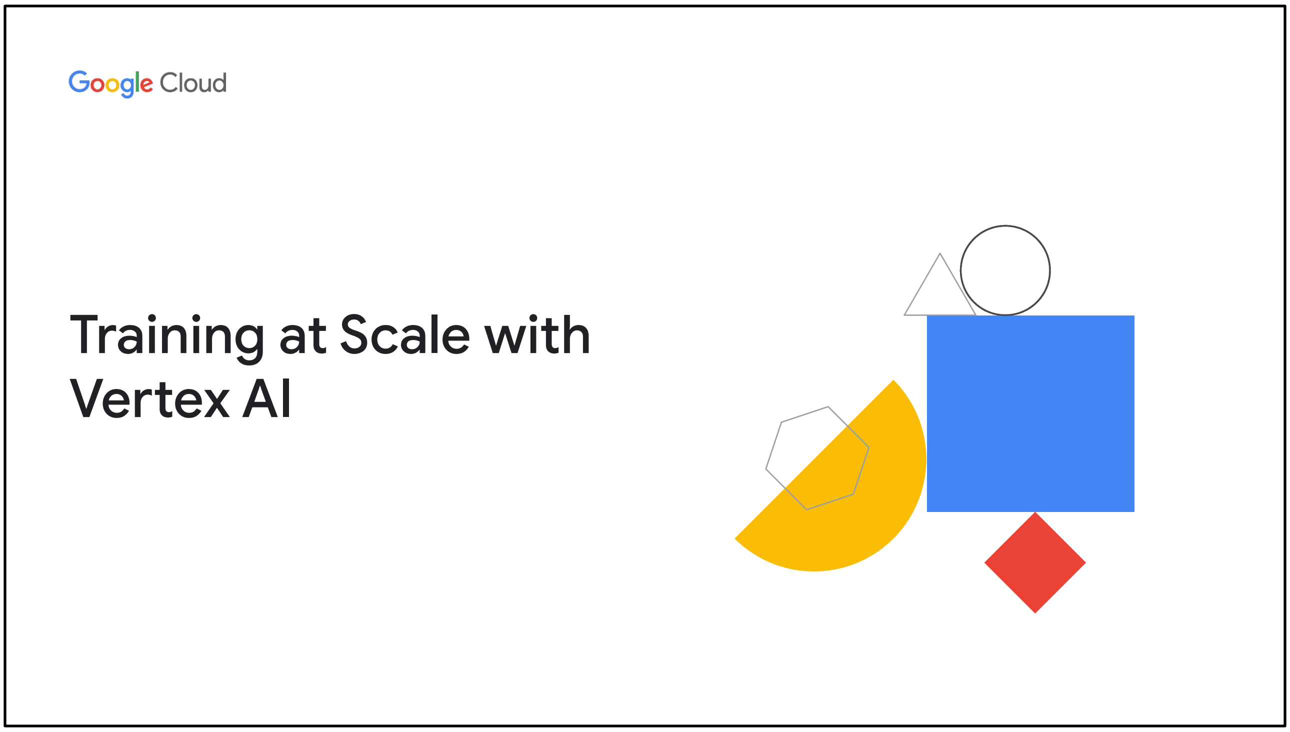
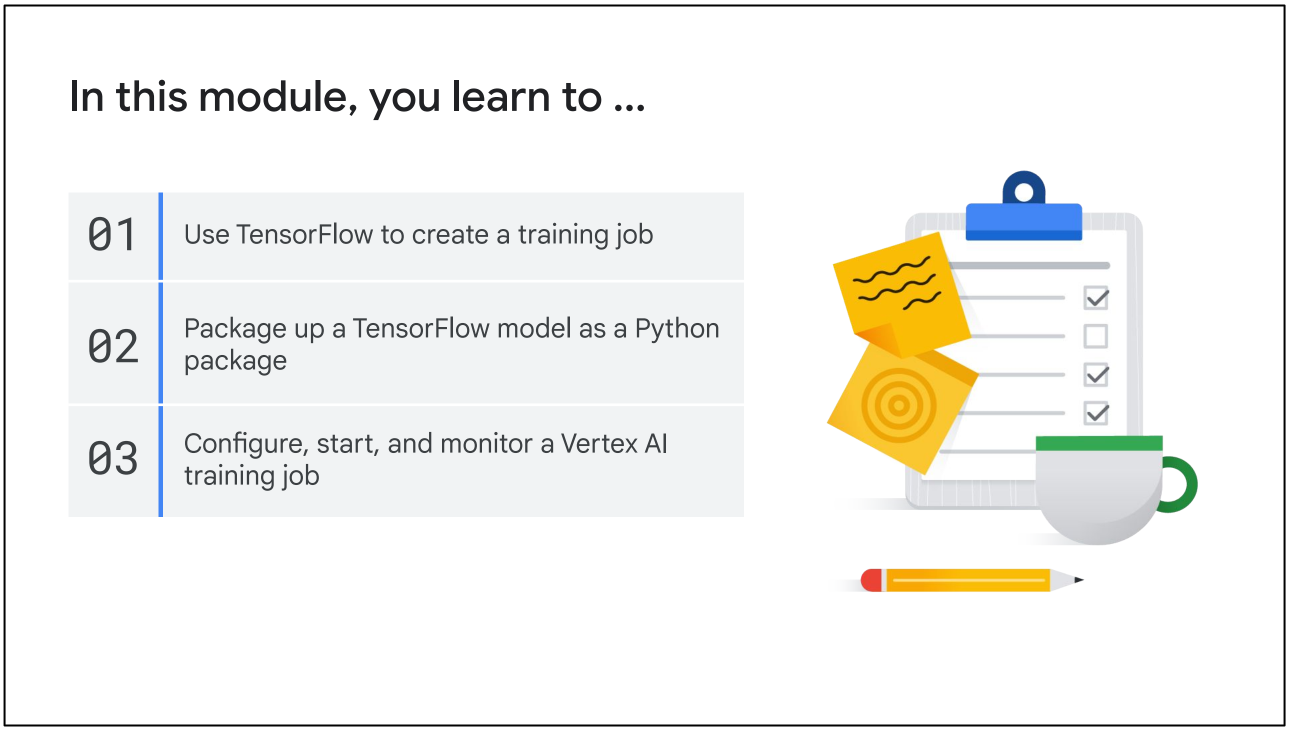


Training at Scale with Vertex AI

Welcome to the **Training at Scale with Vertex AI** module.

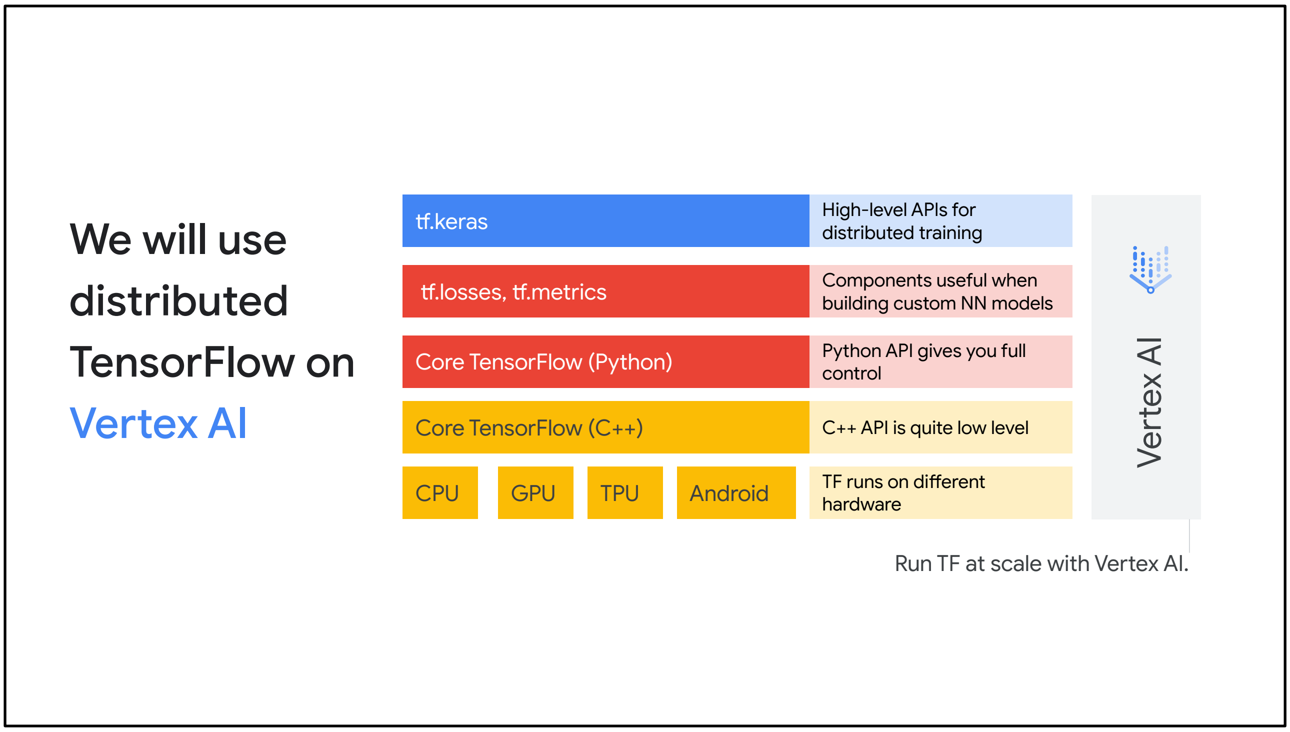


In this module, you learn to ...

01	Use TensorFlow to create a training job
02	Package up a TensorFlow model as a Python package
03	Configure, start, and monitor a Vertex AI training job

In this module, you'll learn to:

- Use TensorFlow to create your training job
- Package up a TensorFlow model as Python package
- Configure, start, and monitor a Vertex AI training job



We will use distributed TensorFlow on Vertex AI

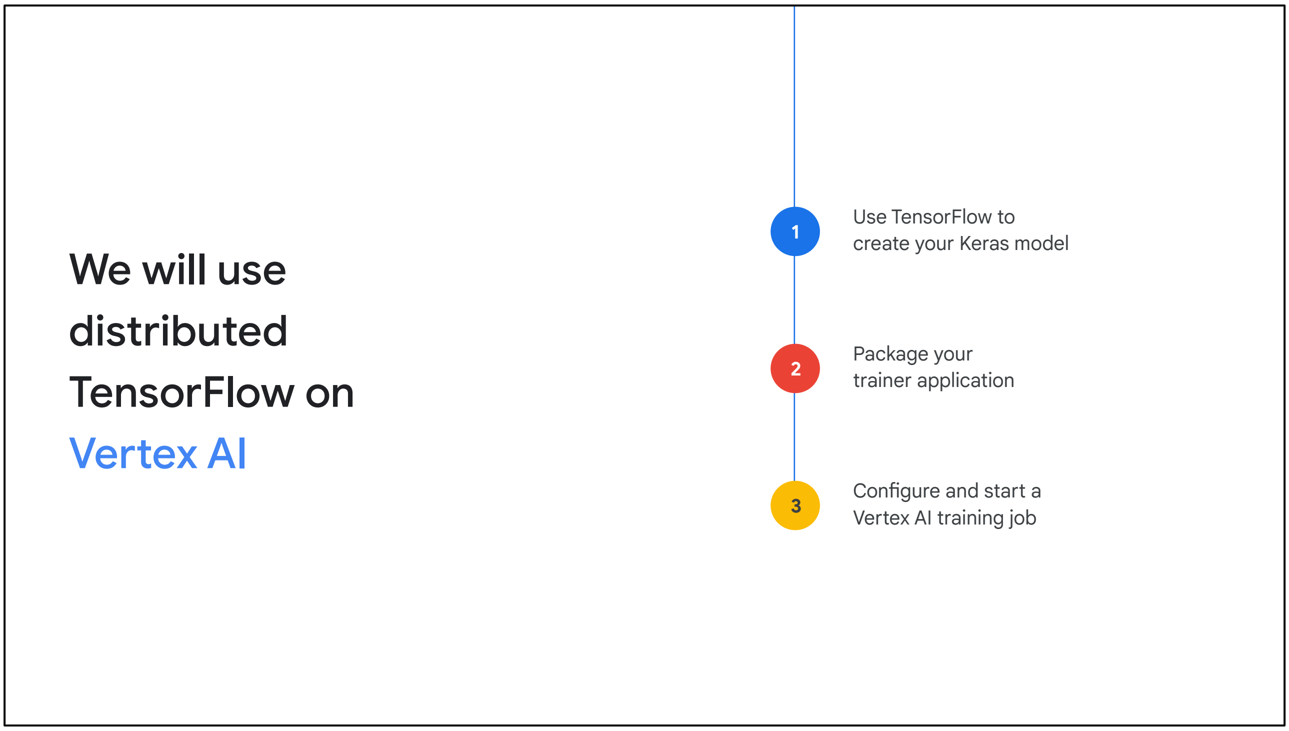
tf.keras
High-level APIs for distributed training
tf.losses, tf.metrics
Components useful when building custom NN models
Core TensorFlow (Python)
Python API gives you full control
Core TensorFlow (C++)
C++ API is quite low level
CPU
GPU
TPU
Android
TF runs on different hardware
Vertex AI
Run TF at scale with Vertex AI.

Now this diagram you've already seen before.

Recall that TensorFlow can run on different hardware; you could program it in the low-level C++ API, but more likely you'll use the Python API as we practice in this course.

And you've already started to see the different abstraction layers for distributed training.

But how do you actually run distributed TF at scale in production?

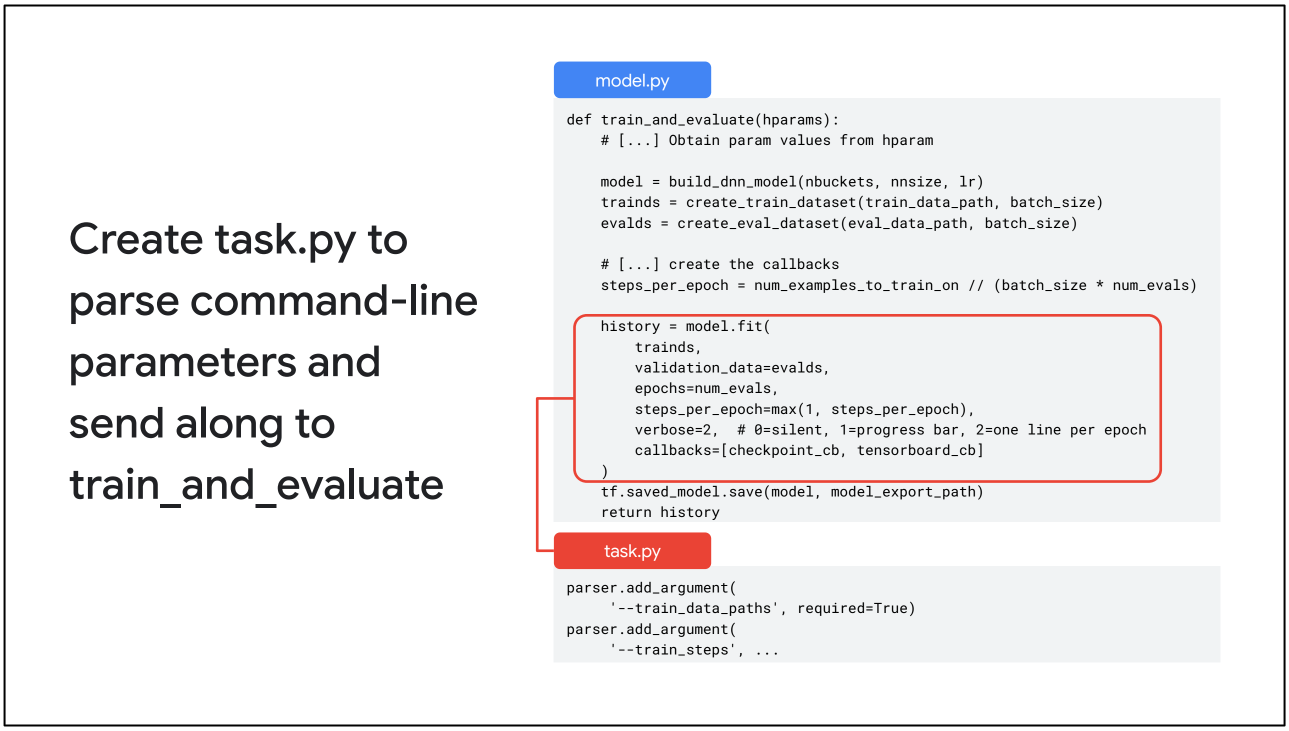


We will use distributed TensorFlow on Vertex AI

1. Use TensorFlow to create your Keras model
2. Package your trainer application
3. Configure and start a Vertex AI training job

We will use distributed TensorFlow on Vertex AI.

Before you begin training though, be sure to (1) gather and prepare (clean, split, engineer features, preprocess features) your training data, and (2) put that training data in an online source that Cloud Machine Learning Engine can access (e.g. Cloud Storage).



Create task.py to parse command-line parameters and send along to train_and_evaluate

model.py

```
def train_and_evaluate(hparams):
    # [...] Obtain param values from hparam

    model = build_dnn_model(nbuckets, nnsize, lr)
    trainds = create_train_dataset(train_data_path, batch_size)
    evalds = create_eval_dataset(eval_data_path, batch_size)

    # [...] create the callbacks
    steps_per_epoch = num_examples_to_train_on // (batch_size * num_evals)

    history = model.fit(
        trainds,
        validation_data=evalds,
        epochs=num_evals,
        steps_per_epoch=max(1, steps_per_epoch),
        verbose=2,  # 0=silent, 1=progress bar, 2=one line per epoch
        callbacks=[checkpoint_cb, tensorboard_cb]
    )
    tf.saved_model.save(model, model_export_path)
    return history
```

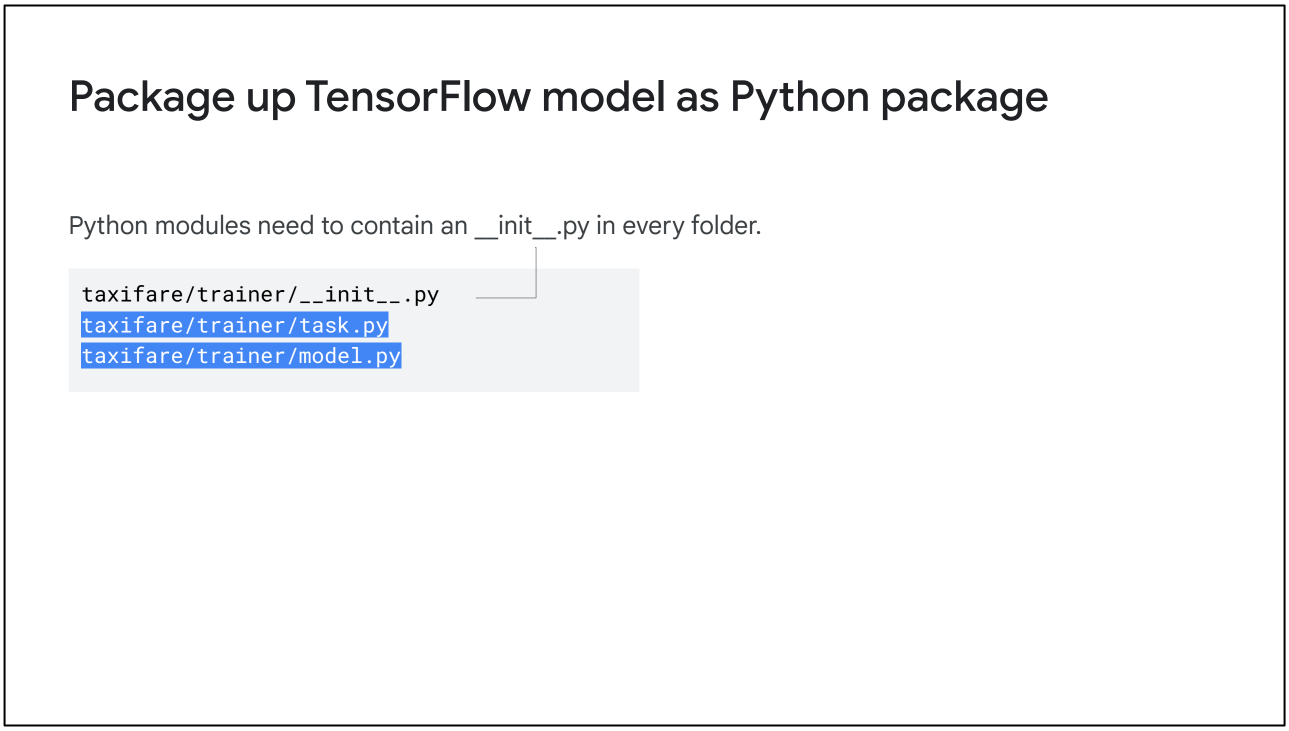
task.py

```
parser.add_argument(
     '--train_data_paths', required=True)
parser.add_argument(
     '--train_steps', ...
```

When sending training jobs to Vertex AI, its common to split most of the logic into a task.py file and a model.py file.

Task.py is the entrypoint to your code that AI Platform will start and knows job-level details like: how to parse the command line arguments, how long to run, where to write the outputs, how to interface with hyperparameter tuning and so on.

To do the core ML, task.py will invoke model.py



Package up TensorFlow model as Python package

Python modules need to contain an __init__.py in every folder.

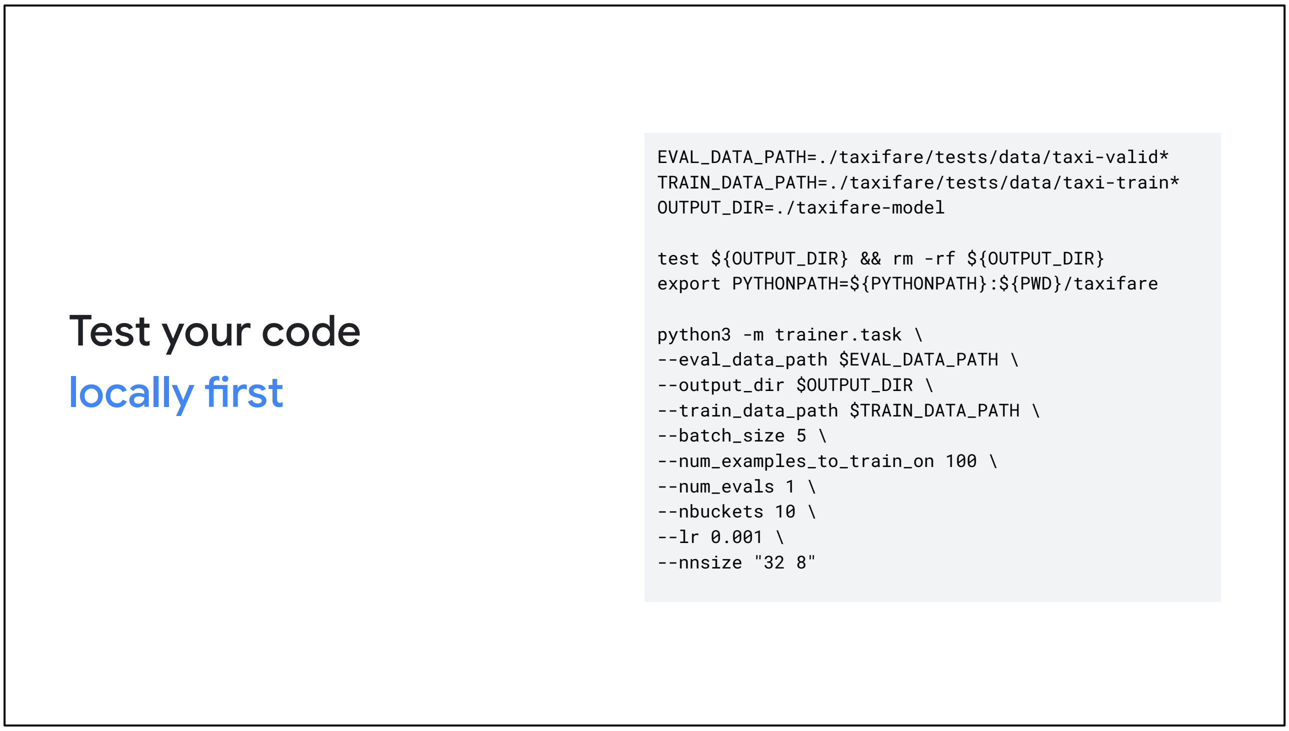
```
taxifare/trainer/__init__.py
taxifare/trainer/task.py
taxifare/trainer/model.py
```

Sharing code between computers always involves some type of packaging. Sending your model to Vertex AI Platform for training is no different.

Tensorflow, and Python in particular, require a very specific, but standardized, packaging structure shown here.

//

Check out more about that here:
http://python-packaging.readthedocs.io/en/latest/minimal.html



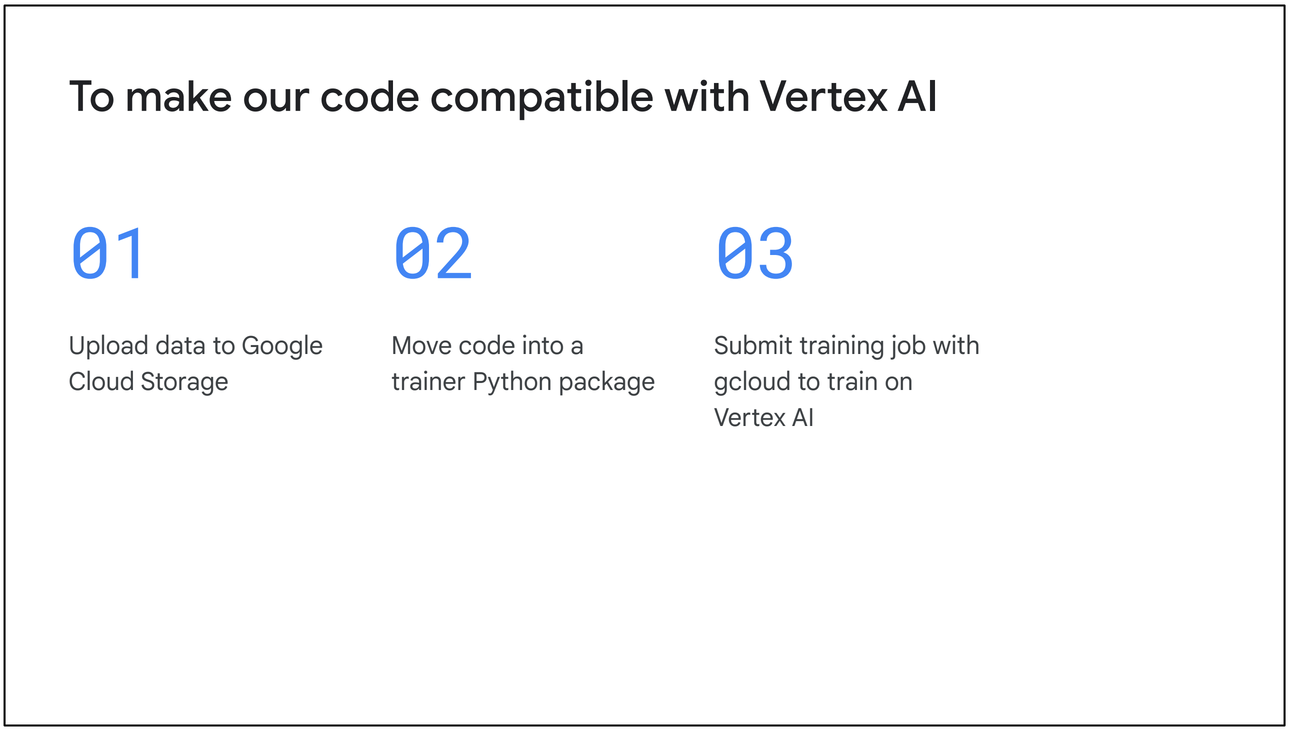
Test your code locally first

```
EVAL_DATA_PATH=./taxifare/tests/data/taxi-valid*
TRAIN_DATA_PATH=./taxifare/tests/data/taxi-train*
OUTPUT_DIR=./taxifare-model

test ${OUTPUT_DIR} && rm -rf ${OUTPUT_DIR}
export PYTHONPATH=${PYTHONPATH}:${PWD}/taxifare

python3 -m trainer.task \
--eval_data_path $EVAL_DATA_PATH \
--output_dir $OUTPUT_DIR \
--train_data_path $TRAIN_DATA_PATH \
--batch_size 5 \
--num_examples_to_train_on 100 \
--num_evals 1 \
--nbuckets 10 \
--lr 0.001 \
--nnsize "32 8"
```

This examples shows the code to test your code locally - which if the first step.



To make our code compatible with Vertex AI

01

Upload data to Google Cloud Storage

02

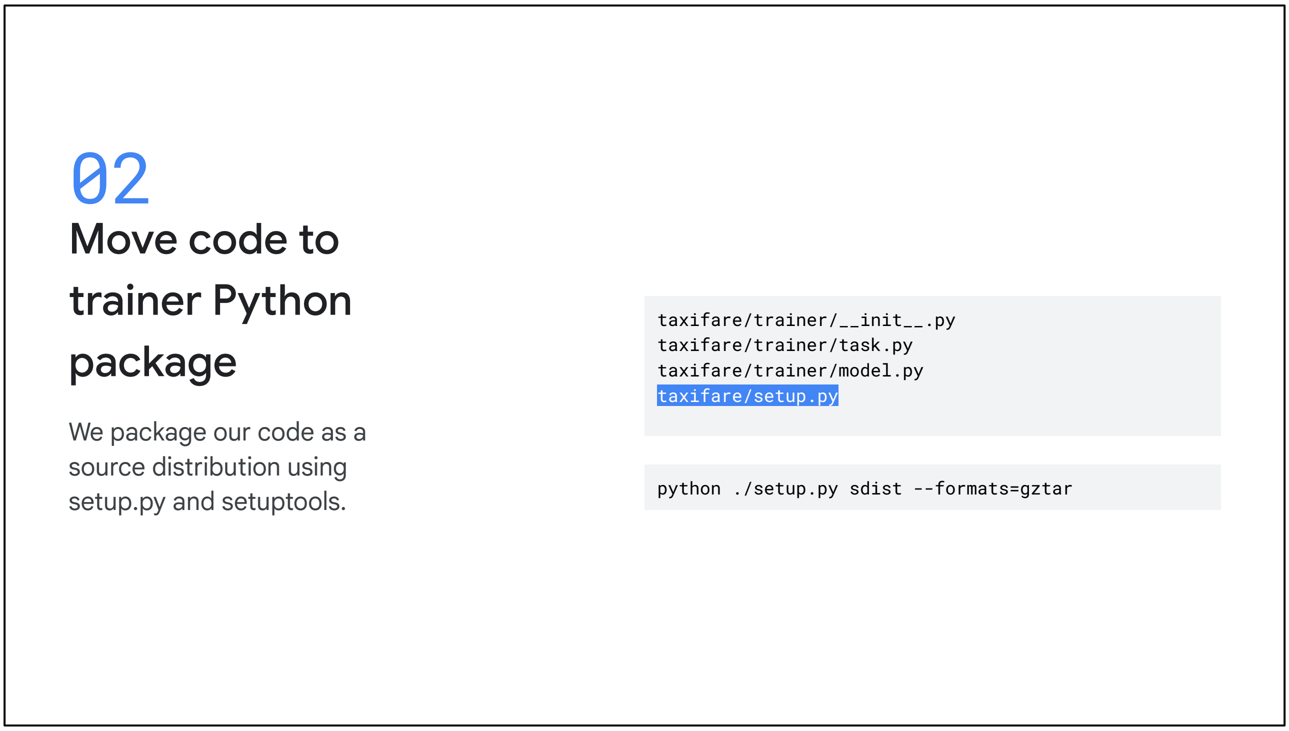
Move code into a trainer Python package

03

Submit training job with gcloud to train on Vertex AI

To make the code compatible with Vertex AI,

- Upload the data to Google Cloud storage
- Move the code into a trainer Python package
- Submit the training job with gcloud to train on Vertex AI



02

Move code to trainer Python package

We package our code as a source distribution using setup.py and setuptools.

```
taxifare/trainer/__init__.py
taxifare/trainer/task.py
taxifare/trainer/model.py
taxifare/setup.py
```

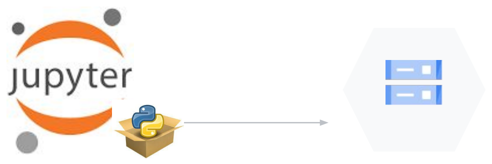
```
python ./setup.py sdist --formats=gztar
```

To move code to a trainer Python package, we package our code as a source distribution using setup.py and setuptools.

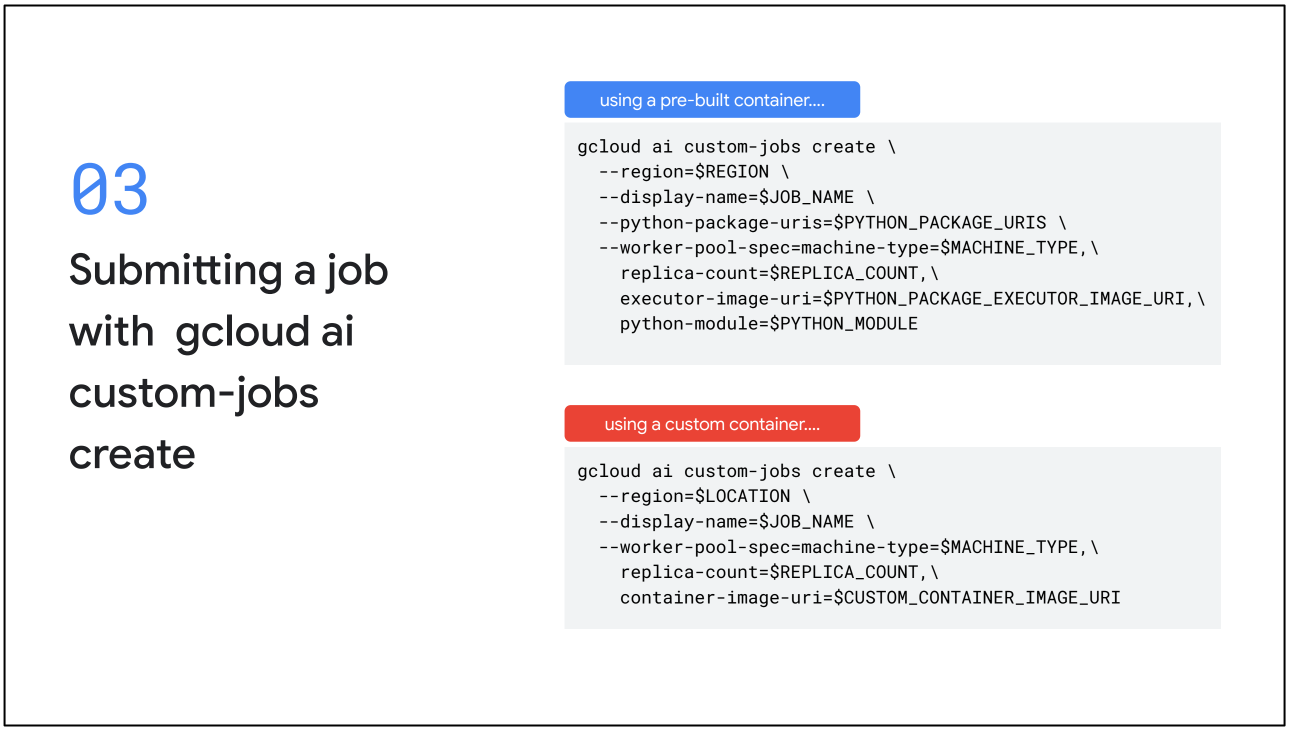
You use the **sdist** command to create a source distribution.

02
Copy that Python package to your GCS bucket

```
gsutil cp taxifare/dist/taxifare_trainer-0.1.tar.gz
gs://${BUCKET}/taxifare/
```



Then copy that Python package to your GCS bucket.



03

Submitting a job with gcloud ai custom-jobs create

using a pre-built container....

```
gcloud ai custom-jobs create \
  --region=$REGION \
  --display-name=$JOB_NAME \
  --python-package-uris=$PYTHON_PACKAGE_URIS \
  --worker-pool-spec=machine-type=$MACHINE_TYPE,\
    replica-count=$REPLICA_COUNT,\
    executor-image-uri=$PYTHON_PACKAGE_EXECUTOR_IMAGE_URI,\
    python-module=$PYTHON_MODULE
```

using a custom container....

```
gcloud ai custom-jobs create \
  --region=$LOCATION \
  --display-name=$JOB_NAME \
  --worker-pool-spec=machine-type=$MACHINE_TYPE,\
    replica-count=$REPLICA_COUNT,\
    container-image-uri=$CUSTOM_CONTAINER_IMAGE_URI
```

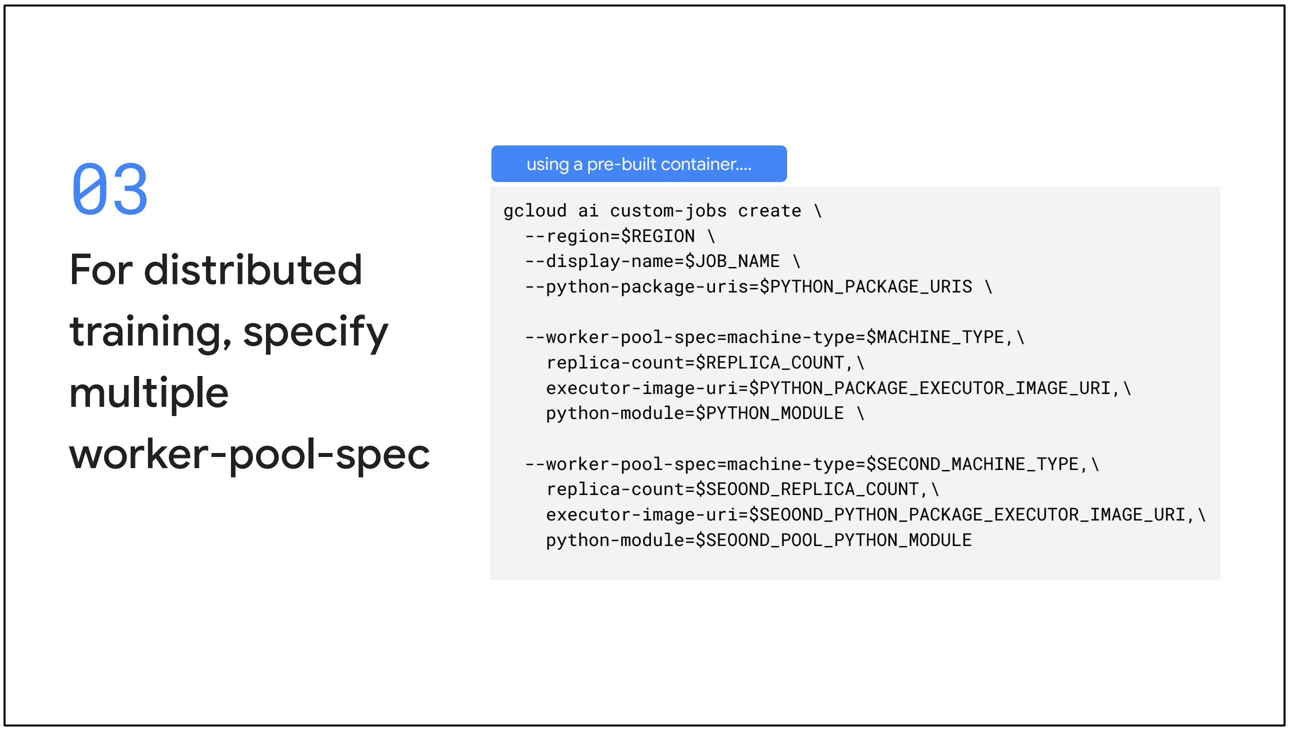
There are two general configurations: with a pre-built container and without.

When you create a *custom job*, you specify settings that Vertex AI needs to run your training code, including:

- **LOCATION**: The region where the container or Python package will be run.
- **JOB_NAME**: Required. A display name for the CustomJob.
- **PYTHON_PACKAGE_URIS**: Comma-separated list of Cloud Storage URIs specifying the Python package files which are the training program and its dependent packages. The maximum number of package URIs is 100.

One worker pool for single-node training (WorkerPoolSpec), or multiple worker pools for distributed training

- **MACHINE_TYPE**: The type of the machine. Refer to available machine types for training.
- **REPLICA_COUNT**: The number of worker replicas to use. In most cases, set this to 1 for your first worker pool.
- **PYTHON_PACKAGE_EXECUTOR_IMAGE_URI**: The URI of the container image that runs the provided Python package. Refer to the available pre-built containers for training.
- **PYTHON_MODULE**: The Python module name to run after installing the packages.



03

For distributed training, specify multiple worker-pool-spec

using a pre-built container....

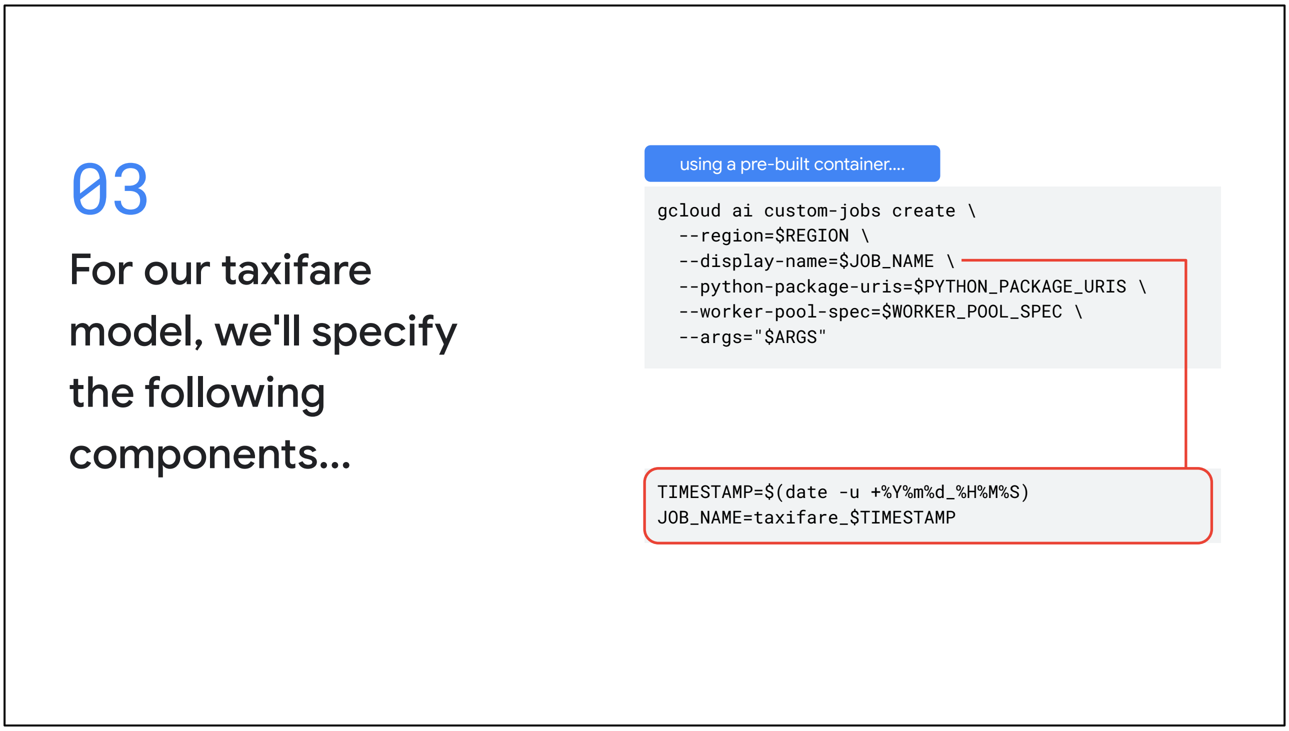
```
gcloud ai custom-jobs create \
  --region=$REGION \
  --display-name=$JOB_NAME \
  --python-package-uris=$PYTHON_PACKAGE_URIS \

  --worker-pool-spec=machine-type=$MACHINE_TYPE,\
    replica-count=$REPLICA_COUNT,\
    executor-image-uri=$PYTHON_PACKAGE_EXECUTOR_IMAGE_URI,\
    python-module=$PYTHON_MODULE \

  --worker-pool-spec=machine-type=$SECOND_MACHINE_TYPE,\
    replica-count=$SEOOND_REPLICA_COUNT,\
    executor-image-uri=$SEOOND_PYTHON_PACKAGE_EXECUTOR_IMAGE_URI,\
    python-module=$SEOOND_POOL_PYTHON_MODULE
```

If you want to perform distributed training, then you can specify the --worker-pool-spec flag multiple times, once for each worker pool.

We'll talk later about modifications to your code to accommodate distributed training.



03

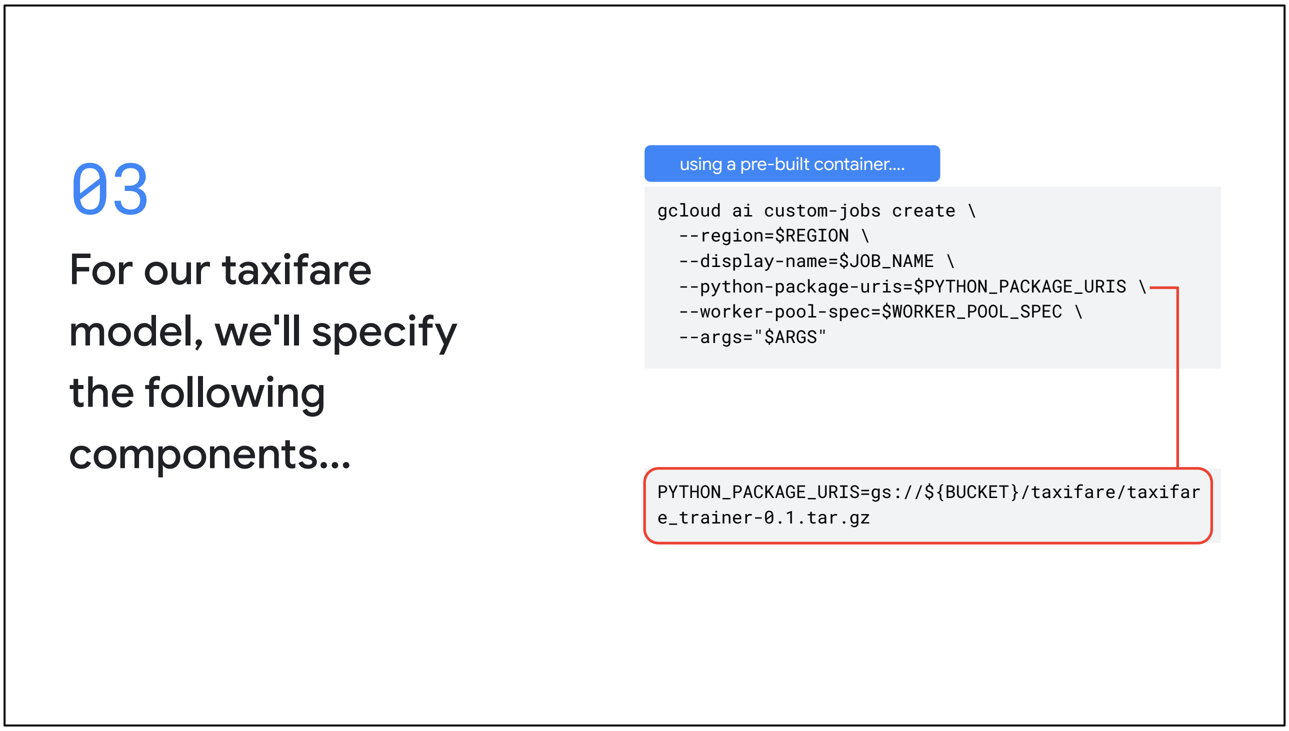
For our taxifare model, we'll specify the following components...

using a pre-built container....

```
gcloud ai custom-jobs create \
  --region=$REGION \
  --display-name=$JOB_NAME \
  --python-package-uris=$PYTHON_PACKAGE_URIS \
  --worker-pool-spec=$WORKER_POOL_SPEC \
  --args="$ARGS"
```

```
TIMESTAMP=$(date -u +%Y%m%d_%H%M%S)
JOB_NAME=taxifare_$TIMESTAMP
```

Let's talk through these various components for our taxifare model...



03

For our taxifare model, we'll specify the following components...

using a pre-built container....

```
gcloud ai custom-jobs create \
  --region=$REGION \
  --display-name=$JOB_NAME \
  --python-package-uris=$PYTHON_PACKAGE_URIS \
  --worker-pool-spec=$WORKER_POOL_SPEC \
  --args="$ARGS"
```

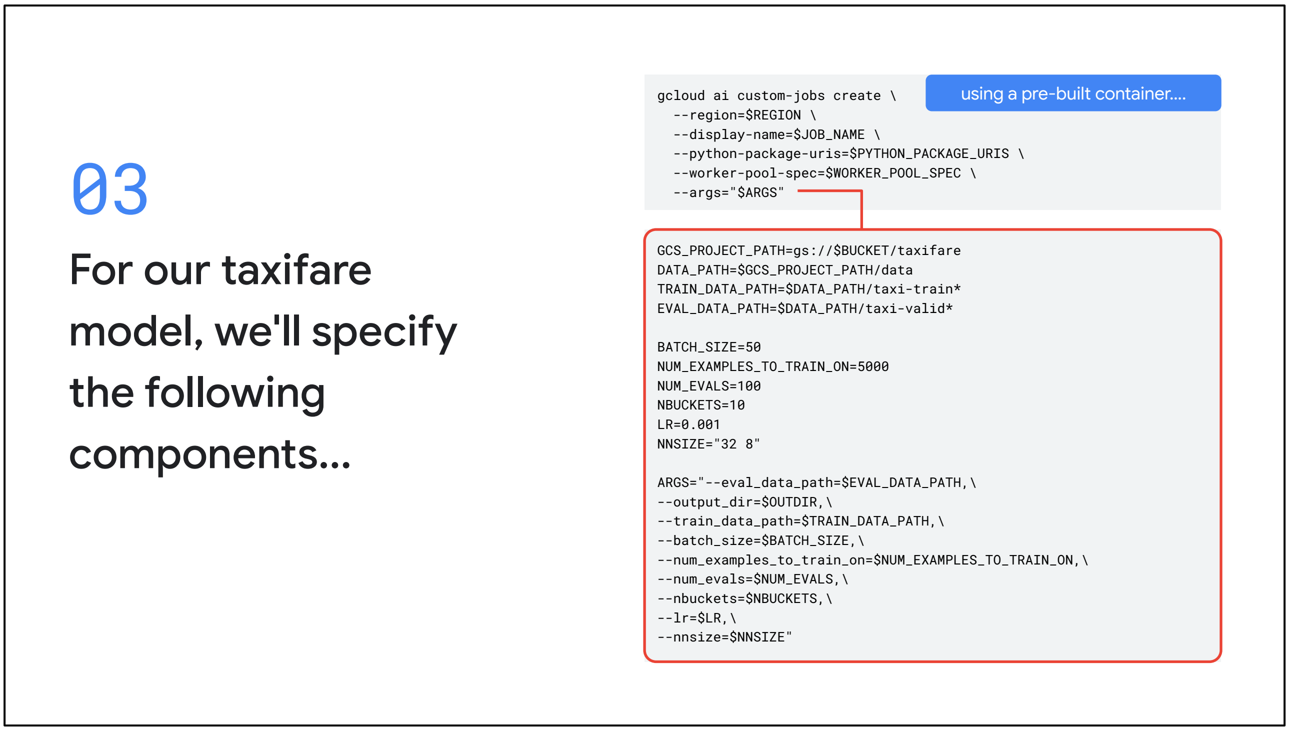
```
PYTHON_PACKAGE_URIS=gs://${BUCKET}/taxifare/taxifar
e_trainer-0.1.tar.gz
```

For our taxifare model, we'll specify the following components: Region, Display name, Python package, worker pool, and any arguments.

Note that, we're using a pre-built container that will run in a specific region. The Python package is held in a Google Cloud Storage bucket.

The spec of the worker pools includes machine type and Docker image.

https://cloud.google.com/vertex-ai/docs/reference/rest/v1/CustomJobSpec



03

For our taxifare model, we'll specify the following components...

```
gcloud ai custom-jobs create \
  --region=$REGION \
  --display-name=$JOB_NAME \
  --python-package-uris=$PYTHON_PACKAGE_URIS \
  --worker-pool-spec=$WORKER_POOL_SPEC \
  --args="$ARGS"
```

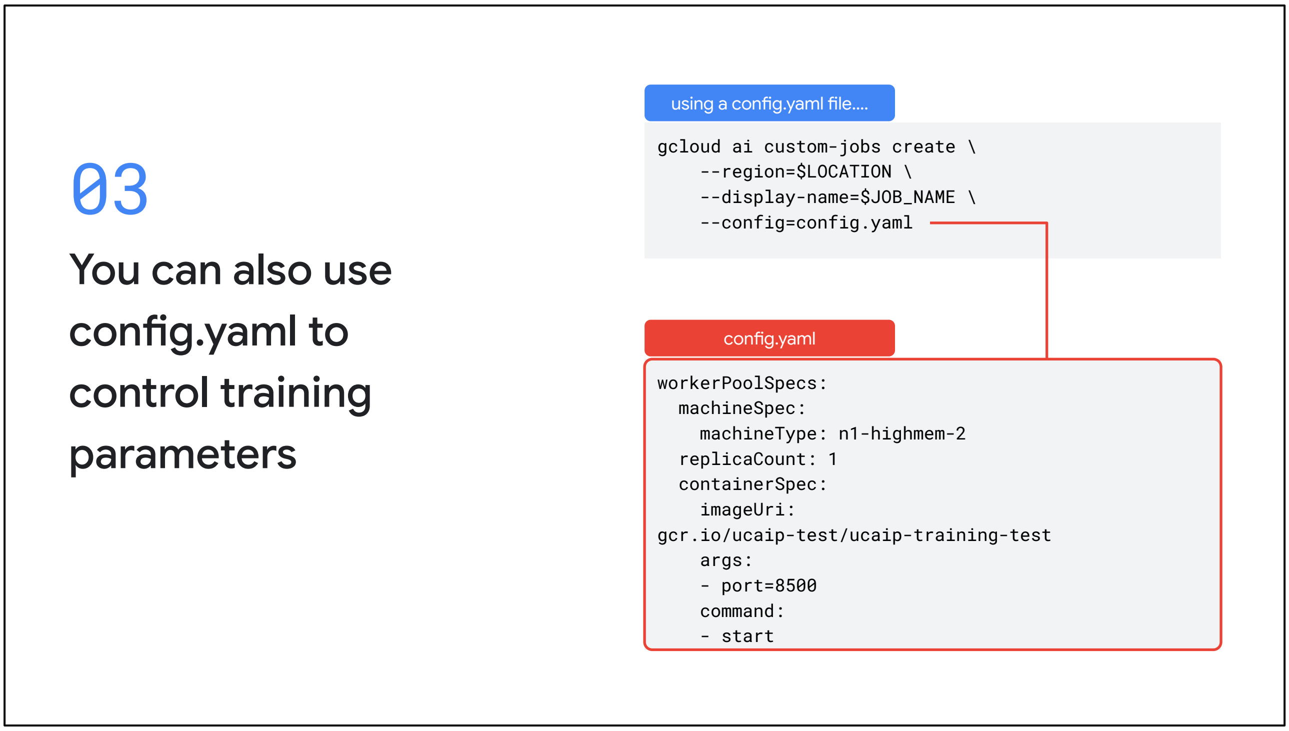
using a pre-built container....

```
GCS_PROJECT_PATH=gs://$BUCKET/taxifare
DATA_PATH=$GCS_PROJECT_PATH/data
TRAIN_DATA_PATH=$DATA_PATH/taxi-train*
EVAL_DATA_PATH=$DATA_PATH/taxi-valid*

BATCH_SIZE=50
NUM_EXAMPLES_TO_TRAIN_ON=5000
NUM_EVALS=100
NBUCKETS=10
LR=0.001
NNSIZE="32 8"

ARGS="--eval_data_path=$EVAL_DATA_PATH,\
--output_dir=$OUTDIR,\
--train_data_path=$TRAIN_DATA_PATH,\
--batch_size=$BATCH_SIZE,\
--num_examples_to_train_on=$NUM_EXAMPLES_TO_TRAIN_ON,\
--num_evals=$NUM_EVALS,\
--nbuckets=$NBUCKETS,\
--lr=$LR,\
--nnsize=$NNSIZE"
```

Arguments include output directory, the training data path, batch size, and the number of examples to train.



03

You can also use config.yaml to control training parameters

using a config.yaml file....

```
gcloud ai custom-jobs create \
    --region=$LOCATION \
    --display-name=$JOB_NAME \
    --config=config.yaml
```

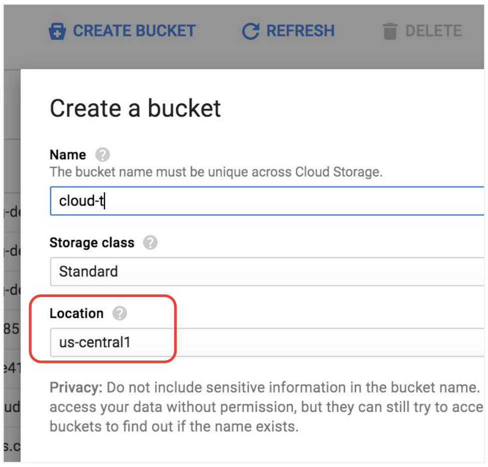
config.yaml

```
workerPoolSpecs:
  machineSpec:
    machineType: n1-highmem-2
  replicaCount: 1
  containerSpec:
    imageUri:
gcr.io/ucaip-test/ucaip-training-test
    args:
    - port=8500
    command:
    - start
```

If you want to specify configuration options that are not available in the preceding examples, you can use the --config flag to specify the path to a config.yaml file in your local environment that contains the fields of CustomJobSpec.

For example:
If an option is specified both in the configuration file **and** via command-line arguments, the command-line arguments override the configuration file.

Tip:
Use single-region bucket for ML



Create a bucket

Name
The bucket name must be unique across Cloud Storage.
cloud-t

Storage class
Standard

Location
us-central1

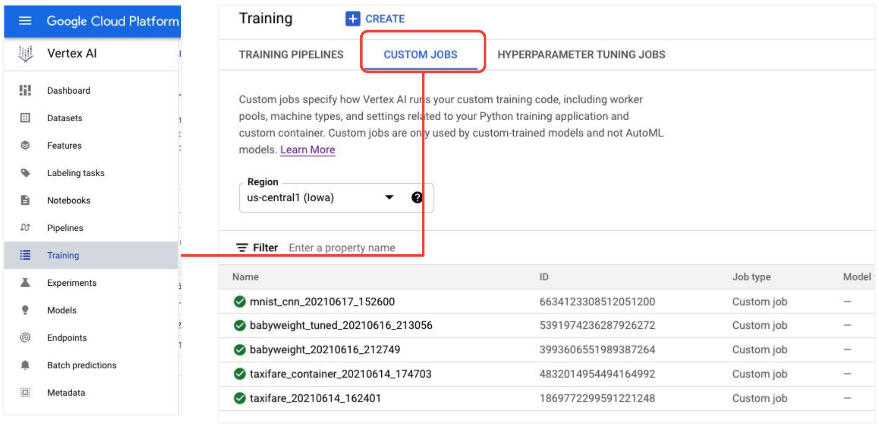
Privacy: Do not include sensitive information in the bucket name.
access your data without permission, but they can still try to acce
buckets to find out if the name exists.

Just a pro-tip here. To get the best performance for ML jobs, make sure you select a single-region bucket in Google Cloud storage.

The default is multi-region which is better suited for web serving than ML training!

Monitor training jobs with GCP console

You can also view CPU and Memory utilization charts for this training job with StackDriver Monitoring.



Custom jobs specify how Vertex AI runs your custom training code, including worker pools, machine types, and settings related to your Python training application and custom container. Custom jobs are only used by custom-trained models and not AutoML models. Learn More

Region: us-central1 (Iowa)

Filter: Enter a property name

Name	ID	Job type	Model
✓ mnist_cnn_20210617_152600	6634123308512051200	Custom job	–
✓ babyweight_tuned_20210616_213056	5391974236287926272	Custom job	–
✓ babyweight_20210616_212749	3993606551989387264	Custom job	–
✓ taxifare_container_20210614_174703	4832014954494164992	Custom job	–
✓ taxifare_20210614_162401	1869772299591221248	Custom job	–

You can monitor training jobs with the Google Cloud console.

The GCP web console has a great UI for monitoring your jobs.

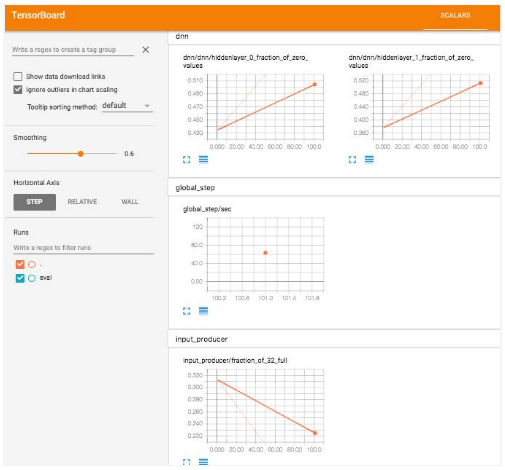
You can see exactly how they were invoked, check out their logs and see how much CPU and memory they are consuming.

Monitor training jobs with TensorBoard

Configure your trainer to save summary data that you can examine and visualize using TensorBoard.

```
tensorboard_cb = callbacks.TensorBoard(tensorboard_path)

history = model.fit(
    trainds,
    validation_data=evalds,
    epochs=num_evals,
    steps_per_epoch=max(1, steps_per_epoch),
    verbose=2,  # 0=silent, 1=progress bar, 2=one line per epoch
    callbacks=[checkpoint_cb, tensorboard_cb]
)
```

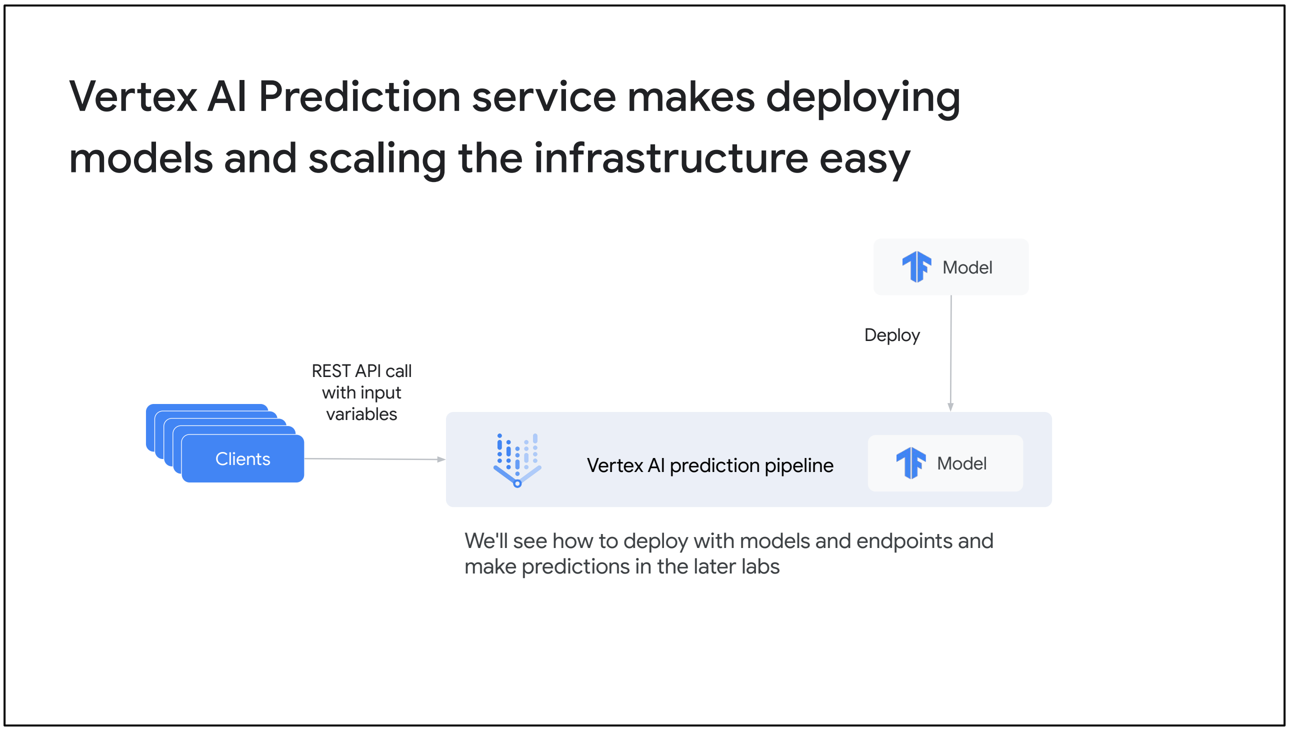

TensorBoard
SCALARS
Write a regex to create a tag group
dnn
dnn/dnn/hiddenlayer_0_fraction_of_zero_values
dnn/dnn/hiddenlayer_1_fraction_of_zero_values
global_step
global_step/sec
input_producer
input_producer/fraction_of_32_full
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
eval

☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.6
Horizontal Axis: STEP

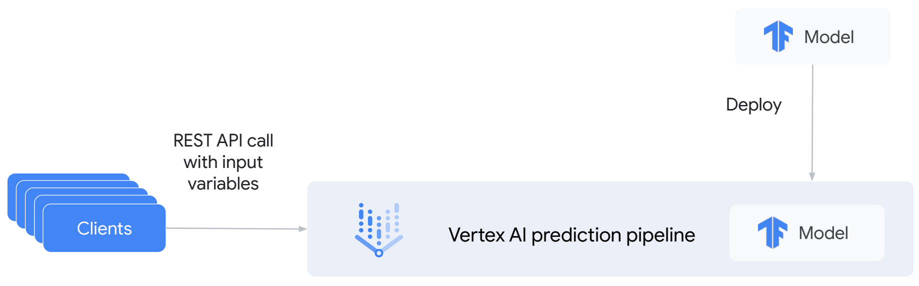
While inspecting log entries may help you debug technical issues like an exception, it's really not the right tool to investigate the ML performance.

Tensorboard however is a great tool. To use it, make sure your job saves summary data to a Google Cloud Storage location, and then when you start Tensorboard simply provide that directory.

It can even handle multiple jobs per folder.



Vertex AI Prediction service makes deploying models and scaling the infrastructure easy

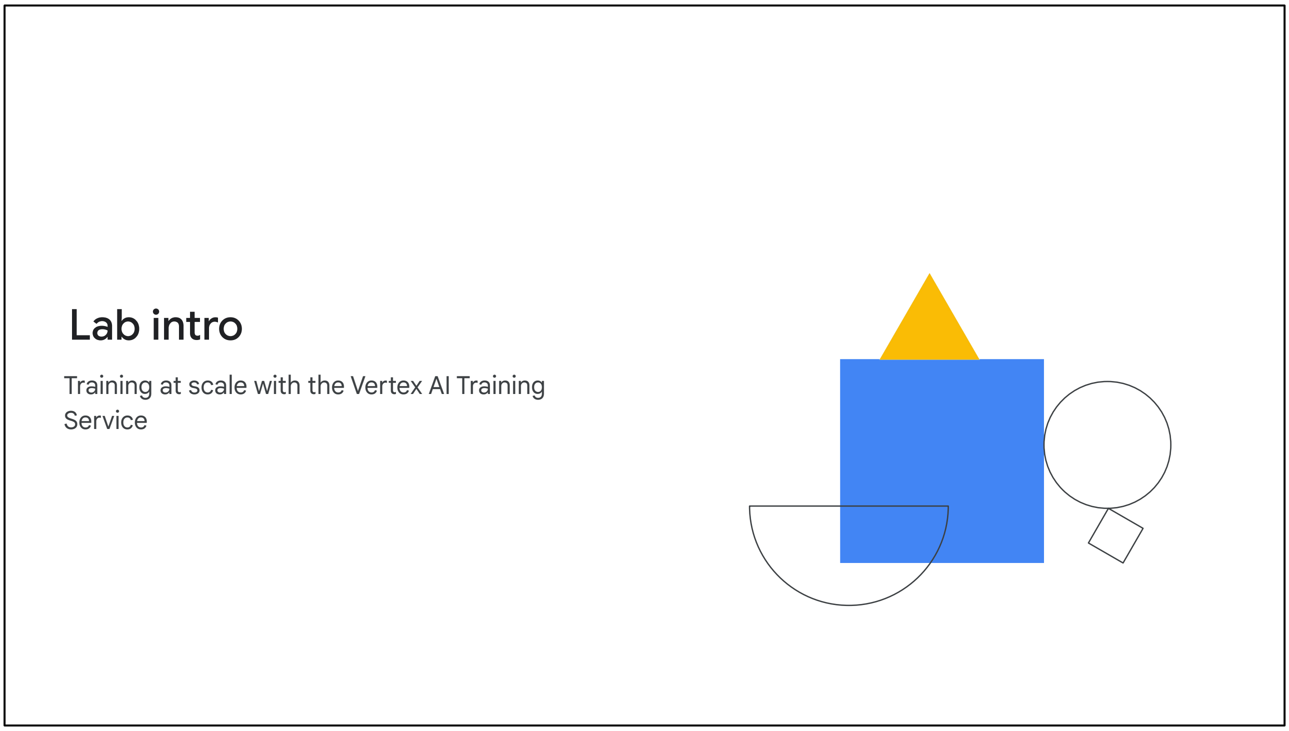

Model
Deploy
REST API call with input variables
Clients
Vertex AI prediction pipeline
Model

We'll see how to deploy with models and endpoints and make predictions in the later labs

Once your training jobs complete, you'll have a TensorFlow model ready to serve for predictions.

AI Platform provides a great infrastructure for this. AI Platform will build you a production-ready web app out of your trained model and offer a batch service for your less latency sensitive predictions.

Since these are both REST APIs, you'll be able to make scalable, secure inferences from whatever language you want to write the client in.



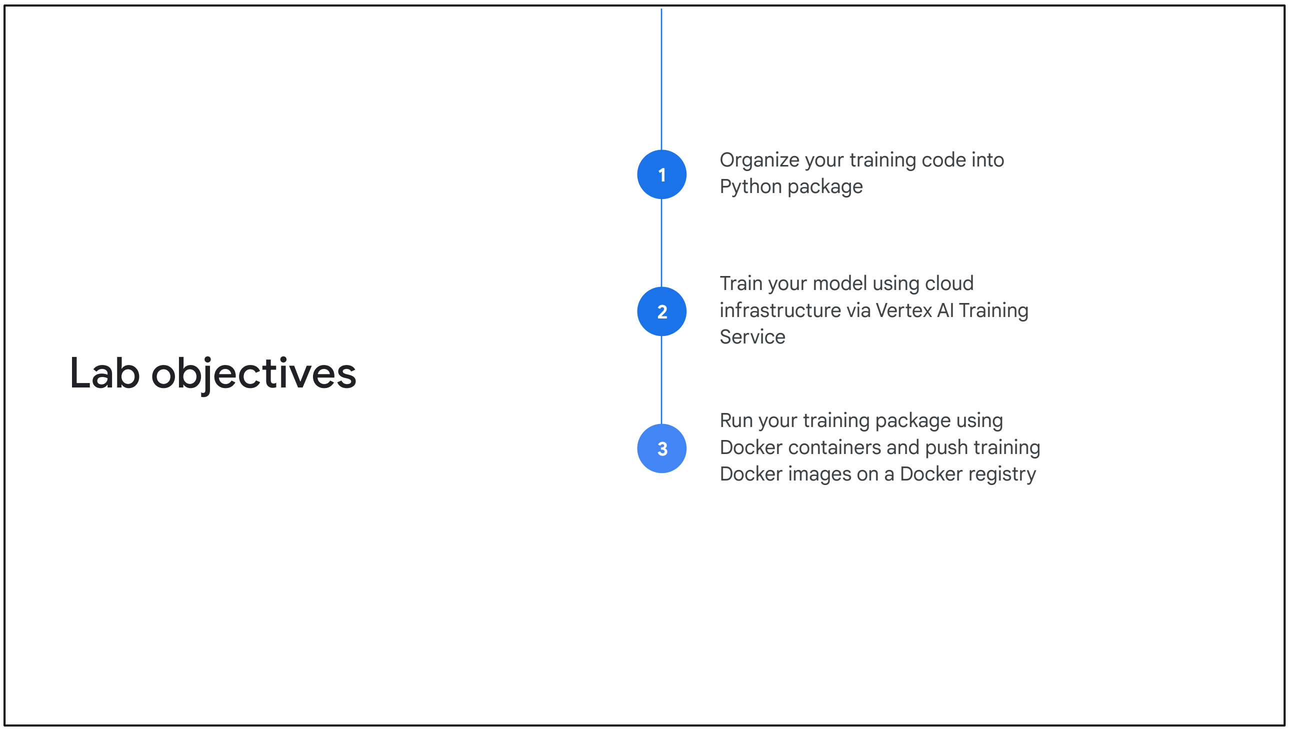
Lab intro

Training at scale with the Vertex AI Training Service

In this lab we'll make the jump from training locally, to do training in the cloud.

We'll take advantage of Google Cloud's Vertex AI Training Service. AI Platform Training Service is a managed service that allows the training and deployment of ML models without having to provision or maintain servers.

The infrastructure is handled seamlessly by the managed service for us.



Lab objectives

1. Organize your training code into Python package
2. Train your model using cloud infrastructure via Vertex AI Training Service
3. Run your training package using Docker containers and push training Docker images on a Docker registry

In this lab you'll organize your training code into a Python package, train your model using cloud infrastructure via Vertex AI Training Service, and run your training package using Docker containers and push training Docker images on a Docker registry.