Apache Airflow and Ray

Orchestrating ML at Scale

Daniel Imberman

Background

- Strategy Engineer at astronomer.io, Airflow PMC member, co-creator of K8sExecutor
- Previously: Building data science platforms at Bloomberg LP
- Obsessed with Data Science Tooling, and building distributed systems

The Airflow Data Science Story

The Airflow Data Science Story

- Airflow is the tool to take you from experiment to production model
- Monitoring and scheduling ensure your models update in time for SLAs
- Connection handling for easily switching between dev -> prod data sources
- Fault tolerant scheduler that can retry jobs in case of failure

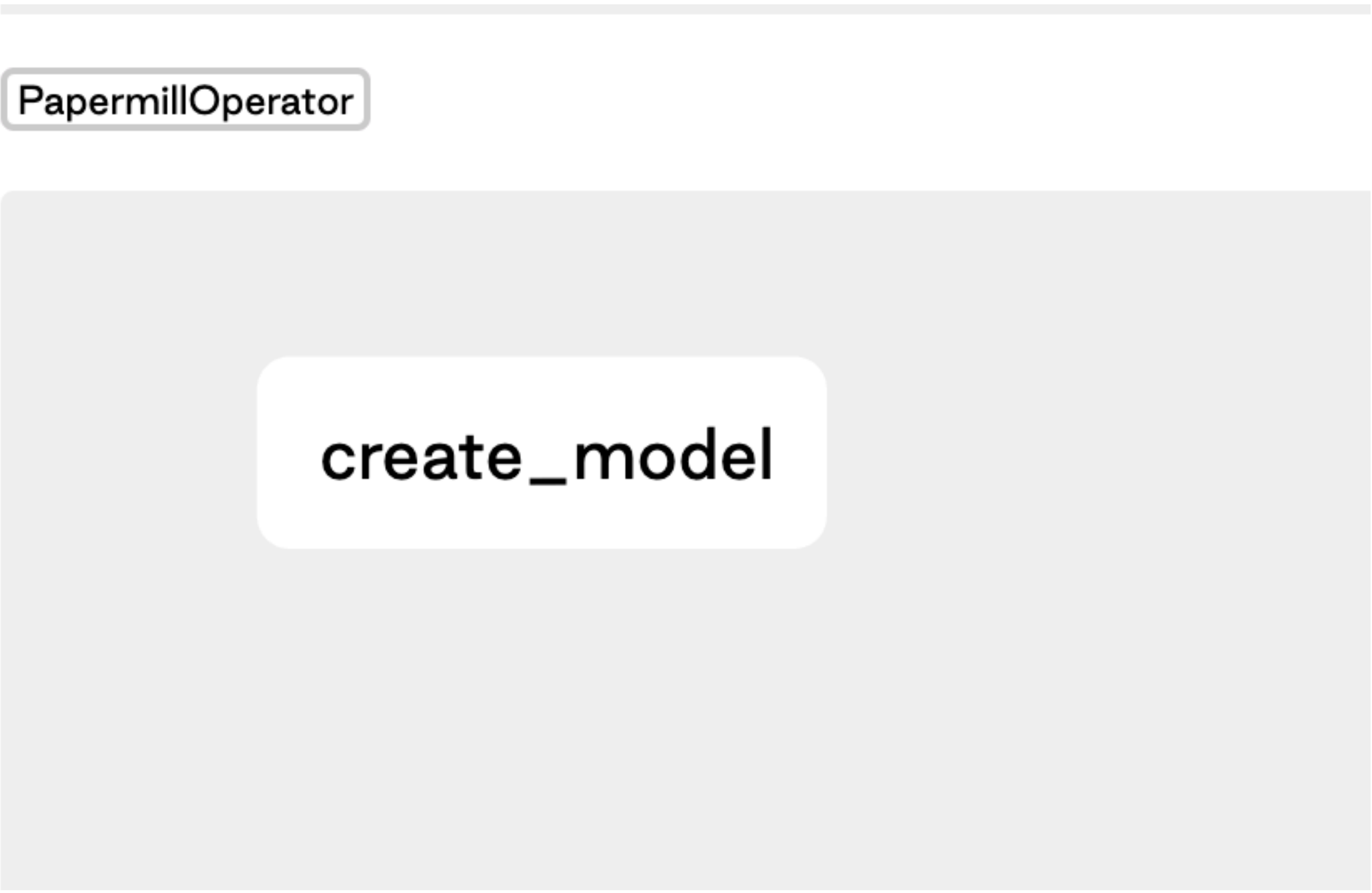
The Airflow Data Science Story

The (Traditional) Airflow Data Science Story

Experiment	Parameterize	Productionize
jupyter	papermill	

Papermill

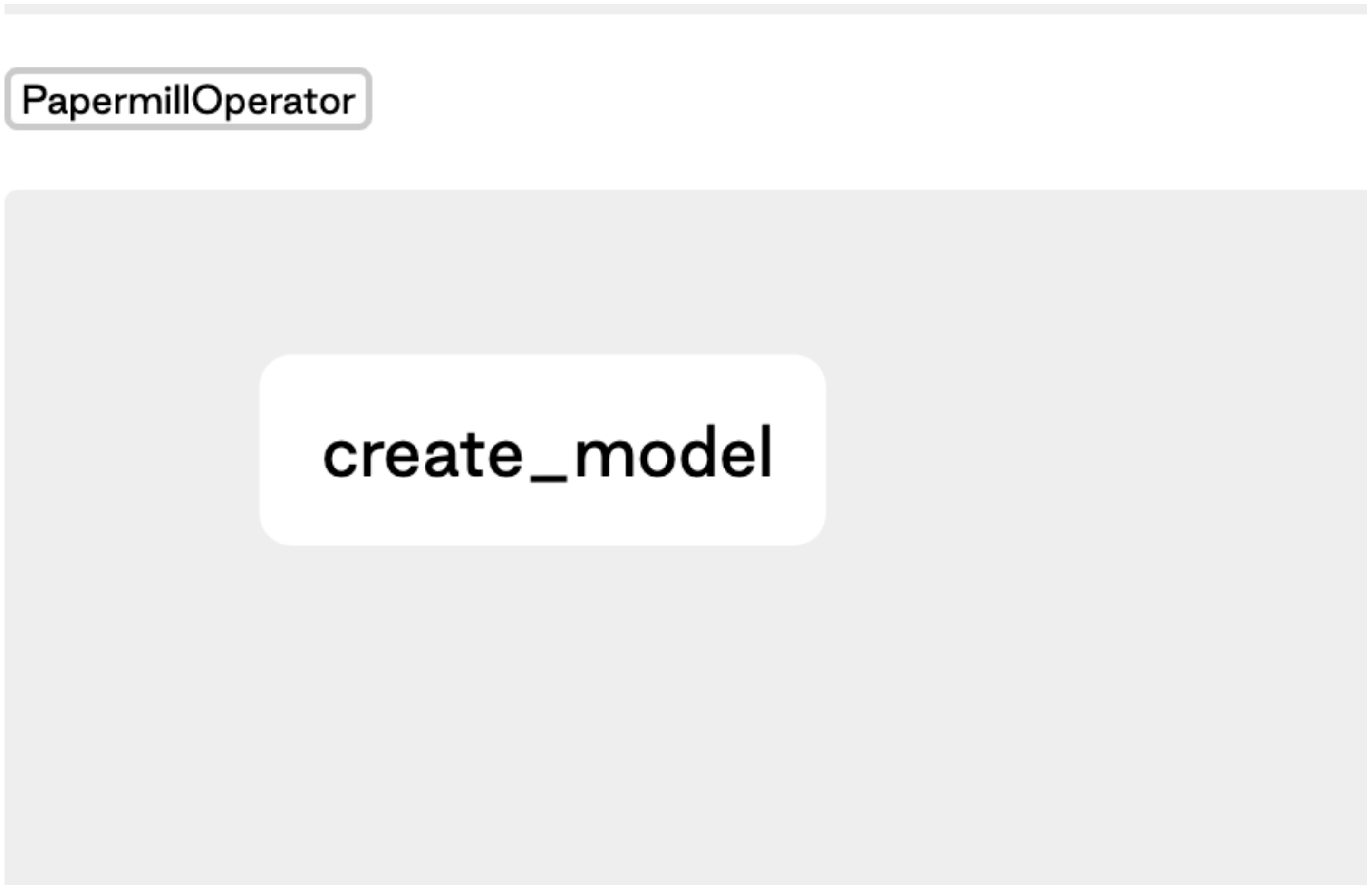
- Parameterize Notebooks through cell tagging
- Stores intermediate notebooks
- Execute using Python API or CLI
- Stores notebooks to S3/GCS




```
PapermillOperator(
    task_id="create_model",
    input_nb="s3://path/to/my_model.ipynb",
    output_nb="s3://path/to/my_model_param.ipynb",

)
```

Issues with this approach

- Entire notebook executes as a single task
- Low visibility, no fault tolerance
- Code is in multiple locations
- Experimentation becomes difficult
- Repeatability becomes messy




```
PapermillOperator(
    task_id="create_model",
    input_nb="s3://path/to/my_model.ipynb",
    output_nb="s3://path/to/my_model_param.ipynb",

)
```

The Next Gen Airflow Data Science Story?

Experiment	Parameterize	Productionize
jupyter	¯_(ツ)_/¯	¯_(ツ)_/¯

The Ideal Story

- Minimal conversion from Jupyter notebook -> Airflow DAG
- Moving large datasets between different tasks should be trivial.
- Should be able to request dedicated resources for the compute job
  - GPU, RAM, CPU, etc.
- Register & deploy, and replicate the resulting models.
- Maintain orchestration and monitoring at scale.

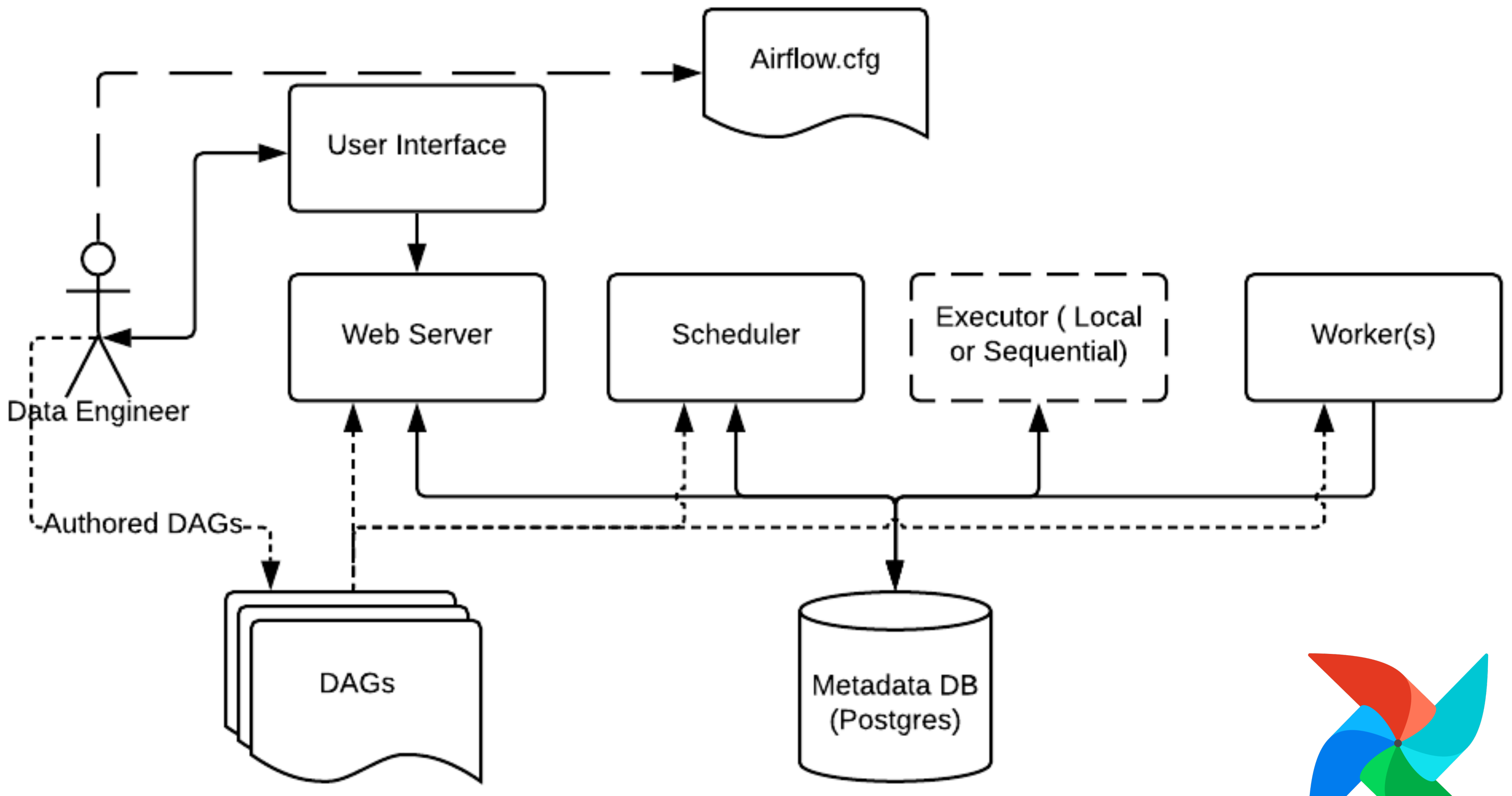
Enter Ray

“[A] distributed execution framework that makes it easy to scale your applications and to leverage state of the art machine learning libraries.

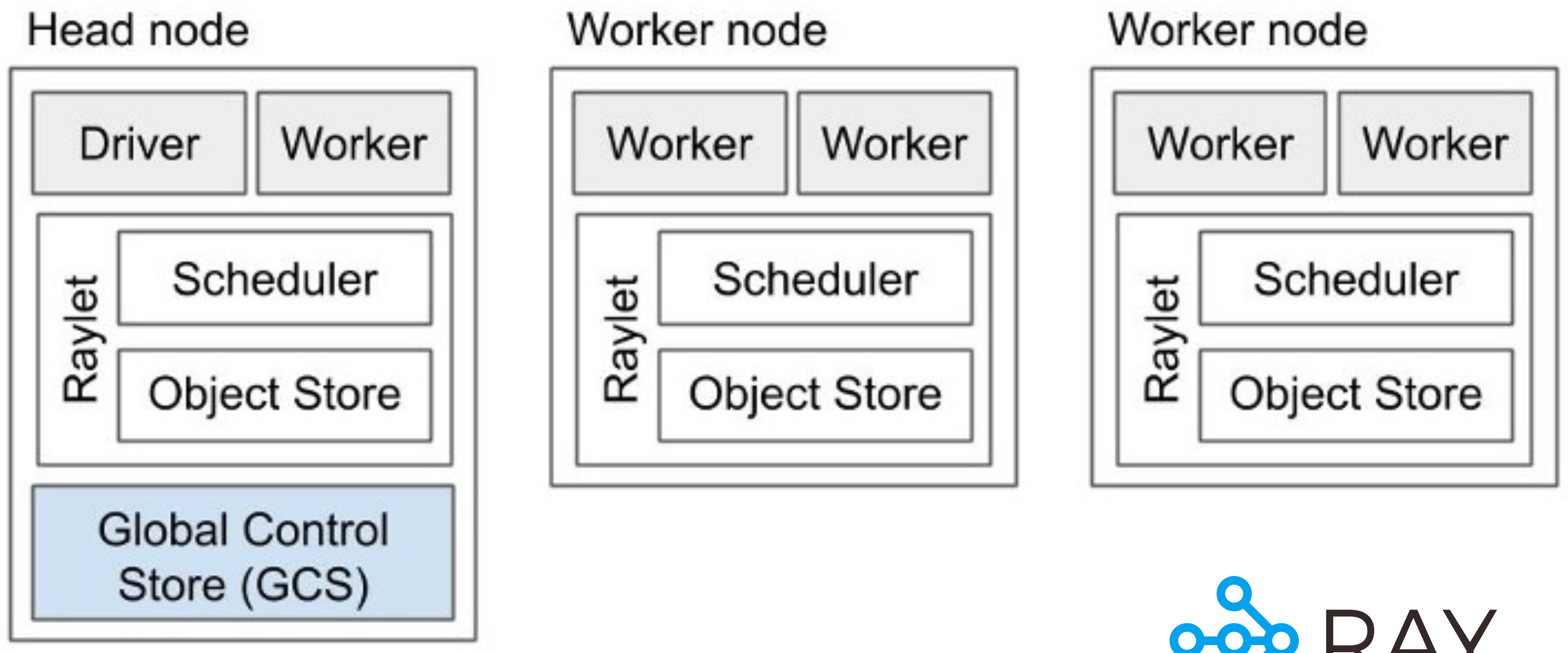
Enter Ray

- Run the same code on your local machine, on an EC2 VM, a hardware machine, etc. with no code change!
- Native Integrations with many ML projects
- Simple setup and native pythonic library
- Options for distributed computation (dask, spark, modin)
- Ray Serve for model serving

The Ideal Story



+



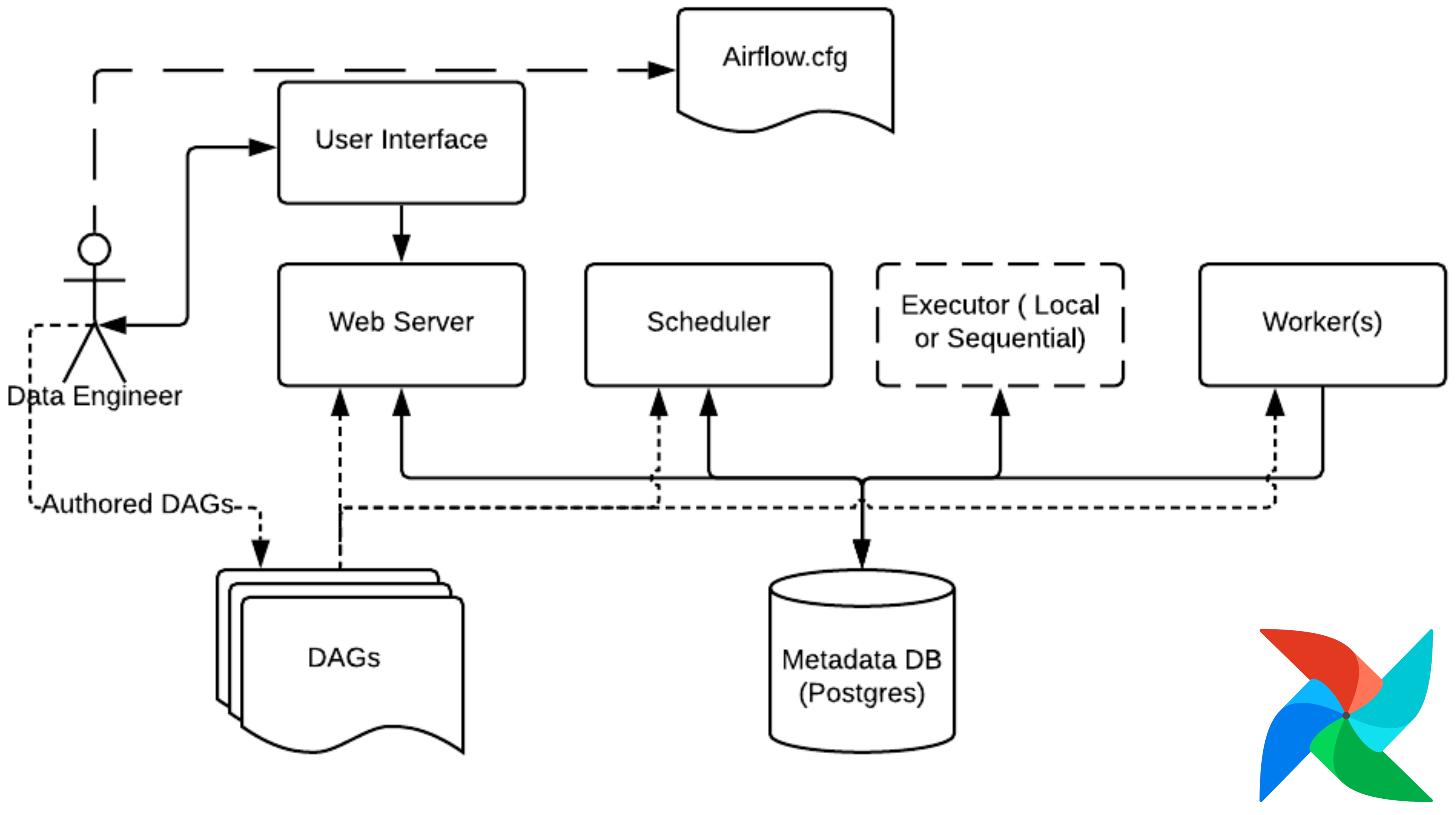
RAY

The Taskflow API

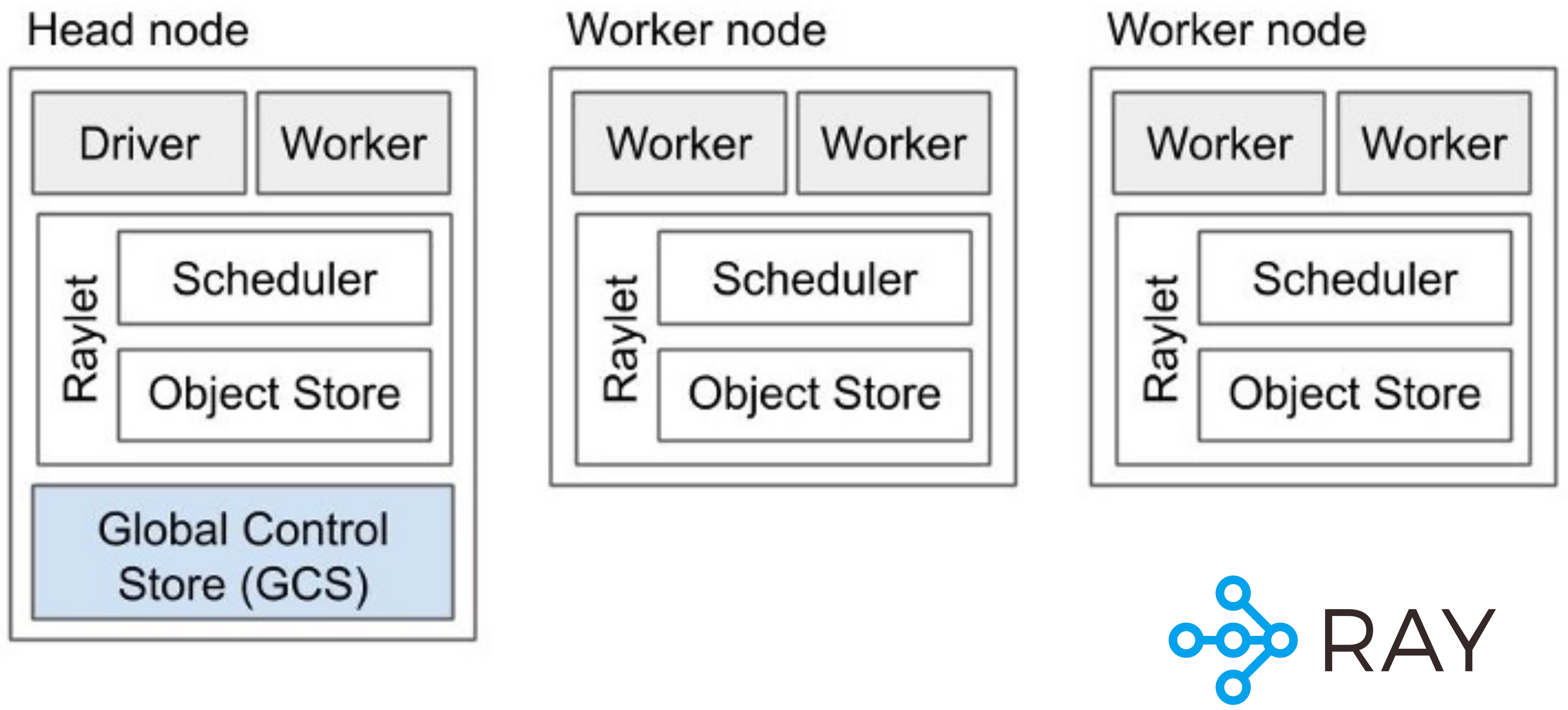
- Introduced Airflow 2.0
- Convert a python function to an Airflow task using just a single decorator!
- Pass data between tasks using functional composition
- But what if we could add the power of Ray?

```python
@task
def get_initial_number():
    return 1

@task
def add_one(value):
    return value + 1

@dag(dag_kwargs=dag_kwargs)
def dag():
    value = get_initial_number()
    for i in range(5):
        value = add_one(value)
```

The Ideal Story



+



`@task`

`@ray.remote(num_cpu=2)`

Introducing The Ray Decorator!

```python
@ray_task
def get_initial_number():
    return 1

@ray_task(num_cpu=2)
def add_one(value):
    return value + 1

@dag(dag_kwargs=dag_kwargs)
def dag():
    value = get_initial_number()
    for i in range(5):
        value = add_one(value)
```

Anyscale

+

ASTRONOMER

Introducing The Ray Decorator!

- Automatically run your airflow tasks in your ray cluster with one line of code!
- Ability to dynamically size tasks and access large ray instances
- Intermediate values automatically stored in the plasma store for ease and data locality!

```
@ray_task
def get_initial_number():
    return 1

@ray_task(num_cpu=2)
def add_one(value):
    return value + 1

@dag(dag_kwargs=dag_kwargs)
def dag():
    value = get_initial_number()
    for i in range(5):
        value = add_one(value)
```

The top-tier ML tooling of Ray with the Stability and Ecosystem of Airflow

From Notebook to Production

Develop

```
In [ ]: @ray.remote
def load_dataframe() -> "ray.ObjectRef":
    """
    build dataframe from breast cancer dataset
    """
    import modin.pandas as mpd
    url = "https://archive.ics.uci.edu/ml/machine-learning-databases/" \
     "00280/HIGGS.csv.gz"
    colnames = ["label"] + ["feature-%02d" % i for i in range(1, 29)]
    data = mpd.read_csv(url, compression='gzip', names=colnames)
    print("loaded higgs")
    return data
```

```
In [ ]: @ray.remote
def split_train_test(data):
    print("Splitting Data to Train and Test Sets")
    df_train = data[(data['feature-01'] < 0.4)]
    colnames = ["label"] + ["feature-%02d" % i for i in range(1, 29)]
    train_set = xgbr.RayDMatrix(df_train, label="label", columns=colnames)
    df_validation = data[(data['feature-01'] >= 0.4)& (data['feature-01'] < 0.8)]
    test_set = xgbr.RayDMatrix(df_validation, label="label")
    print("finished data matrix")
    return train_set, test_set
```

```
In [ ]: def train_model(
    config,
    checkpoint_dir=None,
    data_dir=None,
    data=()
):
    logfile = open("/tmp/ray/session_latest/custom.log", "w")
    def write(msg):
        logfile.write(f"{msg}\n")
        logfile.flush()

    dtrain, dvalidation = data
    evallist = [(dvalidation, 'eval')]
    # evals_result = {}
    config = {
        "tree_method": "hist",
        "eval_metric": ["logloss", "error"],
    }
    print("Start training")
    bst = xgbr.train(
        params=config,
        dtrain=dtrain,
        ray_params=RAY_PARAMS,
        num_boost_round=100,
        evals=evallist,
        callbacks=[TuneReportCheckpointCallback(filename=f"model.xgb")])
```

```
@ray.remote
def do_some_stuff():
    ...

do_some_stuff.remote()
```

```
@ray.remote
def do_some_stuff():
    ...
    return data

do_some_stuff.remote()
```

Experiment

```python
@ray_task(**task_args)
def train_model(
        data
):
    train_df, validation_df = data
    evallist = [(validation_df, 'eval')]
    evals_result = {}
    config = {
        "tree_method": "hist",
        "eval_metric": ["logloss", "error"],
    }
    bst = xgb.train(
        params=config,
        dtrain=train_df,
        evals_result=evals_result,
        ray_params=xgb.RayParams(max_actor_restarts=1, num_actors=8, cpus_per_actor=2),
        num_boost_round=100,
        evals=evallist)
    return bst
```

```python
@ray_task
def train_model():
    …

@dag(…)
def my_dag():
    train_model()
```

Parameterize

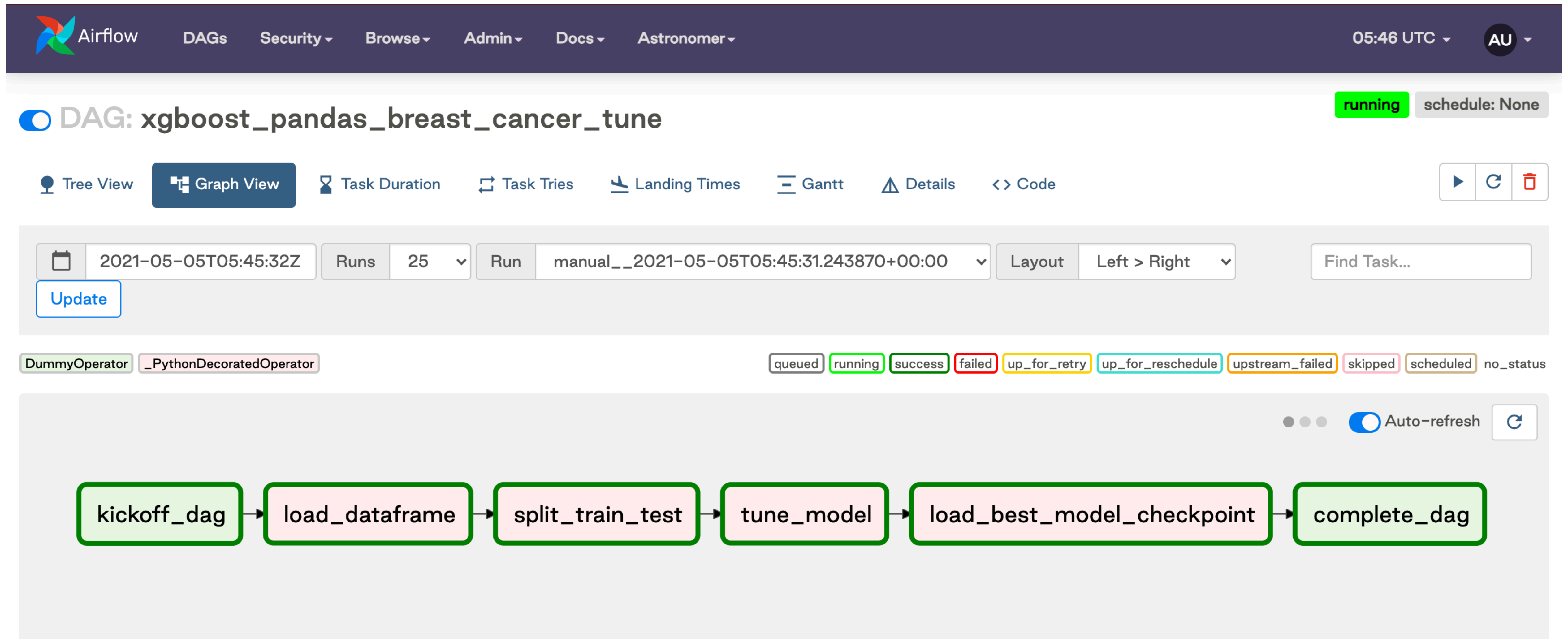
```python
@ray_task(**task_args)
def train_model(
        data
):
    train_df, validation_df = data
    evallist = [(validation_df, 'eval')]
    evals_result = {}
    config = {
        "tree_method": "hist",
        "eval_metric": ["logloss", "error"],
    }
    bst = xgb.train(
        params=config,
        dtrain=train_df,
        evals_result=evals_result,
        ray_params=xgb.RayParams(max_actor_restarts=1, num_actors=8, cpus_per_actor=2),
        num_boost_round=100,
        evals=evallist)
    return bst
```

```python
data_path = \
 "{{ conf.data_path }}"

@ray_task
def train_model(path: str):
    …

@dag(…)
def my_dag():
    train_model(data_path)
```

Productionize

Deploy your DAG!

```python
@dag(default_args=default_args, schedule_interval=None, start_date=days_ago(2), tags=['finished-modin-ex
def task_flow_xgboost_modin():
    build_raw_df = load_dataframe()
    data = create_data(build_raw_df)
    trained_model = train_model(data)

task_flow_xgboost_modin = task_flow_xgboost_modin()
```



Now it's easy to:

- **Add more tasks & parallelize**
- **Tune the model(s)**
- **Schedule fresh updates**
- **Monitor for failures**
- **(Re)Deploy the best model(s)**
- **Connect to the ecosystem**

The Next Gen Airflow Data Science Story

Develop

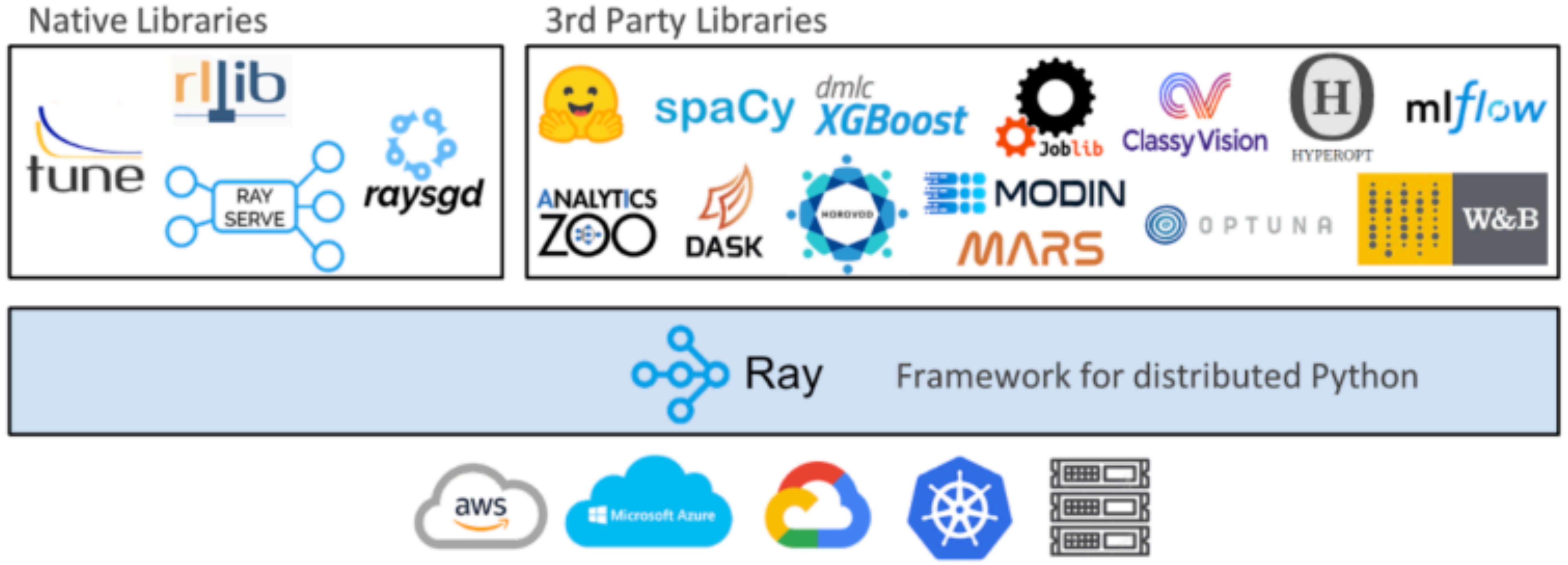
jupyter RAY

Experiment & Parameterize

RAY

Productionize

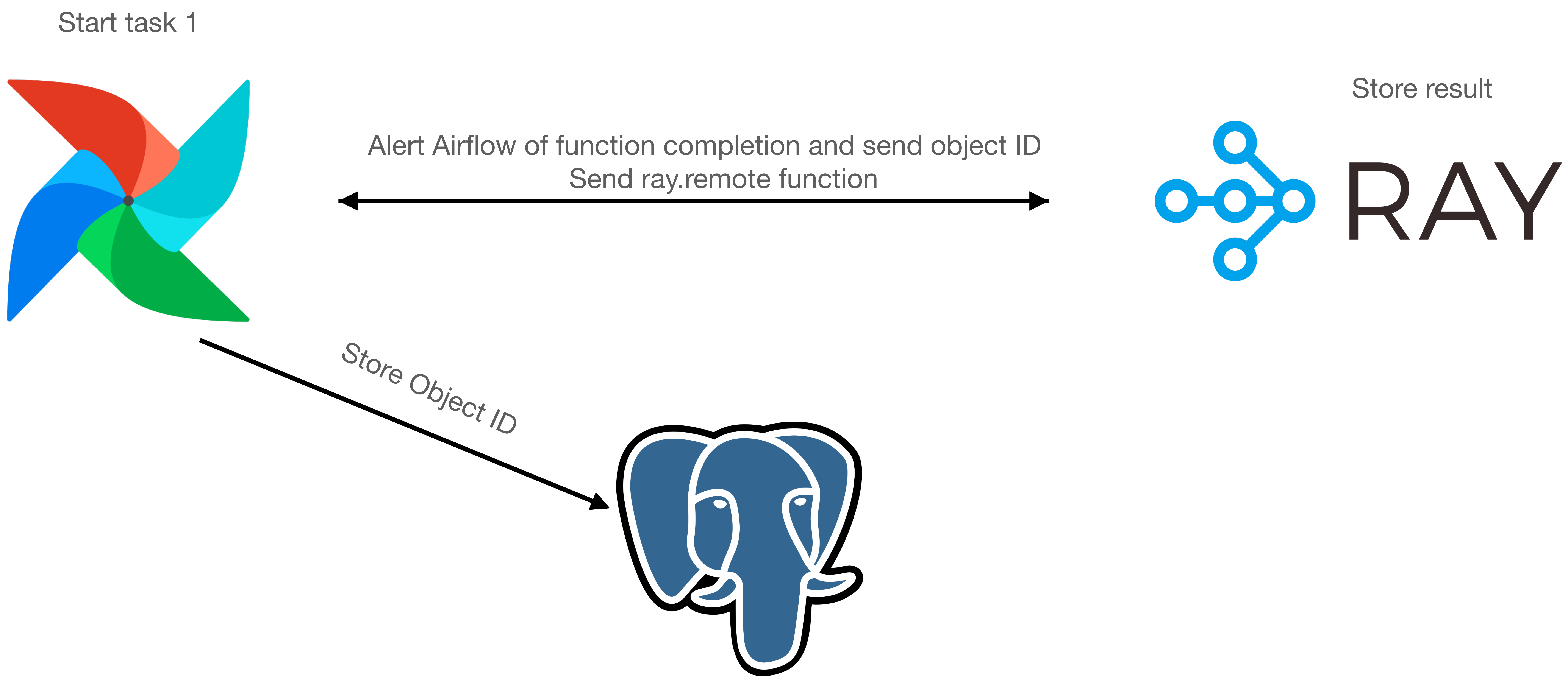
RAY



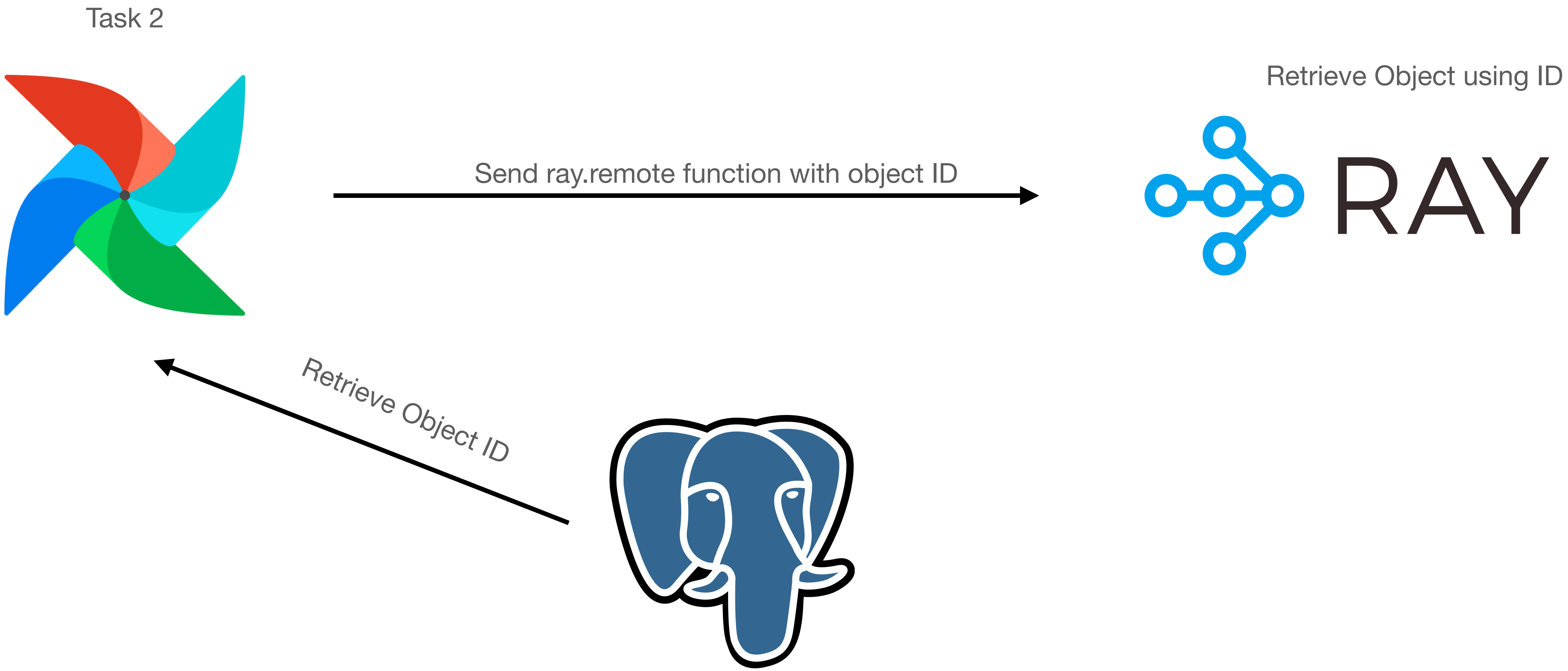
How It Works

How It Works

How It Works

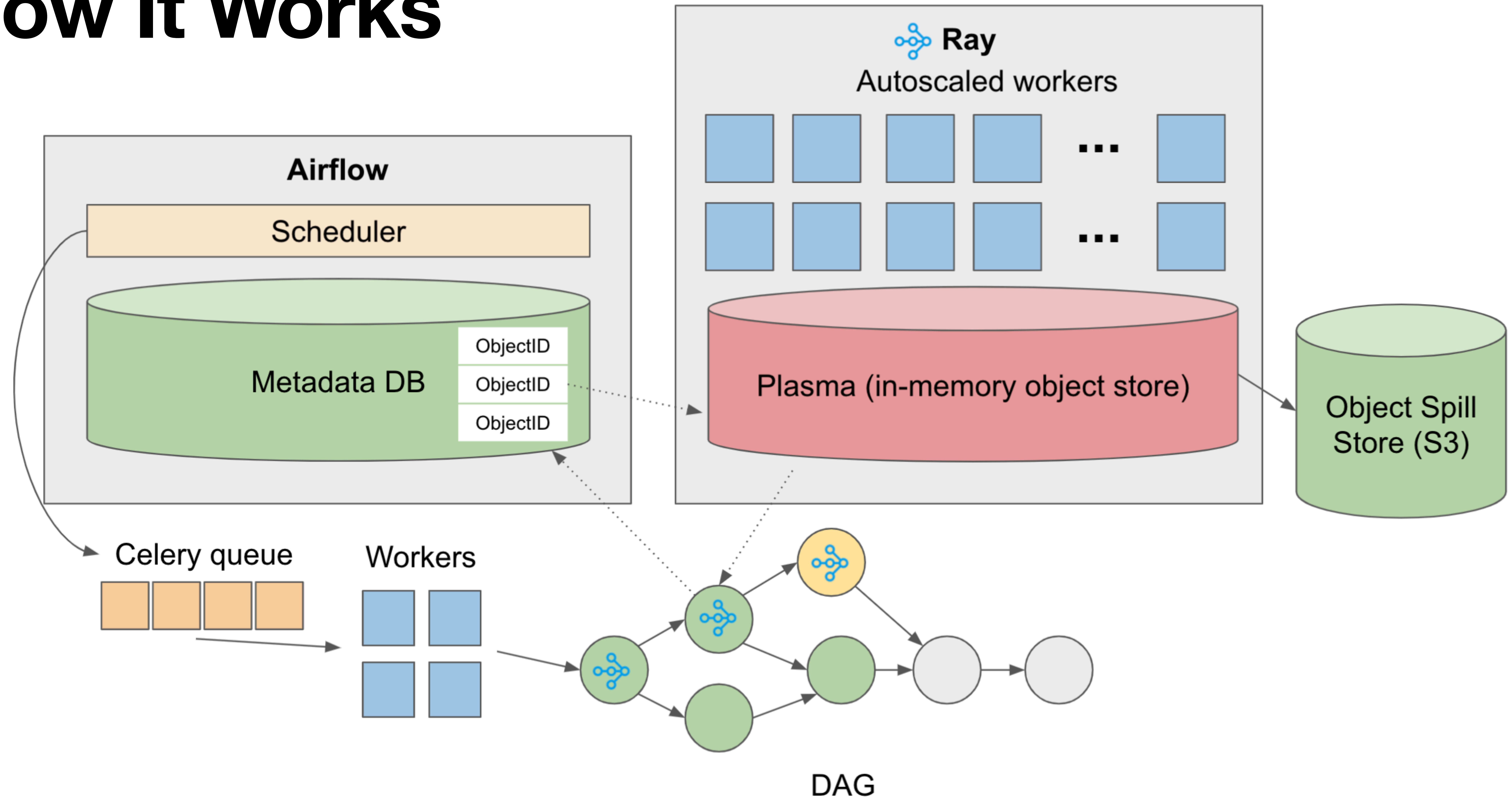

Start task 1
Store result
Alert Airflow of function completion and send object ID
Send ray.remote function
RAY
Store Object ID

How It Works



Task 2
Retrieve Object using ID
Send ray.remote function with object ID
RAY
Retrieve Object ID

How It Works


Ray
Autoscaled workers
...
...
Airflow
Scheduler
Metadata DB
ObjectID
ObjectID
ObjectID
Plasma (in-memory object store)
Object Spill Store (S3)
Celery queue
Workers
DAG

Next Steps

Checkpointing

- Store intermediate data in external data stores
- Re-run failed tasks
- Plug Tune checkpoints to model registries and experiment tracking libs
- Tweak Experiments so even if your ray cluster crashes, you will be able to restart DAG from checkpoint

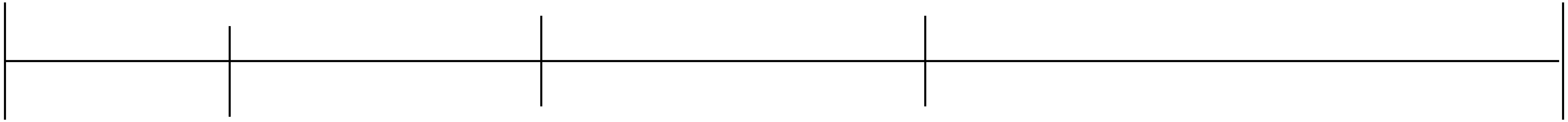
```
@ray_task(checkpoint=True)
def really_long_model():
    …

@ray_serve_task
def serve_model():
    …

@dag(dag_kwargs=dag_kwargs)
def dag():
    model = really_long_model()
    serve_model(model)
```

Ray serve decorator

- Deploy models to your ray cluster via airflow DAGs for instance prediction endpoints
- Composed Models = Multiple models based on business logic
- Parallelize multi-model training with Airflow

```
@ray_task
def create_model_1():
    …
@ray_task
def create_model_2():
    …

@ray_serve_task
def serve_model():
    ComposedModel.deploy()

@dag(dag_kwargs=dag_kwargs)
def dag():
    model = create_model()
    serve_model(model)
```

Road Map



May 2021

- Alpha released
- Modin Support
- Custom sizes

July 2021

- Beta released
- Checkpointing
- Anyscale connector

Q3 2021

- GA released
- Fully tested and supported

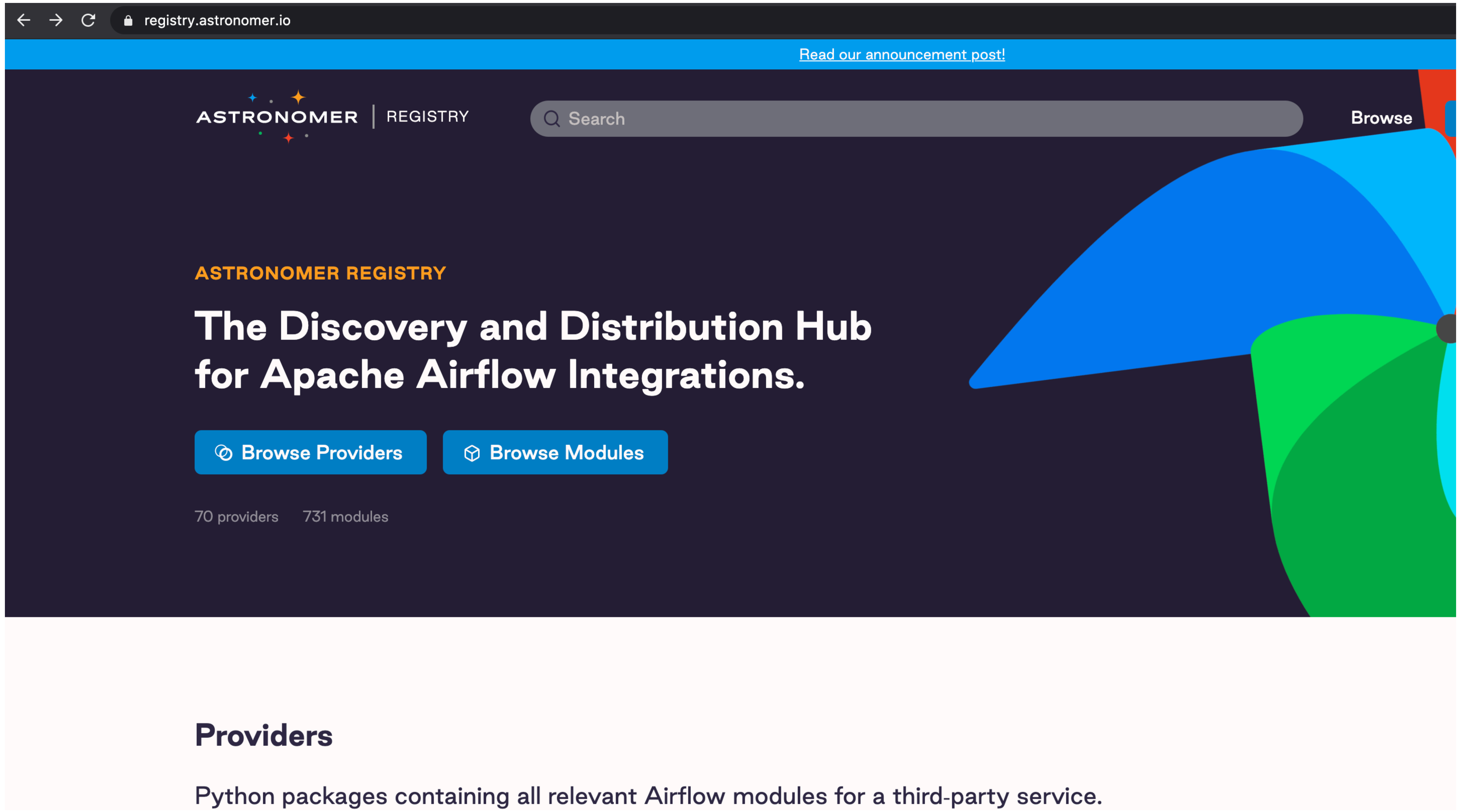
Q4 2021

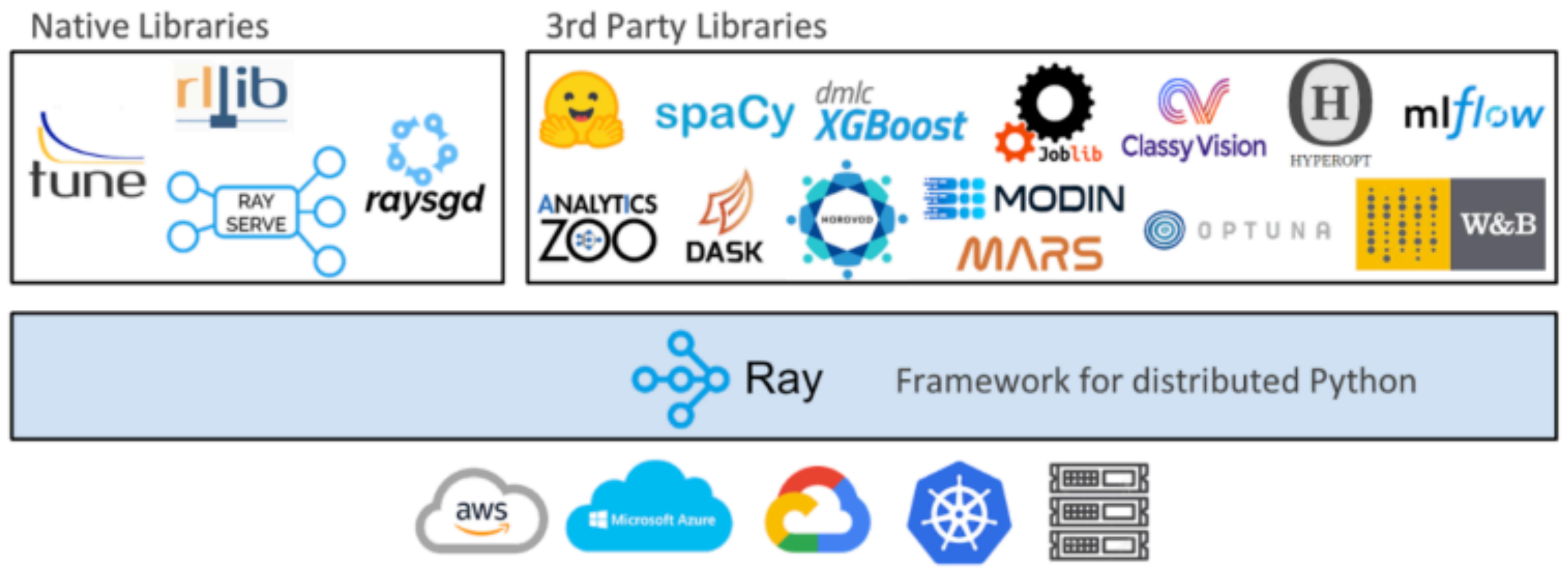
- GA Running with Anyscale cloud

How to Get the Ray Provider

How to Get the Ray Provider

Head to https://registry.astronomer.io/





Anyscale

How to Get the Ray Provider

```
pip install airflow-provider-ray
```

Thank You

ASTRONOMER

@danimberman
@ApacheAirflow

astronomer.io
@astronomerio

Special thanks to:

- Richard Liaw
- Will Drevo
- Charles Greer
- Pete DeJoy
- Rob Deeb
- Plinio Guzman

Anyscale