ML Orchestration

Robert Clements

MSDS Program

University of San Francisco

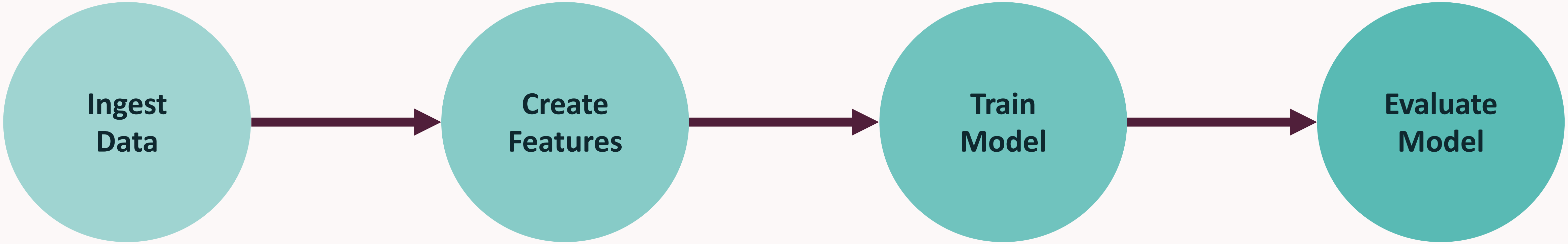
What to Expect

- Goal: to learn about creating reproducible and scalable ML workflows.

- How: we will study ML workflows and practice ML Orchestration with Metaflow and GCP and Kubernetes.

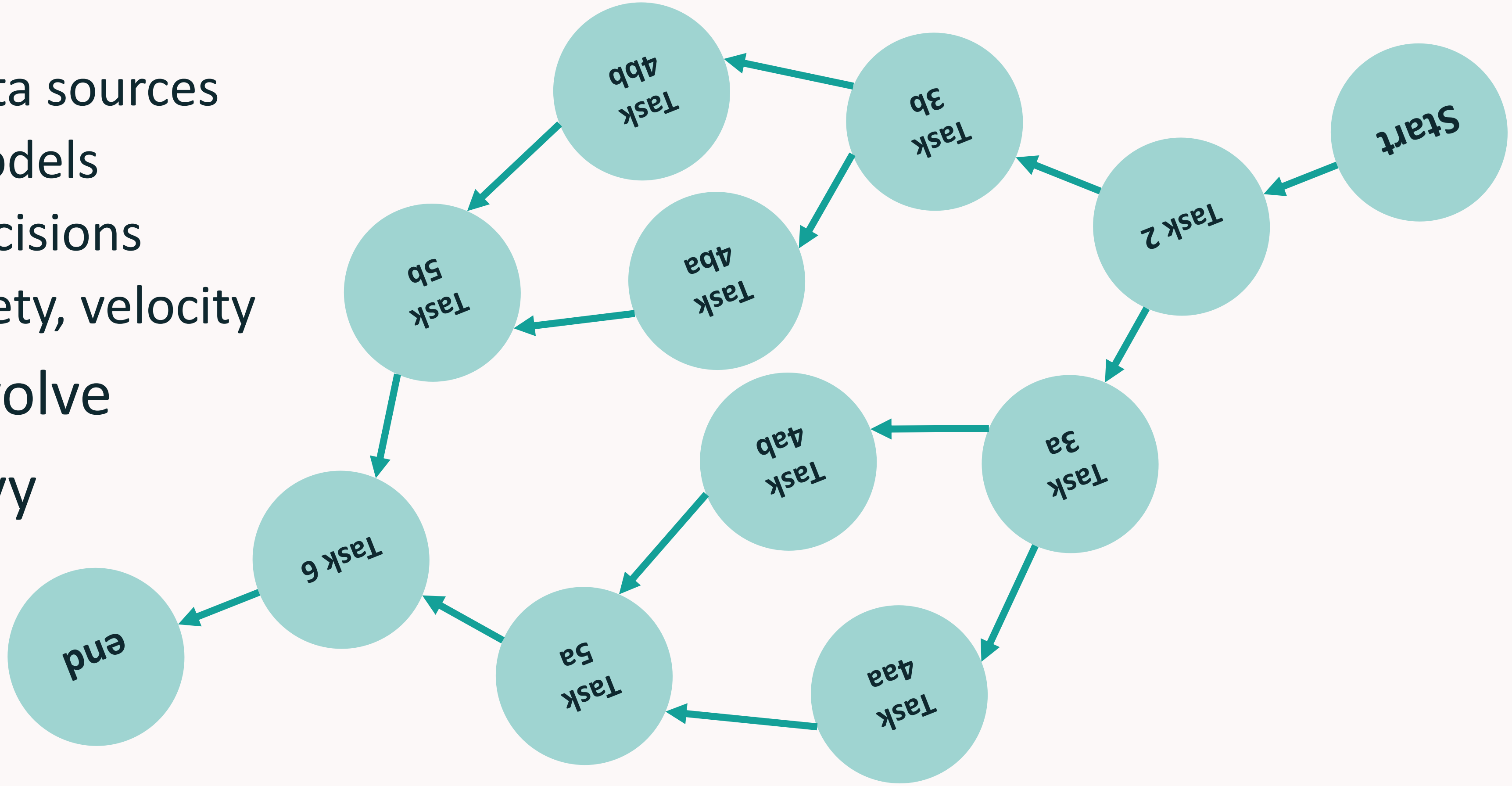
ML Workflows

ML Workflows

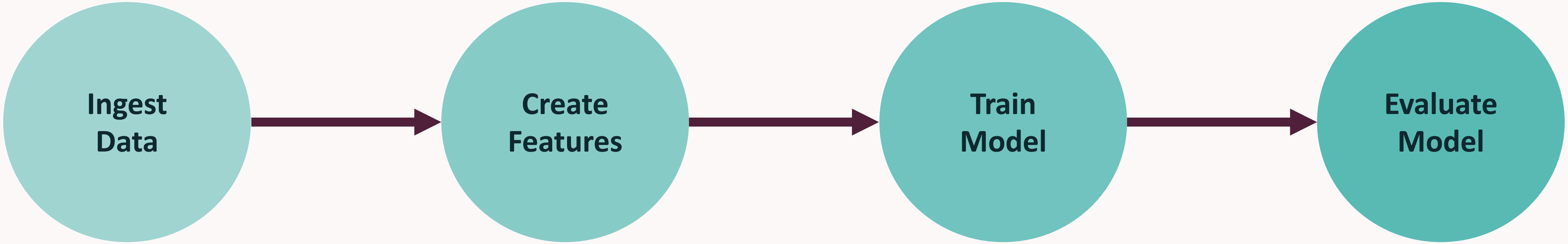


Why do we need ML workflow orchestration?

- ML systems are very complex with many interconnected complex parts:
  - Variety of data sources
  - Variety of models
  - Variety of decisions
  - Volume, variety, velocity
- ML systems evolve
- Resource-heavy



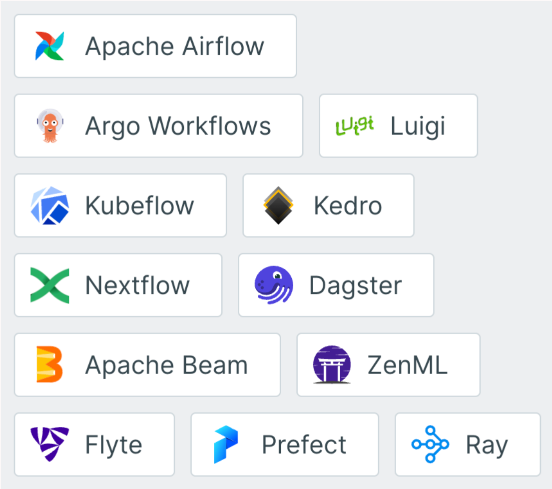
ML Workflows



Versioning: save versions of every run of the flow
Metadata: save the metadata of every run of the flow
Artifacts: save artifacts of every run of the flow
Compute: scale vertically and horizontally
Orchestration: run steps in order, parallel
Deployment: schedule/trigger the flow to run

ML Workflows with Metaflow

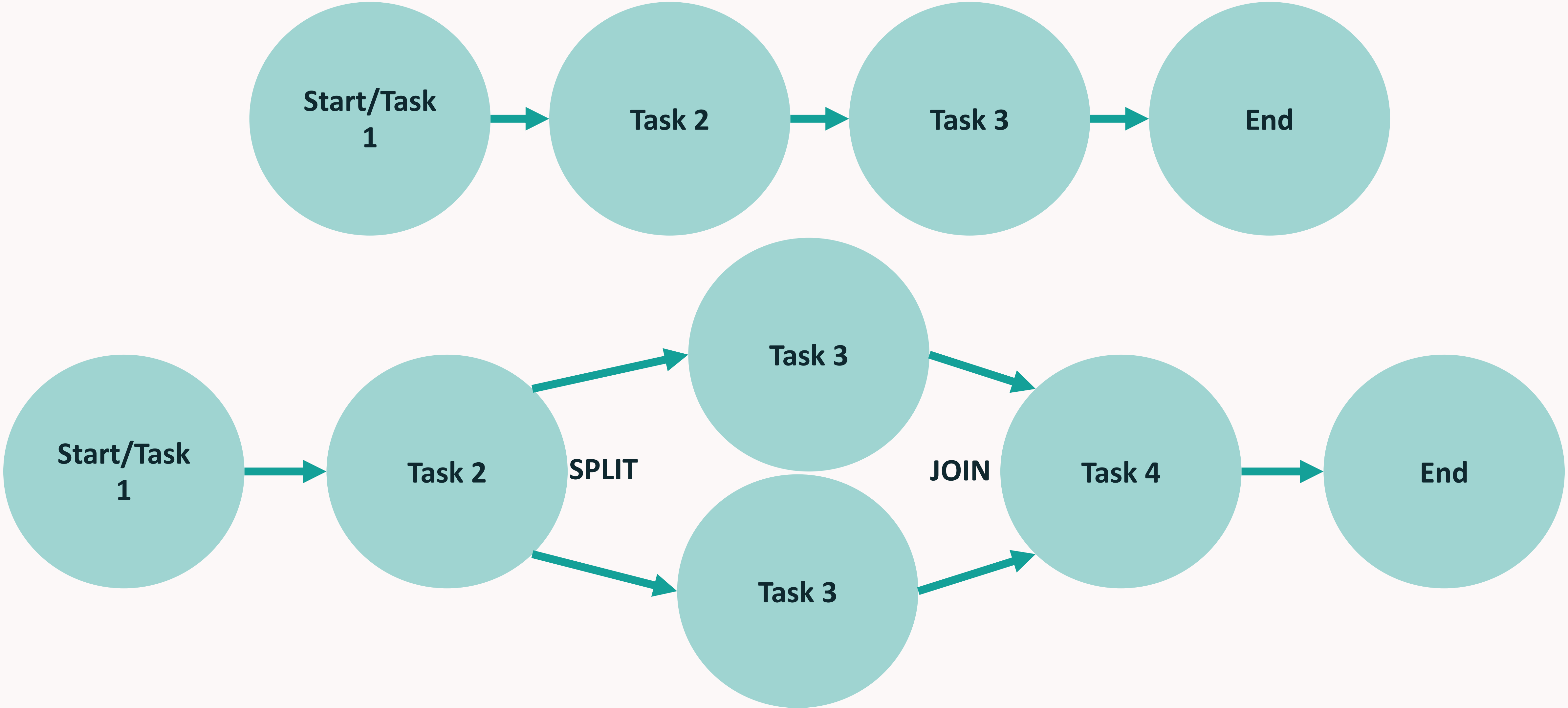
- Metaflow began at Netflix in 2017 by Ville Tuulos
- Book - Effective Data Science Infrastructure
- Open-sourced soon thereafter
- DAG-based tool for creating scalable workflows, but focused on ML
- Runs on
  - Local machine – low-maintenance
  - AWS Batch – low-maintenance prototyping stack
  - AWS Batch + Step Functions – low-maintenance full stack
  - AWS/GCP/Azure K8s – customizable full stack
- Popular alternatives:
  - Airflow
  - Flyte
  - Kubeflow



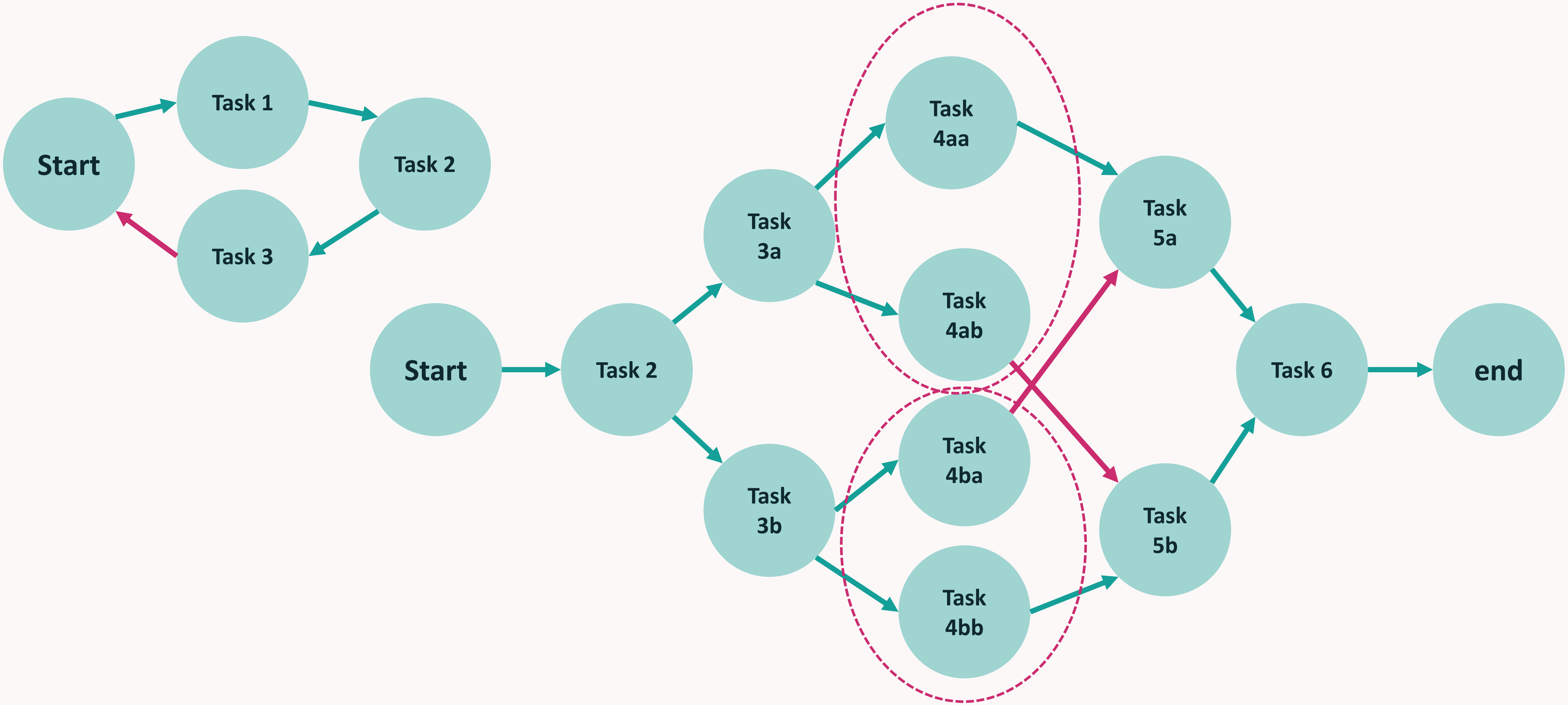
Why Metaflow?

- Designed for ML and for Data Scientists to use
- *Easy* to scale vertically and horizontally
- Reproducible and shareable workflows
- Covers the full stack:
  - Data
  - Compute
  - Orchestration
  - Versioning
  - Deployment
  - Modeling

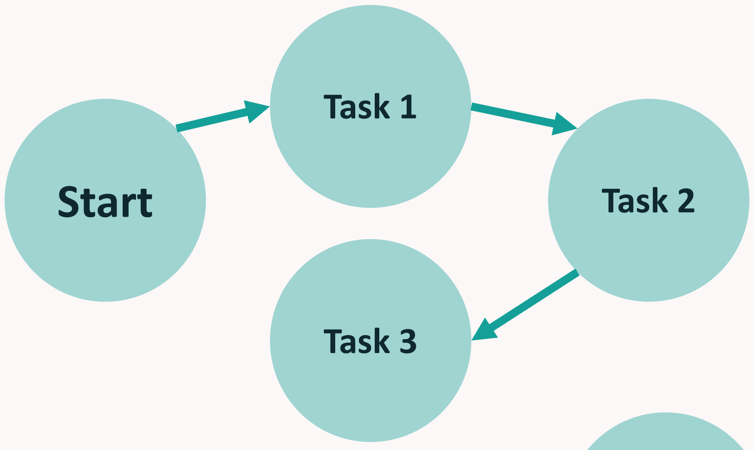
ML Workflows – Valid DAGs

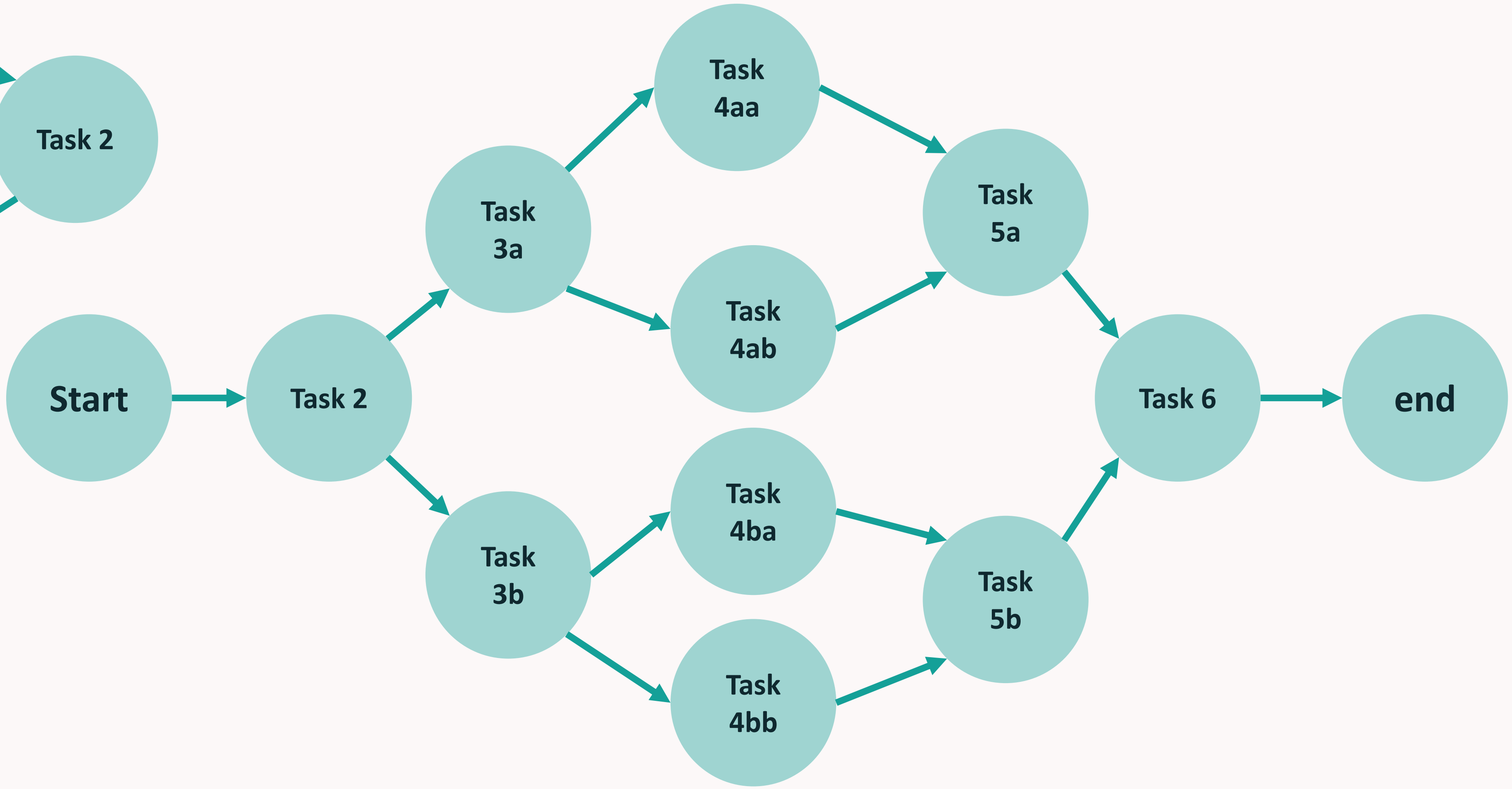


ML Workflows - Invalid DAGs

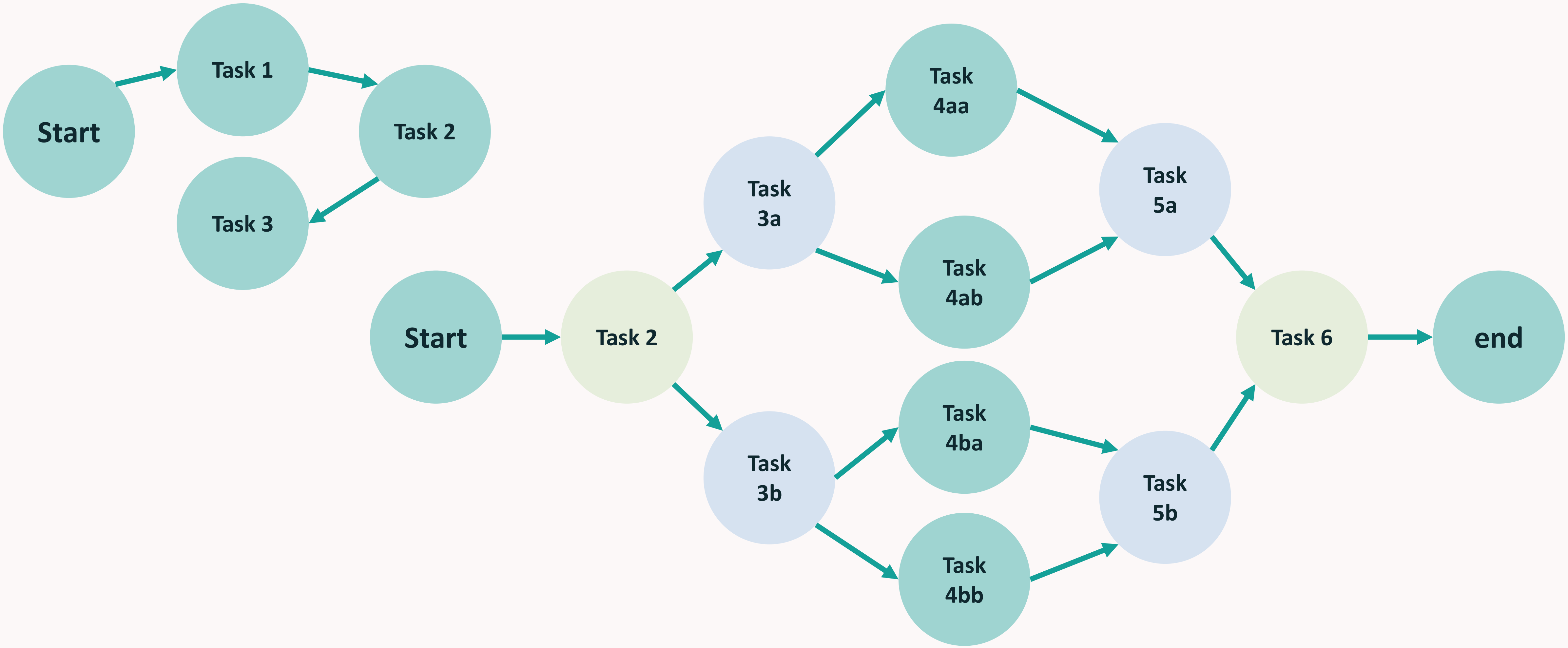


ML Workflows – Fixing Invalid DAGs

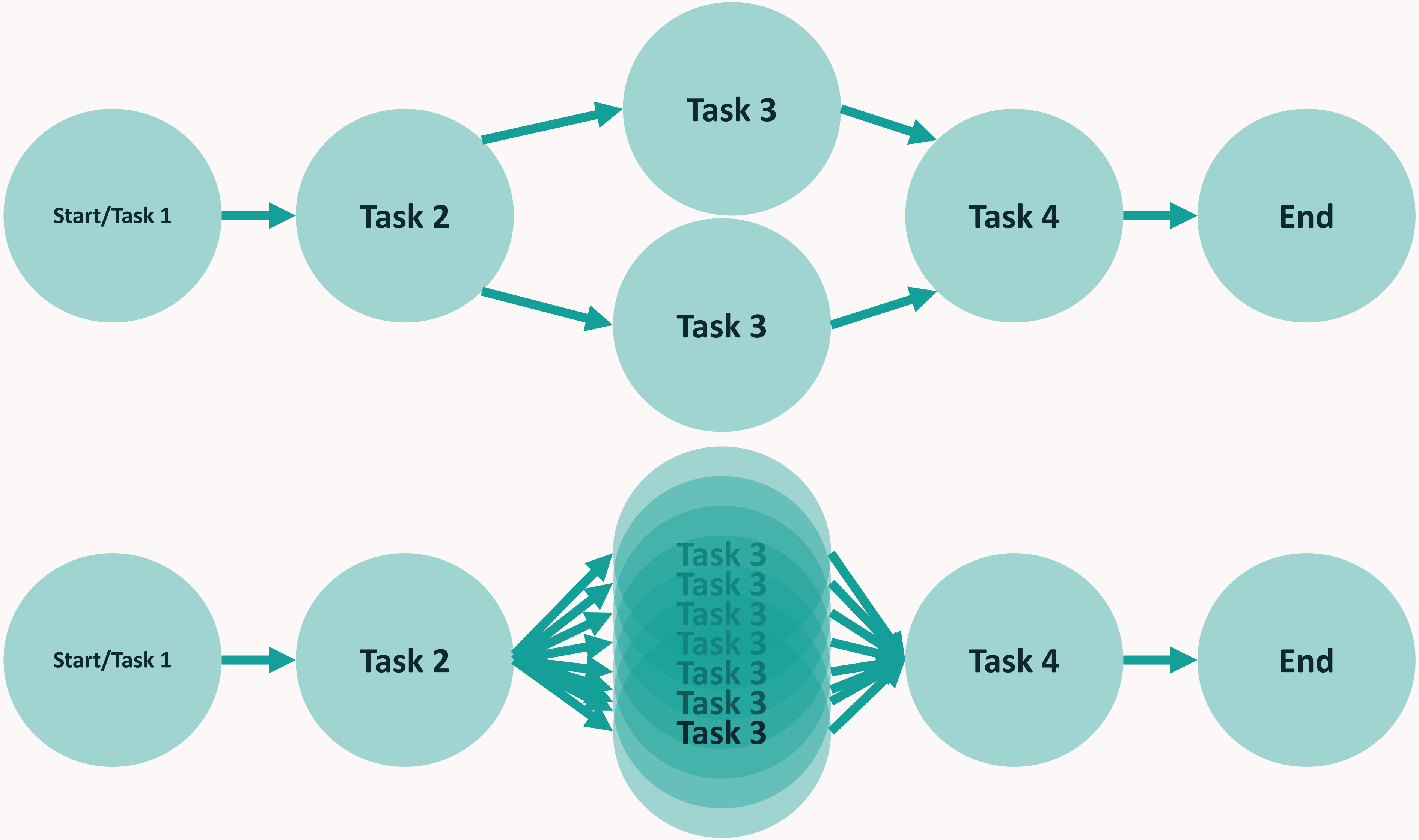




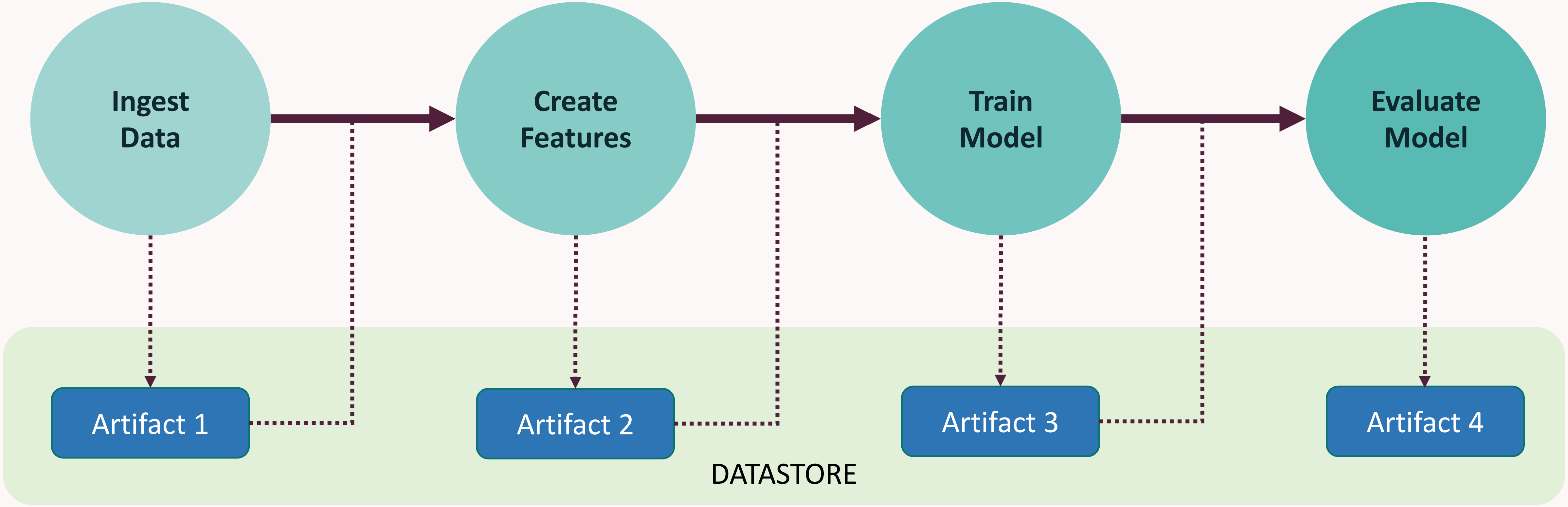
ML Workflows – Fixing Invalid DAGs



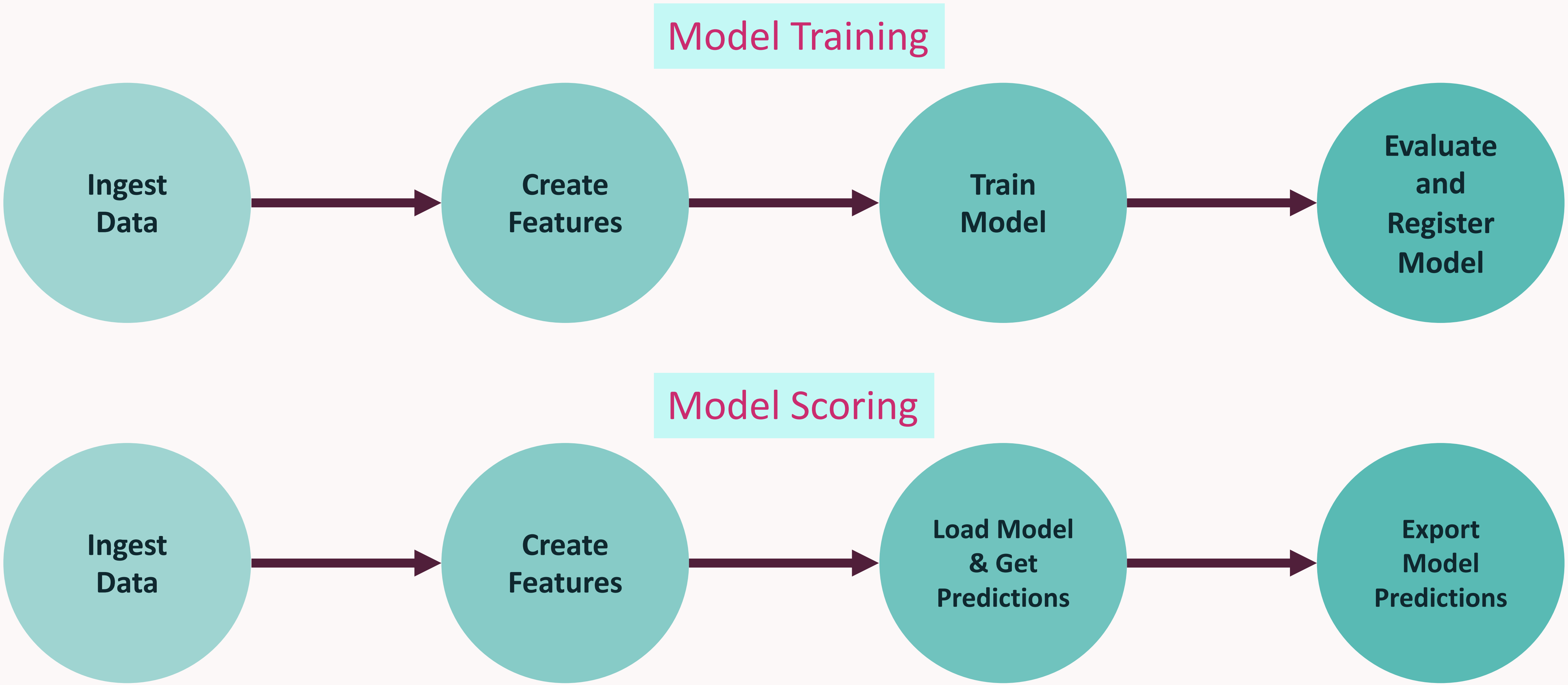
Static and Dynamic DAGs



State Persistence



Two Important Flows



ML Pipeline Tools

- Metaflow
- Prefect
- Airflow
- Dagster
- ZenML
- Mage
- Kubeflow
- So many others…

How do you define the flow?

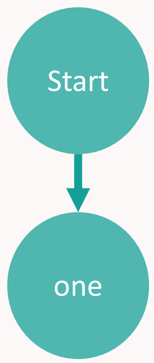
Where and how do you run the flow?

ML Workflows in Metaflow

```
from metaflow import FlowSpec, step

class HelloWorldFlow(FlowSpec):
     @step
     def start(self): # start of flow
          …
          self.next(self.one)
```

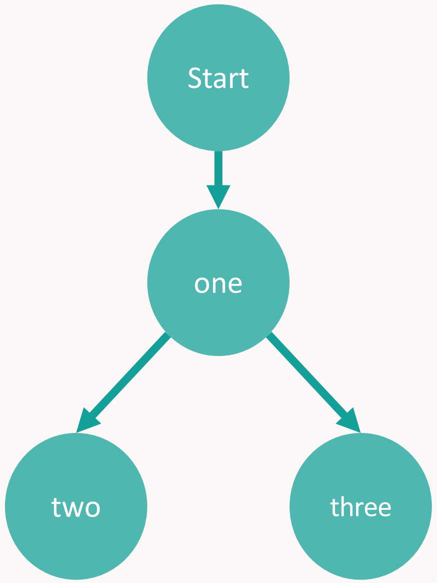
ML Workflows in Metaflow

```
from metaflow import FlowSpec, step

class HelloWorldFlow(FlowSpec):
     @step
     def start(self): # start of flow
          …
          self.next(self.one)

     @step
     def one(self):
          …
          self.next(self.two, self.three) # branching
```



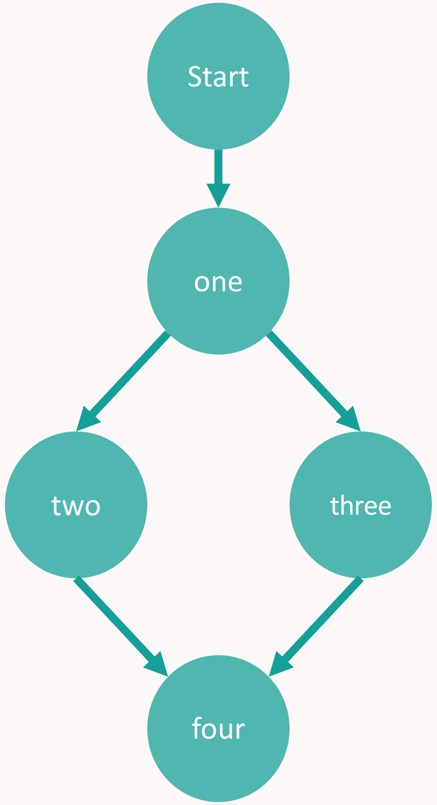
ML Workflows in Metaflow

```
from metaflow import FlowSpec, step

class HelloWorldFlow(FlowSpec):
     @step
     def start(self): # start of flow
          …
          self.next(self.one)

     @step
     def one(self):
          …
          self.next(self.two, self.three) # branching

     @step
     def two(self):
          …
          self.next(self.four)

     @step
     def three(self):
          …
          self.next(self.four)
```



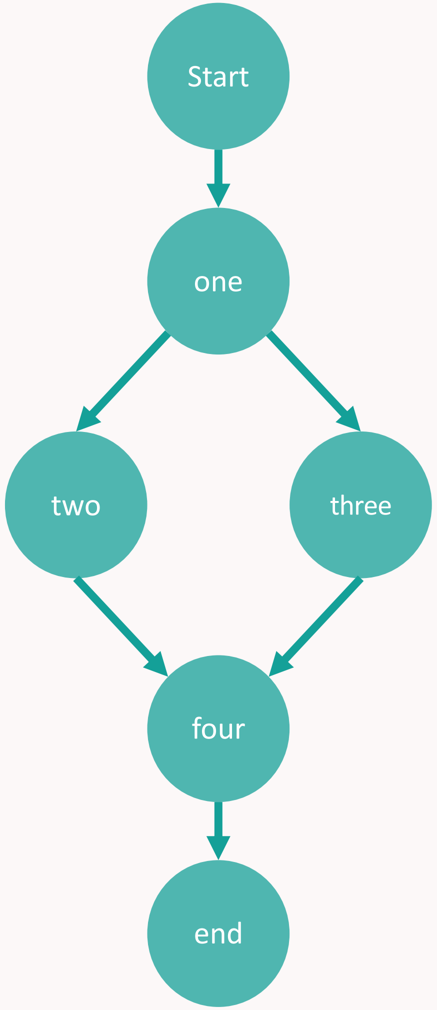
ML Workflows in Metaflow

```
from metaflow import FlowSpec, step

class HelloWorldFlow(FlowSpec):
     @step
     def start(self): # start of flow
          …
          self.next(self.one)

     @step
     def one(self):
          …
          self.next(self.two, self.three) # branching

     @step
     def two(self):
          …
          self.next(self.four)

     @step
     def three(self):
          …
          self.next(self.four)

     @step
     def four(self, inputs): # join step
          …
          self.next(self.end)
```



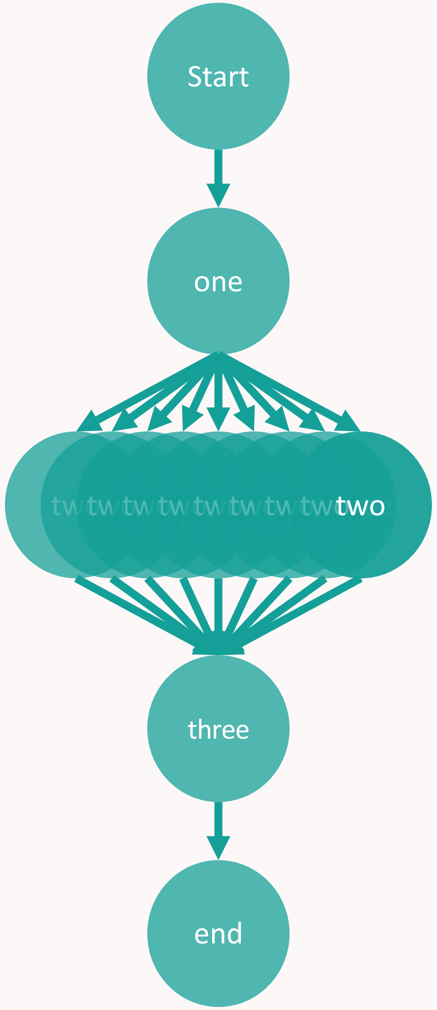
ML Workflows in Metaflow

```
from metaflow import FlowSpec, step

class HelloWorldFlow(FlowSpec):
     @step
     def start(self): # start of flow
          …
          self.next(self.one)

     @step
     def one(self):
          …
          self.next(self.two, self.three) # branching

     @step
     def two(self):
          …
          self.next(self.four)

     @step
     def three(self):
          …
          self.next(self.four)

     @step
     def four(self, inputs): # join step
          …
          self.next(self.end)

     @step
     def end(self):  # end of flow
          …
```

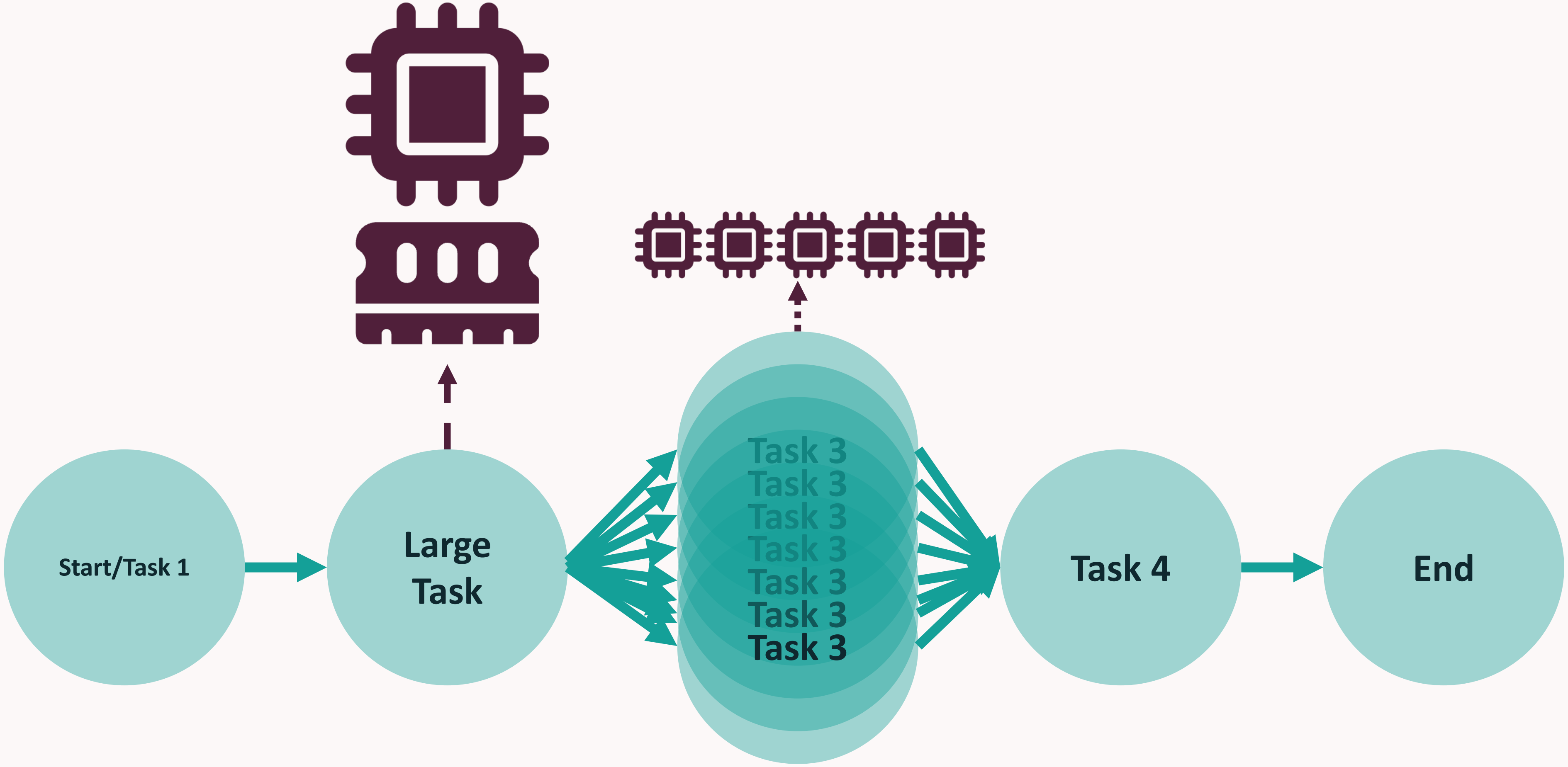


ML Workflows in Metaflow (dynamic)

```
from metaflow import FlowSpec, step

class HelloWorldFlow(FlowSpec):
     @step
     def start(self): # start of flow
          …
          self.next(self.one)

     @step
     def one(self):
          iters = [list of things to iterate over]
          …
          self.next(self.two, foreach='iters') # branching

     @step
     def two(self):
          # do something with self.input
          …
          self.next(self.three)


     @step
     def three(self, inputs): # join step
          …
          self.next(self.end)

     @step
     def end(self):  # end of flow
          …
```



Scaling Vertically and Horizontally



Scaling Vertically Using @resources

```
@resources(memory=60000, cpu=1)
@step
def start(self):
     import numpy
     import time
     big_matrix =
numpy.random.ranf((80000, 80000))
     t = time.time()
     self.sum = numpy.sum(big_matrix)
     self.took = time.time() - t
     self.next(self.end)
```

- Set memory, cpu, gpu, or shared_memory
- Does ***not*** inherently scale up, must be paired with a scalable compute layer, e.g. AWS Batch or K8s

Scaling Horizontally Using @kubernetes

```
$ python BigSum.py run --with kubernetes
```

```
@kubernetes(memory=60000, cpu=1)
@step
def start(self):
     import numpy
     import time
     big_matrix =
numpy.random.ranf((80000, 80000))
     t = time.time()
     self.sum = numpy.sum(big_matrix)
     self.took = time.time() - t
     self.next(self.end)
```

- Running on command line forces **all** steps to run in K8s

OR

- Replace `@resources` with `@kubernetes` (or `@batch` if using AWS) to run specific steps on specific compute layers
- Note: `@kubernetes` has additional arguments available

Being Careful When Using @kubernetes

```
$ python BigSum.py run --with kubernetes --max-num-splits 100
```

- To avoid using too many resources with parallel jobs, use either
  - `--max-num-splits N`
  - `--max-workers N`

Retry Steps Using @retry

```
$ python BigSum.py run --with retry
```

```
@retry
@step
def start(self):
      import numpy
      import time
      big_matrix =
numpy.random.ranf((80000, 80000))
      t = time.time()
      self.sum = numpy.sum(big_matrix)
      self.took = time.time() - t
      self.next(self.end)
```

- For transient platform issues, use `@retry`
- Recommended when using remote compute layer
- To avoid retries for specific steps, use `@retry(times=0)`
- Can set number of retries and `minutes_between_retries`

Catch Exceptions Using @catch

```
@step
def start(self):
     self.params = range(3)
     self.next(self.compute, foreach='params’)

@catch(var='compute_failed')
@step
def compute(self):
     self.div = self.input
     self.x = 5 / self.div
     self.next(self.join)

@step
def join(self, inputs):
     for input in inputs:
          if input.compute_failed:
               print('compute failed for
parameter: %d' % input.div)
     self.next(self.end)
```

- Use `@catch` to catch exceptions that are not transient compute layer issues
- Code has to be rewritten to know how to handle exceptions
- `var` is optional

Timeout Using @timeout

```
@timeout(seconds=5)
@step
def start(self):
     import numpy
     import time
     big_matrix =
numpy.random.ranf((80000, 80000))
     t = time.time()
     self.sum = numpy.sum(big_matrix)
     self.took = time.time() - t
     self.next(self.end)
```

- Use the `@timeout` decorator to avoid stuck code

Accessing Data in Metaflow

Metaflow has the metaflow.S3 module for accessing S3 data, but when it comes to data, just follow best practices:

- Tip: keep data ***loading*** and data ***transformations*** separate
- Whenever possible, use Metaflow artifacts
  - Anything assigned to self will be persisted to subsequent steps
- Try to avoid importing *local* files
- Use larger instances for larger datasets
- Use parquet + Apache Arrow or numpy (not pandas) when possible

Manage Dependencies with @conda

```
python LinearFlow.py --environment=conda run
```

Step-level

```
@conda(libraries={"pandas": "0.22.0"})
def fit_model(self):
...
```

- Create conda environments for each step

Flow-level

```
@conda_base(libraries={'numpy':'1.15.4'}, python=’3.6.5’)
class LinearFlow(FlowSpec):
...
```

- Create conda environment for entire flow
- Can be combined with step-level environments

Metaflow ML Flows Demo

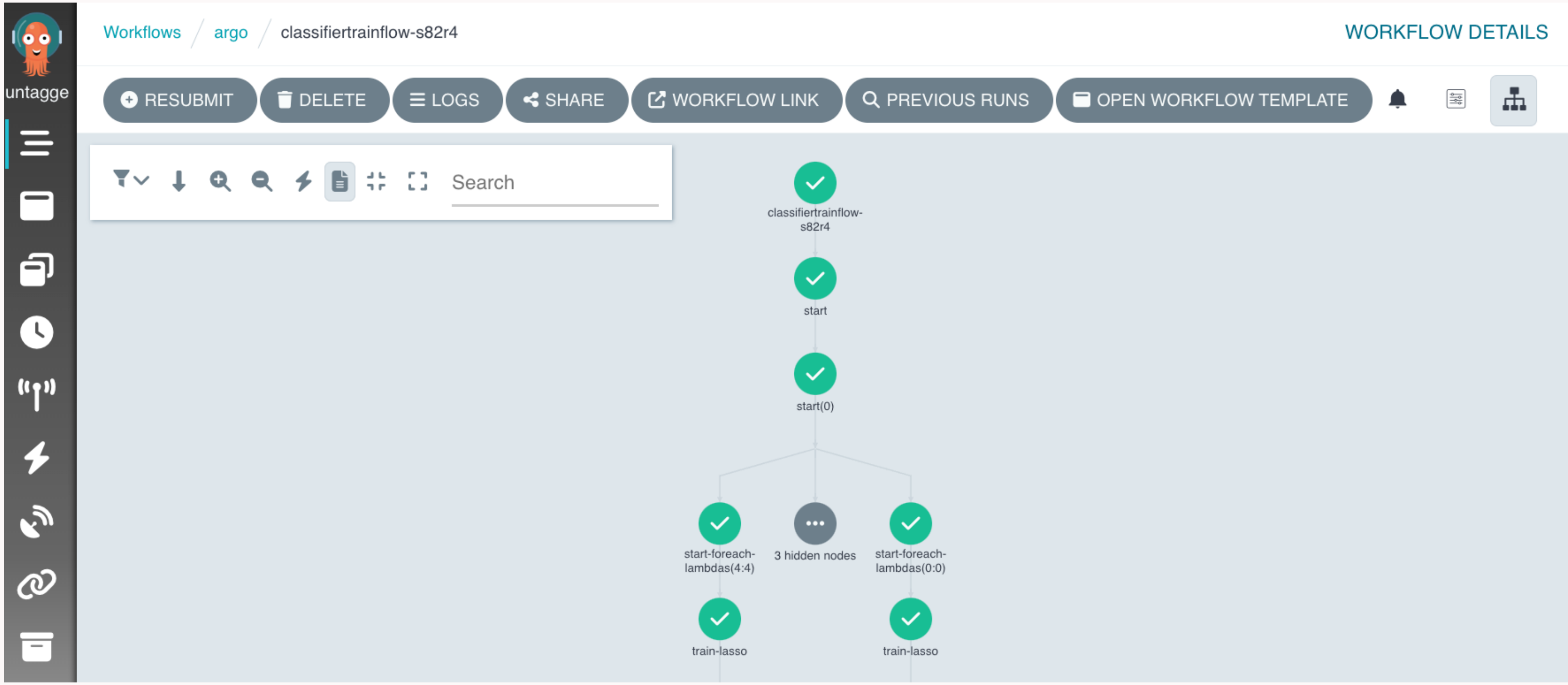
Metaflow ML Flows Lab

Metaflow in Production

- Use @resources and @kubernetes to scale
- Use @conda or @conda_base for packaging up the environment
- Use @timeout, @retry, and @catch to deal with problems
- You will need to:
    - Create a project in Google cloud
    - Install Gcloud CLI
    - Install Terraform
    - Install kubectl
    - Install Kubernetes, google-cloud-storage, google-auth python libraries
    - Patience and luck

Scheduling Jobs with Argo Workflows

```
python LinearFlow.py --environment=conda --with retry argo-workflows create
```



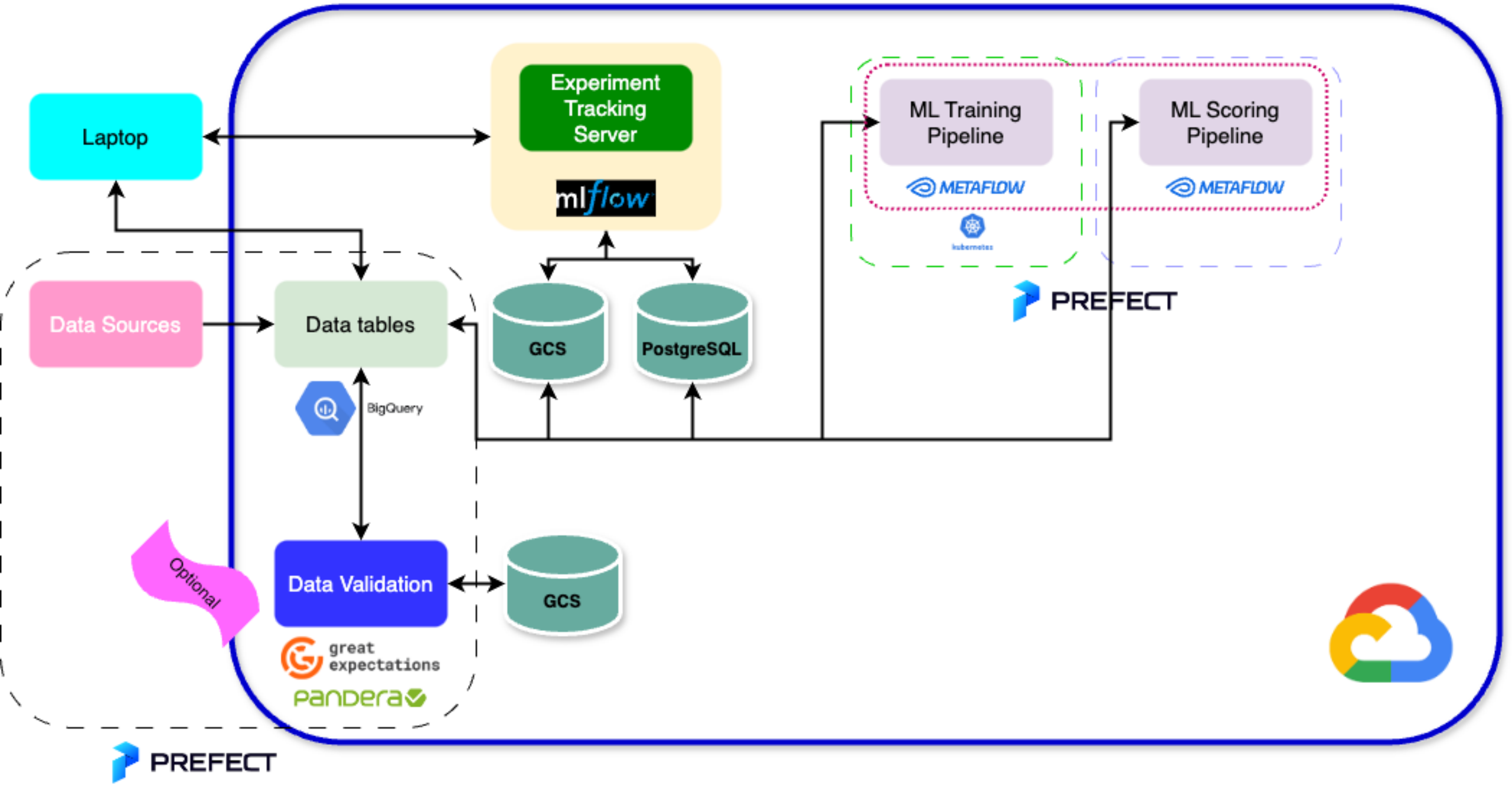
Scheduling Jobs with Argo Workflows

- Use @schedule decorator to schedule hourly, daily, weekly, or using a cron schedule
- Rename the flow (e.g. HelloWorldFlowStage) to deploy and test staging version before deploying production version

```
from metaflow import FlowSpec, step, schedule

@schedule(hourly=True)
class HelloWorldFlow(FlowSpec):
     @step
     def start(self): # start of flow
          …
```

More Things to Cover on Your Own – Read the Docs

- Inspecting Flows with the Client API
- Debugging Flows and the `resume` command
- Visualizing Results with Cards
- Organizing Flows with namespaces and tags



Laptop
Experiment Tracking Server
mlflow
ML Training Pipeline
METAFLOW
ML Scoring Pipeline
METAFLOW
kubernetes
PREFECT
Data Sources
Data tables
GCS
PostgreSQL
BigQuery
Optional
Data Validation
GCS
great expectations
pandera
PREFECT

Metaflow Scaling and Production Demo