Getting Started with Ray Workflows

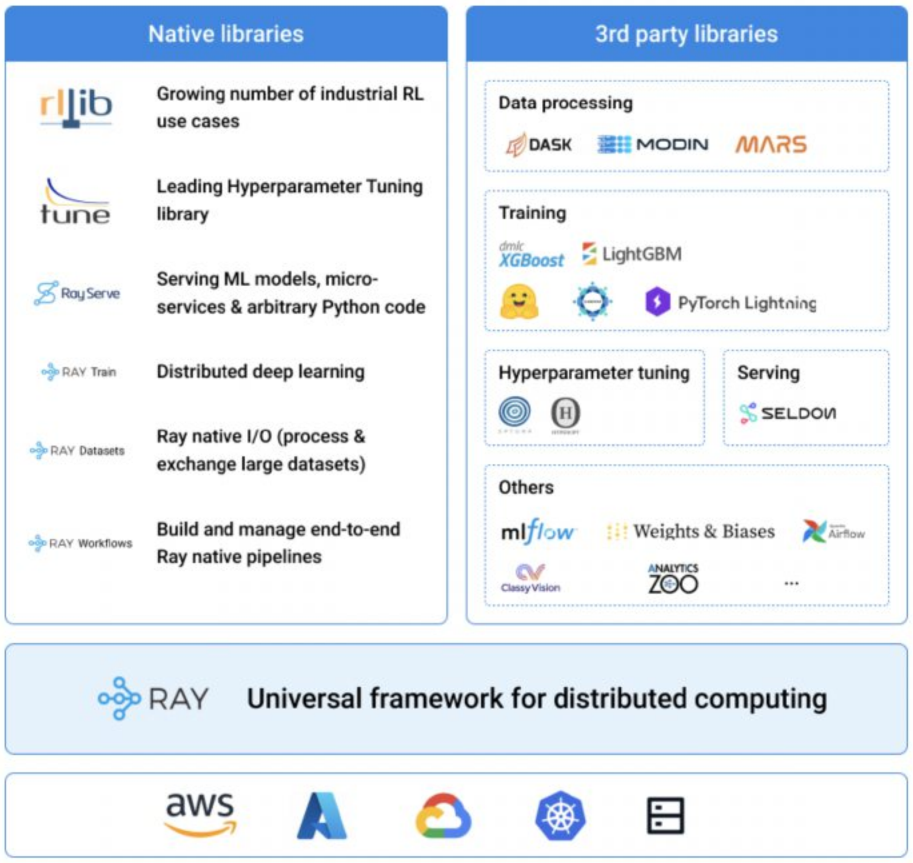
Machine learning projects require significant infrastructure both before and after the core model itself. The open source library Ray has everything needed to go from model experimentation to large-scale training, and ultimately production serving.

Ray provides developer-friendly infrastructure; just bring your own model in PyTorch, TensorFlow, Huggingface, XGBoost, Scikit-learn and many more. The developer experience is central to Ray's mission and reason for existing. Ray is built for modern ML engineers to go work from a laptop, and train on a remote cluster, be that AWS, GCP, or your own servers like we have at Illinois NCSA.

This post focuses on one piece of Ray: Workflows. Workflows is a solution for robust data pipelines and any composable steps in your workflow.

Workflows is comparable to Spotify's Luigi and Apache Airflow, but offers a better developer experience:

- Python-native syntax with excellent defaults and the use of function decorators like `@workflow.step`
- Low-code, more practical to use than heavy-weight Airflow.
- Includes the key features ML engineers need.
- Integration with the rest of Ray for distributed training, excellent hyperparameter tuning and serving in production.


Native libraries
3rd party libraries
rllib
Growing number of industrial RL use cases
tune
Leading Hyperparameter Tuning library
Ray Serve
Serving ML models, micro-services & arbitrary Python code
RAY Train
Distributed deep learning
RAY Datasets
Ray native I/O (process & exchange large datasets)
RAY Workflows
Build and manage end-to-end Ray native pipelines
Data processing
DASK
MODIN
MARS
Training
XGBoost
LightGBM
PyTorch Lightning
Hyperparameter tuning
Serving
SELDON
Others
mlflow
Weights & Biases
Airflow
Classy Vision
ANALYTICS ZOO
...
RAY
Universal framework for distributed computing
aws

Ray's modules and well-supported compatibility, from the Ray Docs.

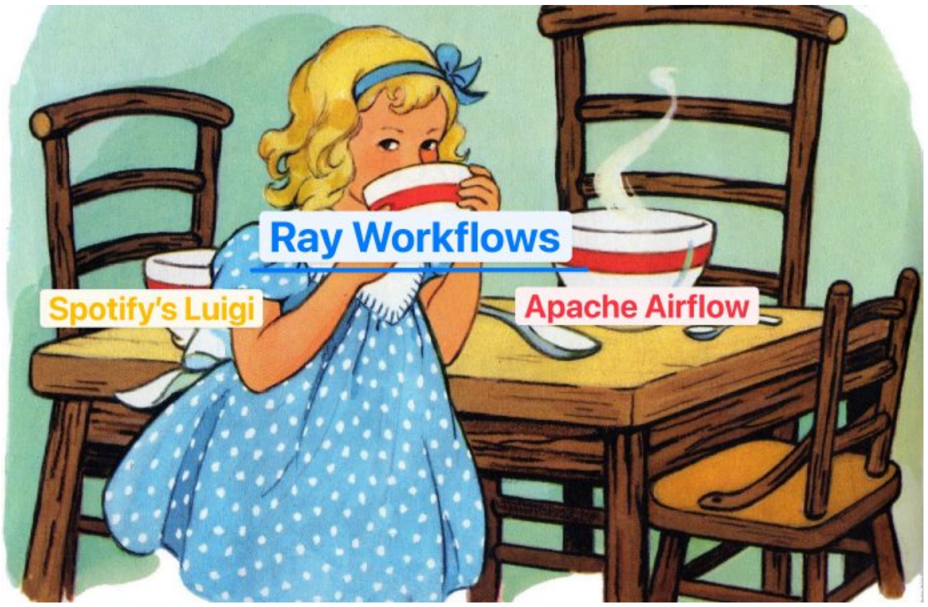
What about existing workflow tools?

For Python developers, I enjoy this analogy. Luigi is like Flask; it's simple, barebones but maintainable. And Airflow is like Django; it's heavyweight, requires configuration, and has more weird syntax. Twitter ML engineers talk about how Airflow adds significant development overhead to ML workflows and they only use it when finalizing production code. Since my teams have even less developer time than Twitter, that makes me nervous to choose Airflow.

Workflows feels as simple to use as Luigi or Flask, uses a modern Pythonic syntax, and still has many of the key features I want: error handling, automatic retires, automatic parallel execution, and durable storage of state. Workflows strikes a developer-friendly balance.

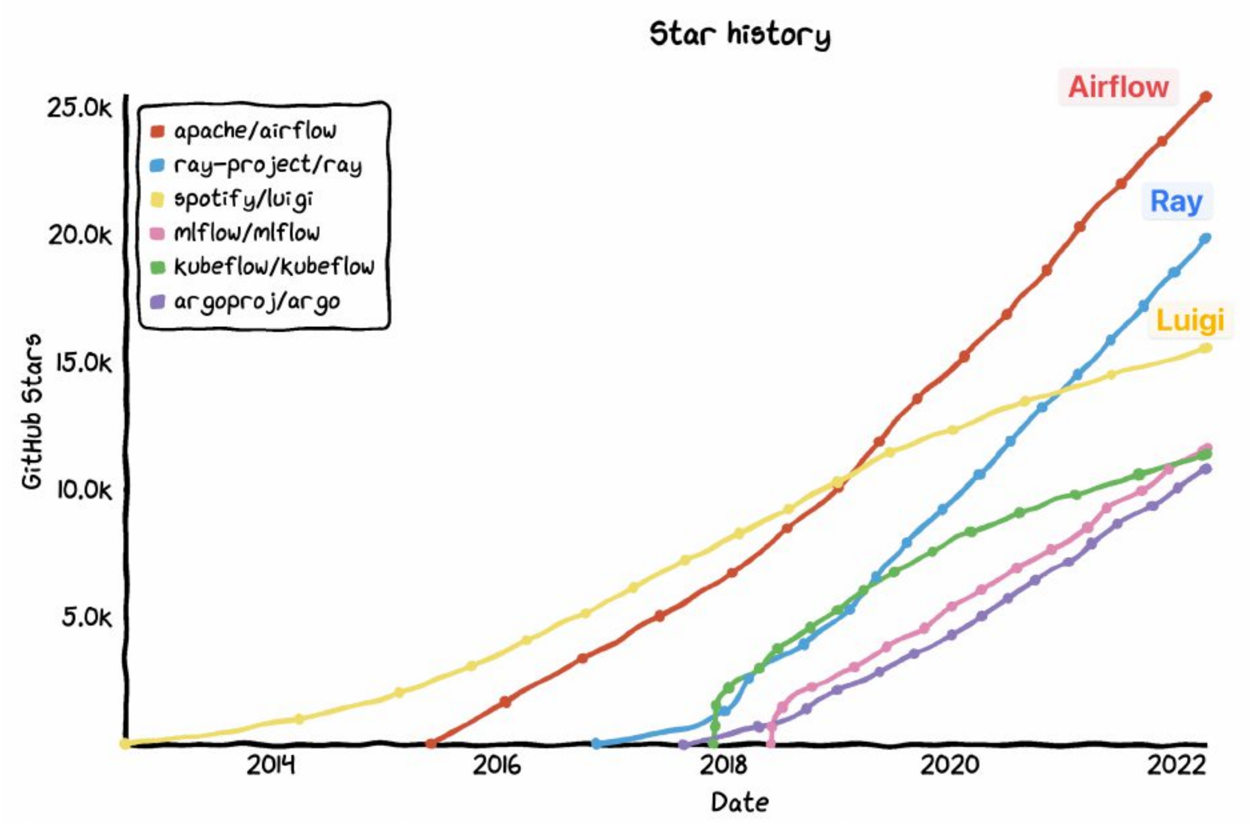
Workflows has a 1st party integration with Airflow and will support your existing Airflow DAGs, as shown in this excellent tutorial.

For an in-depth comparison of workflow tools, see this excellent post.



Workflows strikes the Goldiloks sweet spot between developer experience and important features.

Finally, Ray's open ecosystem has seen surging popularity relative to its competitors.



Ray's explosive popularity, as shown against similar projects.

Interested?

- Look here for Ray install instructions, including Apple Silicon support via Conda.
- Follow this posts' accompanying Colab notebook for a worked example.

Documentation:

- Workflows Getting Started docs, or continue reading here for a friendly intro.
- Workflows API Reference with examples.
- View integrations here, like support for Dask, Horovod, HuggingFace, Scikit Learn, PyTorch Lightning and many more.

Let's go over a simple Ray example and I'll demo my favorite features.

Easy to use

To turn your existing code into a Workflow, simply add the decorator:

```
# add the Workflow decorator
@workflow.step
def load_data():
  return csv
```

Call functions like normal, just add `.step()`

```
# normal Python
return_value = load_data(params)

# Ray Workflow
return_value = load_data.step(params)
```

Run the workflow:

```
from ray import workflow

@workflow.step
def load_data():
  return csv

workflow.init()
return_value = load_data.step(params)
return_value.run()
# ✅ return_value is now populated with the CSV


# - `run()`, returns the value
# - `run_async()`, returns a ObjectRef -- use ray.get(return_val)
```

🎉 Congrats, you've made your code more resilient. Read on to see how to leverage that resiliency.

📒 **Important note for iPython and Jupyter Notebooks:** to eliminate conflicts in Jupyter's async rendering engine, add these lines to the top of any Jupyter Notebook using Workflows.

```
# For compatibility with Jupyter's async rendering
from ray import workflows
import nest_asyncio
nest_asyncio.apply()
```

Automatic retries

By default, each step of a workflow will be retried 3 times, if an error occurs.

You can set this parameter via the Function decorator.

```
@workflow.step(max_retries=5) # <int>
def load_data():
  return csv
```

Catch exceptions

Workflows can prevent your whole pipeline from crashing by catching the exceptions of individual steps.

If catch_exceptions is True, the return value of the function will be converted to `Tuple[Optional[T], Optional[Exception]]`. This can be combined with `max_retries` to try a given number of times before returning the result tuple. This makes it easier to attempt to connect to unreliable services and evaluate the response.

```
@workflow.step(catch_exceptions=True)
def load_data():
  return csv
```

Recovering workflows, thanks to durable storage

If your Workflow fails, the result of each step is stored on disk and can be easily resumed from where it crashed. For example, sometimes data is available in time, but now it's in the right place. Simply resume your Workflow.

```
# Resume all resumable workflows. This won't include failed workflow.
print(workflow.resume_all())
```

Now is a good time to mention **where** the state is stored. By default it's stored in `/tmp/ray/workflow_data`.

```
# specify Workflow's durable storage location
workflow.init(storage="./workflow_storage")
```

Naming Workflows

It’s best to name your Workflows, and each requires a **unique name.** Human-readable dates makes digging through logs more productive. Feel free to use this function.

```
def make_workflow_id(name: str) -> str:
  '''
  🎯 Best practice to ensure unique Workflow names.
  '''
  import pytz
  from datetime import datetime

  # Timezones: US/{Pacific, Mountain, Central, Eastern}
  # All timezones `pytz.all_timezones`. Always use caution with timezones.
  curr_time = datetime.now(pytz.timezone('US/Central'))
  return f"{name}-{str(curr_time.strftime('%h_%d,%Y@%H:%M'))}"

make_workflow_id('import_imagenet')
# 'import_imagenet-Mar_30,2022@10:37' (military time)
```

And as simple as that, you have a multi-step workflow.

```
# init
workflow.init(storage="./workflow_storage")

# define steps
raw_data = load_data.step(<params>)              # 1️⃣ load
clean_data = feature_engineer.step(raw_data)     # 2️⃣ clean

# run
workflow_id = make_workflow_id('import_imagenet')
clean_data.run(workflow_id)                      # 👈 Custom Workflow name
```

Now that your workflows are named, you can easily view and manipulate them in Python.

Viewing workflows (save these Python commands for later)

Since workflows are durably saved to disk, you can retrieve the workflow’s output at any time.

```
# Later, get the output of a workflow from disk storage
result = workflow.get_output(<workflowID>)
```

Or see all the workflows that have run:

```
# List all workflows
print(workflow.list_all())
# [("workflow_id_1", "RUNNING"), ("workflow_id_2", "CANCELED")]
```

And remember these commands for viewing your workflows.

```
status = workflow.get_status(workflow_id="workflow_id")
# assert status in {
#    "RUNNING", "RESUMABLE", "FAILED",
#    "CANCELED", "SUCCESSFUL"}

# View all metadata
workflow.get_metadata(<workflow_id>)

# Resume a workflow.
print(workflow.resume(workflow_id="workflow_id")))
# return is an ObjectRef which is the result of this workflow

# Resume all resumable workflows. This won't include failed workflow
print(workflow.resume_all())

# Cancel a workflow.
workflow.cancel(workflow_id="workflow_id")

# Delete the workflow.
workflow.delete(workflow_id="workflow_id")
```

View the full metadata documentation here.

Async execution

The power of shared memory

Data is passed between Workflow steps using the Ray Object Store. This store is intelligently managed, allowing large objects to be efficiently shared between many tasks. Importantly, the memory share is invisible to the Workflow steps, and developers can behave as if the variables are local Python objects.

The power of Workflows comes from using Ray as the backbone.

1. Easily go distributed: use Ray's distributed libraries within Workflow steps.
   a. The Ray object store passes distributed datasets between steps with **zero-copy overhead.**
   b. Ray intelligently scales your jobs using the powerful and dev-friendly Autoscaler (YouTube).
   c. Deploy on your own HPC cluster or push to any cloud backend (AWS/GCP/Anyscale/etc.)

Objects in memory are `ObjectRef`, and the beauty of the dev experience is you only need `ray.put()` and `ray.get()`. If you use Ray Core's distributed functions, simply add the `@ray.remote` function decorator.

View progress on the Ray Dashboard

The fully-featured Ray web dashboard defaults to port 8265. If you're ssh-ing into a remote cluster, us port forwarding. For more details on ssh port forwarding, see this guide.

```
ssh -L 8265:localhost:8265 <you_username>@<remote_server>
```

Learn more

Start with the excellent Ray getting-started documentation, and work through the docs for creating Ray Remote Tasks.

This example demonstrates the benefits of Ray for massively parallel jobs. These examples work through the many advanced Ray capabilities and use cases.

For more support, join our Clowder community slack!