1
object_ref = ray.put(y)

@ray.remote
def add(x, a=1):
    if x == 'add':
        answer = a + 1
    else:
        answer = a
    time.sleep(5)
    print(answer)

number_add = ray.get(add.remote('add'))
number_none = ray.get(add.remote('hello'))

object_ids = []
st = time.time()
for x in range(2):
    x = x
    y_id = add.remote('add')
    object_ids.append(y_id) # the object ids will print out

## getting the results to pass to another function
objects = ray.get(object_ids)
end = time.time()
print(str(end-st))

#notice that the process ids (pid) have the same number (223) except for the head node which has the number 221
```

```
(add pid=1056) 2
(add pid=1056) 1
(add pid=1056) 2
5.0638556480407715
(add pid=1054) 2
```

- Process IDs for this code included
- 1056 (three IDs)
- 1054 (one id)
- Each set of calculations was assigned a PID

DOMINO

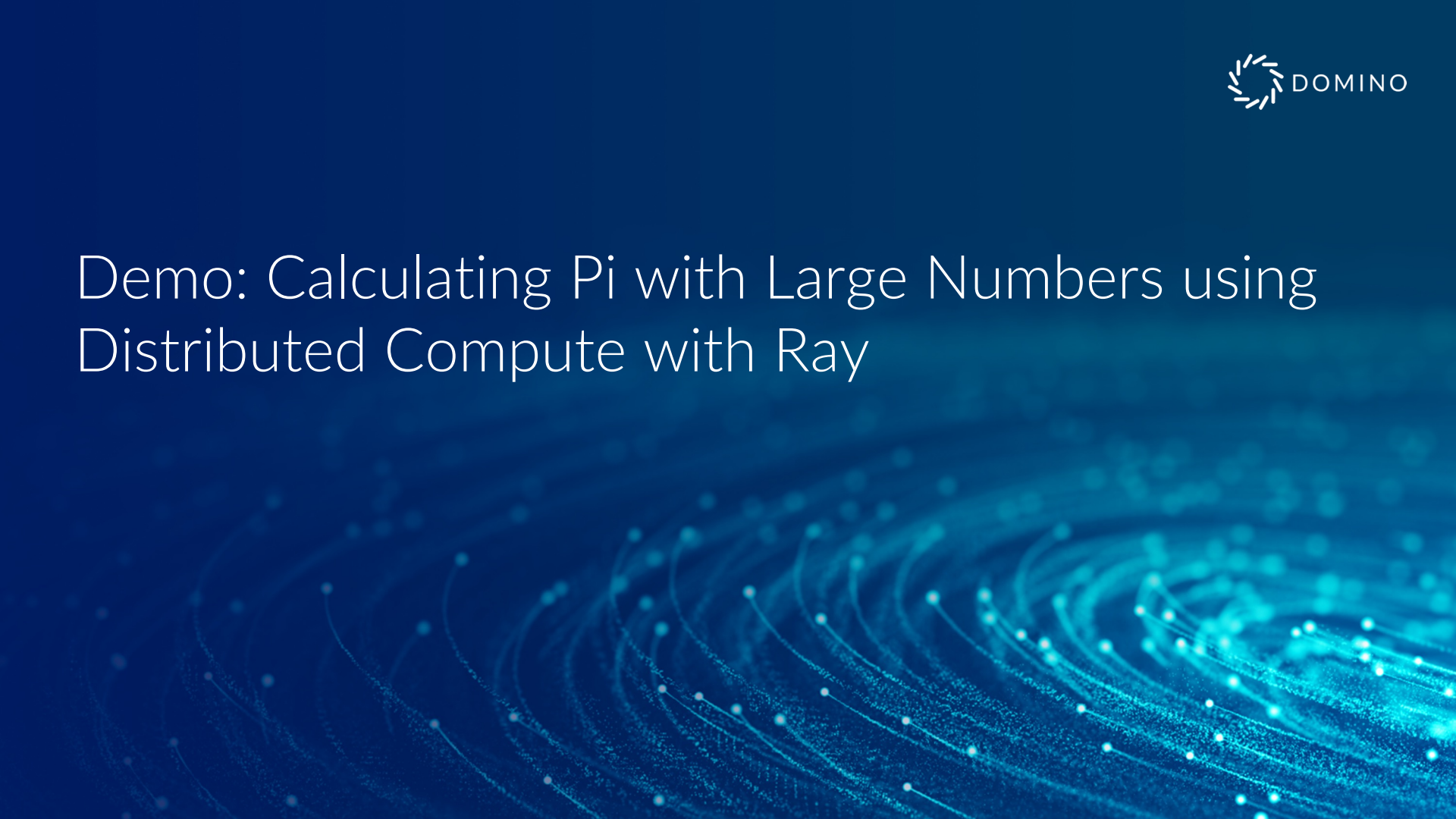
Demo – Create a Remote Function, and Process Identifiers

Break

Calculating Pi with and without Ray

- Demonstrates use of large numbers / data with:
  - No parallelism or distribution of calculations
  - Parallelism using Python's multiprocessing library
  - Distributed and Parallelized code using Ray
- Pay special attention to the speed of the calculations
  - No parallelism or distribution of calculations is slowest
  - Parallelism helps
  - Distributed parallelism is the most efficient

Demo: Calculating Pi with Large Numbers using Distributed Compute with Ray



Recap of What We've Learned

- How to start a workspace
- How to turn a basic function into a Ray distributed Function
- Understanding how Ray works with Process Identifiers
- Calculating pi using Ray to demonstrate Ray's speed with big data

Recap Use Cases Ray

- Deep learning
- Computer vision
- Natural Language processing
- Transfer Learning
- Natural Language Generation
- Converting Tensorflow to distributed compute
- Converting Pytorch to distributed compute
- Hyper-parameter optimization

Dask in Domino

Agenda

- Differences between parallel and distributed computing
- Setting up a Distributed Compute workspace for Dask
- Review of what Dask can do
- Dask Use Cases
- Calculating Pi – ”Hello World” Example
- Dask Pandas-Like Library
- Dask Machine Learning Library
- Recap

Learning Goals

- Starting up your Workspace
- Understanding what Dask can do
- Understanding when Dask is appropriate to use
- Understanding the difference between regular calculations and distributed with Dask
- Understand how to rewrite code to run on Dask

Dask Overview

- Dask provides a set of libraries with familiar syntax that are easy to use and distributes calculations or data operations over several workers (CPUs or GPUs).

- The following libraries that can be used with Dask (but aren't limited to):
  - Pandas-like library (built-in to Dask)
  - Scikit-learn-like library (Dask Machine Learning)
  - Xgboost integration with Dask
  - Ability to take a regular python function and convert it to a Dask (distributed) Function

Dask Distributed Compute | Benefits

Self-serve access
Bring up Dask clusters on-demand directly on the underlying Domino infrastructure with just a few clicks.

Parallelize workloads
Speed up training jobs by scaling beyond a single large machine. Use multiple CPUs, GPUs, or large memory pools.

Use with popular libraries
Tight integration with NumPy, Pandas, Scikit-Learn and other popular data science packages to minimize existing code rewrite.

Dask on Demand in Domino: Use Cases

Working with Large Datasets

Great for scaling up Python data analysis or transformation where processing requires more resources than can be provided by a single machine.

Performed with minimal modification and without having to switch over to a different ecosystem like Spark.

Distributed Training

Dask provides a simple way to parallelize existing scikit-learn models.

For larger memory-bound workloads, there are Dask specific ML libraries (e.g. Parallel Meta-estimators, Incremental Hyperparameter Optimizers)

Custom Distributed Computations

The low-level Dask scheduling APIs can be used to build custom algorithms that can benefit from parallelism.

You control the business logic while Dask handles task dependencies, network communication, workload resilience, diagnostics, etc.

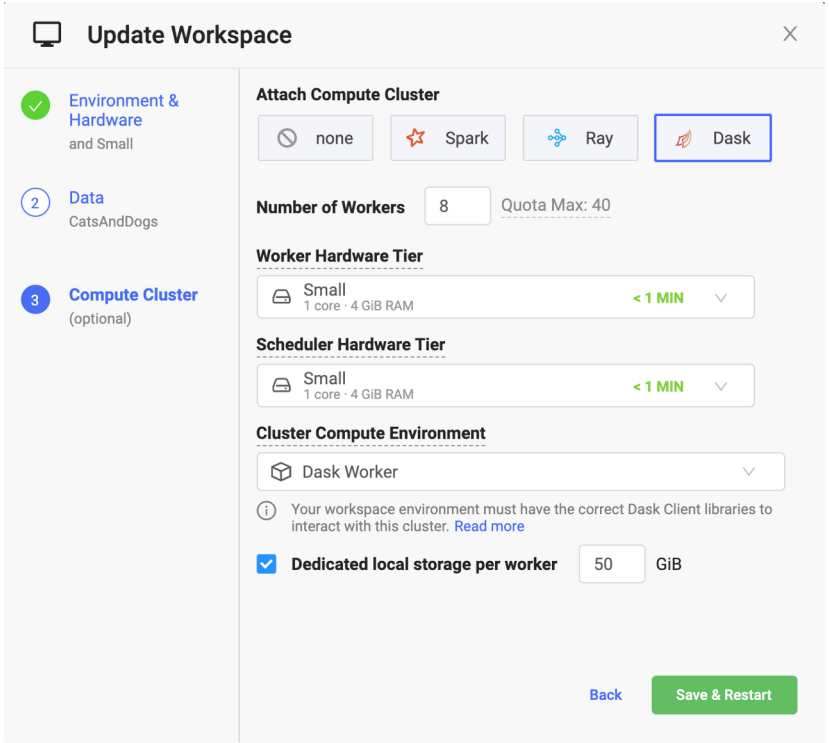
Demo: Creating Distributed Functions in Dask

Steps to setting up a workspace

- Name your workspace
  - Use a name you will remember that indicates whether it is distributed or not and the type of tool used (i.e. in this case Dask)
- Choose your workspace environment
- Ensure your datasets are available
- Choose your scheduler and compute hardware tier
- Choose the number of workers

Distributed Compute | Setting up your workspace

- With only a handful of specified parameters, such as the number of workers and the capacity of each worker, Domino will set up the cluster and seamlessly manage its lifecycle.
- Administrators retain control since the cluster configuration is based on the established Domino concepts of Hardware Tiers and Compute Environments.



Dask Distributed Functions

- Parallelism or distribution of regular python functions
- The decorator `@dask.delayed` is used to tell Dask to set up the function as a series of objects, and the addition of .compute function will tell Dask to execute the process and return the object.
- Must be written correctly or Dask will not work
- Let's look at a code snippet and see how it works

Turning a Regular Function into a Dask / Distributed Function

In [46]:

```
%%time

import dask

@dask.delayed
def inc(x):
   return x + 1

@dask.delayed
def add(x, y):
   return x + y

a = inc(1)       # no work has happened yet
b = inc(2)       # no work has happened yet
c = add(a, b)    # no work has happened yet
c = c.compute()  # work starts and finishes
print(c)
```

```
5
CPU times: user 10.5 ms, sys: 0 ns, total: 10.5 ms
Wall time: 18.6 ms
```

Dask Distribution: Client Set-up

In [25]:
```
from dask.distributed import Client
import os

service_host = os.environ["DASK_SCHEDULER_SERVICE_HOST"]
service_port = os.environ["DASK_SCHEDULER_SERVICE_PORT"]
client = Client(f"{service_host}:{service_port}")

# look at the client and scheduler

client

# you should now be connected to the cluster
# Dashboard link from the client object is clickable but wi
ll not route in Domino
# Use the embedded Dask Web UI tab instead
```

Out[25]:

Client

Client-72a8af49-793c-11ec-8160-da15e17a09f3

Connection method: Direct

Dashboard: http://dask-61e8143c3d51792f465e7f27-dask-scheduler.fieldc272254-compute.svc.cluster.local:8787/status

Scheduler Info

Submitting Jobs to the Client

- Two ways to do this:
  - Dask Delayed function creates a future that is executed with the .compute() is called
  - Client.submit function can take a regular function and submit it to the client to distribute to workers
- Both methods are equivalent except that Dask delayed creates a future function (stored in memory) that is not executed on until the .compute() is called

- Submitting to the client directly can be useful when one does not want to store functions in memory
  - To submit directly the syntax client.submit(function, term) is used

Using Dask Delayed: Example

Wall time: 11.5 s

In [8]:
```
%%time

from dask import delayed

pi = delayed(pi)(10**7)
pi.compute()
```

```
CPU times: user 11.6 ms, sys: 64 µs, total: 11.7 ms
Wall time: 11.2 s
```

Out[8]:
```
3.1415925535897915
```

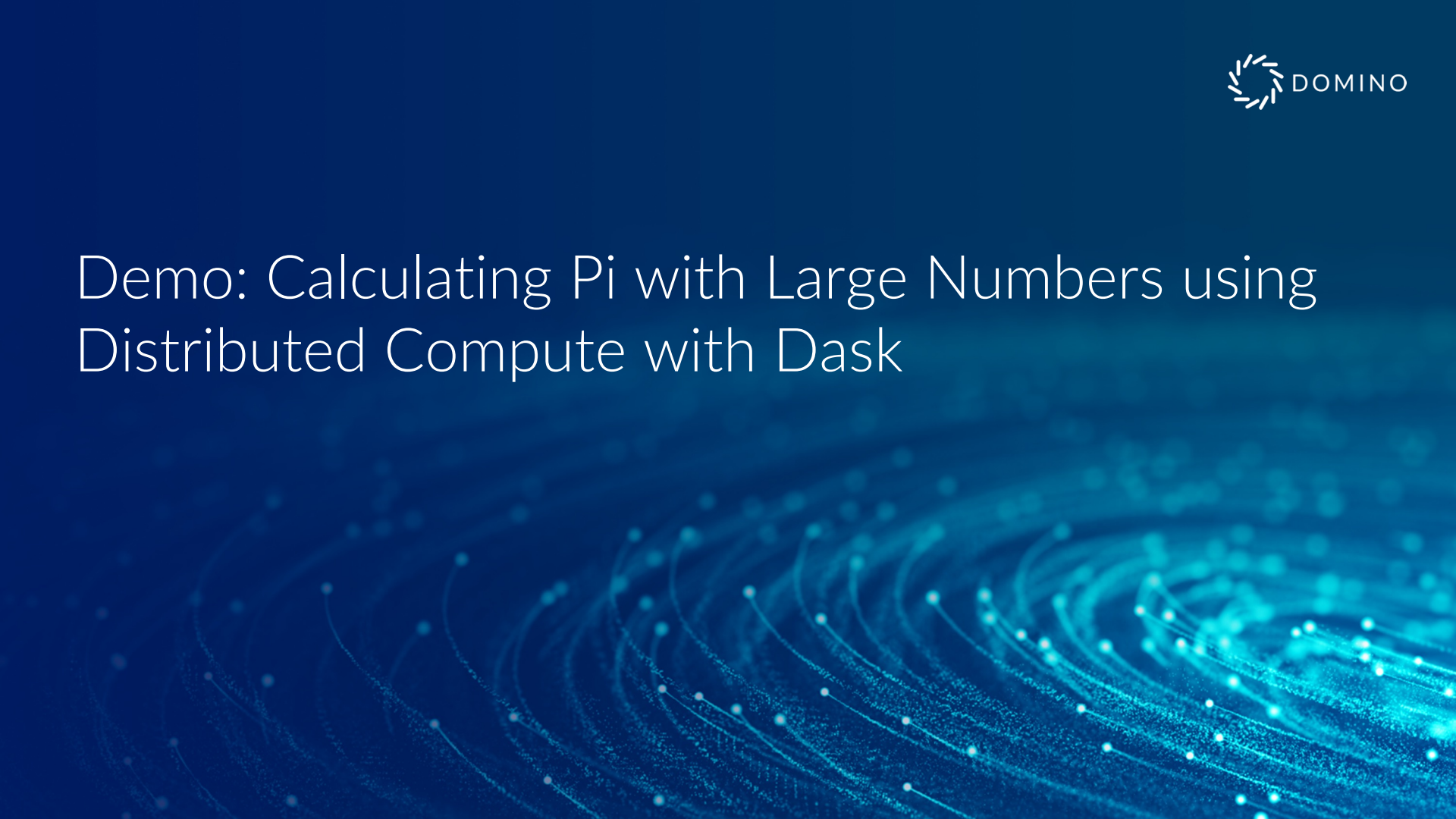
Using Dask's Client: Example

In [6]:
```
import time
## compute pi without Dask - WIP - doesn't seem to be available


def pi(N):
    time.sleep(10)

    # Initialize denominator
    k = 1

    # Initialize sum
    s = 0

    for i in range(N):

        # even index elements are positive
            if i % 2 == 0:
                s += 4/k
            else:
                s -= 4/k

        # denominator is odd
            k += 2
    return s
```

In [7]:
```
%%time

pi_client = client.submit(pi, 10**7)
pi_client = pi_client.result()
print(pi_client)
```

```
3.1415925535897915
CPU times: user 7.33 ms, sys: 3.67 ms, total: 11 ms
Wall time: 11.3 s
```

Demo: Calculating Pi with Large Numbers using Distributed Compute with Dask



Calculating Pi

- Demonstrates use of large numbers / data with:
  - No parallelism or distribution of calculations
  - Parallelism using Python's multiprocessing library
  - Distributed and Parallelized code using Dask
- Pay special attention to the speed of the calculations

Example Code: Calculate Pi without Dask

In [9]:
```
import time

def pi_single_term(i):
    time.sleep(0.01)
    # denominator is odd
    k = 2*i + 1
    # even index elements are positive
    if i % 2 == 0:
        x = 4/k
    # odd index elements are negative
    else:
        x = - 4/k
    return x
```

In [10]:
```
%%time
## compute pi without Dask
s = []
for i in range(10**3):
    s.append(pi_single_term(i))
sum(s)
```

```
CPU times: user 13.5 ms, sys: 3.4 ms, total: 16.9 ms
Wall time: 10.1 s
```

Pi is calculated using inference and a single term without distribution of the calculation

Example Code: Calculate Pi Dask Delayed

In [11]:
```
%%time
## compute pi with Dask using Delayed

s = []
for i in range(10**3):
    s.append(dask.delayed(pi_single_term)(i))
dask.delayed(sum)(s).compute()
```

```
CPU times: user 77.9 ms, sys: 19 ms, total: 96.9 ms
Wall time: 984 ms
```

Out[11]:
```
3.140592653839794
```

Pi is calculated using inference and a single term with distribution of the calculation using Dask Delayed

DOMINO

Example Code: Calculate Pi Dask Client

In [12]:
```
%%time
## compute pi with Dask using Client

s=[]
for i in range(10**3):
    s.append(client.submit(pi_single_term, i))
client.submit(sum, s).result()
```

```
CPU times: user 167 ms, sys: 10.1 ms, total: 177 ms
Wall time: 896 ms
```

Out[12]:
```
3.140592653839794
```

Pi is calculated using inference and a single term distribution of the calculation using Dask client

Demo: Using Pandas with Dask

Pandas versus Dask Dataframe

- Pandas dataframes are stored on a single CPU or GPU and not distributed across workers
- Dask Dataframes can be distributed across workers
- Syntax between the two is almost identical
- Easy to do data munging with very large data using Dask Dataframes

Reading a csv and getting results using Pandas

In [19]:
```
import pandas as pd

df = pd.read_csv("/domino/datasets/local/polioData/polioData.csv")

def get_counts(df):
    by_state = df.groupby("state")
    incidence = by_state["incidence_ratio"]
    return incidence.value_counts()

result = get_counts(df)
result.sort_values(ascending=False, inplace=True)

print(result)
```

```
state  incidence_ratio
DE     0.00              1072
WY     0.00              1043
UT     0.00               949
MT     0.00               940
ID     0.00               940
                         ...
KY     1.14                 1
       1.13                 1
       1.06                 1
       1.05                 1
WY     8.93                 1
Name: incidence_ratio, Length: 6939, dtype: int64
```

Reading a csv and getting results using Dask Dataframes

In [20]:
```
import dask.dataframe as dd

dd = dd.read_csv("/domino/datasets/local/polioData/polioData.csv")

def get_counts(df):
    by_state = df.groupby("state")
    incidence = by_state["incidence_ratio"]
    return incidence.value_counts()

result = get_counts(dd)
result = result.compute(num_workers=4)
result.sort_values(ascending=False, inplace=True)

print(result)
```

```
state  incidence_ratio
DE     0.00              1072
WY     0.00              1043
UT     0.00               949
ID     0.00               940
MT     0.00               940
                         ...
KY     1.19                 1
       1.18                 1
       1.14                 1
       1.13                 1
WY     8.93                 1
Name: incidence_ratio, Length: 6939, dtype: int64
```

Recap of What We've Learned

- How to start a workspace
- How to turn a basic function into a Dask delayed Function
- Understanding how the Dask Client works
- Calculating pi using Dask to demonstrate Dask's speed with big data
- Pandas-like Library in Dask (dask_distributed)

Demo Using Dask – A Simple K-Means Clustering Example

Dask and Scikit-Learn Integration

- Dask has a machine learning library that closely mimics Sci-kit learn Libraries
- Scikit-learn can also be used natively with Dask using the @delayed futures
- Let's look at two ways to create a k-means function
    - Without Dask, just using Sci-kit Learn

K-means Clustering using Scikit Learn

Using Sci-Kit Learn natively on Dask we see a training time as noted below:

In [14]:
```
# import libraries

from sklearn import cluster
import time
from sklearn import datasets
import matplotlib.pyplot as plt
%matplotlib inline
```

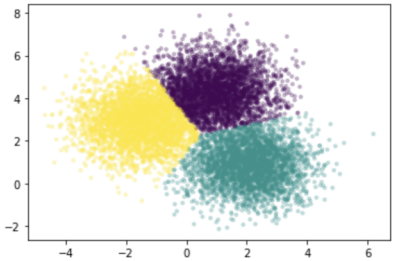
In [15]:
```
X, y = datasets.make_blobs(n_samples=10**8, random_state=0,
                                    centers=3)
```

In [19]:
```
%%time

km = cluster.KMeans(n_clusters=3)
km.fit(X)
```

```
CPU times: user 6min 34s, sys: 44.9 s, total: 7min 19s
Wall time: 3min 38s
```

Out[19]:
```
KMeans(n_clusters=3)
```

In [20]:
```
fig, ax = plt.subplots()
ax.scatter(X[::10000, 0], X[::10000, 1], marker='.', c=km.labels_[::10000],
           cmap='viridis', alpha=0.25);
```



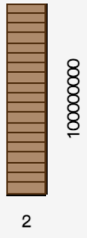
K-means Clustering using Dask: Create Data and Dataframe

In [26]:
```
## import dask ml libraries

import dask_ml.datasets
import dask_ml.cluster
import time
import matplotlib.pyplot as plt
%matplotlib inline
```

In [27]:
```
X, y = dask_ml.datasets.make_blobs(n_samples=10**8,
                                   chunks=1000000,
                                   random_state=0,
                                   centers=3)

X = X.persist()
X
```

Out[27]:

	Array	Chunk
Bytes	1.49 GiB	15.26 MiB
Shape	(100000000, 2)	(1000000, 2)
Count	100 Tasks	100 Chunks
Type	float64	numpy.ndarray



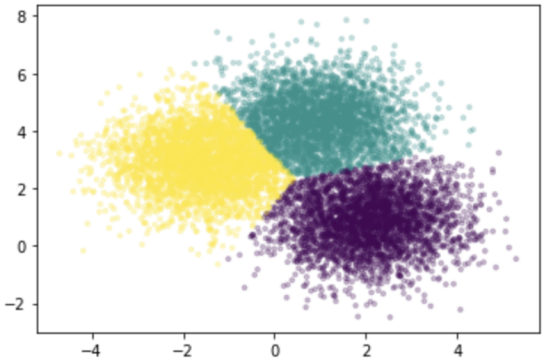
K-means Clustering Using Dask: Perform and Visualize

In [31]:
```
%%time

km = dask_ml.cluster.KMeans(init='k-means||', n_clusters=3, init_max_iter=2, oversampling_factor=10)
km.fit(X)
```

```
CPU times: user 1.81 s, sys: 23.7 ms, total: 1.83 s
Wall time: 33.6 s
```

Out[31]:
```
KMeans(init_max_iter=2, n_clusters=3, oversampling_factor=10)
```

In [24]:
```
fig, ax = plt.subplots()
ax.scatter(X[::10000, 0], X[::10000, 1], marker='.', c=km.labels_[::10000],
           cmap='viridis', alpha=0.25);
```



Recap of What We've Learned

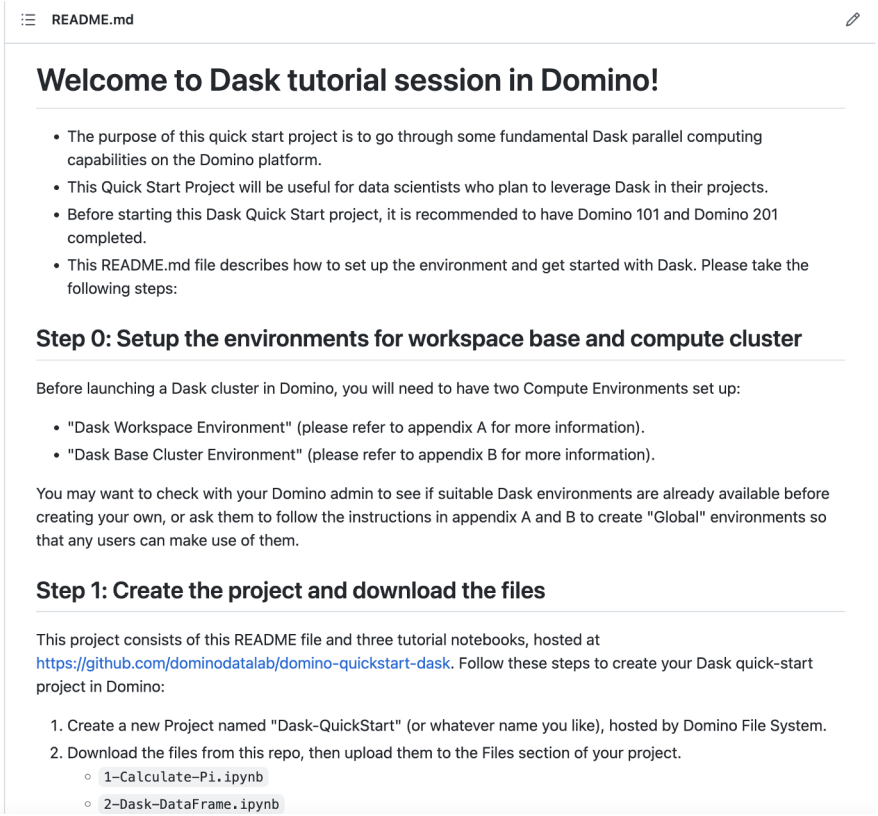
- How to start a workspace
- How to turn a basic function into a Dask delayed Function
- Understanding how the Dask Client works
- Calculating pi using Dask to demonstrate Dask's speed with big data
- Pandas-like Library in Dask (dask_distributed)
- An example of a distributed K-mean Clustering

Recap Use Cases Dask

- Large-scale Machine Learning
- Data munging with Pandas-like functions
- Visualizations
- Xgboost libraries

Dask and Ray Quick-Starts

- Quick-Start Projects are available in Domino Data Lab's open repository
- Dask and Ray quick starts are available
- Dask quick start here: Dask Quick-Start
- Ray quick start here: Ray Quickstart



README.md

Welcome to Dask tutorial session in Domino!

- The purpose of this quick start project is to go through some fundamental Dask parallel computing capabilities on the Domino platform.
- This Quick Start Project will be useful for data scientists who plan to leverage Dask in their projects.
- Before starting this Dask Quick Start project, it is recommended to have Domino 101 and Domino 201 completed.
- This README.md file describes how to set up the environment and get started with Dask. Please take the following steps:

Step 0: Setup the environments for workspace base and compute cluster

Before launching a Dask cluster in Domino, you will need to have two Compute Environments set up:

- "Dask Workspace Environment" (please refer to appendix A for more information).
- "Dask Base Cluster Environment" (please refer to appendix B for more information).

You may want to check with your Domino admin to see if suitable Dask environments are already available before creating your own, or ask them to follow the instructions in appendix A and B to create "Global" environments so that any users can make use of them.

Step 1: Create the project and download the files

This project consists of this README file and three tutorial notebooks, hosted at https://github.com/dominodatalab/domino-quickstart-dask. Follow these steps to create your Dask quick-start project in Domino:

1. Create a new Project named "Dask-QuickStart" (or whatever name you like), hosted by Domino File System.
2. Download the files from this repo, then upload them to the Files section of your project.
   - `1-Calculate-Pi.ipynb`
   - `2-Dask-DataFrame.ipynb`

On-demand Open MPI Clusters

What is MPI?

MPI, or message passing interface, is a communication library widely used in high-performance computing

- It is one underlying communication mechanism for many higher-level machine learning frameworks (e.g. PyTorch, Tensorflow, Keras, MXNet etc)
- Frequently used in conjunction with Horovod but also applicable for broader use cases
- Python API exist called mpi4py

Domino On-demand MPI Use Cases

Domino on-demand MPI clusters are suitable for the following workloads:

- Distributed Multi-GPU Training: Open MPI is ideal for distributed multi-GPU training for Tensorflow, PyTorch, Keras, or MXNet models.
- High Performance Computing MPI clusters are generally faster than other distributed computing systems and highly customizable.
- Domino offers an easy interface for use of MPI (not HPC)

MPI Cluster Setup

MPI Environment Setup

When using on-demand MPI in Domino you will have two separate environments:

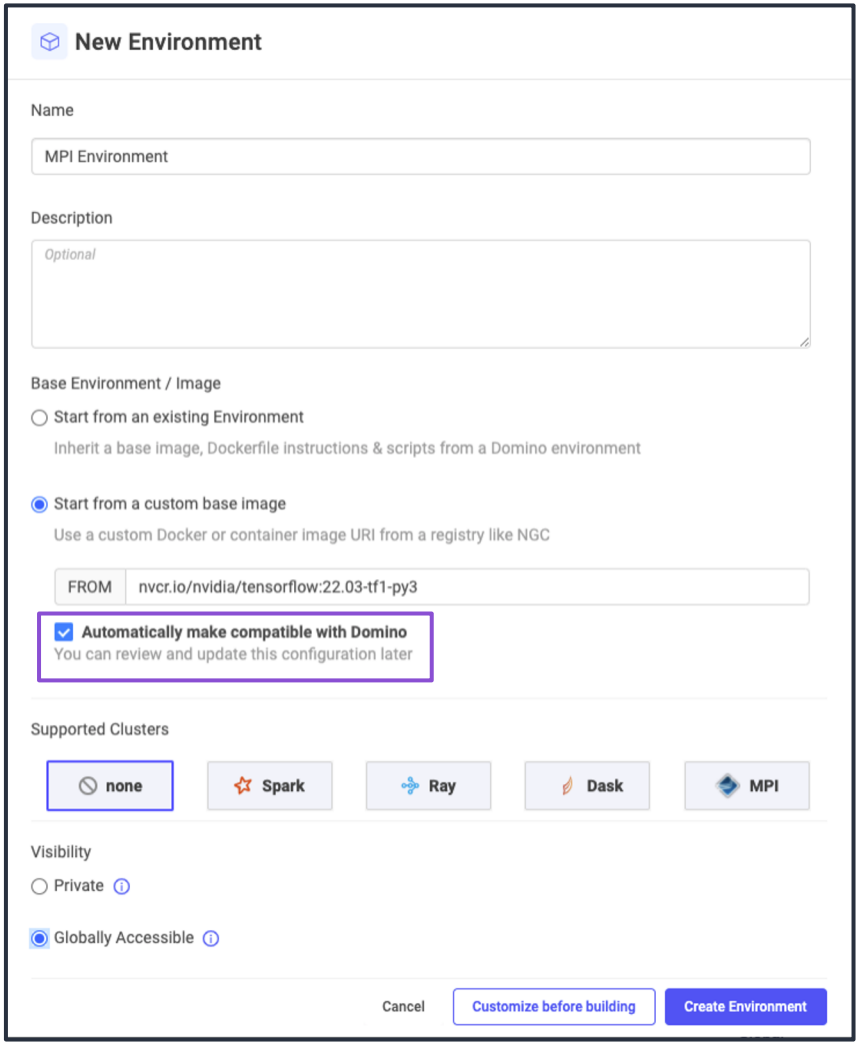
1. A Base environment for the workspace/job
2. The Open MPI cluster environment

The environments must have the same version of Open MPI installed, located in the same directory

MPI Environment Setup

We recommended using an NVIDIA NGC container image with MPI support for both environments

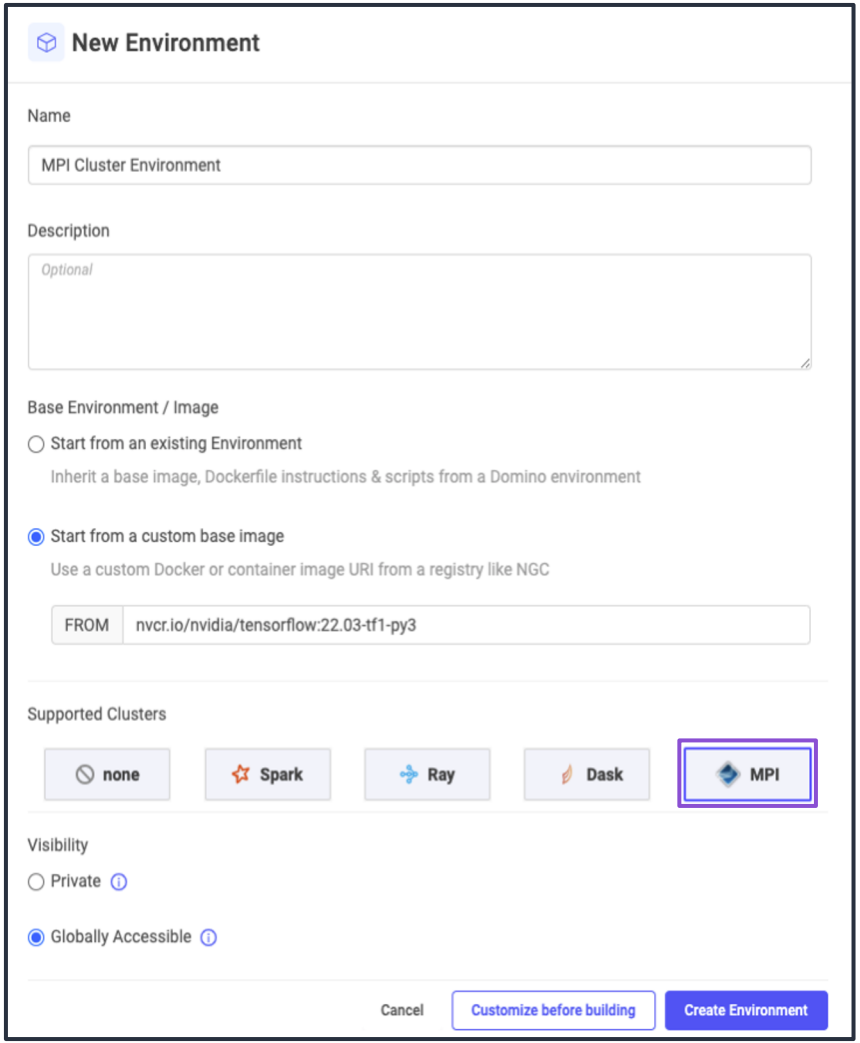
Use the option to make it automatically compatible for the Workspace/Job environment



MPI Environment Setup

We recommended using an NVIDIA NGC container image with MPI support for both environments

Select MPI for the cluster environment



MPI Cluster Environment Setup

For simple, CPU-only workloads, we vendor matching images:

Workspace/Job:

```
quay.io/domino/mpi-environment:ubuntu18-py3.8-r4.1-
mpi4.1.2-domino5.1.1
```

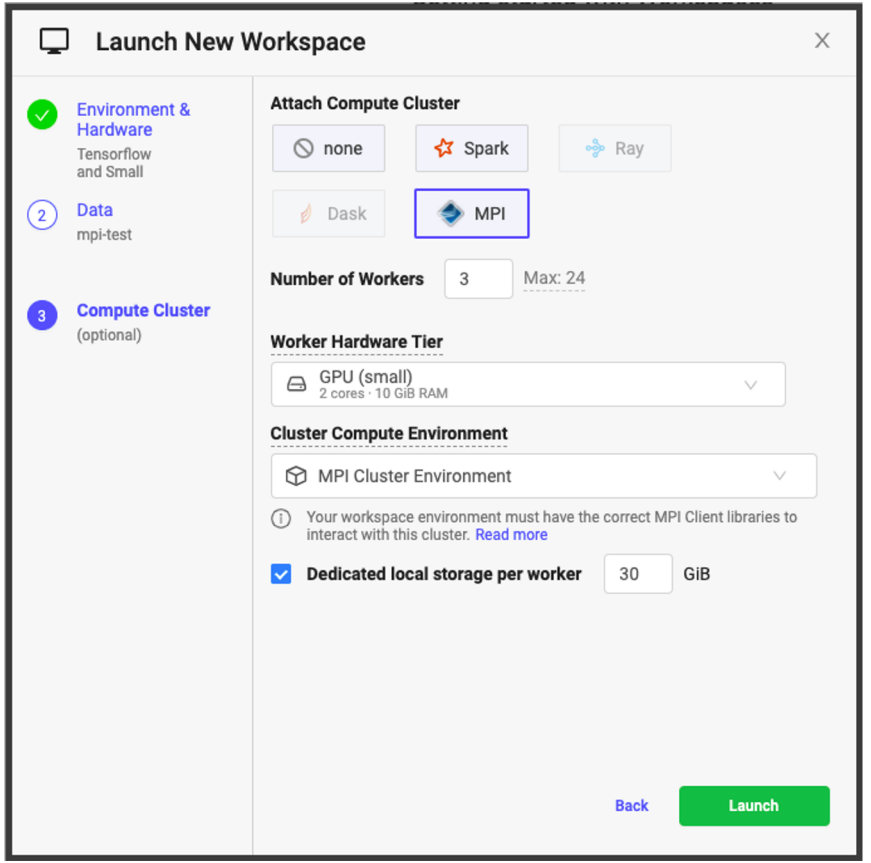
Cluster:

```
quay.io/domino/mpi:Mpi4.1.2-Py3.8.12
```

Working with MPI Clusters

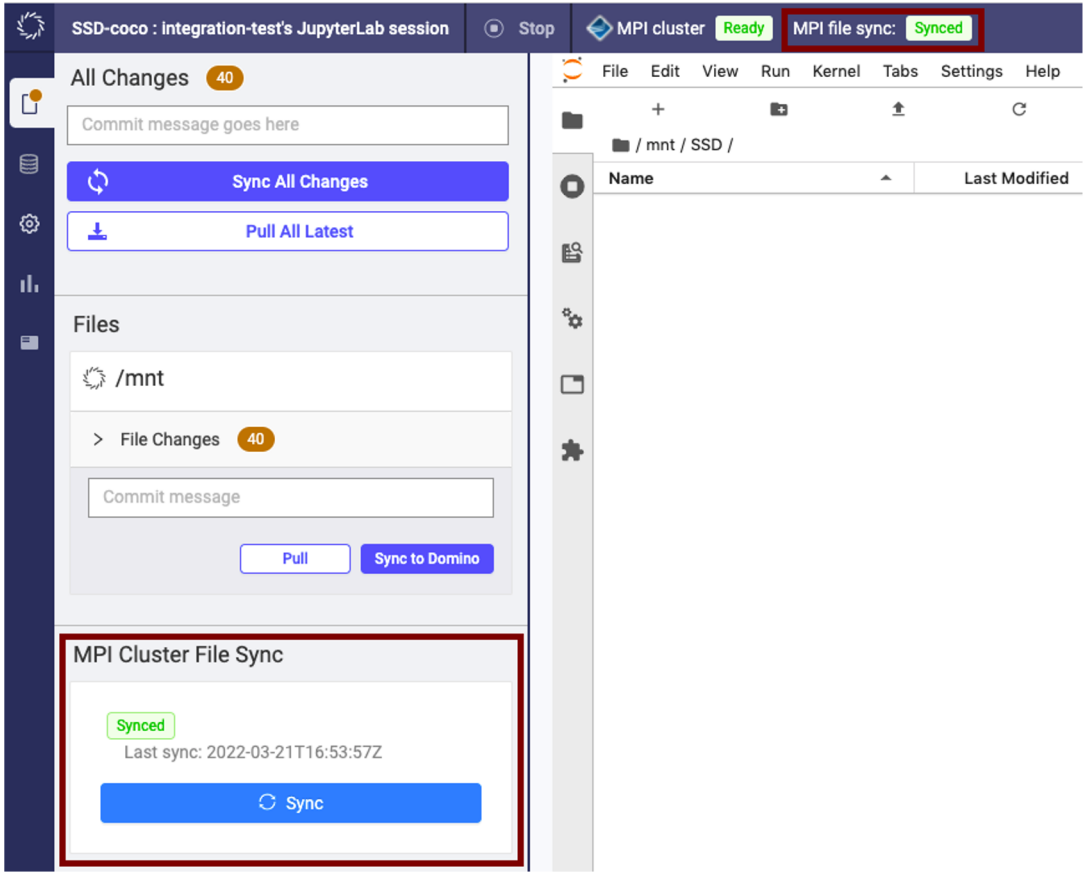
Launch an MPI Cluster

- MPI clusters do *not* have a scheduler node
- Choose number of workers and their resources
- Add temporary local storage if needed: mounted to `/tmp`



Workspace

- There is no MPI user interface (as with other clusters)
- Files must be manually synced to worker nodes if changed after Workspace starts
- Domino datasets are automatically mounted to worker nodes



DOMINO

Connect to the MPI cluster

Default configurations

- A hostfile is created with the name of the worker nodes, located at `/domino/mpi/hosts`
- Hosts are set to all workers. This does not include the Workspace/Job container.
- Slots are based on the CPU/GPU cores in the selected worker hardware tier. The minimum is one.
- Defaults are set for mapping (socket) and binding (core)

Connect to the MPI cluster - Workspaces

Submit `mpirun` jobs within Workspace

You can supply arguments for the `mpirun` command as described in the Open MPI docs, including:

- `--x <env>`: This controls the settings to distribute an existing environment variable to all workers
- `--hostfile <your hostfile>`: Specify a new hostfile with custom slots.
- `--map-by <core>`: Map to the specified object.
- `--bind-to socket`: Bind processes to the specified object.

Connect to the MPI cluster - Workspaces

Code example:

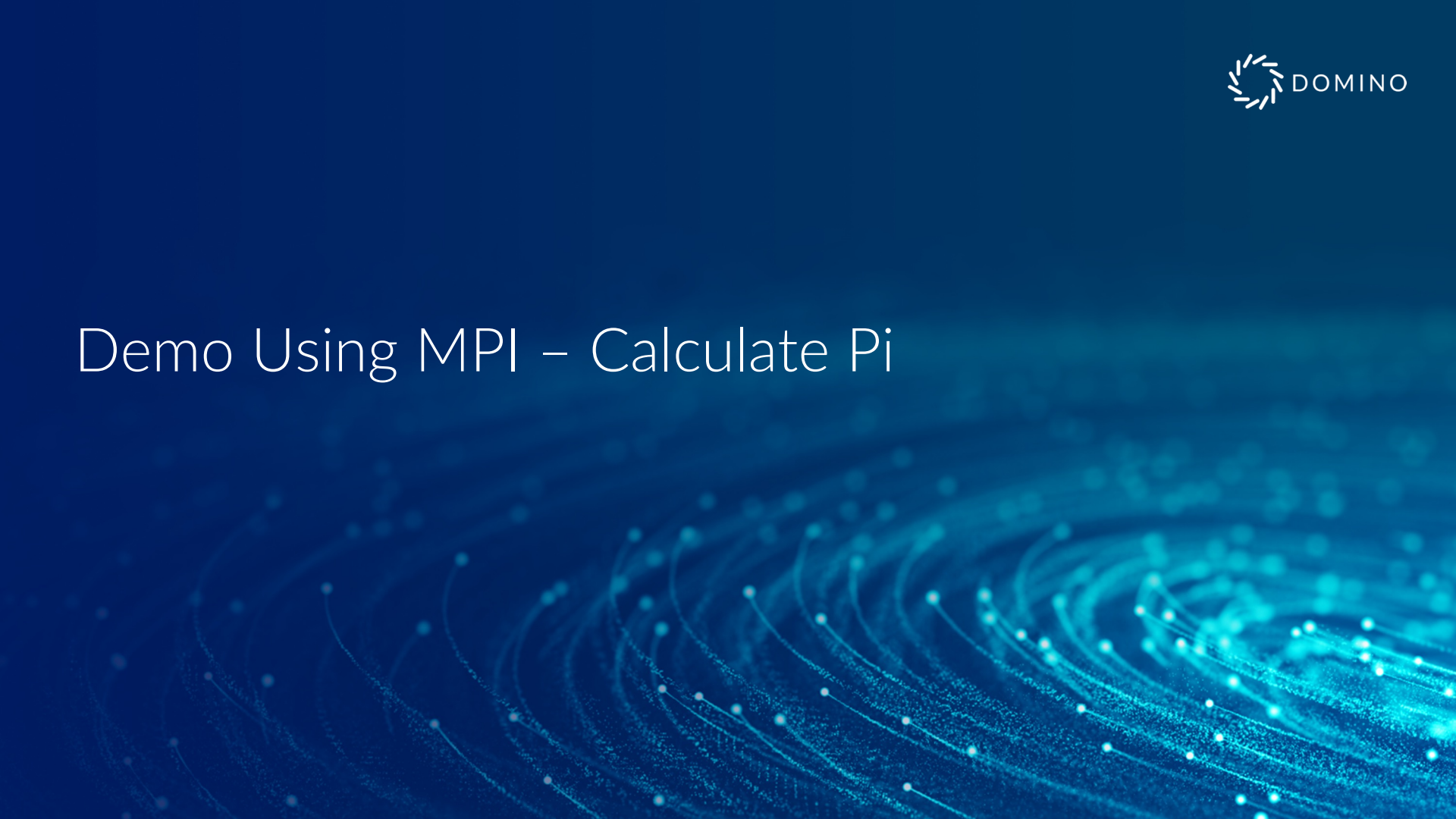
To add `–bind-to none`, which specifies that a training process should not be bound to a single core, and `-map-by slot`, use:

```
mpirun \
    --bind-to none \
    --map-by slot \
    python my-training.py
```

Using Python API for Open MPI

- Python API exists and must be installed on both the workspace and clusters by indicating so in the two environments
- Python wrapper requires very little knowledge of Open MPI
- Documentation can be found here: Documentation (mpi4py)
- For our ‘hello world’ example we will Calculate Pi

Demo Using MPI – Calculate Pi



Submitting a File to the MPI Cluster

- Calculate Pi will use a basic Monte Carlo Method as we have done with our other distributed compute methods
- Code available in C and Python
- Example uses mpi4py API to submit the MPI code to run on Domino
  - Requires a file that is to be submitted to the MPI cluster
  - Requires a command line interface to submit file using python API
  - Very similar to how MPI works with supercomputing applications
- Other methods include using shell files to submit R or Python code

GPUs in Domino

GPUs

Perform multiple, simultaneous computations enabling the distribution of training processes

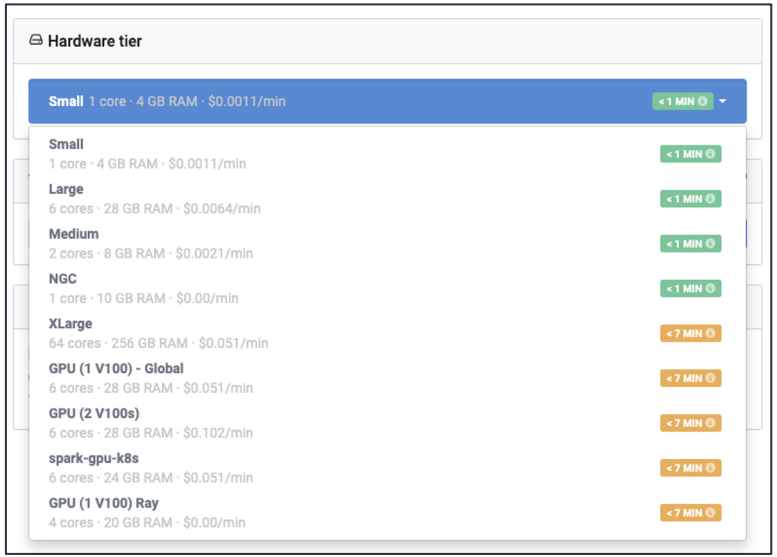
Large number of simple cores vs a few complicated cores (CPU)

Note that GPUs can have a bandwidth bottleneck issue - transferring large amounts of data to the GPU can be slow

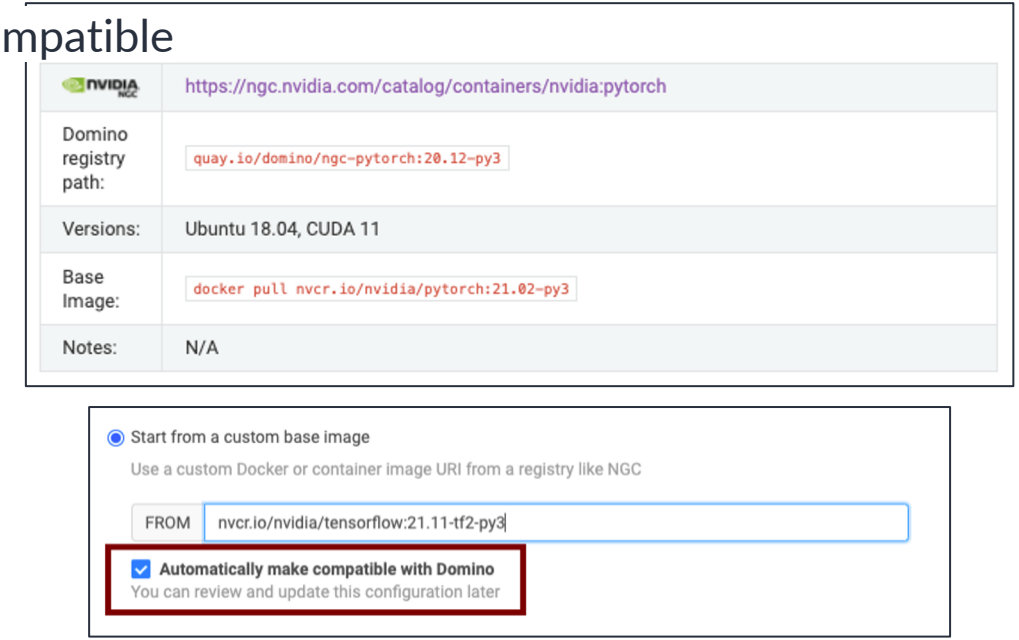
- Optimizing this process is covered in the tutorial notebooks

NVIDIA and Domino

GPUs available as hardware tiers



NGC containers available with Domino configurations or can be made automatically compatible



DOMINO

System Utilization

- While a model is training, you may want to confirm that it's making full use of your hardware.
- If it isn't, that's a sign that you can do things like train larger batches, a larger model, or do more things in parallel.
- `nvidia-smi` shows information about the GPUs in your system and their memory/processor usage.
- `top` is a commonly-used linux command showing the most active processes.

System Utilization

- You may occasionally need to clear cached data from the GPU between model trainings
- `torch.cuda.empty_cache()`

Optimization

In software development, it's good to put effort into making something faster when you know that the part you are working on is the bottleneck.

`line_profiler` is a great Jupyter extension to profile functions line-by-line:

```
%load_ext line_profiler
```

Optimization

It's best practice to use a command like the one below instead of hard-coding GPUs, so your code can still run (very slowly!) on a cpu-only machine:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print('Using {} device'.format(device))
```

Optimizing GPUs Demo

Advanced Course Summary

Great Job! You've completed this course!
The main learning points from this course are:

- Setting your project up for success
- Domino Environments, Custom Images and Best Practices
- Benefits and Use Cases for On-Demand Distributed Computing in Domino
- Spark, Ray, Dask and MPI Integration, Distributed Computations
- GPUs Provide Performance and Optimization in Domino

DOMINO

Thank you!

Any Questions?

Help us improve! Please fill out our post-training survey here:
http://bit.ly/domino_training_survey

Code Repository: https://github.com/jddavis1000/Rev3_Advanced_Data_Science