MLOps on Google Cloud

Training Course

What this course is **NOT about**

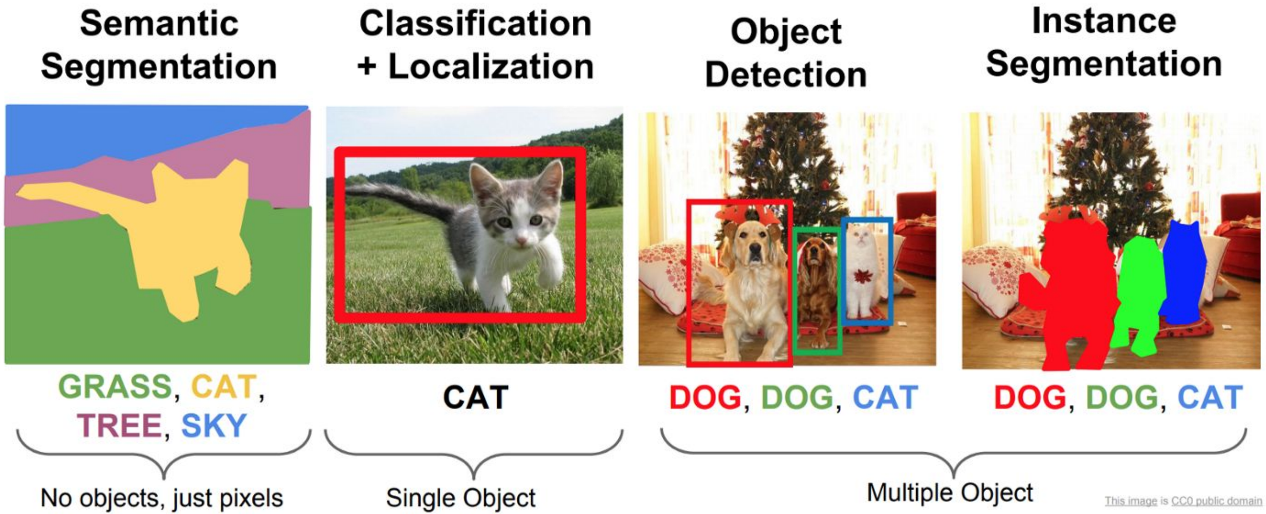
- Data Science
- Designing / Building Models
- Math

What this course is **about**

- Model Deployment / Serving
- Continuous (re)Training : CI/CD/CT
- Automation

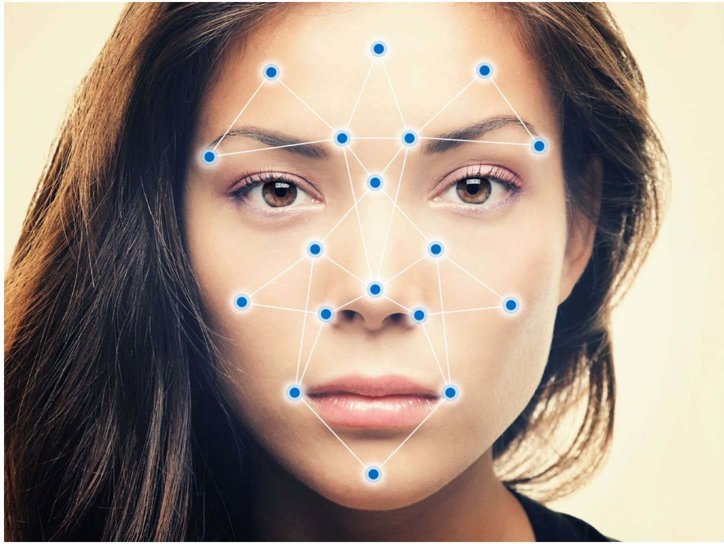
Machine Learning Overview

- Computer Vision (Images, Video)
  - Classification / Localization
  - Object Detection / Tracking
  - Segmentation



Machine Learning Overview

- Computer Vision (Images, Video)
  - Facial Recognition
  - Pose Detection
  - Captioning

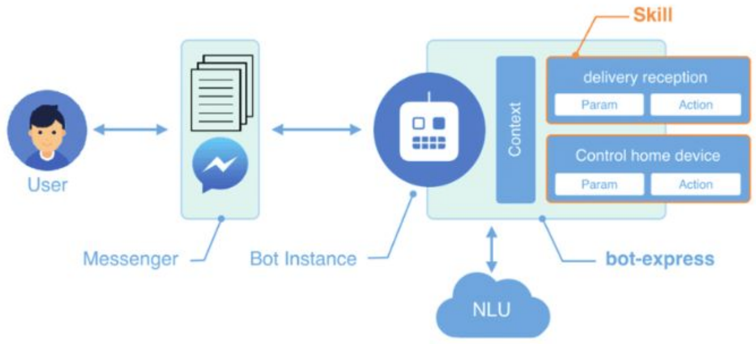


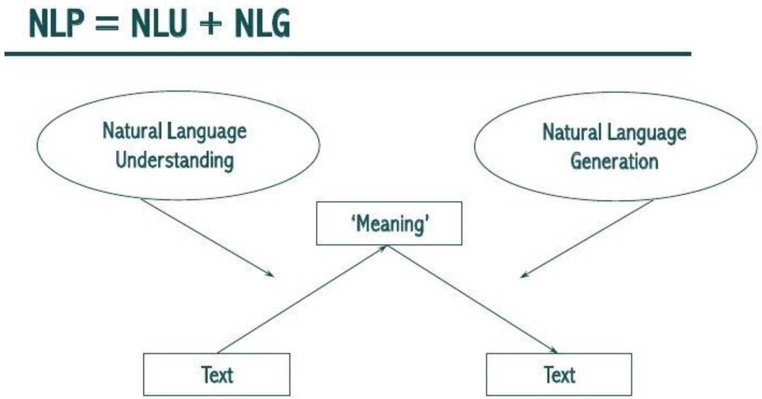

Machine Learning Overview

- Natural Language Understanding (Text)
  - Classification
  - Sentiment
  - Entity Extraction
  - Form Recognition

Machine Learning Overview

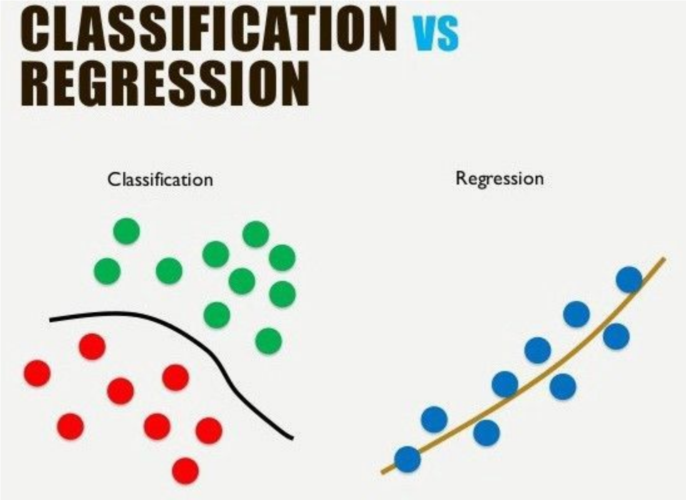
- Natural Language Generation (Text/Audio)
  - Text-2-Speech / Speech-2-Text
  - Summarization
  - Chat

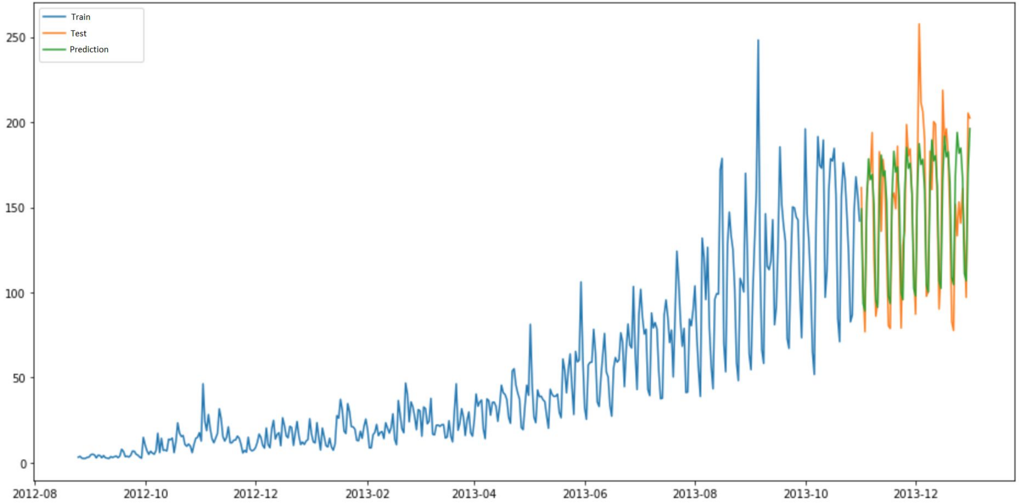




Machine Learning Overview

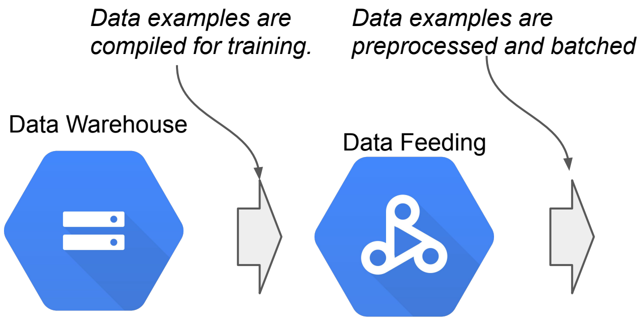
- Structured Data (Tabular, Databases)
  - Classification
  - Regression (Real Number)
  - Forecasting (time-series)





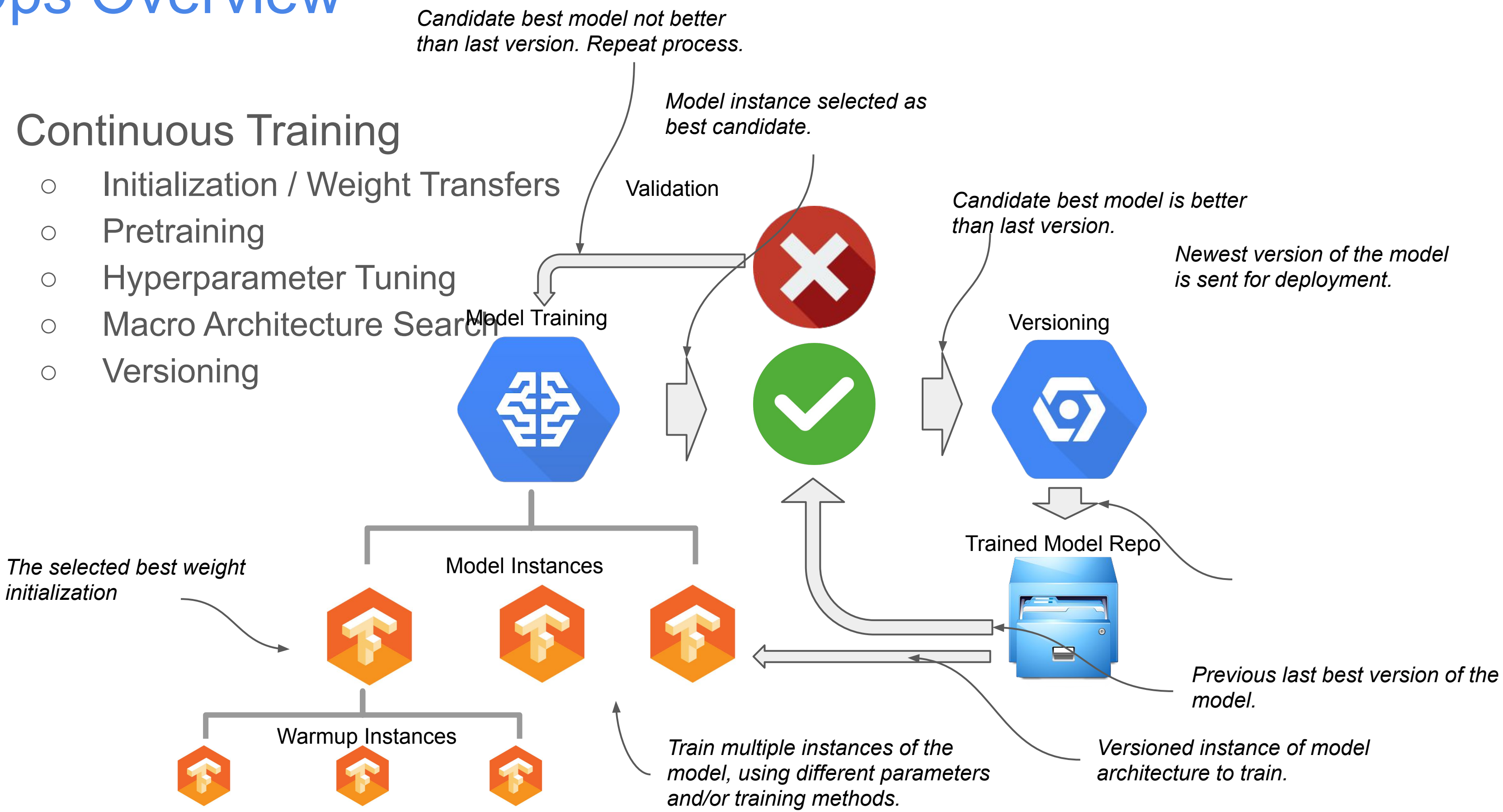
MLOps Overview

- Data Warehousing
  - Storage
  - Retrieval (I/O)
  - Feeding
  - Search / Query



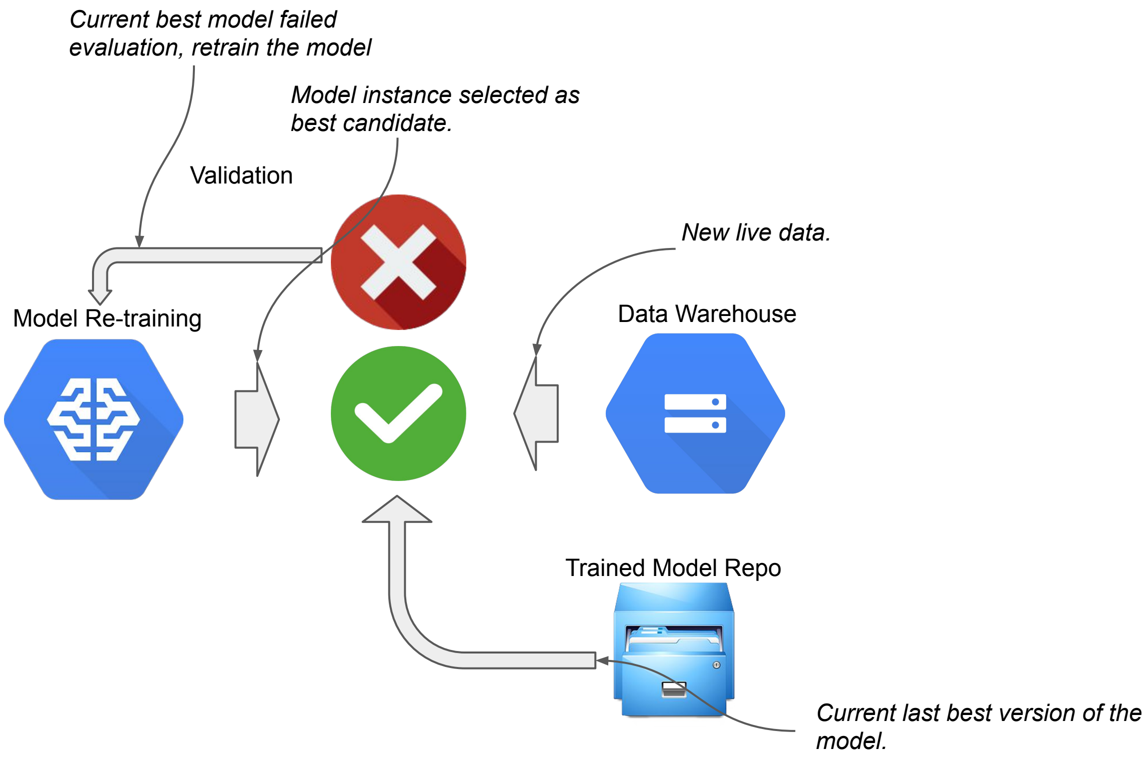
MLOps Overview

- Continuous Training
  - Initialization / Weight Transfers
  - Pretraining
  - Hyperparameter Tuning
  - Macro Architecture Search
  - Versioning



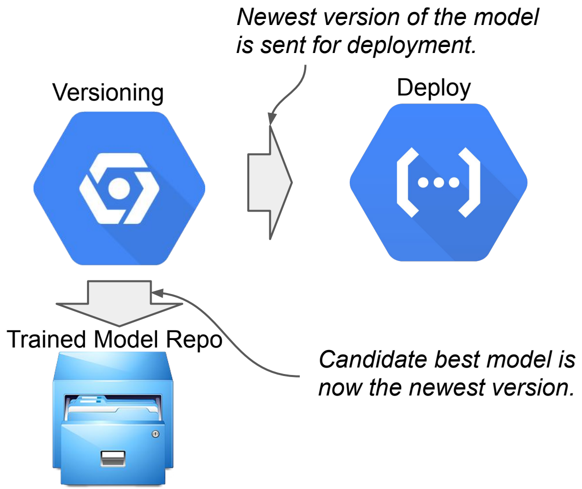
MLOps Overview

- Continuous Evaluation
  - Training Distribution
  - Serving Skew
  - Data Drift
  - A/B Testing



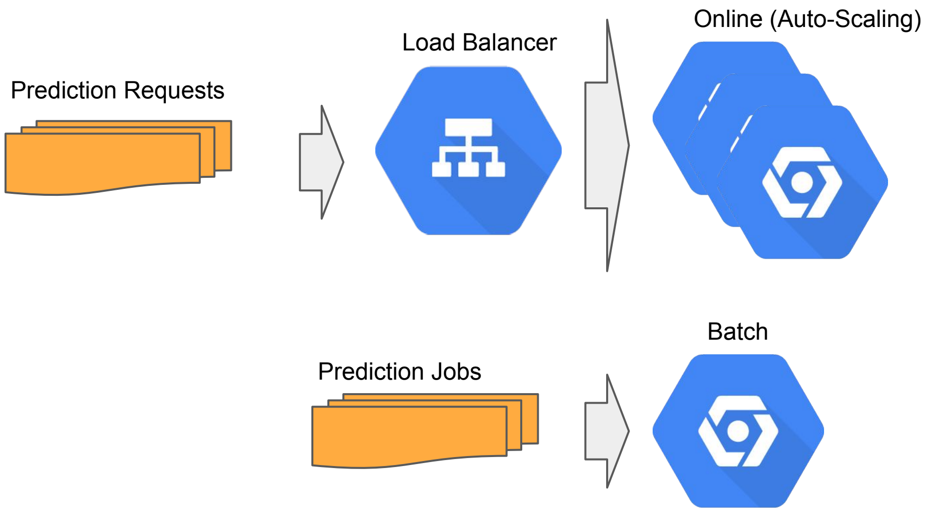
MLOps Overview

- Deployment
    - Scaling
    - Load Balancing
    - Latency
    - Edge



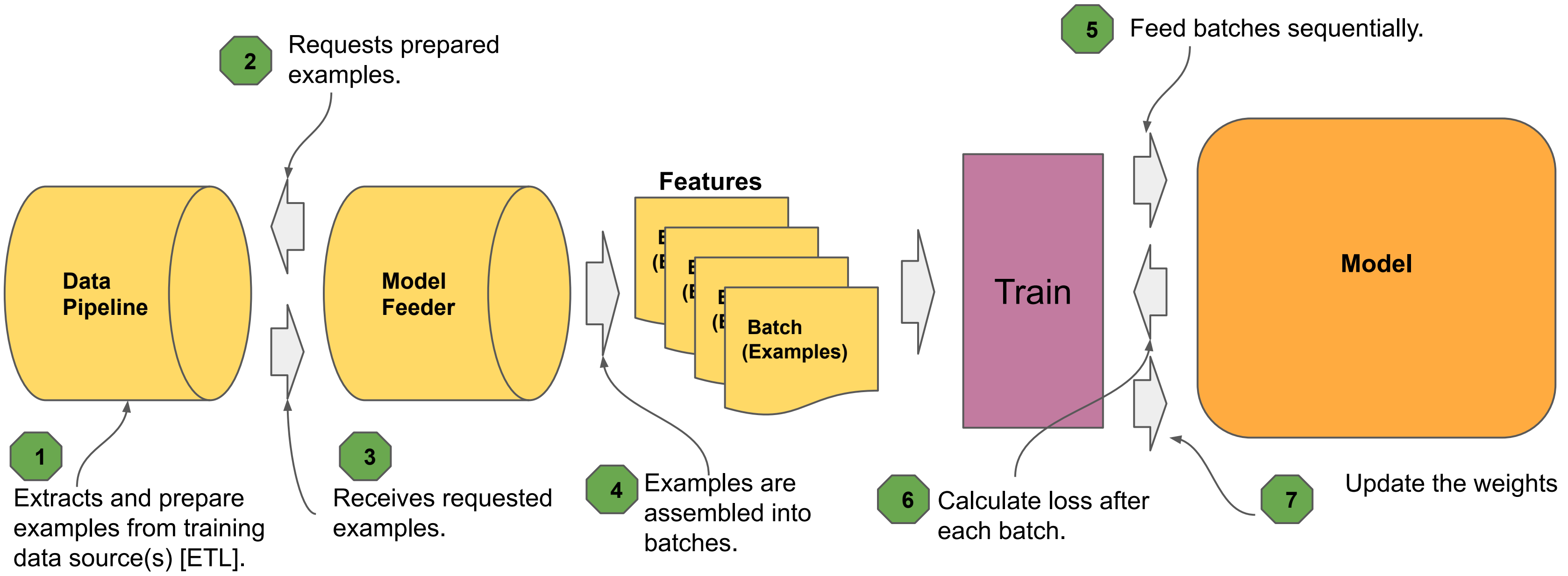
MLOps Overview

- Serving
  - Online (live)
  - Batch
  - Monitoring
  - Data Collection



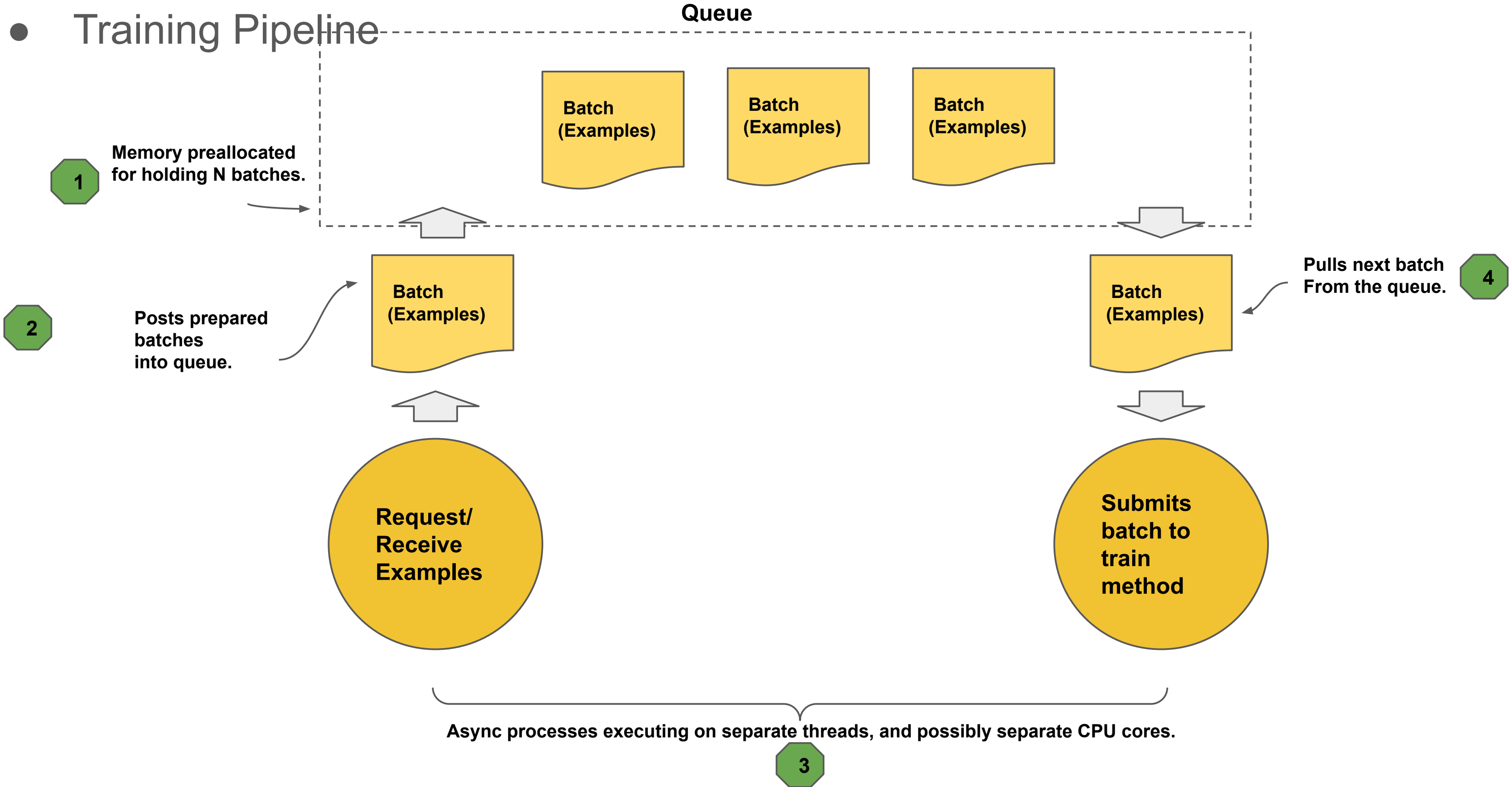
ML end-2-end production pipeline

- Data Pipeline



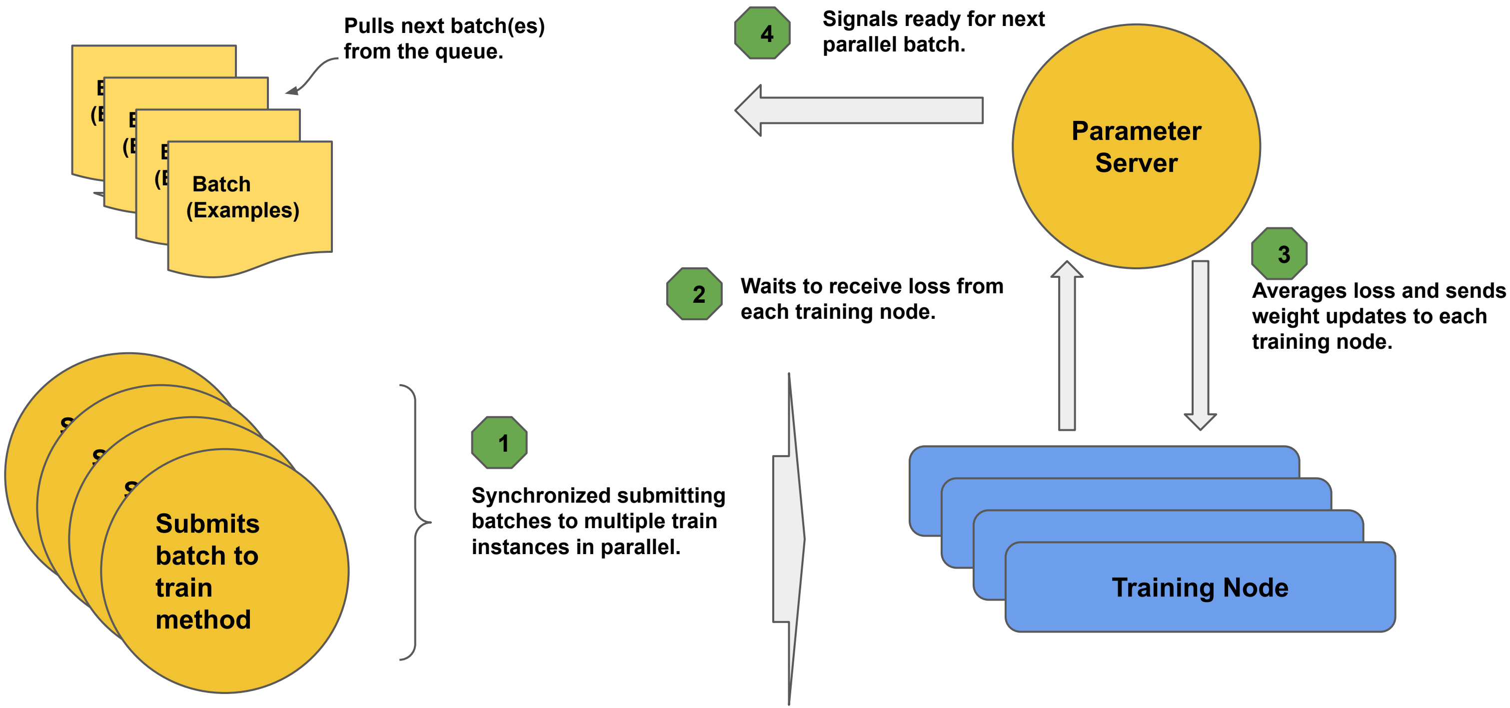
ML end-2-end production pipeline

- Training Pipeline



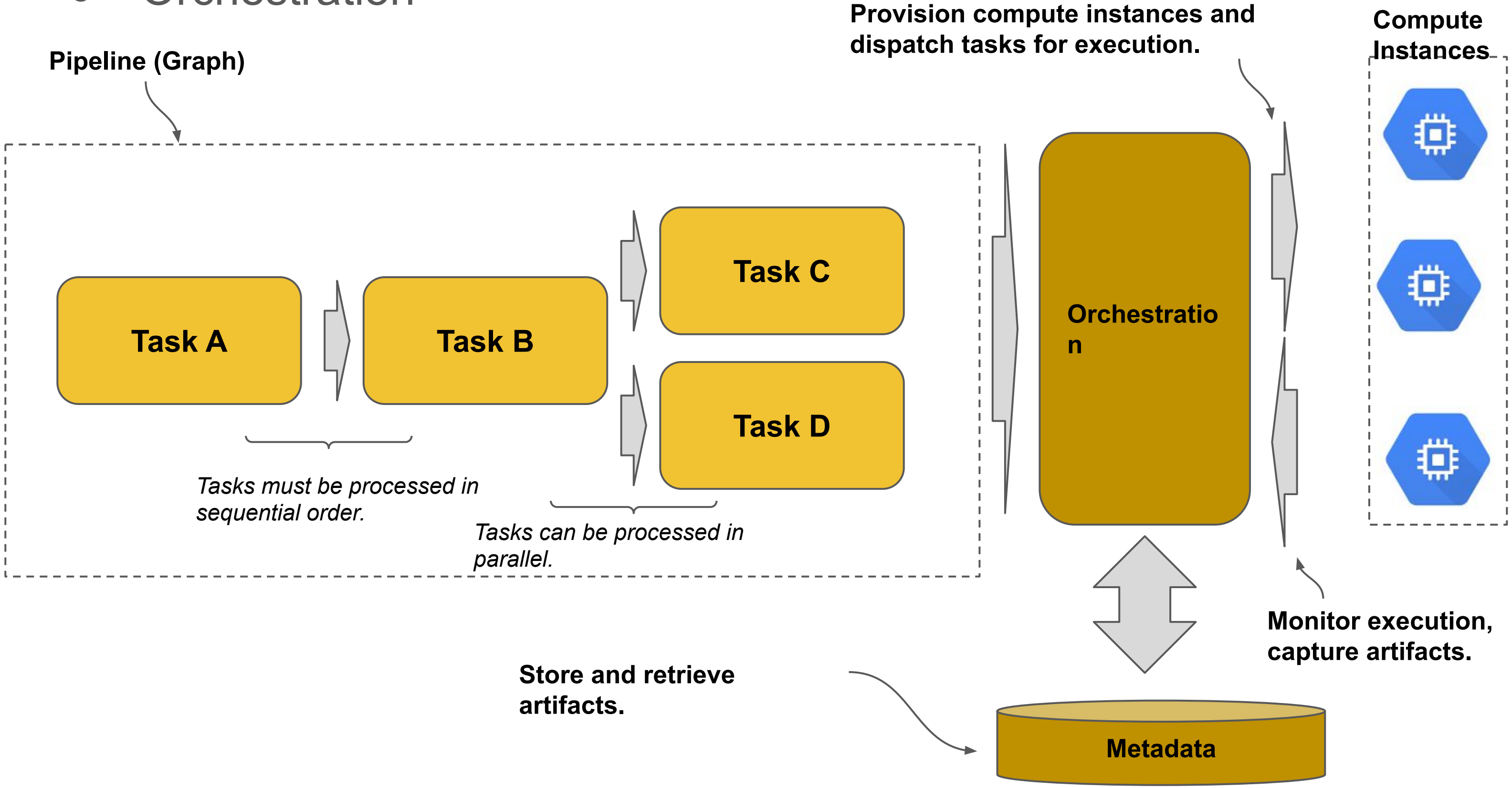
ML end-2-end production pipeline

- Training Pipeline



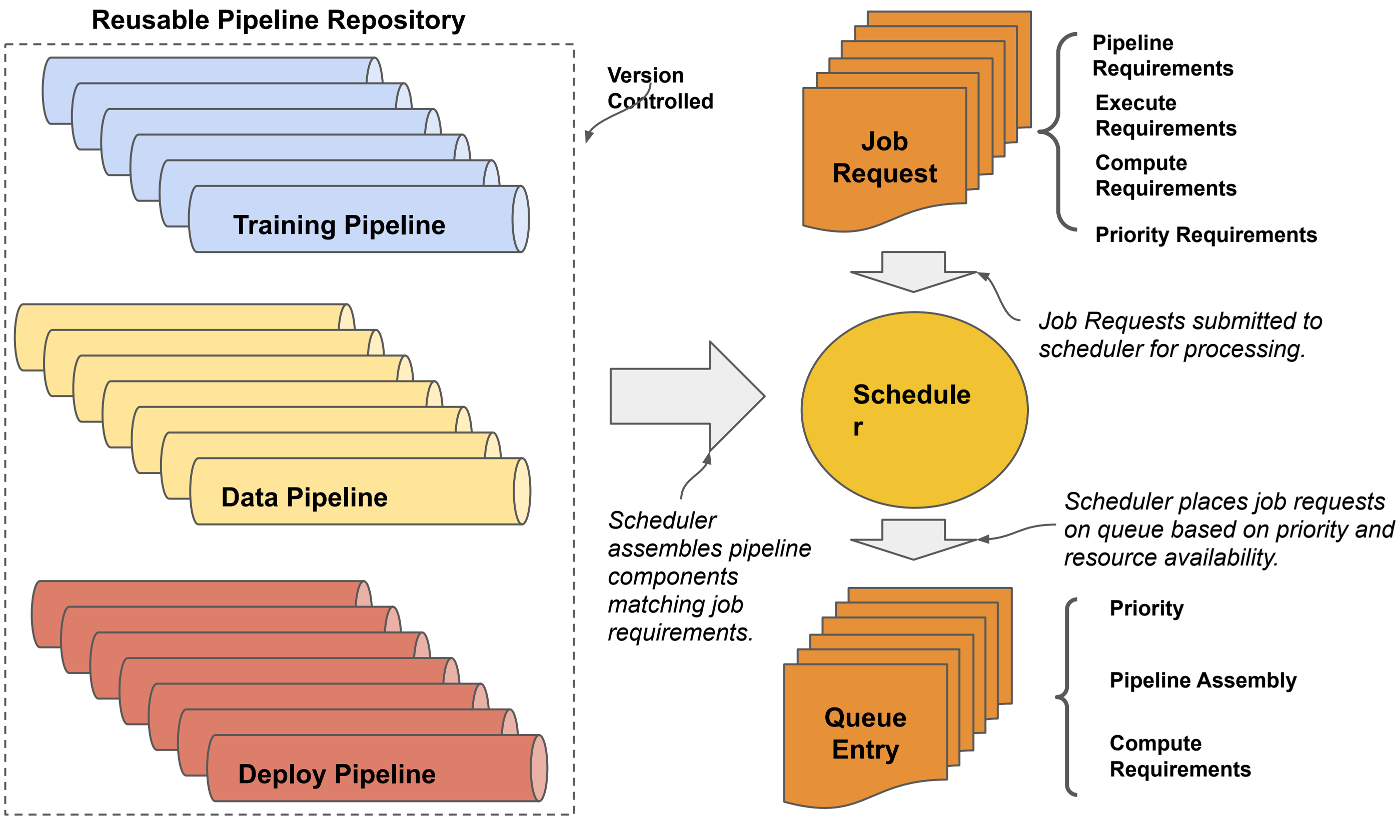
ML end-2-end production pipeline

- Orchestration



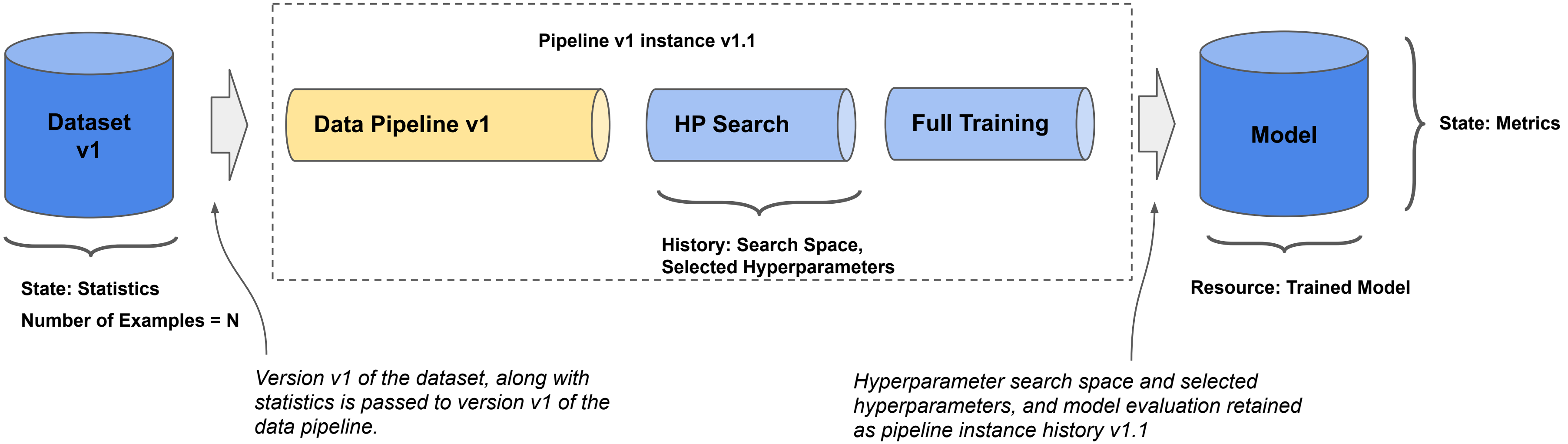
ML end-2-end production pipeline

- Pipeline Components



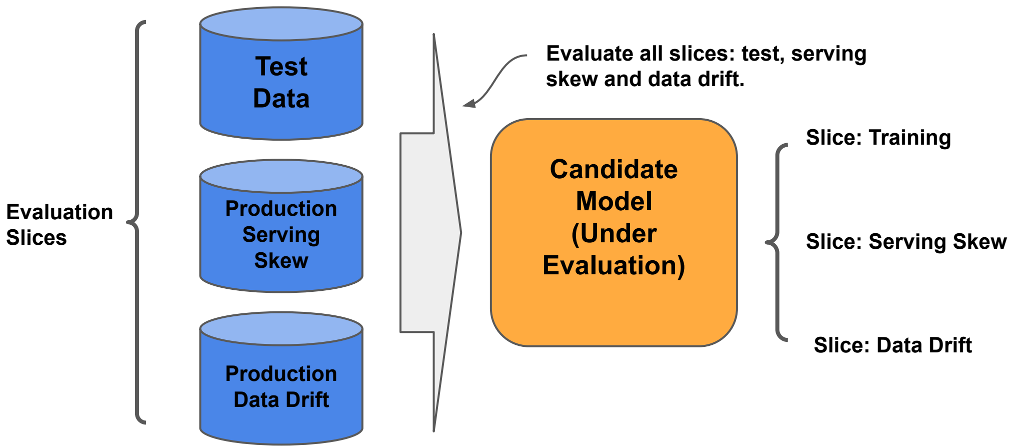
ML end-2-end production pipeline

- Heuristics



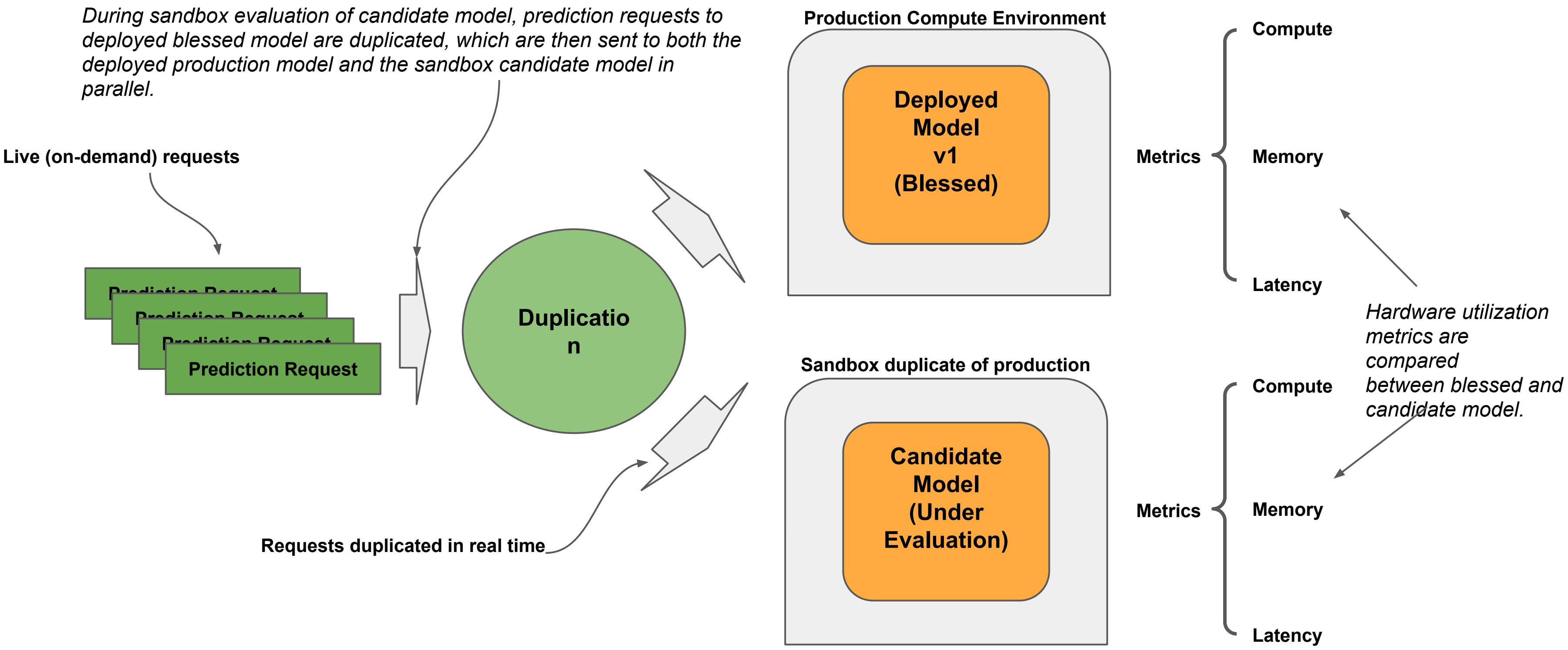
ML end-2-end production pipeline

- Evaluation Slicing



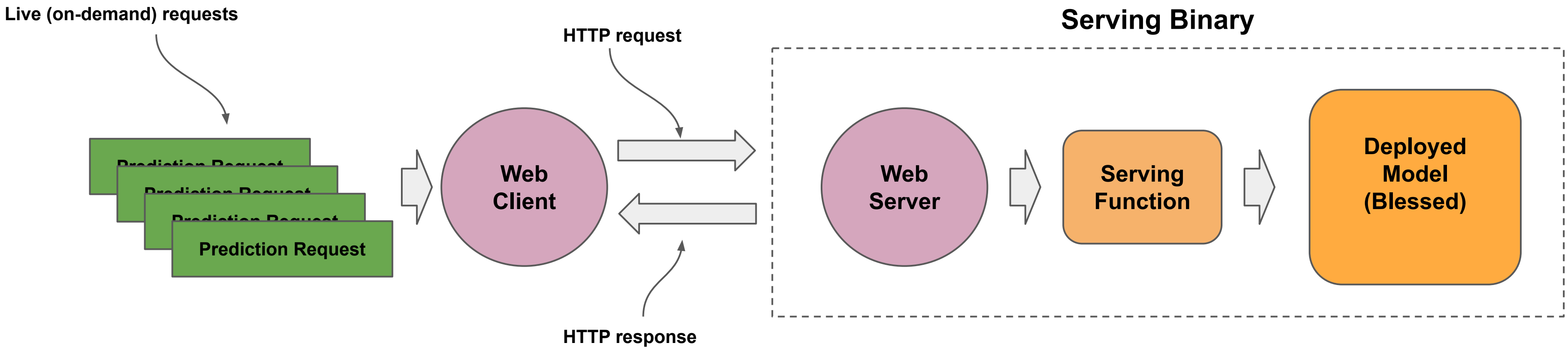
ML end-2-end production pipeline

- Sandboxing



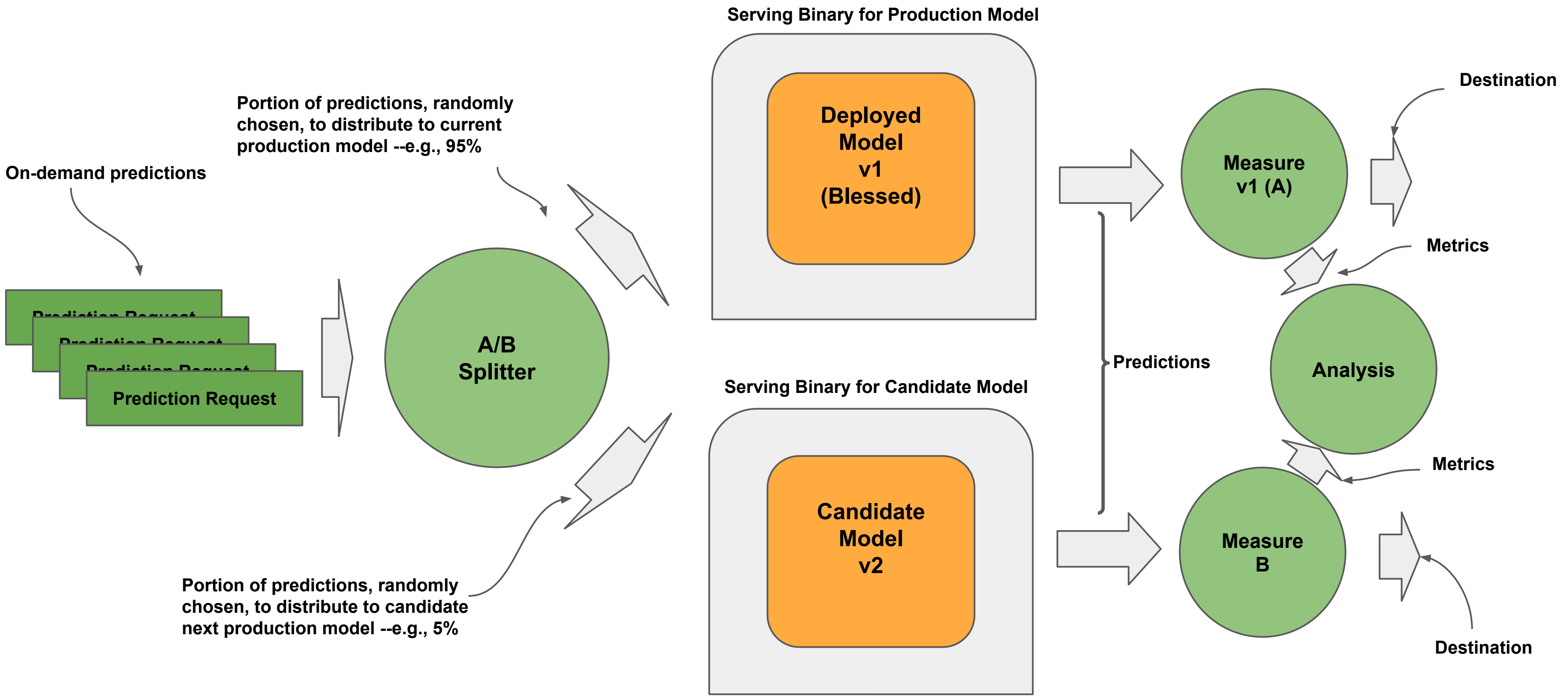
ML end-2-end production pipeline

- Serving Containers



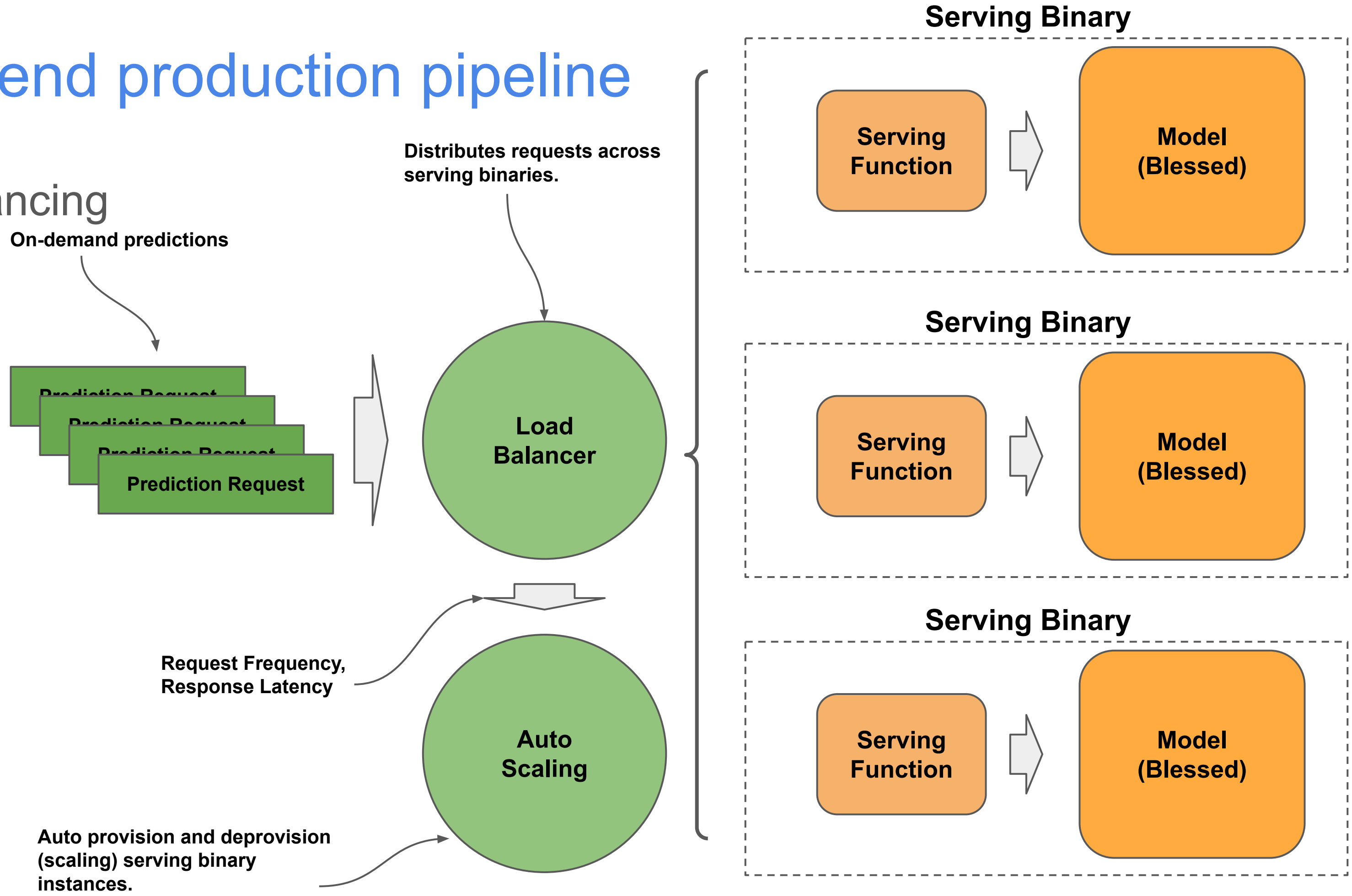
ML end-2-end production pipeline

- A/B Testing



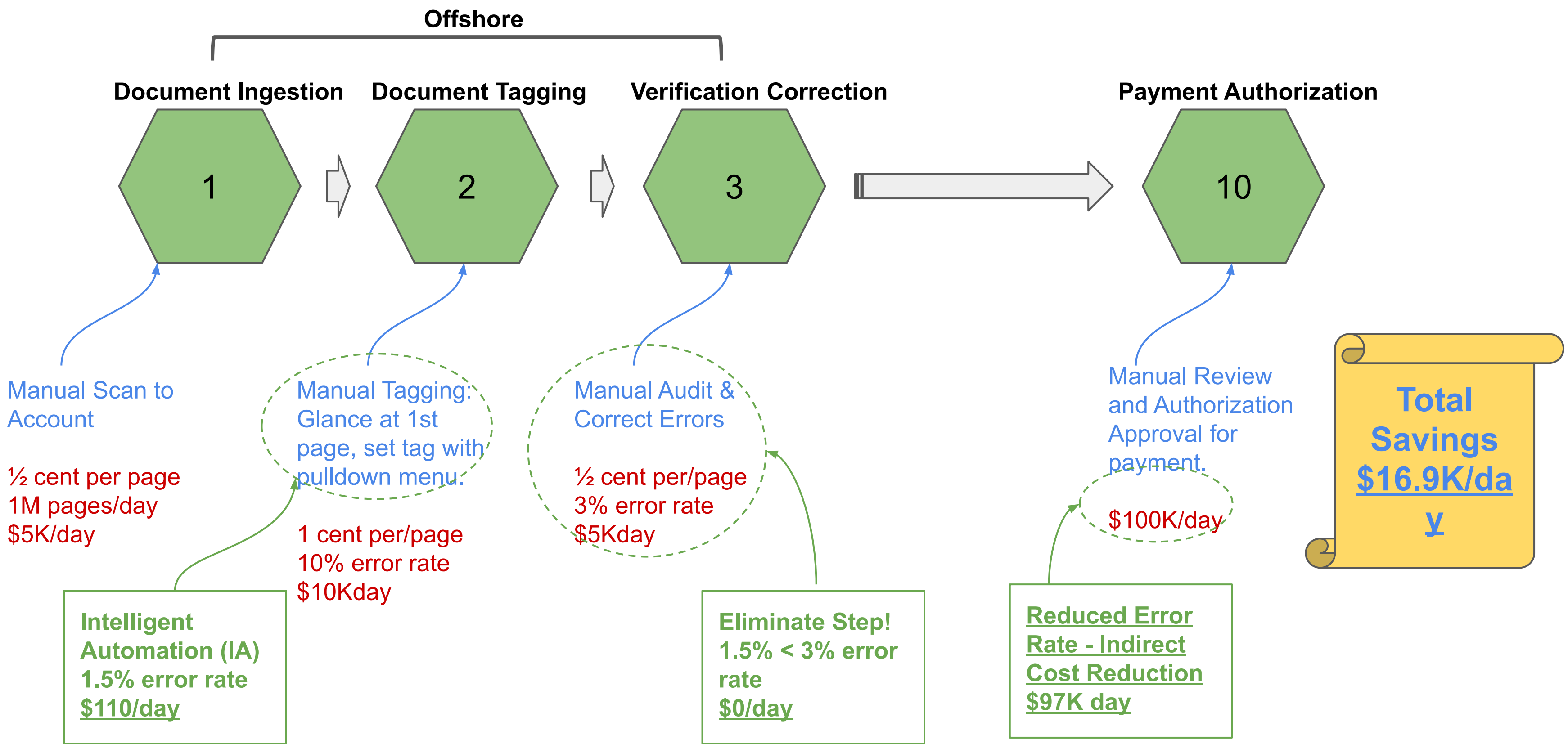
ML end-2-end production pipeline

- Load Balancing

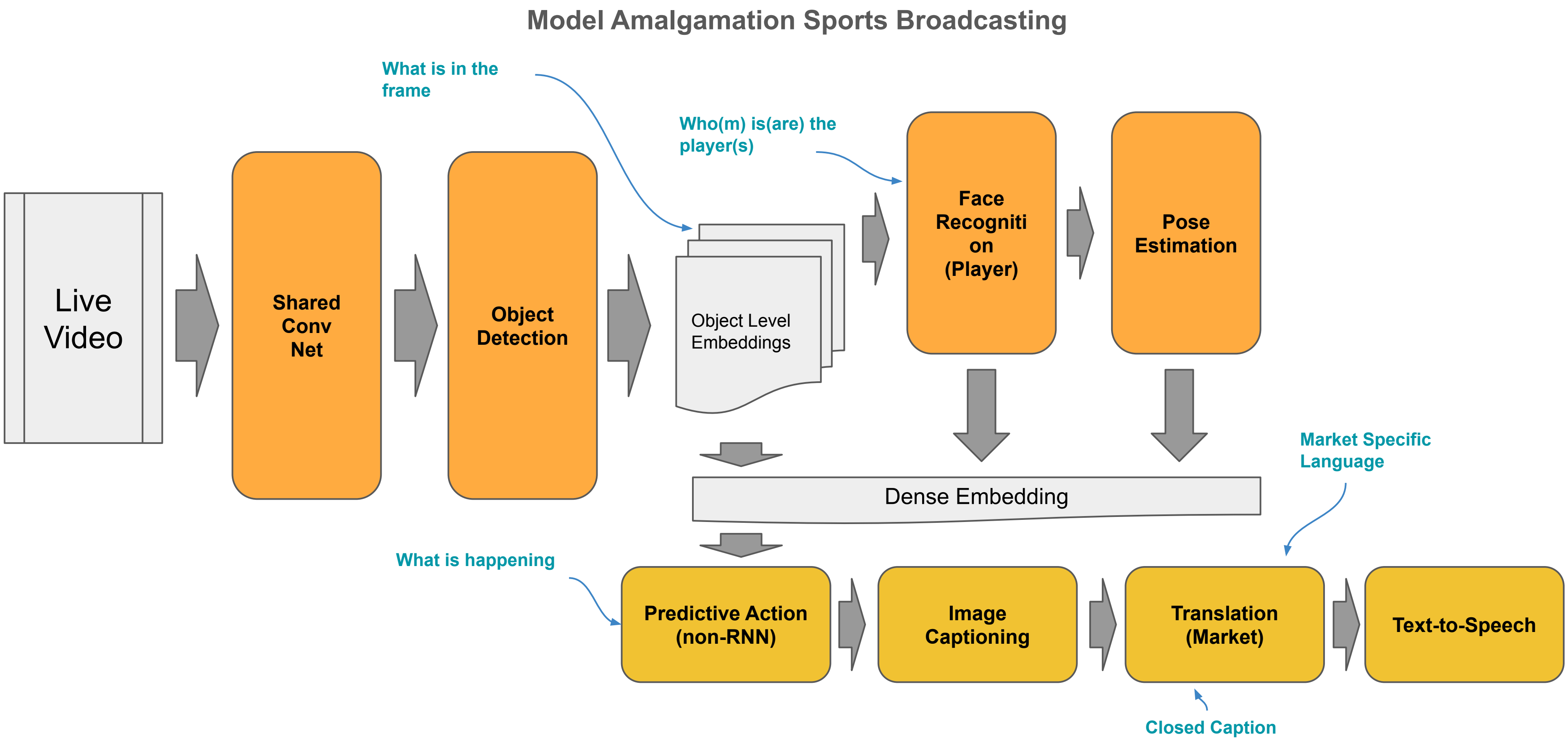


Framing a Business Problem into an e2e Pipeline

Intelligent Automation (IA) Applied to Claim Processing



Framing a Business Problem into an e2e Pipeline



AI Platform (Unified) documentation

Let's visit the official documentation for AI Platform (Unified).

AI Platform (Unified) has the following interfaces:

- User Interface
- Command Line (gcloud)
- REST
- **Client Library (SDK)**

The link below takes you to the home page:
https://cloud.google.com/ai-platform-unified/docs/start/introduction-unified-platform

AI Platform (Unified) walk thru

Let’s now to the AI Platform (Unified) dashboard (UI). I will walk you through:

- Menu options and selections.
- Creating notebook instances.
- Start/Stop/Open notebook instance.
- Resources: Dataset, Model, Endpoint, Pipeline, etc

Reducing Costs

- Notebook Instance
  - You don’t need a GPU for this training course, so don’t select (pay) for one.
  - Select standard instance: 4 vCPUs, 15 GB RAM
  - You pay for each hour the instance is running.
    - 14 cents/hour, ~$3.36/day
  - Shutdown the instance when not using it (from UI console).

Reducing Costs

- Deployed Models
  - You pay for each hour a model is deployed.
  - Deploy the model to the lowest HW configuration
    - single node, n1-standard-4, CPU
  - After an exercise, undeploy the model (optionally from UI console).
  - Custom Models
    - 19 cents/hour, ~$4.50/day
  - AutoML Models are more pricey
    - image classification: $1.25/hour, $30/day
    - object detection: $1.82/hour, $44/day
    - Text models: 5 cents/hour, $1.20/day
    - Tabular models: same as custom, $4.50/day
  - Deployed models get billed a minimum of one hour

Reducing Costs

- Training
  - AutoML Training
    - Image models: $3.15/hour
    - Text models: $3.00/hour
    - Tabular models: $19/hour
    - Video models: $2.94/hour
    - Edge models
      - Classification: $5/hour
      - Object Detection: $18/hour
    - Use very small size datasets

  - Custom Training
    - 19 cents/hour
    - Do only a few epochs

- https://cloud.google.com/ai-platform-unified/pricing

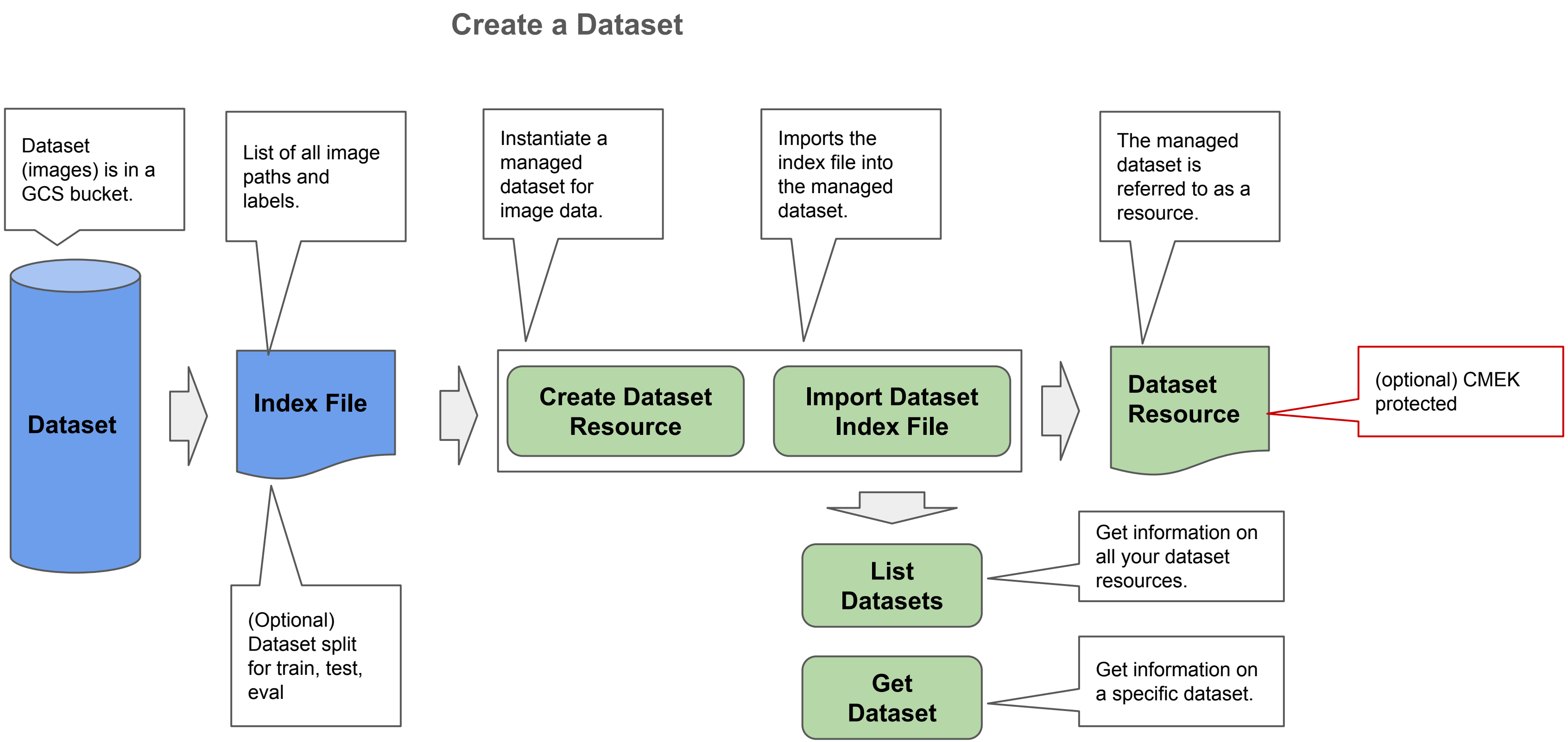
Reducing Costs

- Strategy for workshop notebooks
  - AutoML
    - follow along (execute) upto training
    - From training on, read only
  - Custom Jobs
    - Execute entire notebook

Workshop 1: AutoML Image Classification

- Create a dataset
- Train a model
- Evaluate the model
- Deploy the model for serving
- Do online prediction

Workshop 1: AutoML Image Classification



Create Dataset Resource

Step 1:
- Instantiate a Dataset resource
- Specify schema for data type
- Optionally user-defined metadata.

Step 2:
- Create an instance of the Dataset resource.

Step 3:
- Wait for instance to be created, ~15secs

```
def create_dataset(name, schema, labels=None, timeout=TIMEOUT):
    start_time = time.time()
    try:
        dataset = aip.Dataset(display_name=name,
                              metadata_schema_uri="gs://" + schema,
                              labels=labels)

        operation = clients['dataset'].create_dataset(parent=PARENT, dataset=dataset)

        print("Long running operation:", operation.operation.name)
        result = operation.result(timeout=TIMEOUT)
        print("time:", time.time() - start_time)
        print("response")
        print(" name:", result.name)
        print(" display_name:", result.display_name)
        print(" metadata_schema_uri:", result.metadata_schema_uri)
        print(" metadata:", dict(result.metadata))
        print(" create_time:", result.create_time)
        print(" update_time:", result.update_time)
        print(" etag:", result.etag)
        print(" labels:", dict(result.labels))
        return result
    except Exception as e:
        print("exception:", e)
        return None


result = create_dataset("flowers-" + TIMESTAMP, DATA_SCHEMA)
```

Import Dataset Index File

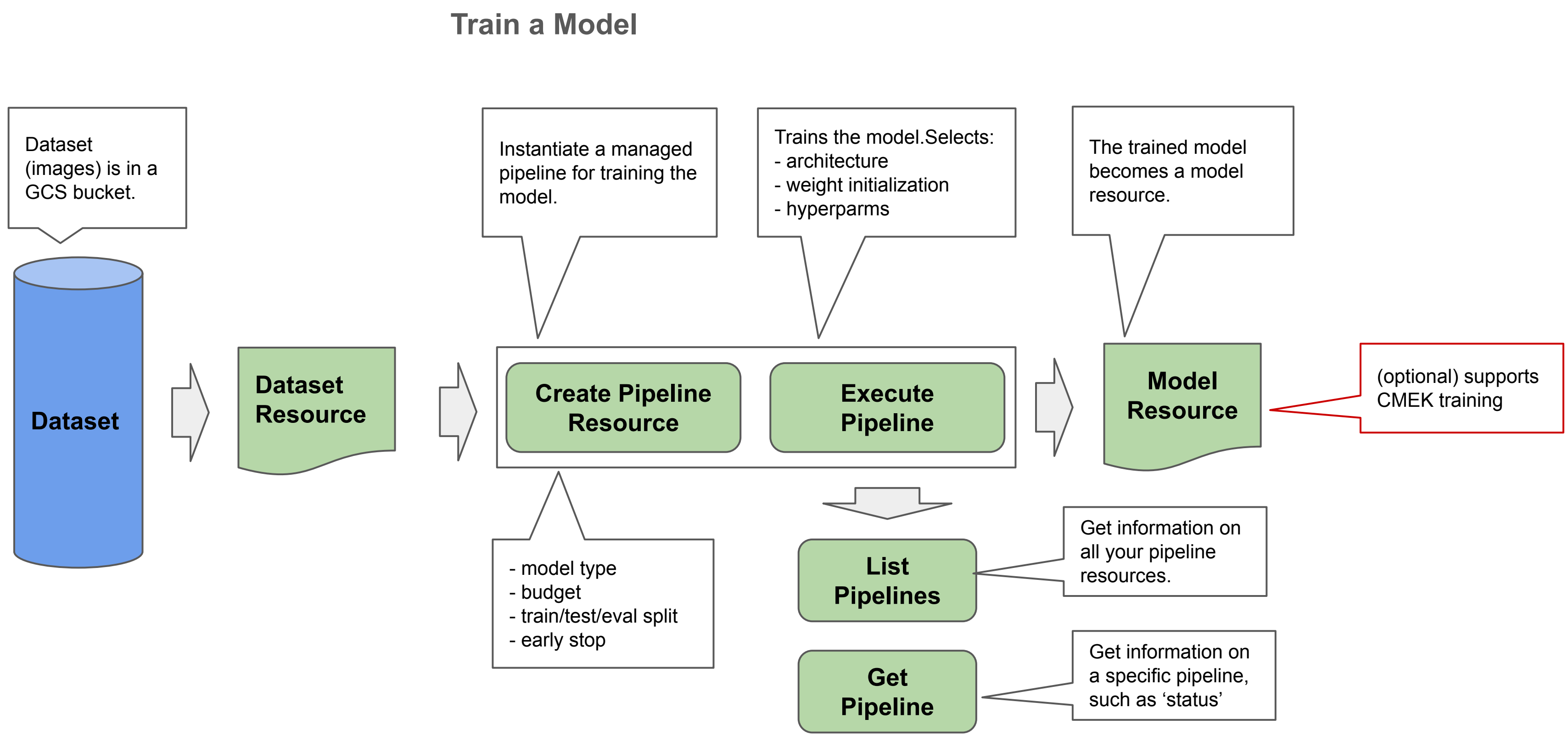
Step 1:
- Set data labeling schema
- Specify one or more index files.

Step 2:
- Import the data.

Step 3:
- Wait for import to complete. Typically a few minutes.

```
def import_data(dataset, gcs_sources, schema):
    config = [{
        'gcs_source': {'uris': gcs_sources},
        'import_schema_uri': schema
    }]

    print("dataset:", dataset_id)
    start_time = time.time()
    try:
        operation = clients['dataset'].import_data(name=dataset_id,
import_configs=config)
        print("Long running operation:", operation.operation.name)

        result = operation.result()
        print("result:", result)
        print("time:", int(time.time() - start_time), "secs")
        print("error:", operation.exception())
        print("meta :", operation.metadata)
        print("after: running:", operation.running(),
                  "done:", operation.done(),
                  "cancelled:", operation.cancelled())

        return operation
    except Exception as e:
        print("exception:", e)
        return None


import_data(dataset_id, [IMPORT_FILE], LABEL_SCHEMA)
```

Workshop 1: AutoML Image Classification



Create Pipeline Resource

Step 1: Specify the training data input
- Specify the dataset
- Specify the training split.

Step 2: Specify the training pipeline.
- Specify training schema
- Specify task requirements
- Specify training data input
- Human readable name for pipeline and uploaded model.

Step 3:
- Start the training ~ asynchronous

```
def create_pipeline(pipeline_name, model_name, dataset, schema, task):

    dataset_id = dataset.split('/')[-1]

    input_config = {'dataset_id': dataset_id,
                    'fraction_split': {
                        'training_fraction': 0.8,
                        'validation_fraction': 0.1,
                        'test_fraction': 0.1
                    }}

    training_pipeline = {
        "display_name": pipeline_name,
        "training_task_definition": schema,
        "training_task_inputs": task,
        "input_data_config": input_config,
        "model_to_upload": {"display_name": model_name},
    }

    try:
        pipeline = clients['pipeline'].create_training_pipeline(parent=PARENT,
                                                                training_pipeline=training_pipeline)
        print(pipeline)
    except Exception as e:
        print("exception:", e)
        return None
    return pipeline
```

Execute Pipeline

Step 1: Query for the training job status.

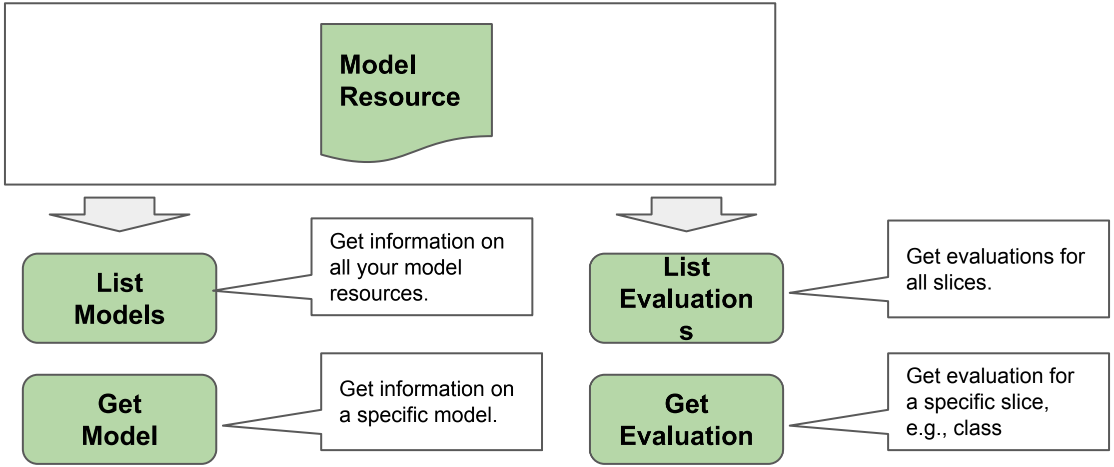
Step 2: return the status

Step 3: Check for status completion. Will automatically deploy trained model to endpoint for serving

```
def get_training_pipeline(name, silent=False):
    response = clients['pipeline'].get_training_pipeline(name=name)
    if silent:
        return response

    print("pipeline")
    print(" name:", response.name)
    print(" display_name:", response.display_name)
    print(" state:", response.state)
    print(" training_task_definition:", response.training_task_definition)
    print(" training_task_inputs:", dict(response.training_task_inputs))
    print(" create_time:", response.create_time)
    print(" start_time:", response.start_time)
    print(" end_time:", response.end_time)
    print(" update_time:", response.update_time)
    print(" labels:", dict(response.labels))
    return response

while True:
    response = get_training_pipeline(pipeline_id, True)
    if response.state != aip.PipelineState.PIPELINE_STATE_SUCCEEDED:
        print("Training job has not completed:", response.state)
        model_to_deploy_id = None
        if response.state == aip.PipelineState.PIPELINE_STATE_FAILED:
            raise Exception("Training Job Failed")
    else:
        model_to_deploy = response.model_to_upload
        model_to_deploy_id = model_to_deploy.name
        print("Training Time:", response.end_time - response.start_time)
        break
    time.sleep(60)

print("model to deploy:", model_to_deploy_id)
```

Workshop 1: AutoML Image Classification

Evaluate the Model



List Models

Get Model

Step 1: Query for information on all trained models (AutoML and Custom)

Step 2: Iterate through the list of model information.

Step 3: Get information on a specific model.

```
def list_models():
  response = clients['model'].list_models(parent=PARENT)
  for model in response:
    print("name", model.name)
    print("display_name", model.display_name)
    print("create_time", model.create_time)
    print("update_time", model.update_time)
    print("container", model.container_spec.image_uri)
    print("artifact_uri", model.artifact_uri)
    print('\n')
    return response


list_models()



def get_model(name):
  response = clients['model'].get_model(name=name)
  print(response)


get_model(model_to_deploy_name)
```

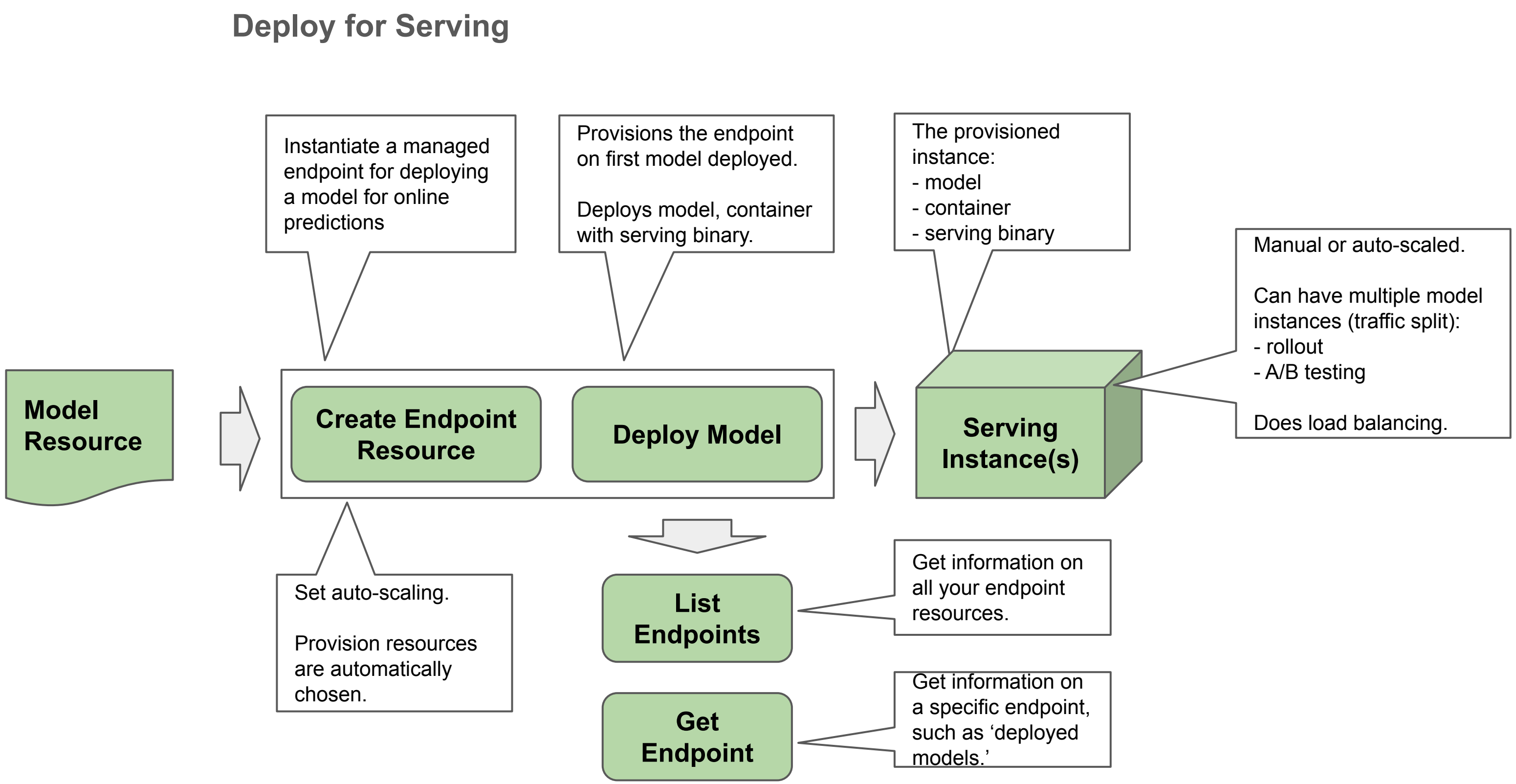
List Evaluations

Step 1: Query for evaluations on all slices of the test/eval data (e.g., by class)

Step 2: Iterate through the list of evaluation slices.

```
def list_model_evaluations(name):
    response = clients['model'].list_model_evaluations(parent=name)
    for evaluation in response:
        print("model_evaluation")
        print(" name:", evaluation.name)
        print(" metrics_schema_uri:", evaluation.metrics_schema_uri)
        metrics = json_format.MessageToDict(evaluation._pb.metrics)
        for metric in metrics.keys():
            print(metric)
        print('logloss', metrics['logLoss'])
        print('auPrc', metrics['auPrc'])


    return response


list_model_evaluations(model_to_deploy_id)
```

Workshop 1: AutoML Image Classification



Create Endpoint Resource

Step 1: Create Endpoint resource. Automatically chooses HW for deployment.

Step 2: Wait for endpoint to be created.

Step 3: Get the endpoint ID

```
def create_endpoint(display_name):
    endpoint = {"display_name": display_name}
    response = clients['endpoint'].create_endpoint(parent=PARENT,
                                                    endpoint=endpoint)
    print("Long running operation:", response.operation.name)

    result = response.result(timeout=300)
    print("result")
    print(" name:", result.name)
    print(" display_name:", result.display_name)
    print(" description:", result.description)
    print(" labels:", result.labels)
    print(" create_time:", result.create_time)
    print(" update_time:", result.update_time)
    return result
```

Deploy Model

Step 1: Specify the model to deploy, and manual/auto-scaling settings.

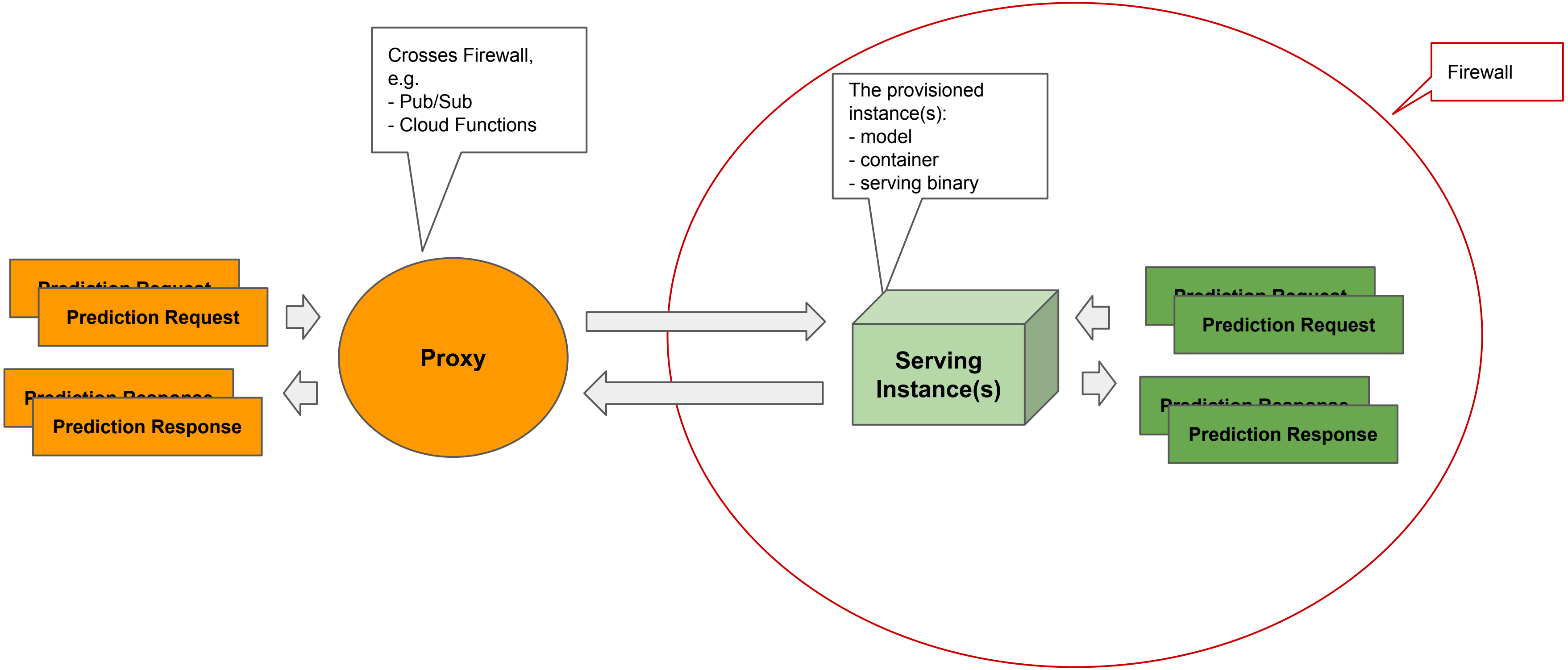
Step 2:
- Specify the traffic split
- Deploy the model

Step 3:
- Wait for model deployed to complete.

```
def deploy_model(model, deployed_model_display_name, endpoint,
                 traffic_split={"0": 100}):

    deployed_model = {
        "model": model,
        "display_name": deployed_model_display_name,
        "automatic_resources": {
            "min_replica_count": MIN_NODES,
            "max_replica_count": MAX_NODES
        },
    }


    response = clients['endpoint'].deploy_model(
        endpoint=endpoint, deployed_model=deployed_model, traffic_split=traffic_split)

    print("Long running operation:", response.operation.name)
    result = response.result()
    print("result")
    deployed_model = result.deployed_model
    print(" deployed_model")
    print("  id:", deployed_model.id)
    print("  model:", deployed_model.model)
    print("  display_name:", deployed_model.display_name)
    print("  create_time:", deployed_model.create_time)

    return deployed_model.id
```

Workshop 1: AutoML Image Classification

Do Online Predictions



Serving

Step 1: Get compressed image bytes

Step 2:
- base64 encode the image

Step 3:
- Construct list of instances to predict.

Step 4:
- Make prediction request
- Set parameters for returning results.

```
def predict_item(filename, endpoint, parameters_dict):

    parameters = json_format.ParseDict(parameters_dict, Value())

    with tf.io.gfile.GFile(filename, "rb") as f:
        content = f.read()

    instances_list = [{"content": base64.b64encode(content).decode("utf-8")}]
    instances = [json_format.ParseDict(s, Value()) for s in instances_list]

    response = clients['prediction'].predict(endpoint=endpoint, instances=instances,
parameters=parameters)
    print("response")
    print(" deployed_model_id:", response.deployed_model_id)
    predictions = response.predictions
    print("predictions")
    for prediction in predictions:
        print(" prediction:", dict(prediction))


predict_item(test_item, endpoint_id,
                          {'confidenceThreshold' : 0.5, 'maxPredictions': 2})
```

Workshop 2: AutoML Image Batch, IOD, ISG, Edge

- Create a batch job for image classification
- Train an image object detection model
- Train an image segmentation
- Export a model for Edge prediction
- Do edge prediction

Workshop 2: AutoML Batch Prediction

Make Batch File

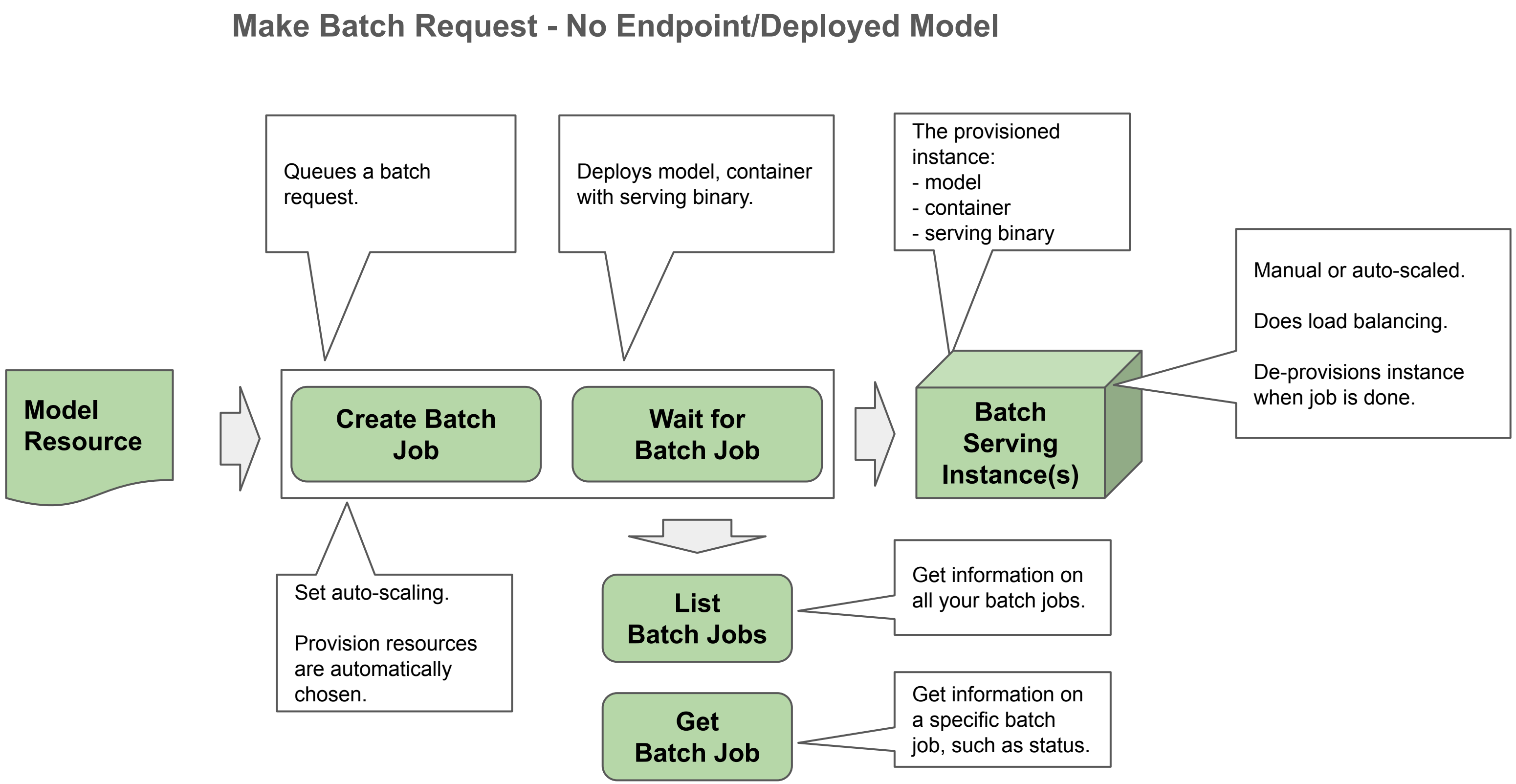
Step 1: Set paths to the images stored in GCS

Step 2: Create JSONL file on GCS

Step 3: Write each instance to predict as a JSON object.

```
test_item_1 = BUCKET_NAME + "/" + file_1
test_item_2 = BUCKET_NAME + "/" + file_2

import tensorflow as tf
import json

gcs_input_uri = BUCKET_NAME + '/test.jsonl'
with tf.io.gfile.GFile(gcs_input_uri, 'w') as f:
    data = {"content": test_item_1, "mime_type": "image/jpeg"}
    f.write(json.dumps(data) + '\n')
    data = {"content": test_item_2, "mime_type": "image/jpeg"}
    f.write(json.dumps(data) + '\n')
```

Workshop 2: AutoML Batch Prediction



Create Batch Job

Step 1: Specify HW resources for each VM instance.

Step 2: Create requirements spec for batch job.

Step 3: Specify one or more batch input files as a list.

Step 4: Specify location on GCS to store the predictions

Step 5: Set manual/auto scaling

Step 6: Submit the batch job

```
def create_batch_prediction_job(display_name, model_name, gcs_source_uri,
                                gcs_destination_output_uri_prefix, parameters):

  if DEPLOY_GPU:
    machine_spec = {
      "machine_type": DEPLOY_COMPUTE,
      "accelerator_type": DEPLOY_GPU,
      "accelerator_count": DEPLOY_NGPU,
    }
  else:
    machine_spec = {
      "machine_type": DEPLOY_COMPUTE,
      "accelerator_count": 0,
    }

  batch_prediction_job = {
    "display_name": display_name,
    "model": model_name,
    "model_parameters": json_format.ParseDict(parameters, Value()),
    "input_config": {
      "instances_format": IN_FORMAT,
      "gcs_source": {"uris": [gcs_source_uri]},
    },
    "output_config": {
      "predictions_format": OUT_FORMAT,
      "gcs_destination": {"output_uri_prefix": gcs_destination_output_uri_prefix},
    },
    "dedicated_resources": {
      "machine_spec": machine_spec,
      "starting_replica_count": MIN_NODES,
      "max_replica_count": MAX_NODES
    }
  }
  response = clients['job'].create_batch_prediction_job(
    parent=PARENT, batch_prediction_job=batch_prediction_job
  )
   return response


IN_FORMAT = 'jsonl'
OUT_FORMAT = 'jsonl'  # [jsonl]

response = create_batch_prediction_job(BATCH_MODEL, model_to_deploy_id, gcs_input_uri, BUCKET_NAME,
                     {'confidenceThreshold': 0.5, 'maxPredictions': 2})
```

Workshop 2: AutoML Image Object Detection

Train Image Object Detection

Image Object Detection (IOD) - Schema

```
# Image Dataset type
DATA_SCHEMA = 'google-cloud-aiplatform/schema/dataset/metadata/image_1.0.0.yaml'
# Image Labeling type
LABEL_SCHEMA = "gs://google-cloud-aiplatform/schema/dataset/ioformat/image_bounding_box_io_format_1.0.0.yaml"
# Image Training task
TRAINING_SCHEMA =
"gs://google-cloud-aiplatform/schema/trainingjob/definition/automl_image_object_detection_1.0.0.yaml"
```

LABEL and TRAINING SCHEMA specific to IOD

Image Object Detection (IOD) - Labeling

For image object detection, the CSV index file has the requirements:

- No heading.
- First column is the Cloud Storage path to the image.
- Second column is the label.
- Third/Fourth columns are the upper left corner of bounding box. Coordinates are normalized, between 0 and 1.
- Fifth/Sixth/Seventh columns are not used and should be 0.
- Eighth/Ninth columns are the lower right corner of the bounding box.

Additional columns for defining the bounding box.

Every bounding box has a separate entry (row).

Image Object Detection (IOD) - Task Requirements

```
task = json_format.ParseDict({'budget_milli_node_hours': 20000,
                        'model_type': "CLOUD",
                        'disable_early_stopping': False
                       }, Value())

response = create_pipeline(PIPE_NAME, MODEL_NAME, dataset_id, TRAINING_SCHEMA, task)
```

Same as ICN, except no multi-label.

model_type:
- CLOUD
- TF_MOBILE_LATENCY_1
- TF_MOBILE_ACCURACY_1
- TF_MOBILE_VERSATILE_1

Image Object Detection (IOD) - Prediction

The `response` object returns a list, where each element in the list corresponds to the corresponding image in the request. You will see in the output for each prediction:

- \- Confidence level in the prediction (confidences).
- \- The predicted label (displayNames).
- \- The bounding box for the label (bboxes).

Additional output for the bounding box of each predicted object label.

Image Object Detection (IOD) - Batch Prediction

Same as image classification

For JSONL file, you make one dictionary entry per line for each data item (instance). The dictionary contains the key/value pairs:

- content: The Cloud Storage path to the image.
- mime_type: The content type. In our example, it is an jpeg file.

For example:

{'content': '[your-bucket]/file1.jpg', 'mime_type': 'jpeg'}

Workshop 2: AutoML Image Segmentation

Image Segmentation (ISG) - Schema

```
# Image Dataset type
DATA_SCHEMA = 'google-cloud-aiplatform/schema/dataset/metadata/image_1.0.0.yaml'
# Image Labeling type
LABEL_SCHEMA = "gs://google-cloud-aiplatform/schema/dataset/ioformat/image_segmentation_io_format_1.0.0.yaml"
# Image Training task
TRAINING_SCHEMA =
"gs://google-cloud-aiplatform/schema/trainingjob/definition/automl_image_segmentation_1.0.0.yaml"
```

LABEL and TRAINING SCHEMA specific to ISG

Image Segmentation (ISG) - Labeling

For image segmentation, the JSONL index file has the requirements:

- - Each data item is a separate JSON object, on a separate line.
- - The key/value pair `image_gcs_uri` is the Cloud Storage path to the image.
- - The key/value pair `category_mask_uri` is the Cloud Storage path to the mask image in PNG format.
- - The key/value pair `'annotation_spec_colors'` is a list mapping mask colors to a label.
-   - The key/value pair pair `display_name` is the label for the pixel color mask.
-   - The key/value pair pair `color` are the RGB normalized pixel values (between 0 and 1) of the mask for the corresponding label.

{ 'image_gcs_uri': image, 'segmentation_annotations': { 'category_mask_uri': mask_image, 'annotation_spec_colors' : [{ 'display_name': label, 'color': {"red": value, "blue", value, "green": value} }, ...] }

All fields except for image path are specific to segmentation

Cleaner to specify as JSON than as CSV.

Image Segmentation (ISG) - Task Requirements

```
task = json_format.ParseDict({'budget_milli_node_hours': 2000,
                              'model_type': "CLOUD_LOW_ACCURACY_1"
                             }, Value())

response = create_pipeline(PIPE_NAME, MODEL_NAME, dataset_id, TRAINING_SCHEMA, task)
```

Model type specific to ISG.

Select either high or low accuracy tradeoff for size/latency.
CLOUD_HIGH_ACCURACY_1

Image Segmentation (ISG) - Prediction

The `response` object returns a list, where each element in the list corresponds to the corresponding image in the request. You will see in the output for each prediction:

- - ConfidenceMask - Confidence level in the prediction
- - CategoryMask - Predictions per pixel.

Output is on a per pixel basis

Image Segmentation (ISG) - Batch Prediction

For JSONL file, you make one dictionary entry per line for each data item (instance). The dictionary contains the key/value pairs:

- content: The Cloud Storage path to the image.
- mime_type: The content type. In our example, it is an jpeg file.

For example:

{'content': '[your-bucket]/file1.jpg', 'mime_type': 'jpeg'}

Same as image classification

Workshop 2: AutoML Image Models, Export to Edge

Deploy for Edge Serving

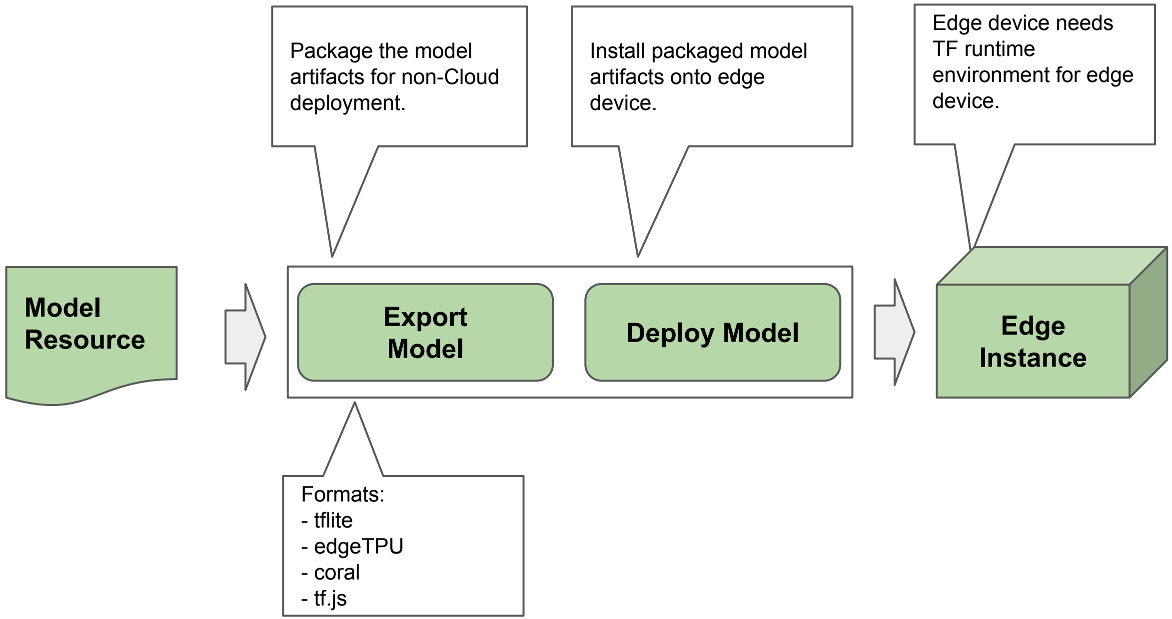


Image Model Exported to Edge - Training

```
PIPE_NAME = "salads_pipe-" + TIMESTAMP
MODEL_NAME = "salads_model-" + TIMESTAMP

task = json_format.ParseDict({'budget_milli_node_hours': 20000,
                              'model_type': "MOBILE_TF_LOW_LATENCY_1",
                              'disable_early_stopping': False
                             }, Value())

response = create_pipeline(PIPE_NAME, MODEL_NAME, dataset_id, TRAINING_SCHEMA, task)
```

Model Type are specific to edge models:
- MOBILE_TF_LOW_LATENCY_1
- MOBILE_TF_HIGH_ACCURACY_1
- MOBILE_TF_VERSATILE_1

Can train edge model for:
- image classification
- object detection

Image Model Exported to Edge - Export

```
def export_model(name, format, gcs_dest):
    output_config = {
        "artifact_destination": {"output_uri_prefix": gcs_dest},
        "export_format_id": format,
    }
    response = clients['model'].export_model(name=name, output_config=output_config)
    print("Long running operation:", response.operation.name)
    result = response.result(timeout=1800)
    metadata = response.operation.metadata
    artifact_uri = str(metadata.value).split("\\\\")[-1][4:-1]
    print("Artifact Uri", artifact_uri)
    return artifact_uri


model_package = export_model(model_to_deploy_id, "tflite", MODEL_DIR)
```

Specify format and GCS location to export the edge packaged model artifacts.

Image Model Exported to Edge - TFLite Interpreter

```
import tensorflow as tf

interpreter = tf.lite.Interpreter(model_path=tflite_path)
interpreter.allocate_tensors()

input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
input_shape = input_details[0]['shape']

print("input tensor shape", input_shape)
```

Instantiate TFLite interpreter for edge model.

TFLite run-time environment must be installed on edge device.

Run-time is smaller than TF run-time to fit into smaller memory.

Image Model Exported to Edge - Image Resizing

```
test_items = ! gsutil cat $IMPORT_FILE | head -n1
test_item = test_items[0].split(',')[0]

with tf.io.gfile.GFile(test_item, "rb") as f:
    content = f.read()
test_image = tf.io.decode_jpeg(content)
print("test image shape", test_image.shape)

test_image = tf.image.resize(test_image, (224, 224))
print("test image shape", test_image.shape, test_image.dtype)

test_image = tf.cast(test_image, dtype=tf.uint8).numpy()
```

Must resize the image to the edge model input size, either upstream or on edge device.

Image Model Exported to Edge - Prediction

Step 1: Fit image instance into a batch.

Step 2: Bind the batch to the input tensor.

Step 3: Invoke the model to do the prediction.

Step 4: Get the prediction output.

```
import numpy as np

data = np.expand_dims(test_image, axis=0)    # np.asarray([test_image])

interpreter.set_tensor(input_details[0]['index'], data)

interpreter.invoke()

softmax = interpreter.get_tensor(output_details[0]['index'])

label = np.argmax(softmax)

print(label)
```

Workshop 3: Text Models

- Text Classification
- Text Sentiment Analysis
- Text Entity Extraction

Workshop 3: AutoML Text Classification

Text Classification (TCN) - Schema

```
# Text Dataset type
DATA_SCHEMA = 'google-cloud-aiplatform/schema/dataset/metadata/text_1.0.0.yaml'
# Text Labeling type
LABEL_SCHEMA =
"gs://google-cloud-aiplatform/schema/dataset/ioformat/text_classification_single_label_io_format_1.0.0.yaml"
# Text Training task
TRAINING_SCHEMA = "gs://google-cloud-aiplatform/schema/trainingjob/definition/automl_text_classification_1.0.0.yaml"
```

DATA specific to Text
LABEL and TRAINING SCHEMA
specific to TCN

Text Classification - Labeling

For text classification, the CSV file has a few requirements:

- No heading.
- First column is the text example or GCS path to text file (.txt suffix).
- Second column the label.

Same column fields as image classification.

Data items (examples) are text files.

Text Classification (TCN) - Task Requirements

```
PIPE_NAME = "happydb_pipe-" + TIMESTAMP
MODEL_NAME = "happydb_model-" + TIMESTAMP

task = json_format.ParseDict({'multi_label': False,
                    }, Value())

response = create_pipeline(PIPE_NAME, MODEL_NAME, dataset_id, TRAINING_SCHEMA, task)
```

Cloud only model.
Can pick between single or multi-label classification.

Text Classification (TCN) - Prediction

Format:

{ ‘content’: text_item }

The `response` object returns a list, where each element in the list corresponds to the corresponding text item in the request. You will see in the output for each prediction:

- - Confidence level in the prediction (`confidences`).
- - The predicted label (`displayNames`).

Either text example, or GCS path to text file.

Same as image classification

Text Classification - Batch Prediction

For JSONL file, you make one dictionary entry per line for each data item (instance). The dictionary contains the key/value pairs:

- `content`: The Cloud Storage path to the file with the text item.

- `mime_type`: The content type. In our example, it is an `text` file.

For example:

{'content': '[your-bucket]/file1.txt', 'mime_type': 'text'}

Same as image model, except content is text file.

Workshop 3: AutoML Text Sentiment Analysis

Text Sentiment Analysis (TST) - Schema

```
# Text Dataset type
DATA_SCHEMA = 'google-cloud-aiplatform/schema/dataset/metadata/text_1.0.0.yaml'
# Text Labeling type
LABEL_SCHEMA = "gs://google-cloud-aiplatform/schema/dataset/ioformat/text_sentiment_io_format_1.0.0.yaml"
# Text Training task
TRAINING_SCHEMA = "gs://google-cloud-aiplatform/schema/trainingjob/definition/automl_text_sentiment_1.0.0.yaml"
```

LABEL and TRAINING SCHEMA specific to TST

Text Sentiment Analysis - Labeling

For text sentiment analysis, the CSV file has a few requirements:

- No heading.
- First column is the text example or GCS path to text file.
- Second column the label (i.e., sentiment).
- Third column is the maximum sentiment value. For example, if the range is 0 to 3, then the maximum value is 3.

Label column is the sentiment.

Has additional column for the maximum possible sentiment value.

Text Sentiment Analysis (TST) - Task Requirements

```
PIPE_NAME = "claritin_pipe-" + TIMESTAMP
MODEL_NAME = "claritin_model-" + TIMESTAMP

task = json_format.ParseDict({'sentiment_max' : SENTIMENT_MAX,
                    }, Value())

response = create_pipeline(PIPE_NAME, MODEL_NAME, dataset_id, TRAINING_SCHEMA, task)
```

Cloud only model.

Specify the maximum sentiment.

Text Sentiment Analysis - Prediction

Same as text classification

The sentiment rating

Format:

{ ‘content’: text_item }

The response object returns a list, where each element in the list corresponds to the corresponding image in the request. You will see in the output for each prediction -- in our case there is just one:

- The sentiment rating

Text Sentiment Analysis - Batch Prediction

Same as text classification

For JSONL file, you make one dictionary entry per line for each data item (instance). The dictionary contains the key/value pairs:

- `content`: The Cloud Storage path to the file with the text item.

- `mime_type`: The content type. In our example, it is an `text` file.

For example:

{'content': '[your-bucket]/file1.txt', 'mime_type': 'text'}

Workshop 3: AutoML Text Entity Extraction

Text Entity Extraction (TEN) - Schema

```
# Text Dataset type
DATA_SCHEMA = 'google-cloud-aiplatform/schema/dataset/metadata/text_1.0.0.yaml'
# Text Labeling type
LABEL_SCHEMA = "gs://google-cloud-aiplatform/schema/dataset/ioformat/text_extraction_io_format_1.0.0.yaml"
# Text Training task
TRAINING_SCHEMA = "gs://google-cloud-aiplatform/schema/trainingjob/definition/automl_text_extraction_1.0.0.yaml"
```

LABEL and TRAINING SCHEMA specific to TEN

Text Entity Extraction - Labeling

For text entity extraction, the JSONL file has a few requirements:

- Each data item is a separate JSON object, on a separate line.
- The key/value pair `text_segment_annotations` is a list of character start/end positions in the text per entity with the corresponding label.
 - `display_name`: The label.
 - `start_offset/end_offset`: The character offsets of the start/end of the entity.
- The key/value pair `text_content` is the text.

For example:
 {'text_segment_annotations': [{'end_offset': value, 'start_offset': value, 'display_name': label}, ...], 'text_content': text}

Each entity is specified with a start and end position in the text.

Text Entity Extraction (TEN) - Task Requirements

```
task = json_format.ParseDict({'multi_label': False,
                              'budget_milli_node_hours': 8000,
                              'model_type': "CLOUD",
                              'disable_early_stopping': False
                             }, Value())
```

Cloud only.
Entities can have multiple labels.

Text Entity Extraction - Prediction

Same as text classification

The location of each entity, label of each entity, and confidence score

Format:

{ ‘content’: text_item }

The `response` object returns a list, where each element in the list corresponds to the corresponding data item in the request. You will see in the output for each prediction -- in our case there is just one:

- `prediction`: A list of IDs assigned to each entity extracted from the text.
- `confidences`: The confidence level between 0 and 1 for each entity.
- `display_names`: The label name for each entity.
- `textSegmentStartOffsets`: The character start location of the entity in the text.
- `textSegmentEndOffsets`: The character end location of the entity in the text.

Text Entity Extraction - Batch Prediction

For JSONL file, you make one dictionary entry per line for each data item (instance). The dictionary contains the key/value pairs:

- `content`: The Cloud Storage path to the file with the text item.

- `mime_type`: The content type. In our example, it is an `text` file.

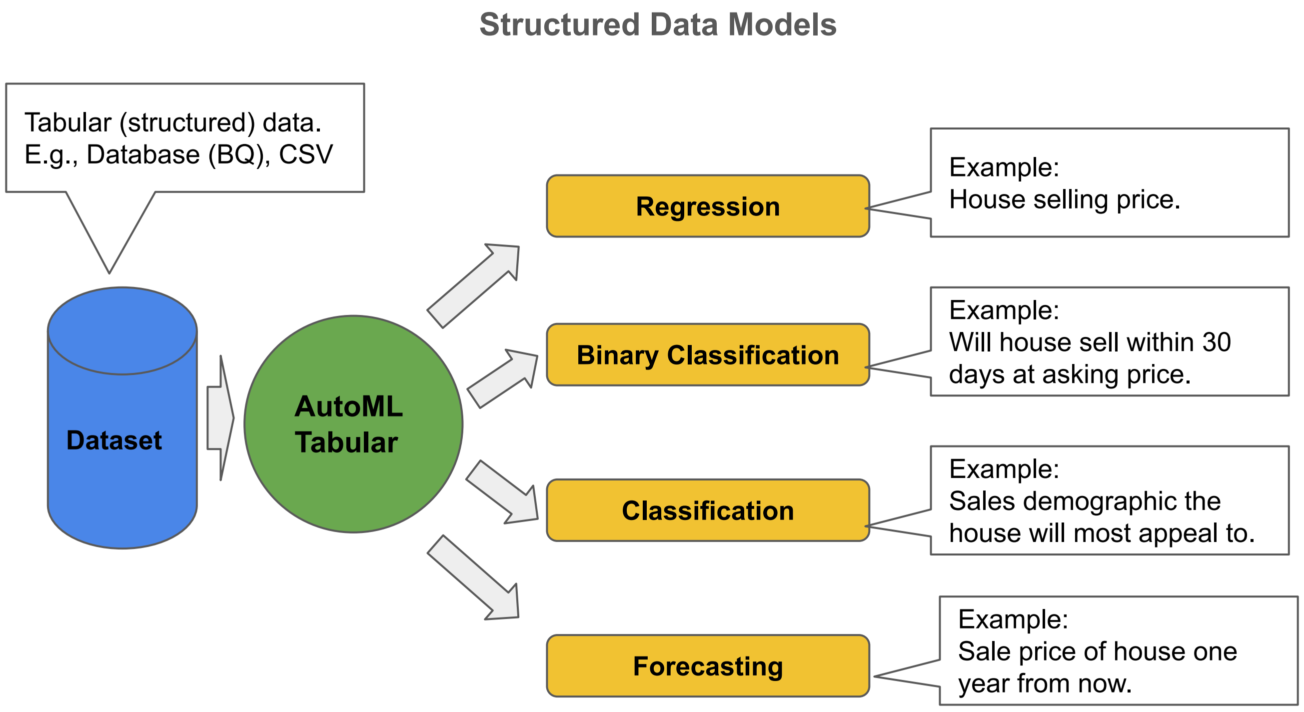
For example:

{'content': '[your-bucket]/file1.txt', 'mime_type': 'text'}

Same as text classification and sentiment analysis.

Workshop 4: Tabular Models

- Models for structured data
- BigQuery input
- Model export for other cloud or on-prem serving


Structured Data Models
Tabular (structured) data.
E.g., Database (BQ), CSV
Dataset
AutoML
Tabular
Regression
Binary Classification
Classification
Forecasting
Example:
House selling price.
Example:
Will house sell within 30
days at asking price.
Example:
Sales demographic the
house will most appeal to.
Example:
Sale price of house one
year from now.

Workshop 4: AutoML Tabular Models

Tabular (LRG, LBN, LCN) - Schema

```
# Tabular Dataset type
DATA_SCHEMA = 'google-cloud-aiplatform/schema/dataset/metadata/tables_1.0.0.yaml'
# Tabular Labeling type
LABEL_SCHEMA = 'gs://google-cloud-aiplatform/schema/dataset/ioformat/table_io_format_1.0.0.yaml'
# Tabular Training task
TRAINING_SCHEMA = "gs://google-cloud-aiplatform/schema/trainingjob/definition/automl_tables_1.0.0.yaml"
```

Schemas specific to Tabular

Schema is the same for regression, binary and classification.

Tabular (CSV) - Labeling

For tabular classification, the CSV file has a few requirements:

- The first row must be the heading

- All but one column are features.

- One column is the label, which you will specify when you subsequently create the training pipeline.

Note how this is different from Vision, Video and Language where the requirement is no heading.

Specific to tabular data.

All rows must have the same number of columns and match the heading.

Tabular (LRG, LBN, LCN) - Task Requirements

1. Name of column that is the label.
2. Type of model (classification, etc)
3. Feature engineering.

```
task = Value(struct_value=Struct(
          fields={
             'target_column': Value(string_value=label_column),
             'prediction_type': Value(string_value="classification"),
             'transformations': json_format.ParseDict(TRANSFORMATIONS, Value())

             'train_budget_milli_node_hours': Value(number_value=1000),
             'disable_early_stopping': Value(bool_value=False),          }
      ))
```

Must specific transformation (feature engineering) for each feature – even if defaulting to automatic.

Tabular (LRG, LBN, LCN) - Prediction Request

Format:

{"feature_1": “value”, "feature_2": “value”, "feature_3”: “value”, ...}

Each feature has a key/value pair.

Values must be specified as strings, even if they are a number.

Tabular (LBN, LCN) - Prediction Response

The `response` object returns a list, where each element in the list corresponds to the corresponding image in the request. You will see in the output for each prediction -- in this case there is just one:

- `confidences`: Confidence level in the prediction.
- `displayNames`: The predicted label.

Same for all classification models

Tabular (LRG) - Prediction Response

The `response` object returns a list, where each element in the list corresponds to the corresponding image in the request. You will see in the output for each prediction -- in this case there is just one:

- `value`: The predicted value.

A real number

Tabular (LRG, LBN, LCN) - Batch Prediction Request

Each instance is a CSV row

Make a batch input file, which you will store in your local #(GCS) bucket. Unlike image, video and text, the batch input file for tabular is only supported for CSV. For CSV file, you make:

- The first line is the heading with the feature (fields) heading names.

- Each remaining line is a separate prediction request with the corresponding feature values.

For example:

```
"feature_1", "feature_2". …
 value_1, value_2, ...
```

Tabular (LCN) - Batch Prediction Response

For each prediction:

- The first nth fields are the values (features) you did the prediction on.
- The remaining fields are the confidence values, between 0 and 1, for each prediction.

The number of fields is the number of features.

The number of fields is the number of classes

Workshop 4: AutoML Tabular Forecasting

Tabular (Forecasting) - Schema

```
# Forecasting Dataset type
DATA_SCHEMA = 'google-cloud-aiplatform/schema/dataset/metadata/time_series_1.0.0.yaml'
# Forecasting Training task
TRAINING_SCHEMA = "gs://google-cloud-aiplatform/schema/trainingjob/definition/automl_forecasting_1.0.0.yaml"
```

Forecasting is also known as time series.

Tabular (Forecasting) - Labeling

Same as other tabular models

Specific to forecasting

For tabular forecasting, the CSV file has a few requirements:

- The first row must be the heading

- All but one column are features.
- One column is the label, which you will specify when you subsequently create the training pipeline.

- One column is the time column, which you will specify when you subsequently create the training pipeline.
- One column is the time series identifier column, which you will specify when you subsequently create the training pipeline.

Tabular (Forecasting) - Task Requirements

```
task = Value(struct_value=Struct(
        fields={
            'target_column': Value(string_value=label_column),
            'train_budget_milli_node_hours': Value(number_value=1000),

            'time_column': Value(string_value=time_column),
            'time_series_identifier_column': Value(string_value=time_series_identifier_column),
            "period": json_format.ParseDict(PERIOD, Value()),
            "forecast_window_end": Value(number_value=10),
            "time_variant_past_only_columns": json_format.ParseDict(PAST_ONLY_COLUMNS, Value()),
            "static_columns": json_format.ParseDict(STATIC_COLUMNS, Value()),
            "time_variant_past_and_future_columns": json_format.ParseDict(PAST_AND_FUTURE_COLUMNS, Value()),
"optimization_objective": Value(string_value=optimization_objective),
            'transformations': json_format.ParseDict(TRANSFORMATIONS, Value())
        }
    ))
```

Specific to forecasting

Tabular (Forecasting) - Batch Prediction

TODO

Workshop 4: AutoML Tabular BQ Input

Create Dataset Resource

Tabular BigQuery Input - Dataset Creation

`metadata = {"input_config": {"bigquery_source": {"uri": [bq_uri]}}}`

The format for a BigQuery path is:

bq://[collection].[dataset].[table]

Note that the `uri` field is a list, whereby you can input multiple CSV files or BigQuery tables when your data is split across files.

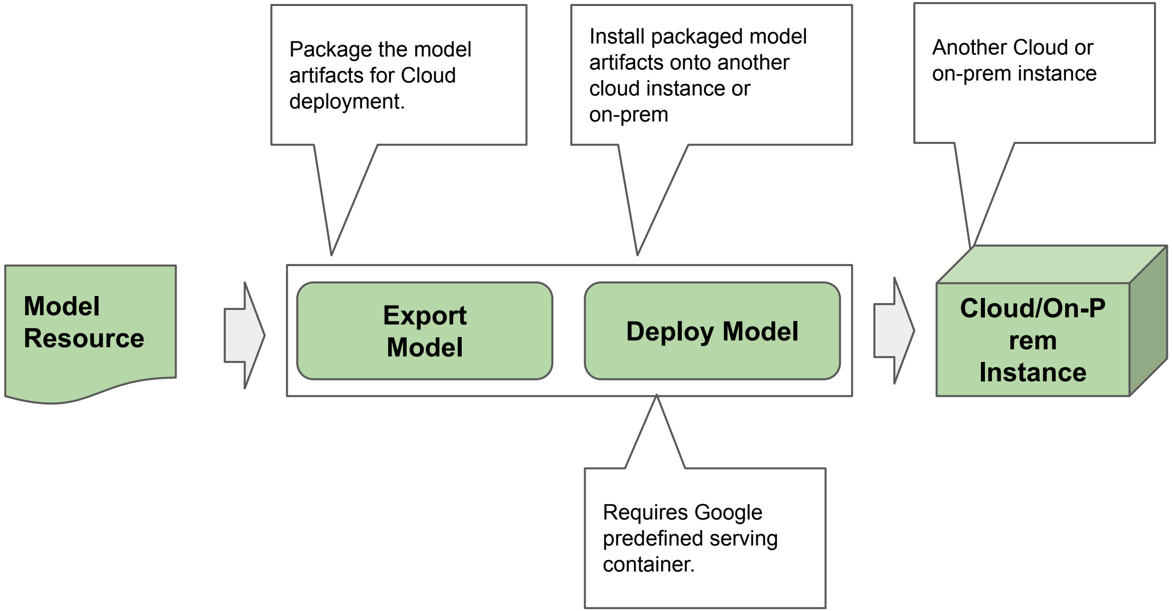
Format for specifying a BigQuery dataset.

Everything else is the same.

Workshop 4: AutoML Tabular Export

Workshop 4: AutoML Tabular Models, Export to Cloud

Deploy for other Cloud/On-Prem Serving



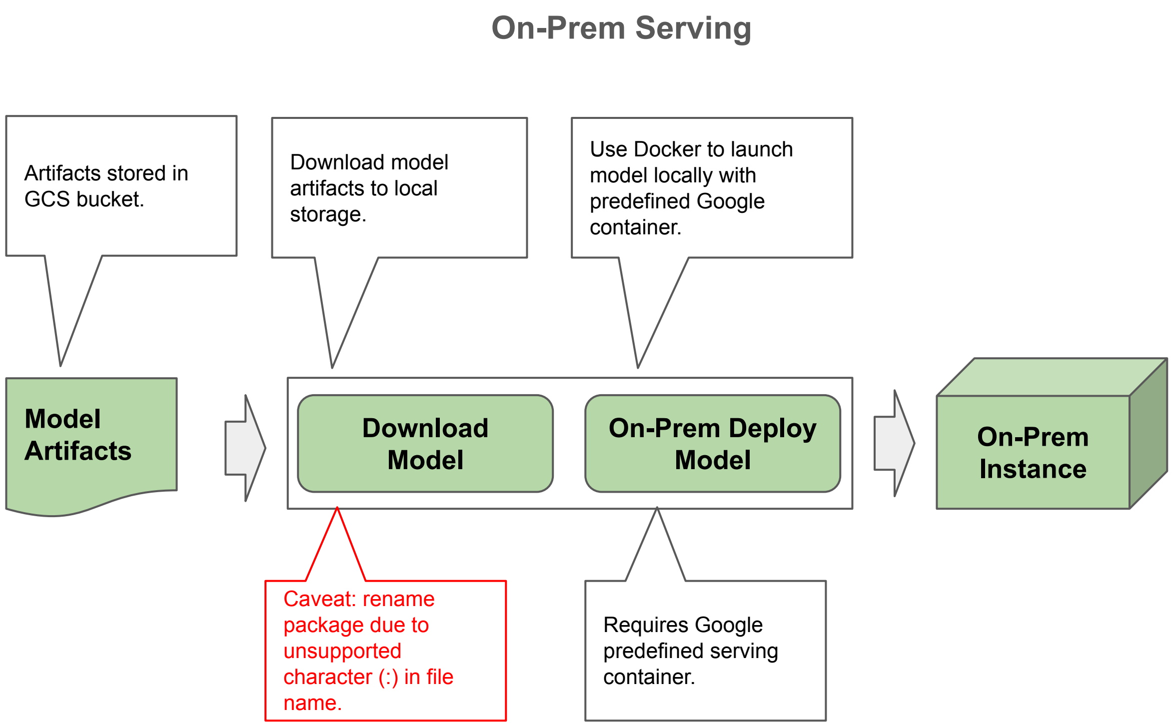
Tabular Model Exported to Cloud - Export

```
def export_model(name, format, gcs_dest):
   output_config = {
      "artifact_destination": {"output_uri_prefix": gcs_dest},
      "export_format_id": format,
   }
   response = clients['model'].export_model(name=name, output_config=output_config)
   print("Long running operation:", response.operation.name)
   result = response.result(timeout=1800)
   metadata = response.operation.metadata
   artifact_uri = str(metadata.value).split("\\")[-1][4:-1]
   print("Artifact Uri", artifact_uri)
   return artifact_uri


model_package = export_model(model_to_deploy_id, "tf-saved-model", MODEL_DIR)
```

Specify format and GCS location to export the model artifacts.

Only TF SavedModel format supported.

Tabular Model Exported to Cloud - On-Prem Execution



On-Prem Deploy Model

Tabular Model Exported to on-Prem - Docker Startup

```
MODEL_SERVER = "gcr.io/cloud-aiplatform/automl_tables/prediction_server"

PORT = 8081

docker_id = ! docker run -d -v `pwd`/tbl_exported:/models/default -p 8081:8080 -it $MODEL_SERVER
```

Google predefined serving container.

Run locally the container with the export model.

Renamed model artifacts folder.

Health Status

Tabular Model Exported to on-Prem - Health Status

```
import time
time.sleep(10)

! curl -X GET http://localhost:8081/health
```

Predefined URL path /health
Acts like a ping.

Healthy is 200 response

Prediction

Tabular Model Exported to on-Prem - Prediction

Format of prediction request.

Predefined URL path /predict

iThe format for the prediction request is a JSON object of the form:

{ "instances": [{ "column_name_1": value, "column_name_2": value, ... } , ...] }

Place your prediction request in a text file, such as:

test.json

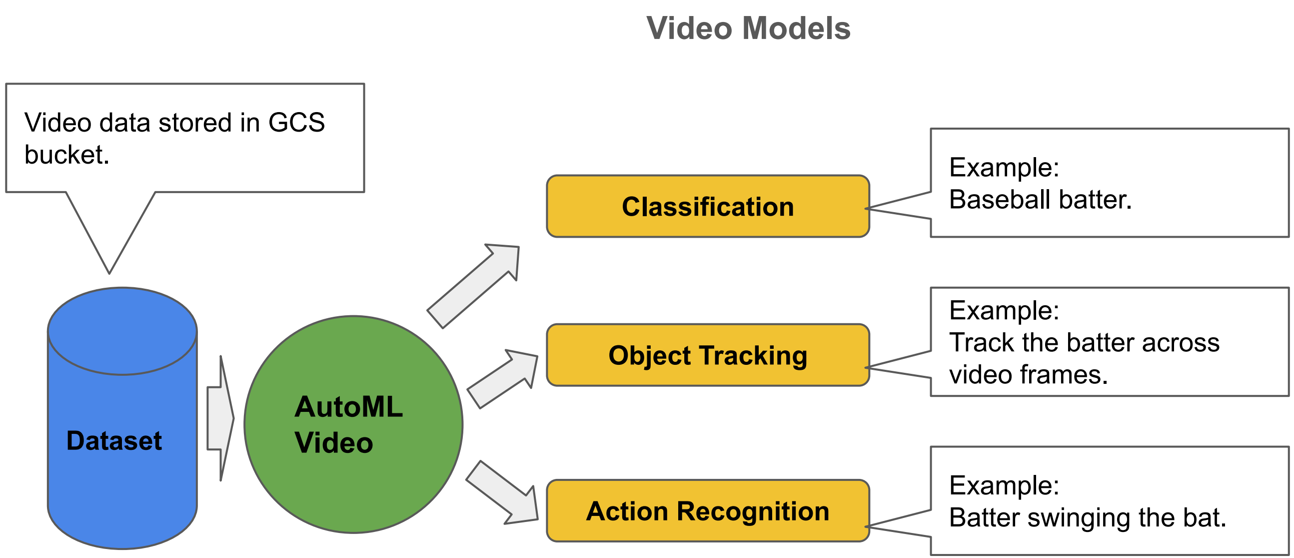
You can the send the prediction request using CURL:

```
curl -X POST --data @test.json http://localhost:8081/predict
```

Workshop 5: AutoML Wrap Up

- AutoML Video
- AutoML Explanablity

Workshop 5: AutoML Video


Video Models
Video data stored in GCS bucket.
Dataset
AutoML Video
Classification
Object Tracking
Action Recognition
Example: Baseball batter.
Example: Track the batter across video frames.
Example: Batter swinging the bat.

Video Models - Summary

- **Training and Prediction: Essentially the same as image models.**

- Differences:
  - Schemas specific to Video models.
  - Datasets include time segments in video.
  - Only batch prediction supported
  - Predictions include time segments.

Video - Schema

```
# Video Dataset type
DATA_SCHEMA = 'google-cloud-aiplatform/schema/dataset/metadata/video_1.0.0.yaml'

# Video Labeling type
LABEL_SCHEMA = "gs://google-cloud-aiplatform/schema/dataset/ioformat/video_classification_io_format_1.0.0.yaml"
# Video Training task
TRAINING_SCHEMA = "gs://google-cloud-aiplatform/schema/trainingjob/definition/automl_video_classification_1.0.0.yaml"

# Video Labeling type
LABEL_SCHEMA = "gs://google-cloud-aiplatform/schema/dataset/ioformat/video_object_tracking_io_format_1.0.0.yaml"
# Video Training task
TRAINING_SCHEMA =
"gs://google-cloud-aiplatform/schema/trainingjob/definition/automl_video_object_tracking_1.0.0.yaml"

# Video Labeling type
LABEL_SCHEMA =
"gs://google-cloud-aiplatform/schema/dataset/ioformat/video_action_recognition_io_format_1.0.0.yaml"
# Video Training task
TRAINING_SCHEMA =
"gs://google-cloud-aiplatform/schema/trainingjob/definition/automl_video_action_recognition_1.0.0.yaml"
```

LABEL and TRAINING schemas specific to model type.

Video Classification (VCN) - Labeling

Common to all video models

Specific to classification

For video action recognition, the CSV index file has a few requirements:

- No heading.
- First column is the Cloud Storage path to the video.

-
- Second column is the label.

Video Object Tracking (VOT) - Labeling

Specific to object tracking

For video action recognition, the CSV index file has a few requirements:

- No heading.
- First column is the Cloud Storage path to the video.

- Second column is the label.
- Third column is **not used**
- Fourth column is **not used**
- Fifth/Sixth columns are the upper left corner of bounding box. Coordinates are normalized, between 0 and 1.
- Seventh/Eigth/Ninth columns are not used and should be 0.
- Tenth/Eleventh columns are the lower right corner of the bounding box.

Video Action Recognition (VAR) - Labeling

For video action recognition, the CSV index file has a few requirements:

- No heading.
- First column is the Cloud Storage path to the video.

- Second column is the time offset for the start of the video segment to analyze.
- Third column is the time offset for the end of the video segment to analyze.
- Fourth column is label for the action (e.g., swing).

Specific to action recognition

Video - Task Requirements

```
task = json_format.ParseDict({'model_type': "CLOUD",
                        }, Value())
```

For all video models

Video - Batch File Format

Same as for image models

For all video models

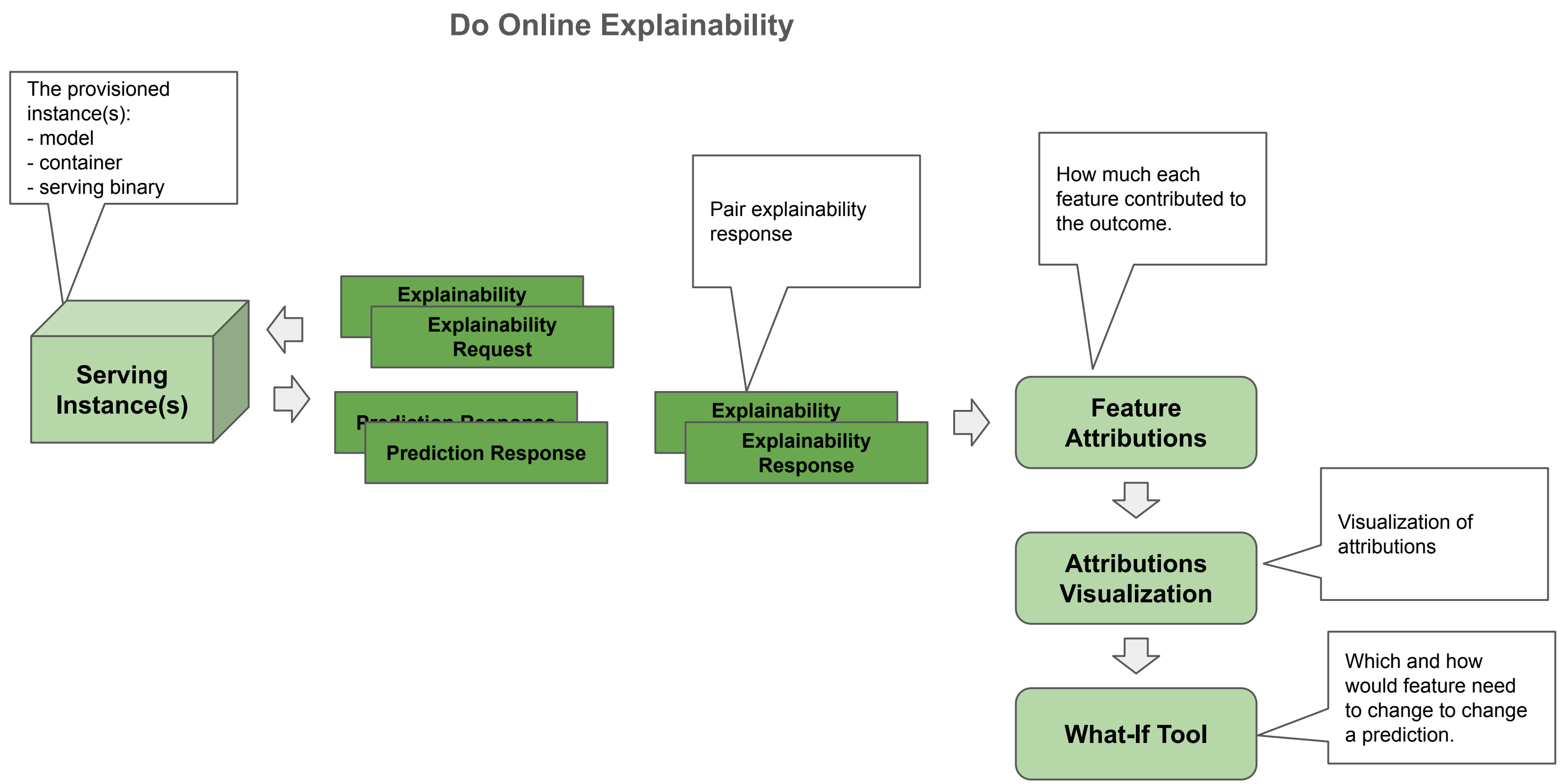
For JSONL file, you make one dictionary entry per line for each video. The dictionary contains the key/value pairs:

- `content`: The Cloud Storage path to the video.
- `mimeType`: The content type. In our example, it is an `avi` file.

- `timeSegmentStart`: The start timestamp in the video to do prediction on. *Note*, the timestamp must be specified as a string and followed by s (second), m (minute) or h (hour).
- `timeSegmentEnd`: The end timestamp in the video to do prediction on.

Workshop 5: AutoML Explainability

Workshop 5: AutoML Explainability



Explainability Request

Tabular Explanation Call/Attributions

```
def explain_item(data_items, endpoint, parameters_dict, deployed_model_id, silent=False):
    parameters = json_format.ParseDict(parameters_dict, Value())

    # The format of each instance should conform to the deployed model's prediction input schema.
    instances = [json_format.ParseDict(s, Value()) for s in data_items]

    response = clients['prediction'].explain(endpoint=endpoint, instances=instances,
                                             parameters=parameters, deployed_model_id=deployed_model_id)
    if silent:
        return response

    print("response")
    print(" deployed_model_id:", response.deployed_model_id)

    explanations = response.explanations
    print("explanations")
    for explanation in explanations:
        print(explanation)
    return response


response = explain_item([INSTANCE], endpoint_id, None, None)
```

Call explain method instead of predict

Feature Attributions

Tabular Explanation Visualization

Package that builds charts

Show in chart how much each feature contributed.

```
from tabulate import tabulate

feature_names = ["petal_length", "petal_width", "sepal_length", "sepal_width"]
attributions = response.explanations[0].attributions[0].feature_attributions

rows = []
for i, val in enumerate(feature_names):
    rows.append([val, INSTANCE[val], attributions[val]])
print(tabulate(rows, headers=['Feature name', 'Feature value', 'Attribution value']))
```

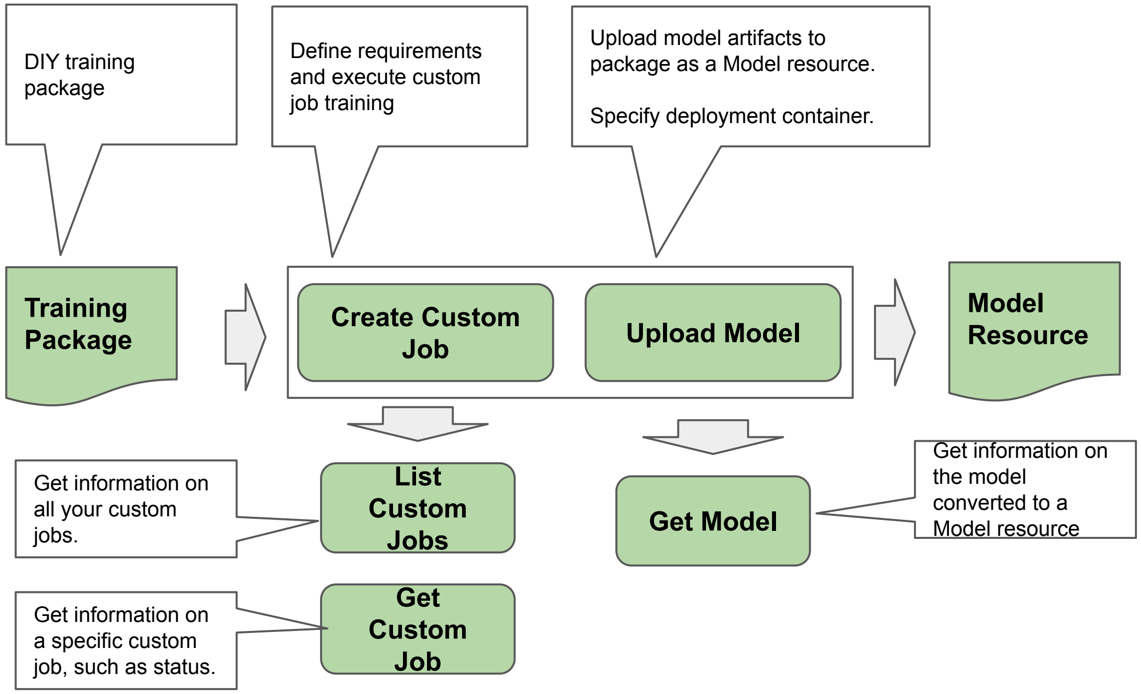
Workshop 6: Custom Jobs

- Custom Jobs Fundamentals
- Serving Functions
- Custom Job Image Classification

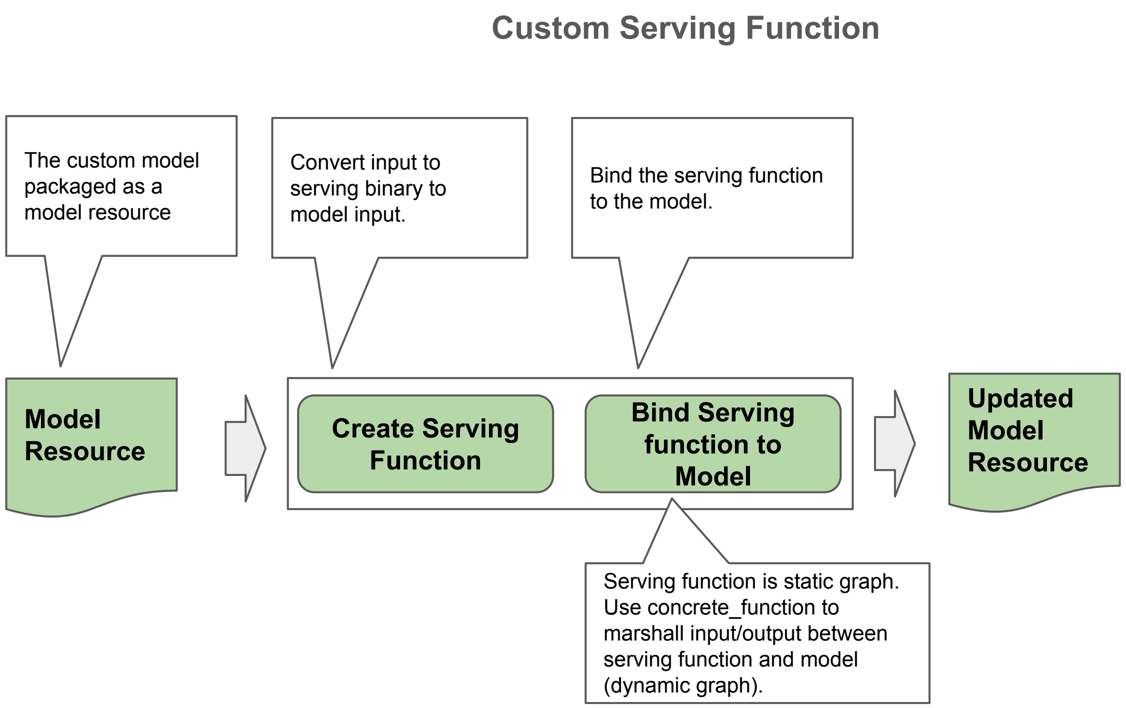
Workshop 6: Custom Job Fundamentals

Custom Jobs for all Model Types

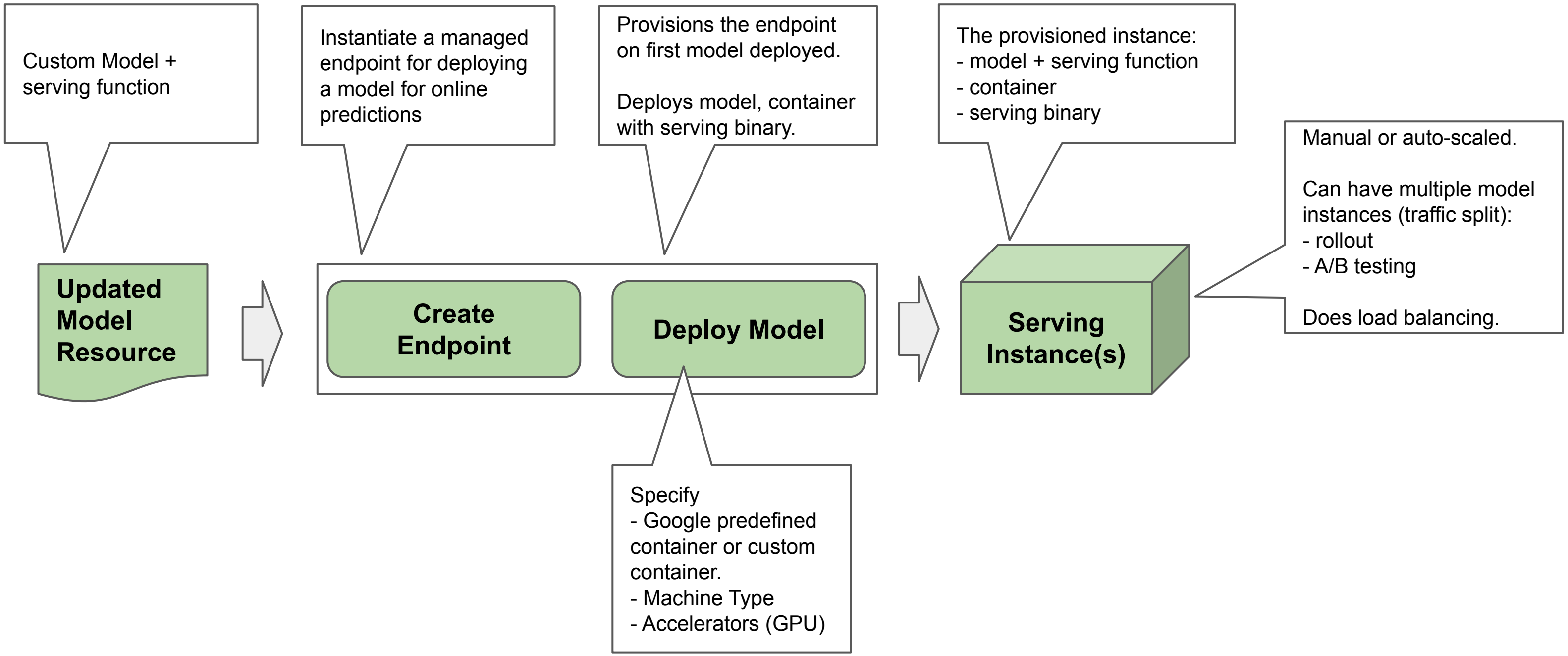
Custom Training



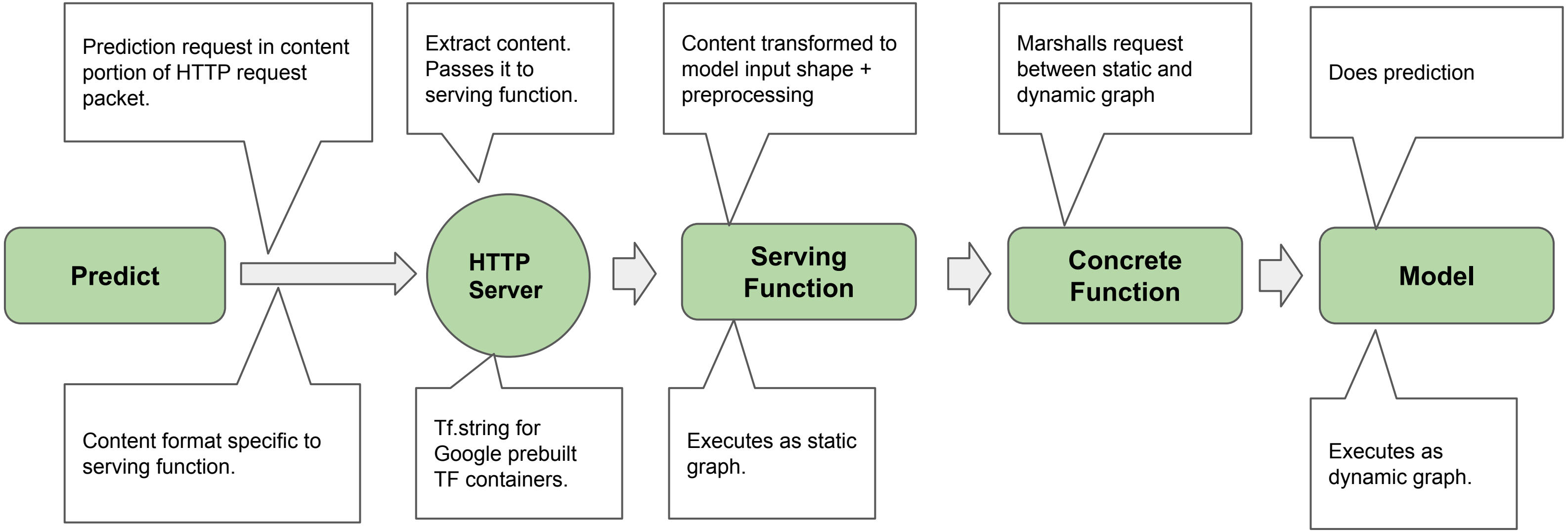
Custom Serving Function - Model data type specific

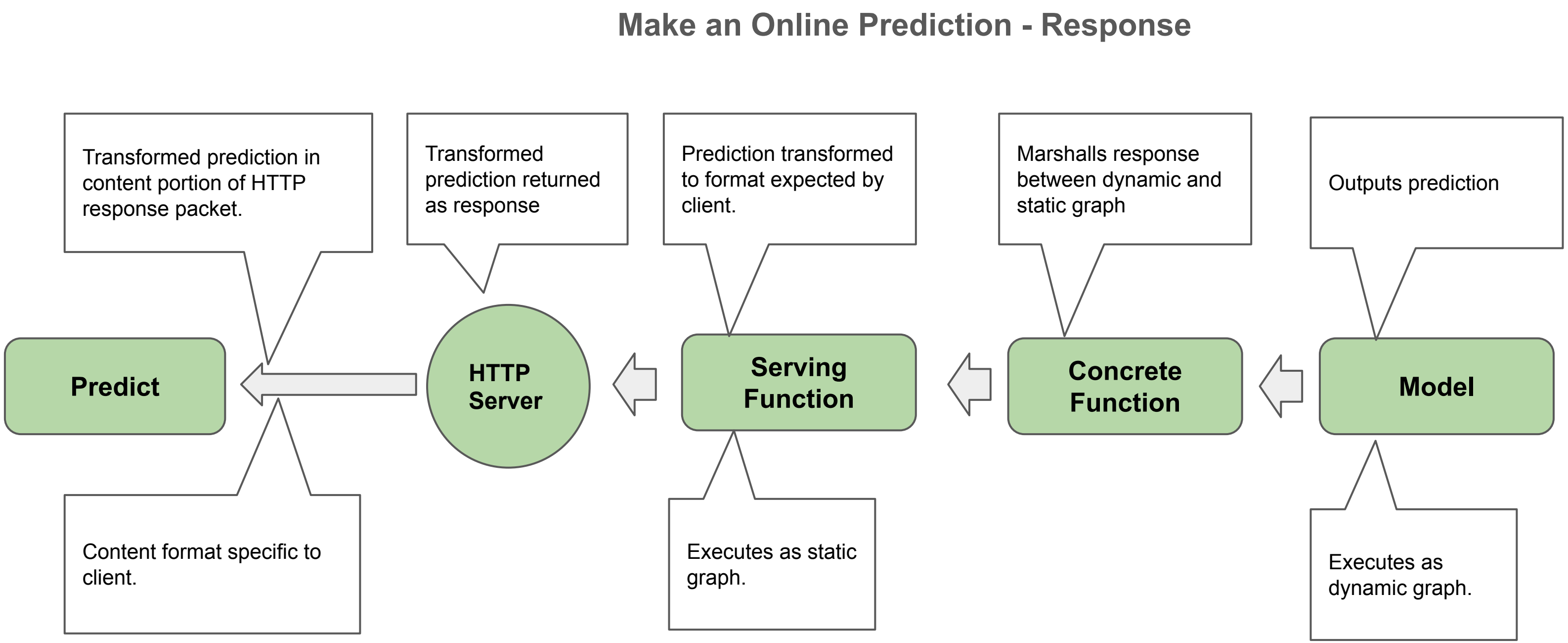


Deploy Custom Model



Make an Online Prediction - Request




Make an Online Prediction - Response
Transformed prediction in content portion of HTTP response packet.
Transformed prediction returned as response
Prediction transformed to format expected by client.
Marshalls response between dynamic and static graph
Outputs prediction
Predict
HTTP Server
Serving Function
Concrete Function
Model
Content format specific to client.
Executes as static graph.
Executes as dynamic graph.

Workshop 6: Custom Job Image Classification

VM Instances

Hardware Accelerators

Zero or more GPUs (and type) per VM training instance.

Zero or more GPUs (and type) per VM deployment instance.

```
if os.getenv("AUTORUN_TRAIN_GPU"):
    TRAIN_GPU, TRAIN_NGPU = (aip.AcceleratorType.NVIDIA_TESLA_K80, int(os.getenv("AUTORUN_TRAIN_GPU")))
else:
    TRAIN_GPU, TRAIN_NGPU = (aip.AcceleratorType.NVIDIA_TESLA_K80, 1)

if os.getenv("AUTORUN_DEPOLY_GPU"):
    DEPLOY_GPU, DEPLOY_NGPU = (aip.AcceleratorType.NVIDIA_TESLA_K80, int(os.getenv("AUTORUN_DEPOLY_GPU")))
else:
    DEPLOY_GPU, DEPLOY_NGPU = (None, None)
```

VM Instances

Machine Type

Machine type and number of CPUs per VM training instance.

Machine type and number of CPUs per VM deployment instance.

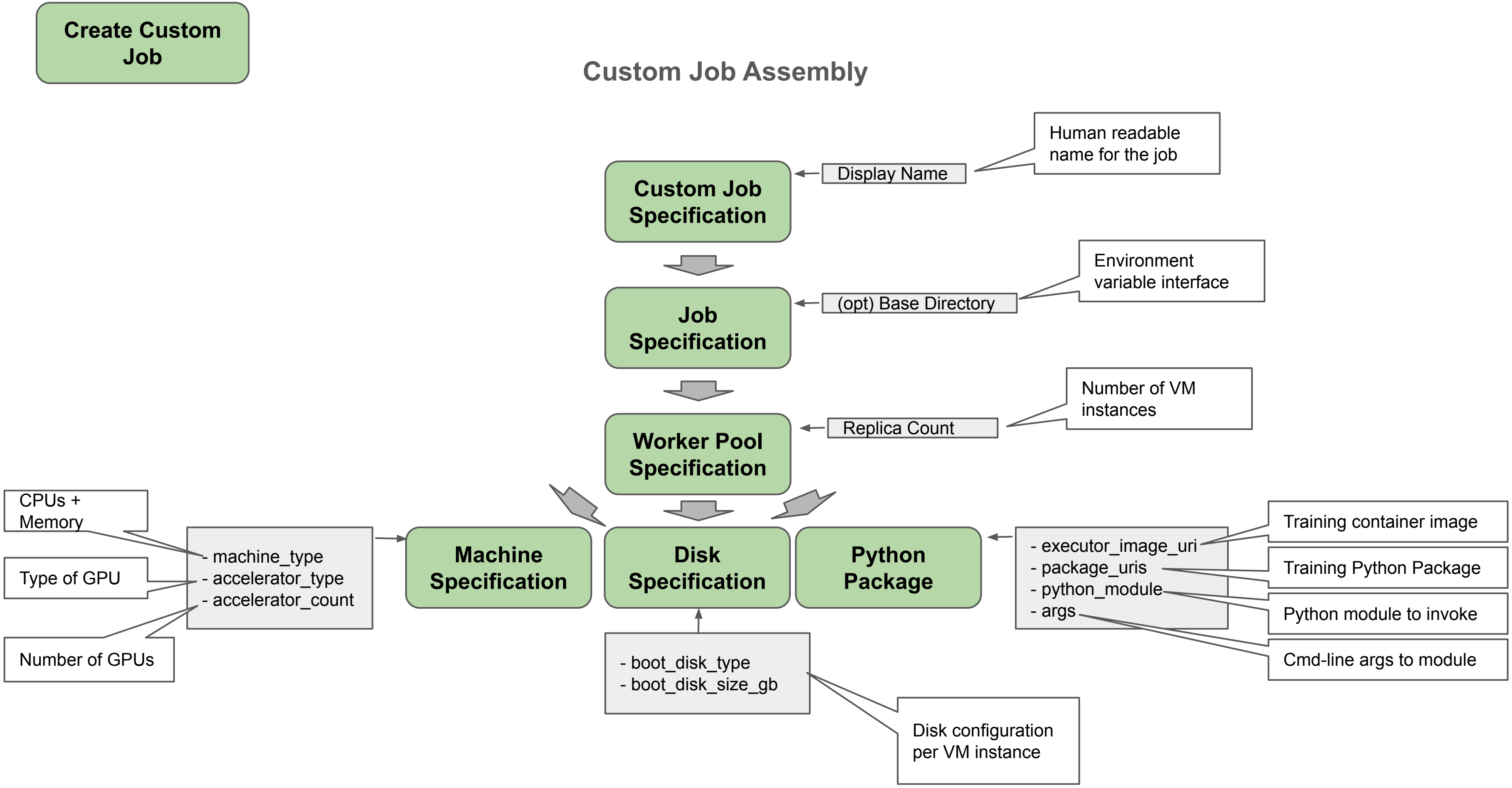
```
if os.getenv("AUTORUN_TRAIN_MACHINE"):
    MACHINE_TYPE = os.getenv("AUTORUN_TRAIN_MACHINE")
else:
    MACHINE_TYPE = 'n1-standard'

VCPU = '4'
TRAIN_COMPUTE = MACHINE_TYPE + '-' + VCPU
print('Train machine type', TRAIN_COMPUTE)


if os.getenv("AUTORUN_DEPLOY_MACHINE"):
    MACHINE_TYPE = os.getenv("AUTORUN_DEPLOY_MACHINE")
else:
    MACHINE_TYPE = 'n1-standard'

VCPU = '4'
DEPLOY_COMPUTE = MACHINE_TYPE + '-' + VCPU
print('Deploy machine type', DEPLOY_COMPUTE)
```

VM Instances

Prebuilt Container Images

Google prebuilt containers for training and prediction.

```
if os.getenv("AUTORUN_TF"):
   TF = os.getenv("AUTORUN_TF")
else:
   TF = '2-1'

if TF[0] == '2':
   if TRAIN_GPU:
      TRAIN_VERSION = 'tf-gpu.{}'.format(TF)
   else:
      TRAIN_VERSION = 'tf-cpu.{}'.format(TF)
   if DEPLOY_GPU:
      DEPLOY_VERSION = 'tf2-gpu.{}'.format(TF)
   else:
      DEPLOY_VERSION = 'tf2-cpu.{}'.format(TF)
else:
   if TRAIN_GPU:
      TRAIN_VERSION = 'tf-gpu.{}'.format(TF)
   else:
      TRAIN_VERSION = 'tf-cpu.{}'.format(TF)
   if DEPLOY_GPU:
      DEPLOY_VERSION = 'tf-gpu.{}'.format(TF)
   else:
      DEPLOY_VERSION = 'tf-cpu.{}'.format(TF)

TRAIN_IMAGE  = "gcr.io/cloud-aiplatform/training/{}:latest".format(TRAIN_VERSION)
DEPLOY_IMAGE = "gcr.io/cloud-aiplatform/prediction/{}:latest".format(DEPLOY_VERSION)

print("Training:", TRAIN_IMAGE, TRAIN_GPU, TRAIN_NGPU)
print("Deployment:", DEPLOY_IMAGE, DEPLOY_GPU, DEPLOY_NGPU)
```


Create Custom Job
Custom Job Assembly
Custom Job Specification
Display Name
Human readable name for the job
Job Specification
(opt) Base Directory
Environment variable interface
Worker Pool Specification
Replica Count
Number of VM instances
CPUs + Memory
Type of GPU
Number of GPUs
- machine_type
- accelerator_type
- accelerator_count
Machine Specification
Disk Specification
Python Package
- boot_disk_type
- boot_disk_size_gb
Disk configuration per VM instance
- executor_image_uri
- package_uris
- python_module
- args
Training container image
Training Python Package
Python module to invoke
Cmd-line args to module

Custom Specification

Custom Job Specification Parameters

```
JOB_NAME = "custom_job_" + TIMESTAMP

custom_job = {
   "display_name": JOB_NAME,
   "job_spec": job_spec
}
```

Human readable name for the job

Job Specification

Job Specification Parameters

Using cmd-line arguments from service to training package, e.g., location to store model artifacts.

Using environment variables from service to training package.

```
MODEL_DIR = '{}/{}'.format(BUCKET_NAME, JOB_NAME)

if DIRECT:
   job_spec = {
      "worker_pool_specs": worker_pool_spec
   }
else:
   job_spec = {
      "worker_pool_specs": worker_pool_spec,
      "base_output_directory": {"output_uri_prefix": MODEL_DIR}
   }
```

Environment variable interface

Worker Pool Specification

Worker Pool Specification Parameters

```
worker_pool_spec = [
   {
      "replica_count": 1,
      "machine_spec": machine_spec,
      “disk_spec”: disk_spec,
      "python_package_spec": python_package_spec
   }
]
```

Number of VM instances, e.g., single machine

Machine Specification

Machine Specification Parameters

1 or more GPUs

CPU(s) only

```
if TRAIN_GPU:
  machine_spec = {
    "machine_type": TRAIN_COMPUTE,
    "accelerator_type": TRAIN_GPU,
    "accelerator_count": TRAIN_NGPU
  }
else:
  machine_spec = {
    "machine_type": TRAIN_COMPUTE,
    "accelerator_count": 0
  }
```

CPUs + Memory

Type of GPU

Number of GPUs

Disk Specification

Disk Specification Parameters

SSD is default

```
DISK_TYPE = "pd-ssd" # [ pd-ssd, pd-standard]
DISK_SIZE = 200 # GB

disk_spec = {
   "boot_disk_type": DISK_TYPE,
   "boot_disk_size_gb": DISK_SIZE
}
```

Disk configuration per VM instance

Machine Specification

Python Package Specification Parameters

```
EPOCHS = 20
STEPS = 100


if DIRECT:
   CMDARGS = [
            "--model-dir=" + MODEL_DIR,
            "--epochs=" + str(EPOCHS),
            "--steps=" + str(STEPS),
            "--distribute=" + TRAIN_STRATEGY
          ]
else:
   CMDARGS = [
            "--epochs=" + str(EPOCHS),
            "--steps=" + str(STEPS),
            "--distribute=" + TRAIN_STRATEGY
          ]

python_package_spec = {
        "executor_image_uri": TRAIN_IMAGE,
        "package_uris": [BUCKET_NAME + "/trainer_cifar10.tar.gz"],
        "python_module": "trainer.task",
        "args": CMDARGS,
      }
```

Using cmd-line arguments from service to specify hyperparameters.

Model output directory specified at cmd-line

Model output directory specified as environment variable

Training image must be Google pre-built container.

Training container image

Training Python Package

“folder/file.py” path specified as: “folder.file”

Python module to invoke

Cmd-line args to module

Create CustomJob

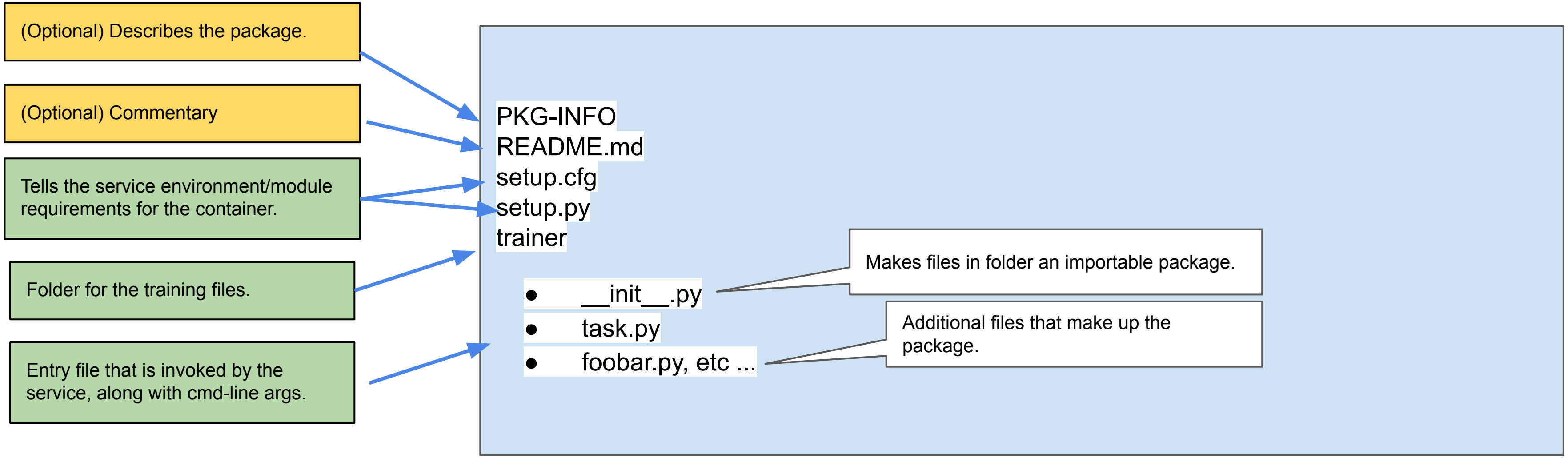
Execute a custom job

Custom Job runs asynchronous

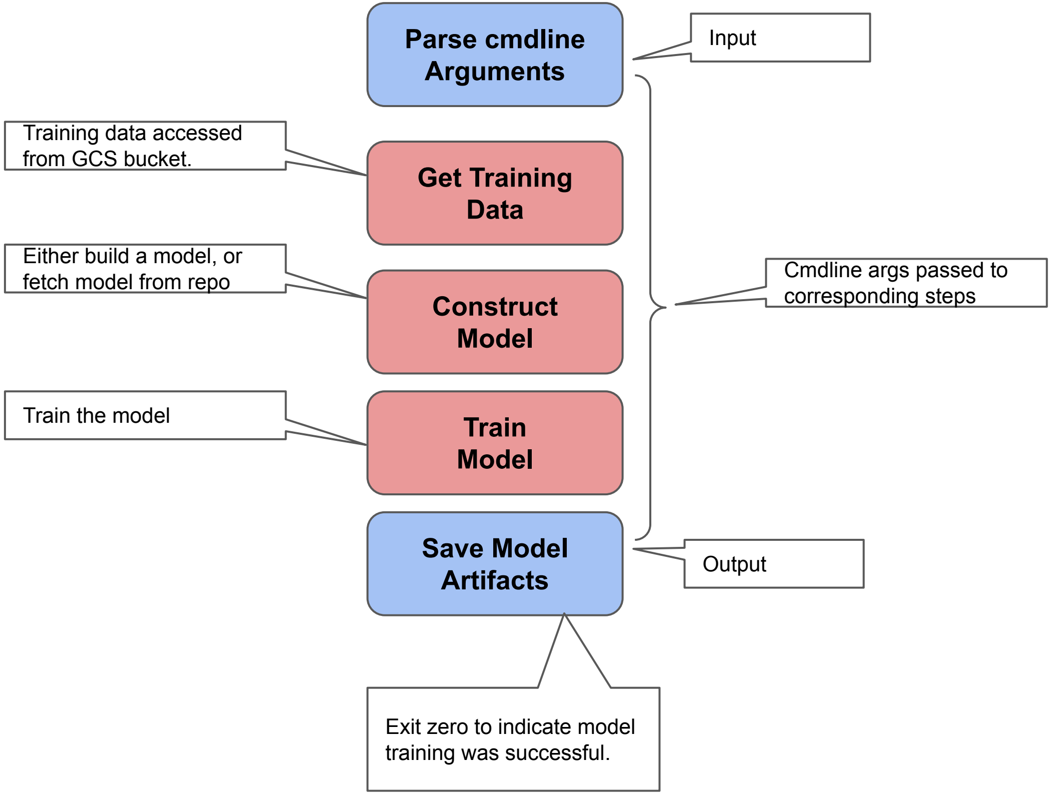
Poll job to get status:
PENDING
RUNNING
SUCCESSFUL
FAILED

```
def create_custom_job(custom_job):
  response = clients['job'].create_custom_job(parent=PARENT, custom_job=custom_job)
  print("name:", response.name)
  print("display_name:", response.display_name)
  print("state:", response.state)
  print("create_time:", response.create_time)
  print("update_time:", response.update_time)
  return response

# Save the job name
response = create_custom_job(custom_job)
```

Python Package



(Basic) Training File Contents



Parse cmdline Arguments

Parse the Cmd-Line Arguments

Otherwise get them from service's environment variable.

Get the location to store the model artifacts.

Hyperparameters

Distribution strategy.

Default to single instance.

```
parser = argparse.ArgumentParser()
parser.add_argument('--model-dir', dest='model_dir',
            default=os.getenv("AIP_MODEL_DIR"), type=str, help='Model dir.')

parser.add_argument('--lr', dest='lr',
            default=0.01, type=float,
            help='Learning rate.')
parser.add_argument('--epochs', dest='epochs',
            default=10, type=int,
            help='Number of epochs.')
parser.add_argument('--steps', dest='steps',
            default=200, type=int,
            help='Number of steps per epoch.')

parser.add_argument('--distribute', dest='distribute', type=str, default='single',
            help='distributed training strategy')
args = parser.parse_args()
```

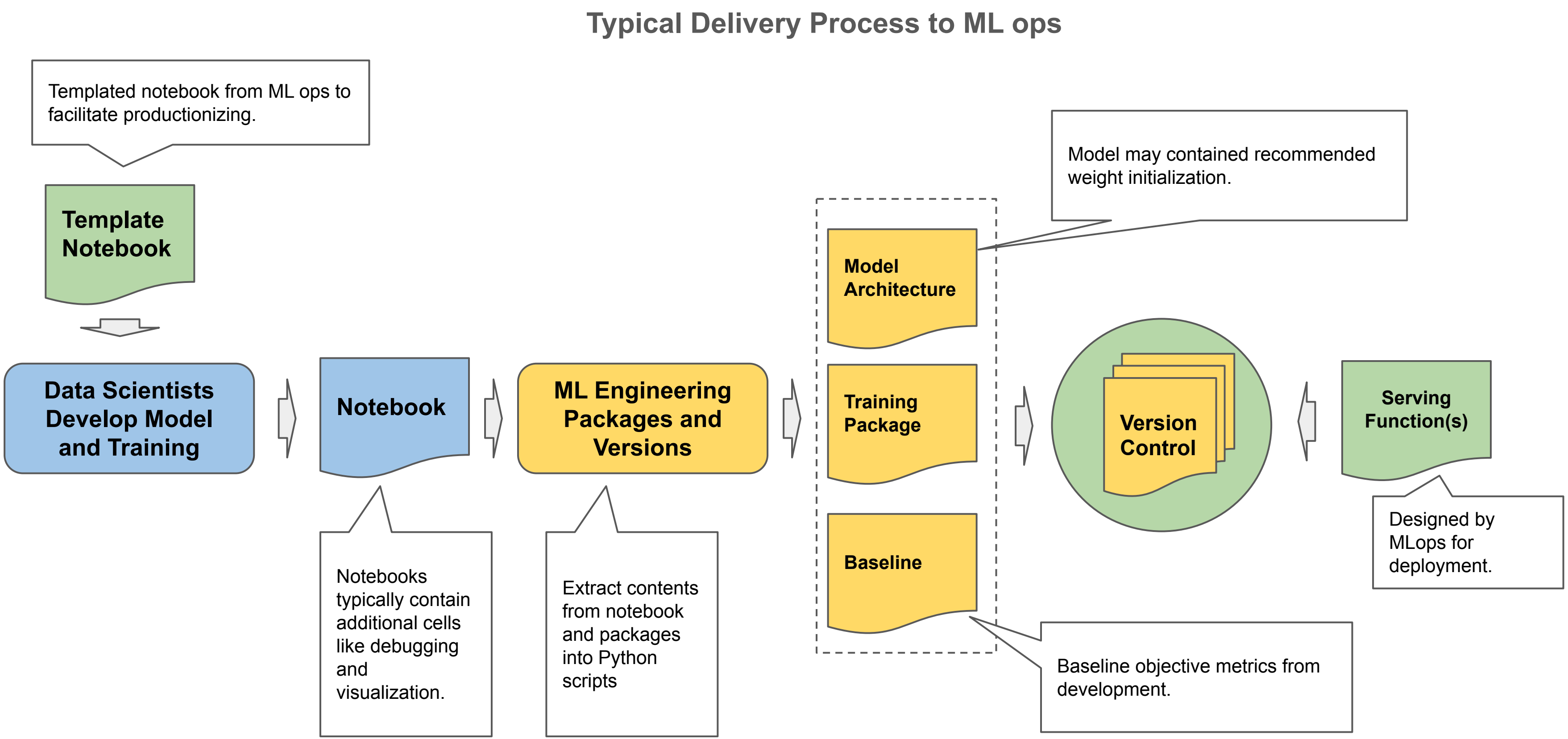
Save Model Artifacts

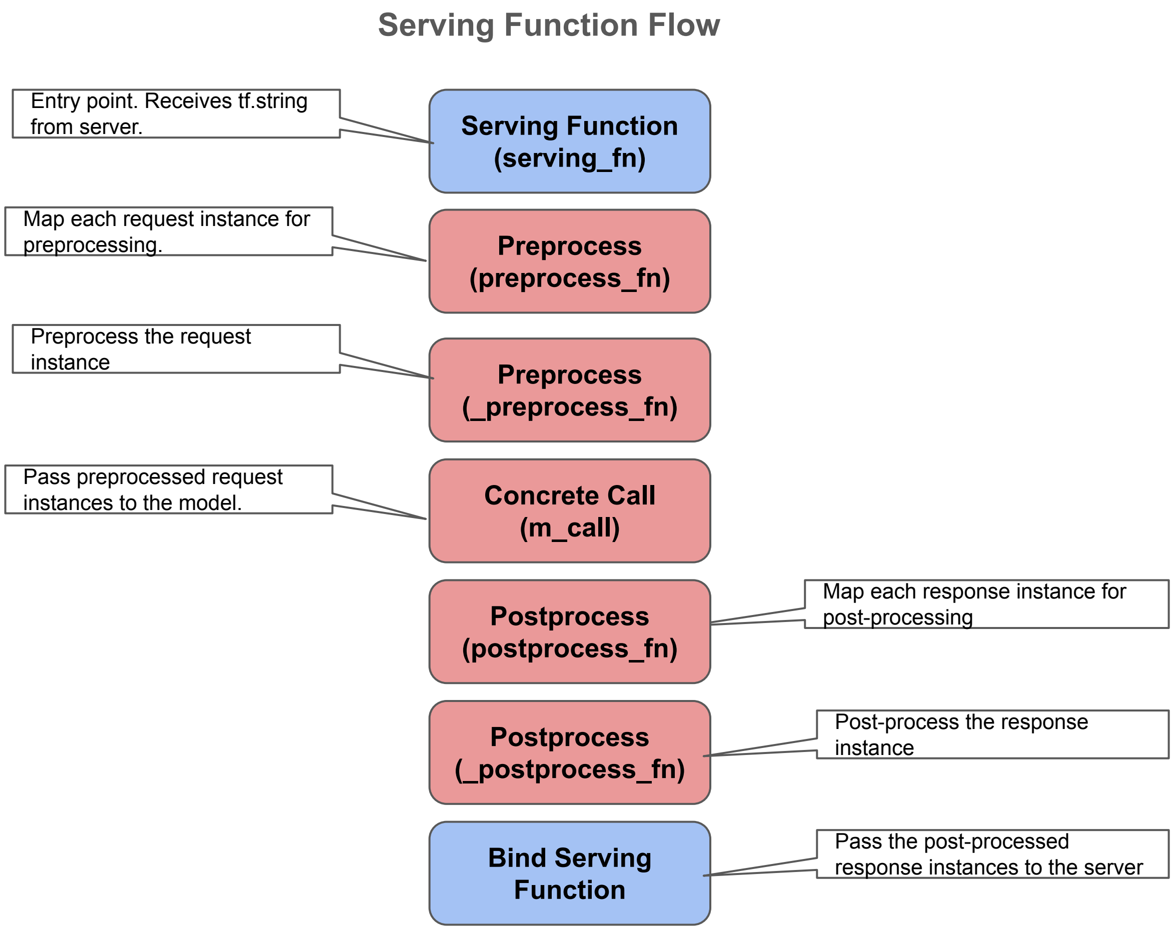
Save the Model Artifacts to GCS

```
model.fit(x=train_dataset, epochs=args.epochs, steps_per_epoch=args.steps)
model.save(args.model_dir)
```

Save mode artifacts after training.

Saved to a GCS bucket.


Typical Delivery Process to ML ops
Templated notebook from ML ops to facilitate productionizing.
Template Notebook
Data Scientists Develop Model and Training
Notebook
ML Engineering Packages and Versions
Notebooks typically contain additional cells like debugging and visualization.
Extract contents from notebook and packages into Python scripts
Model Architecture
Training Package
Baseline
Model may contained recommended weight initialization.
Baseline objective metrics from development.
Version Control
Serving Function(s)
Designed by MLops for deployment.


Serving Function Flow
Entry point. Receives tf.string from server.
Serving Function (serving_fn)
Map each request instance for preprocessing.
Preprocess (preprocess_fn)
Preprocess the request instance
Preprocess (_preprocess_fn)
Pass preprocessed request instances to the model.
Concrete Call (m_call)
Map each response instance for post-processing
Postprocess (postprocess_fn)
Post-process the response instance
Postprocess (_postprocess_fn)
Pass the post-processed response instances to the server
Bind Serving Function

Serving Function (serving_fn)

Serving Function Outline

1) @tf.function decoration converts to static graph ops by AutoGraph compiler

2) Preprocess the request instances

4) Postprocess the prediction output

5) Return the post-processed prediction output to the server (e.g., HTTP).

1) Request content received as tf.string

```
@tf.function(input_signature=[tf.TensorSpec([None], tf.string)])
def serving_fn(bytes_inputs):
  instances = preprocess_fn(bytes_inputs)
  predictions = m_call(**instances)
  post = postprocess_fn(predictions)
  return post
```

1) Placeholder for instances (number of) size

3) Call the model with the preprocessed instances

```
tf.saved_model.save(model, model_path_to_deploy, signatures={
   'serving_default': serving_fn,
})
```

6) Will bind serving function to model.

Preprocess (preprocess_fn)

Serving Preprocessing Function

```
CONCRETE_INPUT = "numpy_inputs"

@tf.function(input_signature=[tf.TensorSpec([None], tf.string)])
def preprocess_fn(bytes_inputs):
  preprocessed_instances = tf.map_fn(_preprocess, bytes_inputs, dtype=tf.float32, back_prop=False)
  return {CONCRETE_INPUT: preprocessed_instances}
```

1) User assigned name for the interface layer between serving function and model.

2) Request content received as instances of tf.string

3) Returned preprocessed instances are of data type FLOAT32.

2) @tf.function decoration converts to static graph ops by AutoGraph compiler

3) Map (tf.map_fn) each instance in instances to the function _preprocess.

4) Return as a dictionary:
Key: The layer to pass data to (concrete interface)
Value: The preprocessed request (data)

Postprocess (postprocess_fn)

Serving Postprocessing Function

1) @tf.function decoration converts to static graph ops by AutoGraph compiler

2) Map (tf.map_fn) each instance in instances to the function _postprocess.

1) Replace with output shape of model.

```
@tf.function(input_signature=[tf.TensorSpec([None, <??>])])
def postprocess_fn(bytes_outputs):
  postproces_instances = tf.map_fn(_postprocess, bytes_outputs, dtype=tf.<??>, back_prop=False)
  return postprocess_instances
```

2) Replace with output type of preprocessed instances.

Preprocess (_preprocess_fn)

Serving Preprocessing Function - Images
TensorFlow “Common Image Input Format”

1) Decompress the image as raw uncompressed byte tensor of shape (H, W, C)

2) Typecast the pixels to FLOAT32

3) Resize HxW to input shape of model

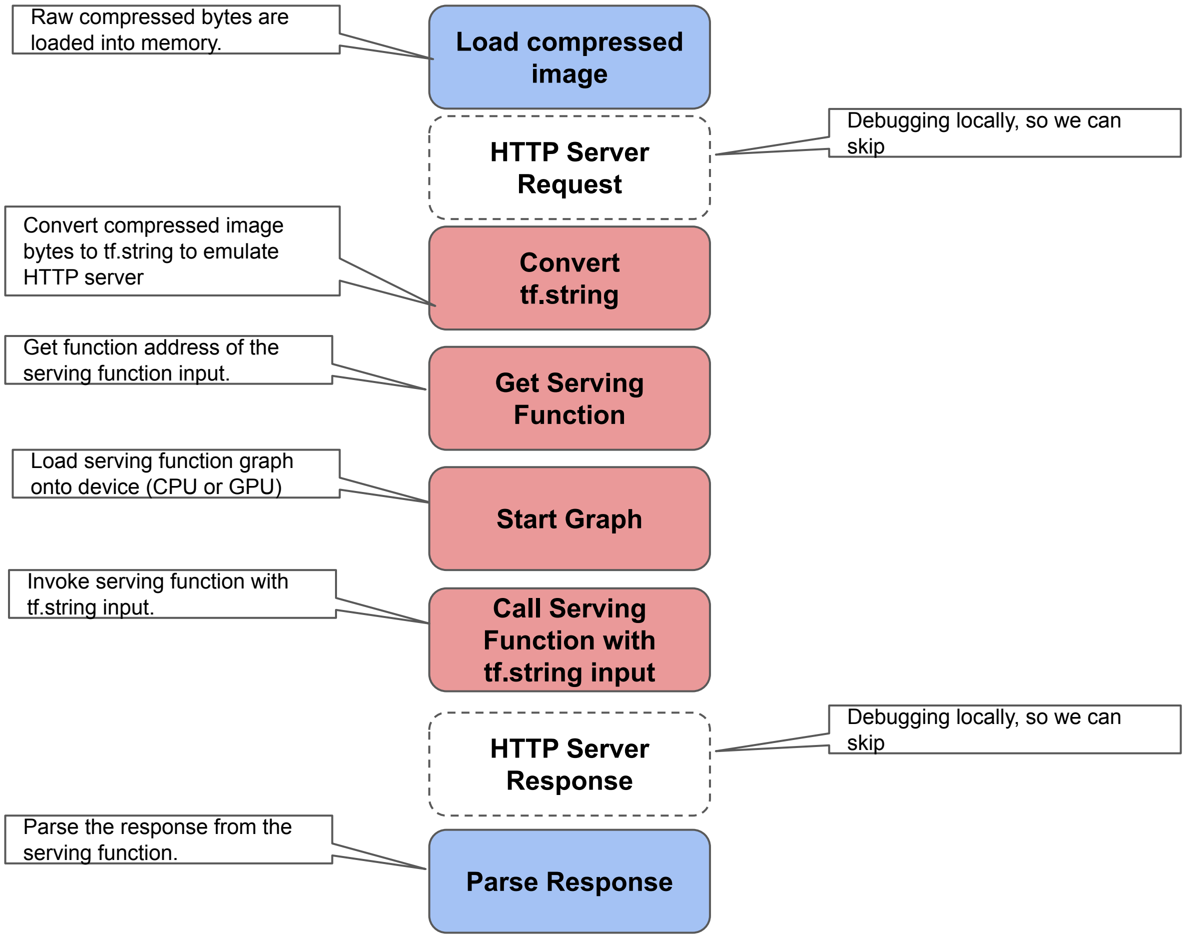
4) Rescale (normalization) pixel data between 0 and 1.

1) Compressed image as tf.string

```
def _preprocess(bytes_input):
    decoded = tf.io.decode_jpeg(bytes_input, channels=3)
    casted  = tf.image.convert_image_dtype(decoded, tf.float32)
    resized = tf.image.resize(casted, size=(<H>, <W>))
    rescale = tf.cast(resized / 255.0, tf.float32)
    return rescale
```

3) Replace with Height and Width of input shape of model

Local Debugging a Serving Function



Debug Serving Function - Image

Load compressed image bytes into memory.

Convert bytes to tf.string

Get function entry for serving function.

Load serving function graph onto CPU or GPU

```
bytes = tf.io.read_file('tmp.jpg')
tensor = tf.convert_to_tensor( [bytes], tf.string)

serving_function = loaded.signatures["serving_default"]

with tf.device("cpu:0"):
   response = serving_function(tensor)

print(response)
```

Invoke the serving function

Review the response

Upload Model

Upload Custom Model into Model Resource
Google Prebuilt Prediction Container

Human readable name of Model resource.

GCS location of model artifacts: TF, Pytorch, XGBoost, SciLearn

The prebuilt prediction container image.

API call to service (async)

Block (sync) for completion

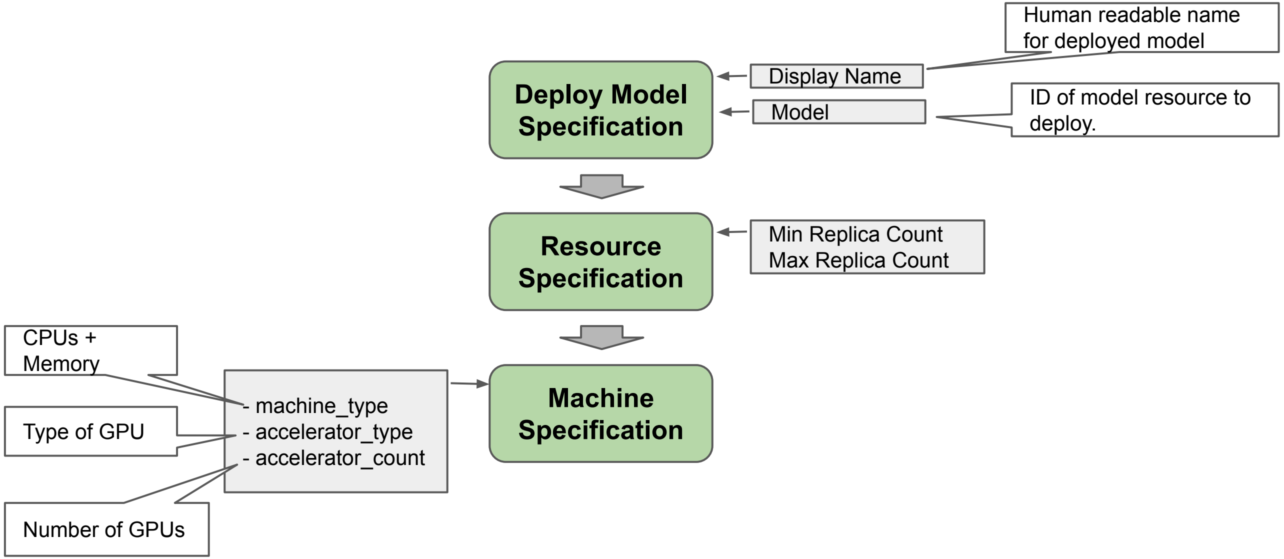
```
def upload_model(display_name, image_uri, model_uri):
  model = {
    "display_name": display_name,
    "metadata_schema_uri": "",
    "artifact_uri": model_uri,
    "container_spec": {
      "image_uri": image_uri,
    },
  }

  response = clients['model'].upload_model(parent=PARENT, model=model)

  upload_model_response = response.result(timeout=180)
  return rupload_model_response.model

model_to_deploy_id = upload_model("mymodel" + TIMESTAMP, IMAGE_URI, model_path_to_deploy)
```

Not used for custom models.

The ID of the model resource.

Deploy Model

Deploy Model Assembly



Deploy Model

Deploy Custom Model Resource

```
def deploy_model(model, deployed_model_display_name, endpoint, traffic_split={"0": 100}):

  if DEPLOY_GPU:
    machine_spec = {
      "machine_type": DEPLOY_COMPUTE,
      "accelerator_type": DEPLOY_GPU,
      "accelerator_count": DEPLOY_NGPU,
    }
  else:
    machine_spec = {
      "machine_type": DEPLOY_COMPUTE,
      "accelerator_count": 0,
    }

  deployed_model = {
    "model": model,
    "display_name": deployed_model_display_name,
    "dedicated_resources": {
      "min_replica_count": MIN_NODES,
      "max_replica_count": MAX_NODES,
      "machine_spec": machine_spec,
    },
  }


  response = clients['endpoint'].deploy_model(
    endpoint=endpoint, deployed_model=deployed_model, traffic_split=traffic_split)

  result = response.result()
```

Designages the percent of prediction requests go to this model.

Machine type + Accelerators

Machine type w/o Accelerators

Scaling

Machine specification

API call to service (async)

Block (sync) for completion

Predict

Prediction for Custom Image Models

Prepare image for common image input format.

List of one or more instances.
{ serving_input: { ‘b4’: content}}

Serving input layer name.

Base64 compressed image

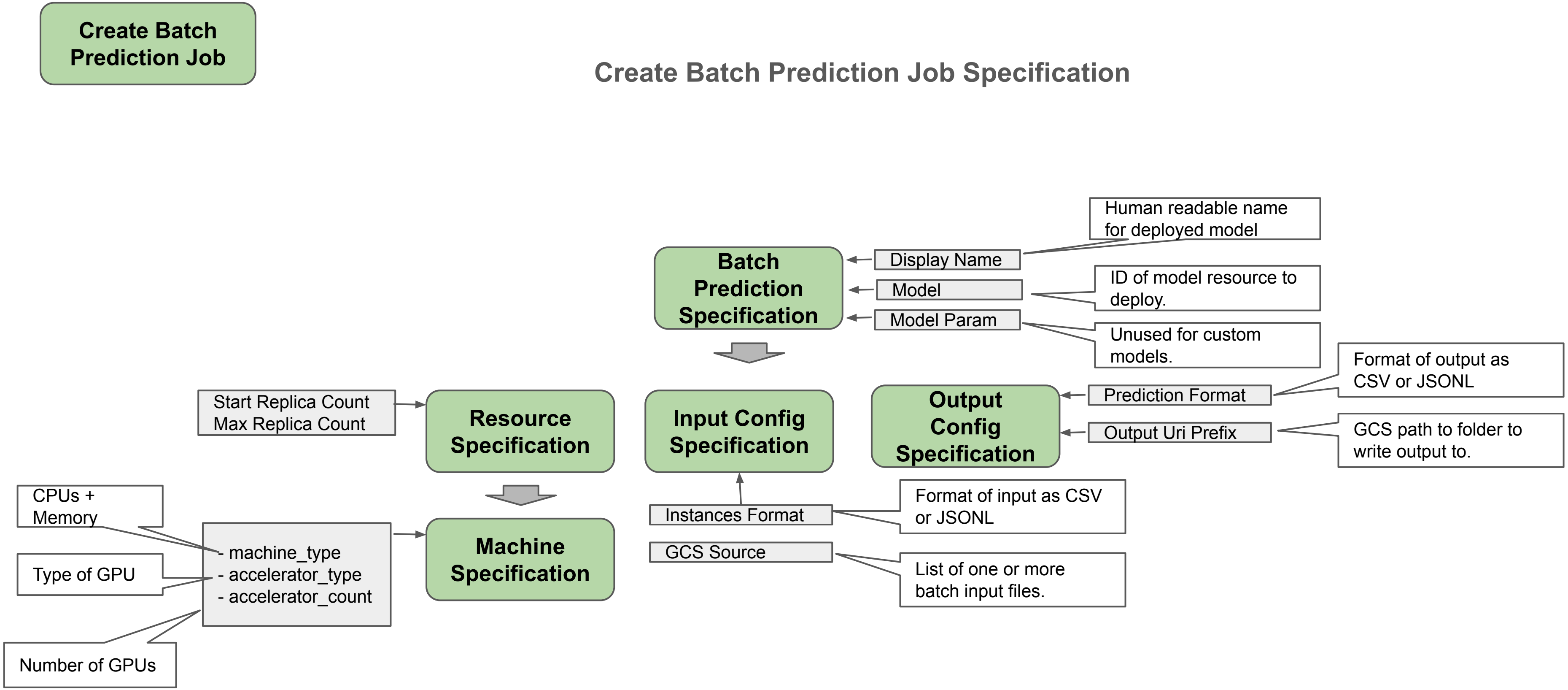
API call for prediction.

JSON object (dictionary) prediction response for each instance

```
loaded = tf.saved_model.load(model_path_to_deploy)

serving_input = list(loaded.signatures['serving_default'].structured_input_signature[1].keys())[0]

bytes = tf.io.read_file('myimage.jpg')
b64str = base64.b64encode(bytes.numpy()).decode('utf-8')

def predict_image(content, endpoint):

  instances_list = [{serving_input: {'b64': content}}]
  instances = [json_format.ParseDict(s, Value()) for s in instances_list]

  response = clients['prediction'].predict(endpoint=endpoint, instances=instances, parameters=None)

  print(" deployed_model_id:", response.deployed_model_id)

  predictions = response.predictions
  print("predictions")
  for prediction in predictions:
    print(" prediction:", prediction)


predict_image(b64str, endpoint_id)
```

Which model on endpoint did the prediction.

No builtin post-processing for custom models.


Create Batch Prediction Job
Create Batch Prediction Job Specification
Human readable name for deployed model
Batch Prediction Specification
Display Name
Model
ID of model resource to deploy.
Model Param
Unused for custom models.
Format of output as CSV or JSONL
Start Replica Count
Max Replica Count
Resource Specification
Input Config Specification
Output Config Specification
Prediction Format
Output Uri Prefix
GCS path to folder to write output to.
CPUs + Memory
Format of input as CSV or JSONL
Instances Format
- machine_type
- accelerator_type
- accelerator_count
Machine Specification
GCS Source
Type of GPU
List of one or more batch input files.
Number of GPUs

Batch Predict

Batch Request File Format

GCS location of batch file.

Write each instance to batch file as a JSONL object

Format:
{ serving_function: {'b64' : content }}

Serving input layer name.

Base64 compressed image

```
gcs_input_uri = BUCKET_NAME + "/" + "test.jsonl"

gcs_input_uri = "gs://" + BUCKET_NAME + "/" + "test.jsonl"
with tf.io.gfile.GFile(gcs_input_uri, 'w') as f:
   data = {serving_input: {'b64': b64str}}
   f.write(json.dumps(data) + '\n')
   data = {serving_input: {'b64': b64str}}
   f.write(json.dumps(data) + '\n')
```

Workshop 7: Custom Jobs (cont)

- Custom Jobs Text Models
- Custom Jobs Tabular Models
- Hyperparameter Tuning Jobs
- Local (on-Prem) Training

Workshop 7: Custom Job Text Models

Predict

Prediction for Custom Text Models

```
data = text.numpy()
data = data.tolist()

def predict_data(data, endpoint, parameters_dict):

    instances_list = [{serving_input: data}]
    instances = [json_format.ParseDict(s, Value()) for s in instances_list]

    response = clients['prediction'].predict(endpoint=endpoint, instances=instances, parameters=parameters_dict)
    print("response")
    print(" deployed_model_id:", response.deployed_model_id)
    predictions = response.predictions

    print("predictions")
    for prediction in predictions:
        print(" prediction:", prediction)

predict_data(test_data, endpoint_name, None)
```

Convert text string into an array of bytes

Convert array of bytes to list of characters.

List of one or more instances.
{ serving_input: data}

Serving input layer name.

Text as list of characters

Which model on endpoint did the prediction.

API call for prediction.

JSON object (dictionary) prediction response for each instance

Batch Predict

Batch Request File Format - Text

GCS location of batch file.

Write each instance to batch file as a JSONL object

Format:
{ serving_function: content }

Serving input layer name.

List of characters

```
test_item_1 = text.numpy()
test_item_1= test_item_1.tolist()

gcs_input_uri = BUCKET_NAME + "/" + "test.jsonl"

with tf.io.gfile.GFile(gcs_input_uri, 'w') as f:
   data = {serving_name: test_item_1}
   f.write(json.dumps(data) + '\n')
```

Convert text string into an array of bytes

Convert array of bytes to list of characters.

Workshop 7: Custom Job Tabular Models

Predict

Prediction for Custom Tabular Models

```
data = data.tolist()

def predict_data(data, endpoint, parameters_dict):

  instances_list = [{serving_input: data}]
  instances = [json_format.ParseDict(s, Value()) for s in instances_list]

  response = clients['prediction'].predict(endpoint=endpoint, instances=instances, parameters=parameters_dict)
  print("response")
  print(" deployed_model_id:", response.deployed_model_id)
  predictions = response.predictions

  print("predictions")
  for prediction in predictions:
        print(" prediction:", prediction)


predict_data(test_data, endpoint_name, None)
```

Convert tabular data (e.g., array) to list.

E.g., [1, 2.0, 3.1, 2,2]

List of one or more instances.
{ serving_input: data}

Serving input layer name.

Tabular as list of floating point values.

API call for prediction.

JSON object (dictionary) prediction response for each instance

Which model on endpoint did the prediction.

Batch Predict

Batch Request File Format - Tabular

GCS location of batch file.

Write each instance to batch file as a JSONL object

Format:
{ serving_function: content }

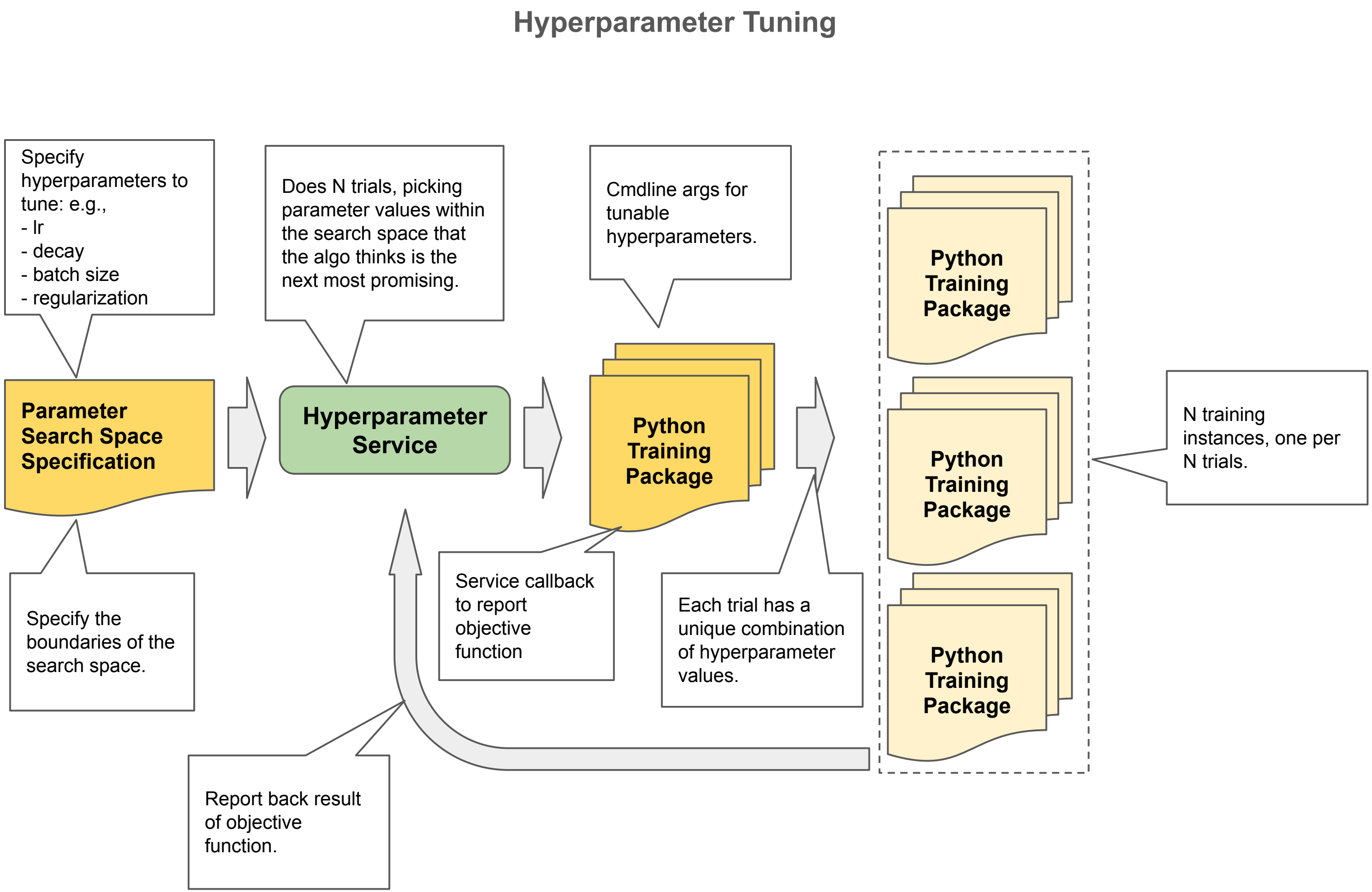
Serving input layer name.

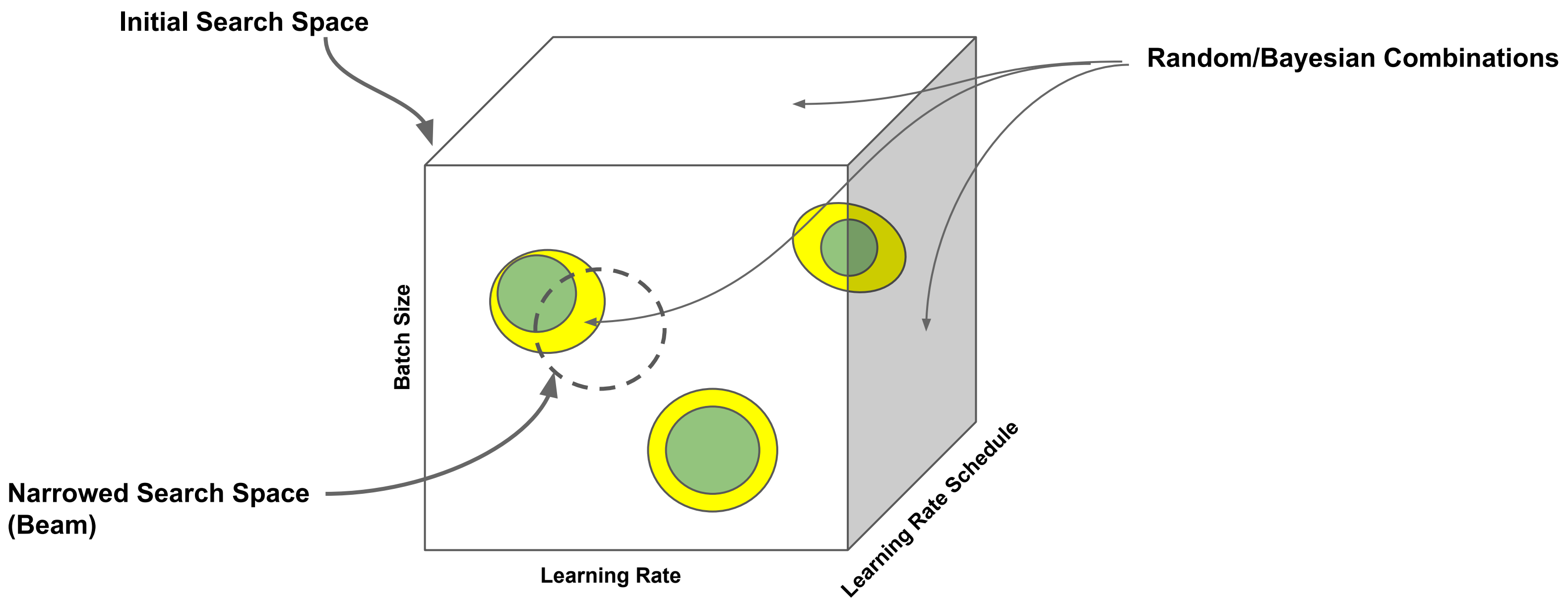
List of floating point values

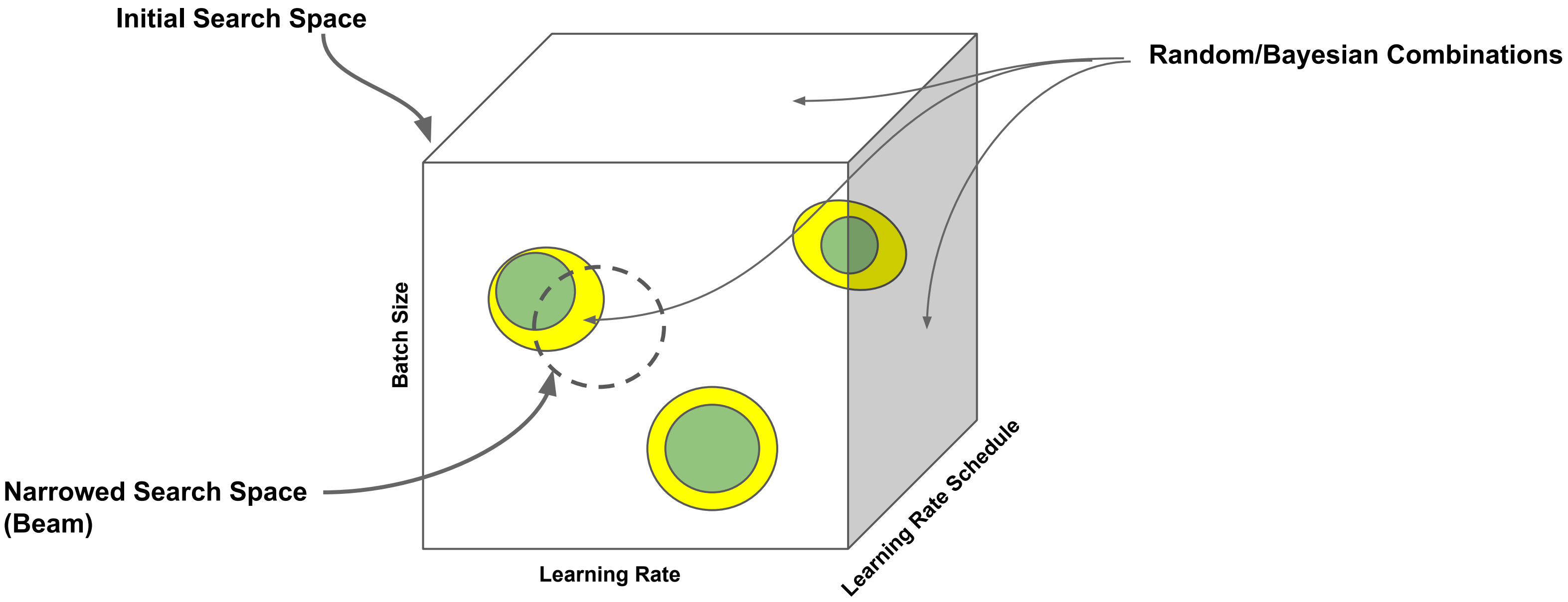
```
test_item_1= test_item_1.tolist()

gcs_input_uri = BUCKET_NAME + "/" + "test.jsonl"

with tf.io.gfile.GFile(gcs_input_uri, 'w') as f:
    data = {serving_name: test_item_1}
    f.write(json.dumps(data) + '\n')
```

Convert array of bytes to list of floating point values.

Workshop 7: Hyperparameter Tuning


Hyperparameter Tuning
Specify hyperparameters to tune: e.g.,
- lr
- decay
- batch size
- regularization
Does N trials, picking parameter values within the search space that the algo thinks is the next most promising.
Cmdline args for tunable hyperparameters.
Parameter Search Space Specification
Hyperparameter Service
Python Training Package
Python Training Package
Python Training Package
Python Training Package
N training instances, one per N trials.
Specify the boundaries of the search space.
Service callback to report objective function
Each trial has a unique combination of hyperparameter values.
Report back result of objective function.


Initial Search Space
Random/Bayesian Combinations
Batch Size
Narrowed Search Space
(Beam)
Learning Rate
Learning Rate Schedule


Initial Search Space
Random/Bayesian Combinations
Batch Size
Narrowed Search Space
(Beam)
Learning Rate
Learning Rate Schedule

Modifying Training Package for HP Tuning

- Cmdline args are added for tunable hyperparameters.
- Callback added to training step (e.g., fit() method) to report result of objective (e.g., logloss, accuracy, etc).
- Cmdline args for epochs, and/or steps to reduce training time for tuning trials vs the full training.

HPT
Python Package

CmdLine Arguments for HPT training Package

```
# Command Line arguments
parser = argparse.ArgumentParser()

parser.add_argument('--model-dir', dest='model_dir',
                default=os.getenv('AIP_MODEL_DIR'), type=str, help='Model dir.')

parser.add_argument('--lr', dest='lr',
                default=0.001, type=float,
                help='Learning rate.')
parser.add_argument('--decay', dest='decay',
                default=0.01, type=float,
                help='Decay rate')
parser.add_argument('--batch_size', dest='batch_size',
                default=32, type=int,
                help='Batch size')

parser.add_argument('--epochs', dest='epochs',
                default=200, type=int,
                help='Number of epochs")
parser.add_argument('--steps', dest='steps',
                default=1000, type=int,
                help='Number of batches per epoch")

args = parser.parse_args()
```

Arguments set by hyperparameter service.

Arguments set for length of training (trial and full).

Trial runs typically use shorter training cycles.

HPT Python Package

Objective Reporting in HPT training Package

Optimizer arguments set by hyperparameter service.

Duration of training set in job specification

Training arguments set by hyperparameter service.

```
# Compile the model
model.compile(
        optimizer=Adam(lr=args.lr,
                       decay=args.decay
        ),
        loss='categorical_crossentry)

# Train the model
model.fit(train_dataset,
        epochs=args.epochs,
        steps_per_epoch=args.epochs,
        batch_size=args.batch_size,
        callbacks=[HPTCallback()])
```

Trial runs typically use shorter training cycles.

HPT
Python Package

Objective Reporting in HPT training Package

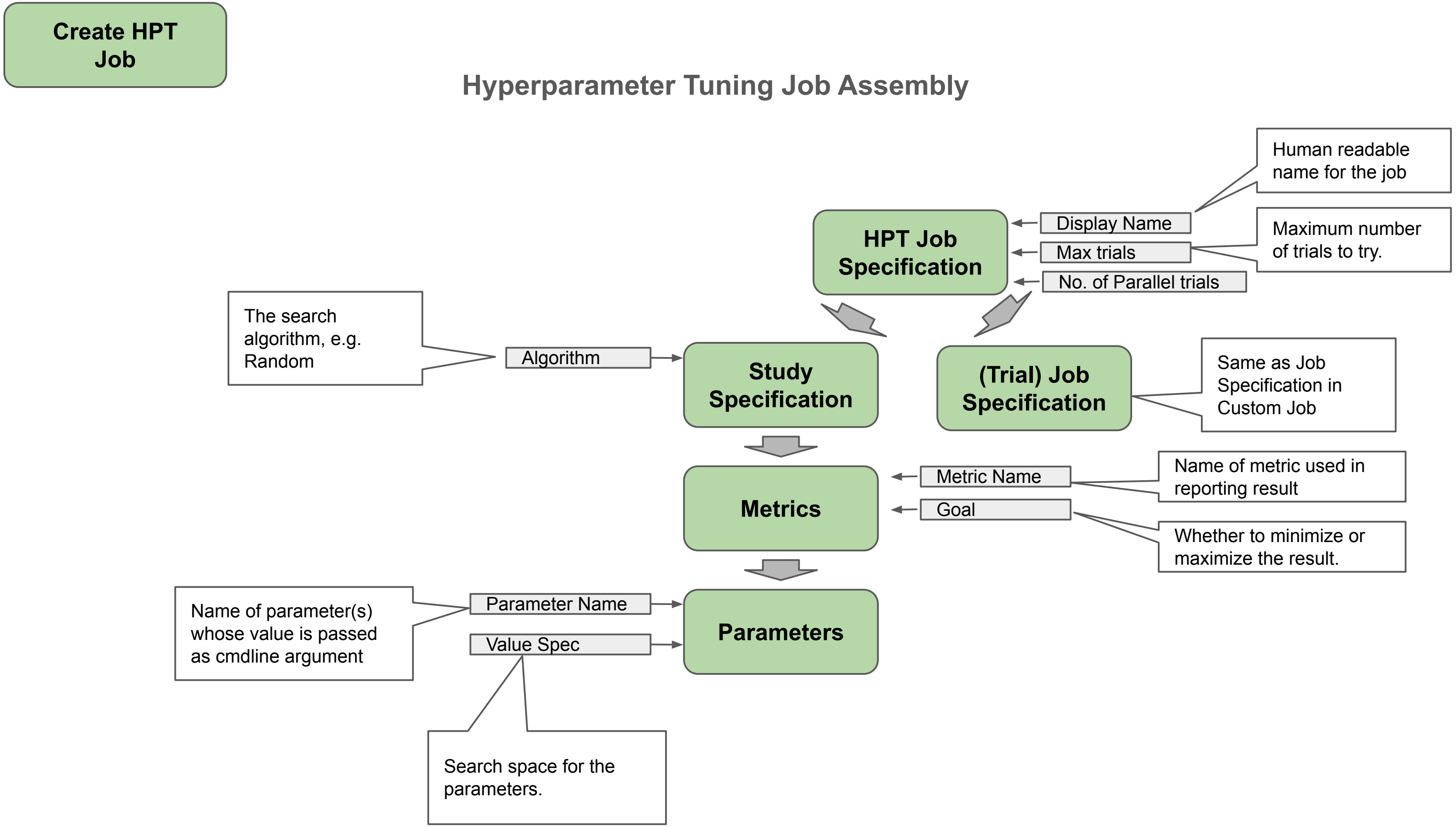
Derive a training callback for reporting back to the service.

Overwrite the on_epoch_end() to report objective results at the end of each epoch.

Report the objective result

```
# Instantiate the HyperTune reporting object
hpt = HyperTune()

# Reporting callback
class HPTCallback(tf.keras.callbacks.Callback):

  def on_epoch_end(self, epoch, logs=None):
    global hpt
    hpt.report_hyperparameter_tuning_metric(
          hyperparameter_metric_tag='val_accuracy',
          metric_value=logs['val_accuracy'],
          global_step=epoch)
```

Specify the parameter name and result value.


Create HPT Job
Hyperparameter Tuning Job Assembly
Human readable name for the job
Display Name
HPT Job Specification
Max trials
Maximum number of trials to try.
No. of Parallel trials
The search algorithm, e.g. Random
Algorithm
Study Specification
(Trial) Job Specification
Same as Job Specification in Custom Job
Metric Name
Name of metric used in reporting result
Metrics
Goal
Whether to minimize or maximize the result.
Name of parameter(s) whose value is passed as cmdline argument
Parameter Name
Value Spec
Parameters
Search space for the parameters.

HPT Job Specification

HPT Job Specification

Same as Custom Job specification

Hyperparameters and search space.

```
if DIRECT:
   TRIAL_JOB_SPEC = {
      "worker_pool_specs": WORKER_POOL_SPEC
   }
else:
   TRIAL_JOB_SPEC = {
      "worker_pool_specs": WORKER_POOL_SPEC,
      "base_output_directory": {"output_uri_prefix": MODEL_DIR}
   }

HPT_JOB = {
   "display_name": JOB_NAME,
   "trial_job_spec": TRIAL_JOB_SPEC,
   "study_spec" : STUDY_SPEC,
   "max_trial_count": 6,
   "parallel_trial_count": 1
}
```

If max trial count is small, it will likely execute all the trials.

HPT Job Specification

Study Specification

One or more objectives (metric) to with minimize or maximize value.

The name of the parameter that is passed as a cmdline arg.

The search space.

```
STUDY_SPEC = {
   "metrics": [{
         "metric_id": "val_accuracy",
         "goal": aip.StudySpec.MetricSpec.GoalType.MAXIMIZE
      }],
      "parameters": [{
          "parameter_id": "lr",
          "discrete_value_spec": { 'values': [ 0.001, 0.01, 0.1 ] },
          "scale_type": aip.StudySpec.ParameterSpec.ScaleType.UNIT_LINEAR_SCALE
      },
      {
          "parameter_id": "decay",
          "double_value_spec": { 'min_value': 1e-6, 'max_value': 1e-2 },
          "scale_type": aip.StudySpec.ParameterSpec.ScaleType.UNIT_LINEAR_SCALE
      }],
   "algorithm": aip.StudySpec.Algorithm.RANDOM_SEARCH
}
```

Discrete is a finite set of values.

Double is an infinite set of values.

Int is essential infinite, though the range is bounded.

Create HPT Job

Create Hyperparameter Tuning Job

Called with job client.

```
def get_hyperparameter_tuning_job(name, silent=False):
    response = clients['job'].get_hyperparameter_tuning_job(name=name)
    if silent:
        return response

    print("name:", response.name)
    print("display_name:", response.display_name)
    print("state:", response.state)
    print("create_time:", response.create_time)
    print("update_time:", response.update_time)
    return response


job_response = get_hyperparameter_tuning_job(JOB_ID)
```

Trials are done when state says completed.

Async method. Poll for current status

Create HPT Job

Review Trial Results

Response has results for each trial

Find trial with the best objective outcome.

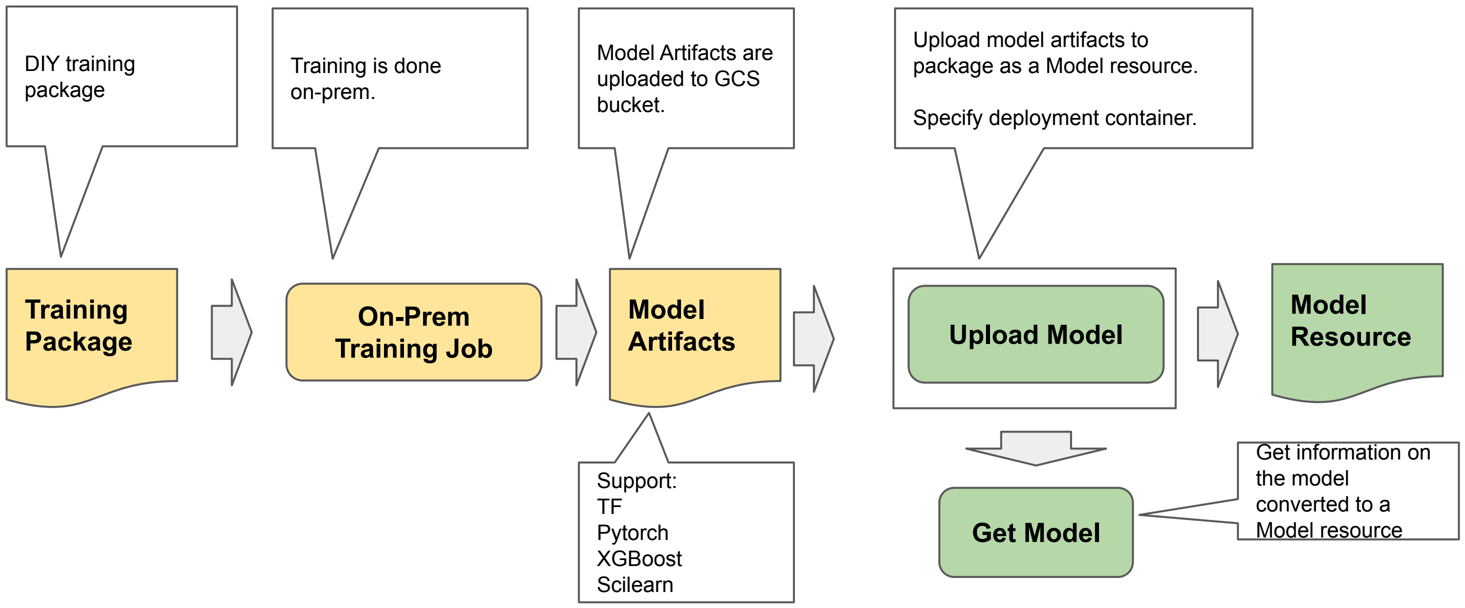
```
best = (None, None, None, 0.0)
for trial in job_response.trials:
   print(trial)
   # Keep track of the best outcome
   try:
      if float(trial.final_measurement.metrics[0].value) > best[3]:
         best = (trial.id, float(trial.parameters[0].value), float(trial.parameters[1].value),
               float(trial.final_measurement.metrics[0].value))
   except:
      pass


print("Decay", best[1])
print("Learning Rate", best[2])
```

Save these values for full custom training.

Workshop 7: Local (On-Prem) Training

On-Prem Training / Deployment

Converting On-Prem Model to Model Resource

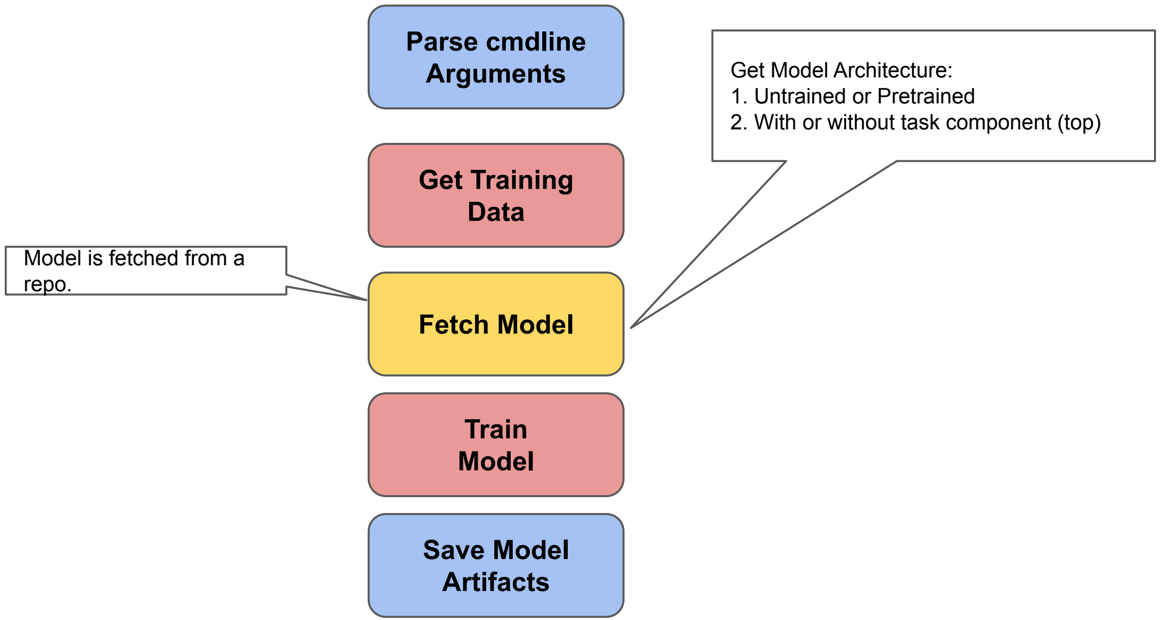


Workshop 8: Advanced Custom Jobs

- Custom Jobs using Model Gardens/Zoos
- Custom Training Containers
- Custom Deployment Containers
- Explainability for Custom Jobs
- A/B Testing
- Sandboxing

Workshop 8: Custom Jobs using Model Gardens

(Basic) Training File Contents when using Model Garden/Zoo



Fetch Model

Fetch Model from TFHub

Fetch a pertained without the task component.

Freeze existing layers

Add new task component.

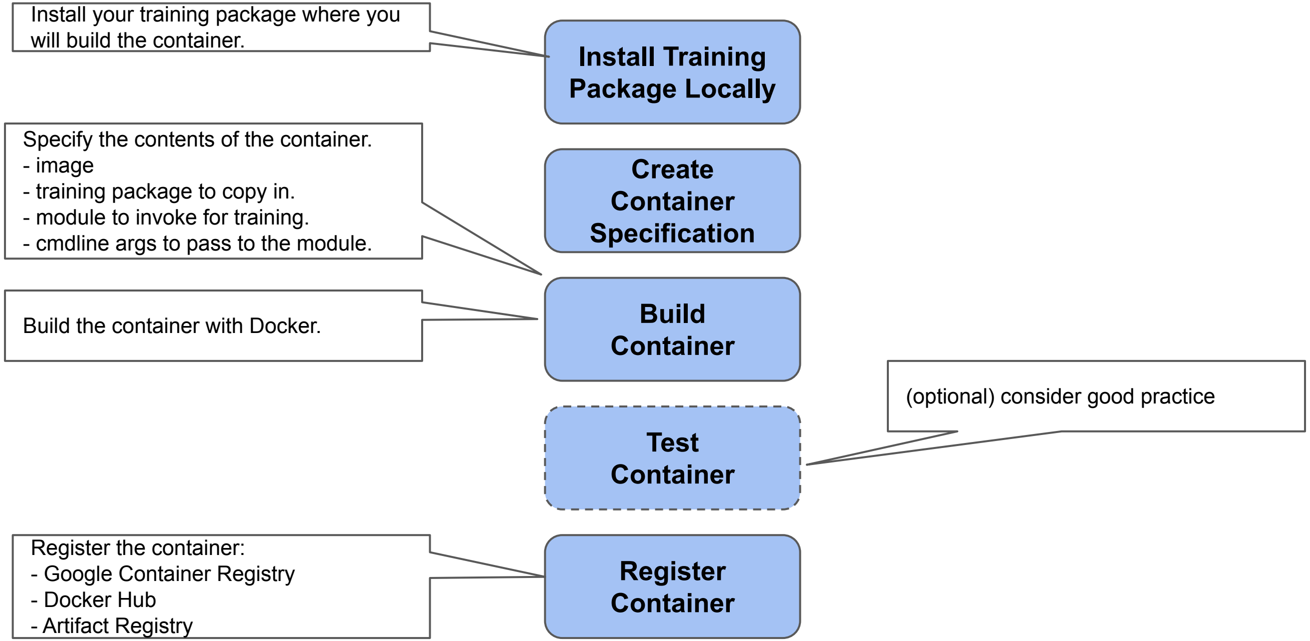
Fine-tune (transfer learning) the model.

Feature vector means to drop the task component.

```
import tensorflow as tf
import tensorflow_hub as hub

IMAGE_SHAPE = (224, 224)

model = tf.keras.Sequential([
   hub.KerasLayer("https://tfhub.dev/google/imagenet/resnet_v1_50/feature_vector/4",
              input_shape=IMAGE_SHAPE+(3,))
])

for layer in model.layers:
  model.trainable = False

model.add(GlobalAveragePooling())
model.add(Dense(10))

model.fit(data, ....... )

model_path_to_deploy = BUCKET_NAME + '/resnet'
```

Workshop 8: Custom Training Containers

Building a Custom Training Container



Create Container Specification

Creating a Container Specification

Install an existing image (alternately you can build the image requirements).

Copy your training package into the container.

Specify the module to invoke

```
FROM gcr.io/deeplearning-platform-release/tf2-cpu.2-1
WORKDIR /root

WORKDIR /

# Copies the trainer code to the docker image.
COPY trainer /trainer

# Sets up the entry point to invoke the trainer.
ENTRYPOINT ["python", "-m", "trainer.task"]
```

Build/Test/Register Container

Last Steps to Get Container Ready

- Specify a name for the container
- Build the container.
- Optionally test the container.
- Push the container to a registry.

```
TRAIN_IMAGE = "gcr.io/" + PROJECT_ID + "/cifar10:v1"

docker build <location_of_container_spec> -t $TRAIN_IMAGE

docker run $TRAIN_IMAGE --epochs=5

docker push $TRAIN_IMAGE
```

Create Custom Job

Create container specification for custom job

```
if DIRECT:
   CMDARGS = [
            "--model-dir=" + MODEL_DIR,
            "--epochs=" + str(EPOCHS),
            "--steps=" + str(STEPS),
          ]
else:
   CMDARGS = [
            "--epochs=" + str(EPOCHS),
            "--steps=" + str(STEPS),
          ]

container_spec = {
   "image_uri": TRAIN_IMAGE,
   "args": CMDARGS,
}
```

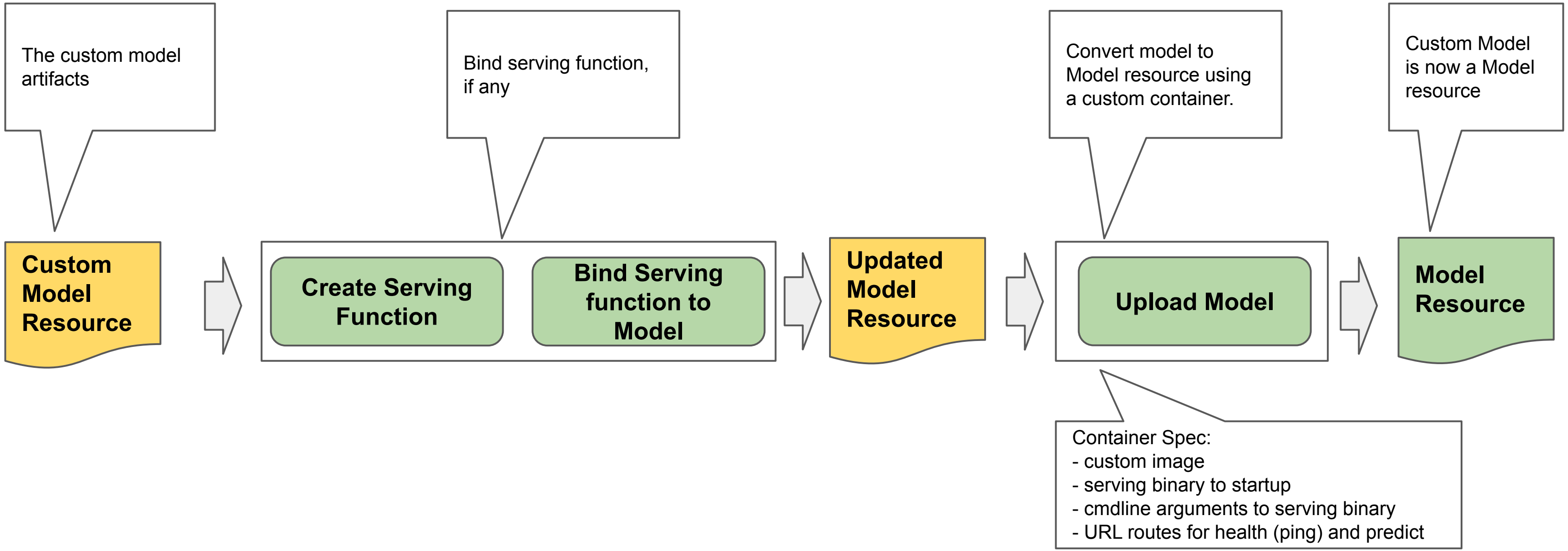
Specify the container image in the registry.

Specify cmdline args to pass to the module that is invoked in the container.

Worker pool specification will not have a python package specification – it is already in the custom container.

Workshop 8: Custom Deployment Containers

Custom Serving Container



Upload Model

Upload Custom Model with Custom Container

Specify the custom container image for serving

Specify the serving binary to startup (e.g. TF Serving).

Specify cmdline args to pass to the serving binary.

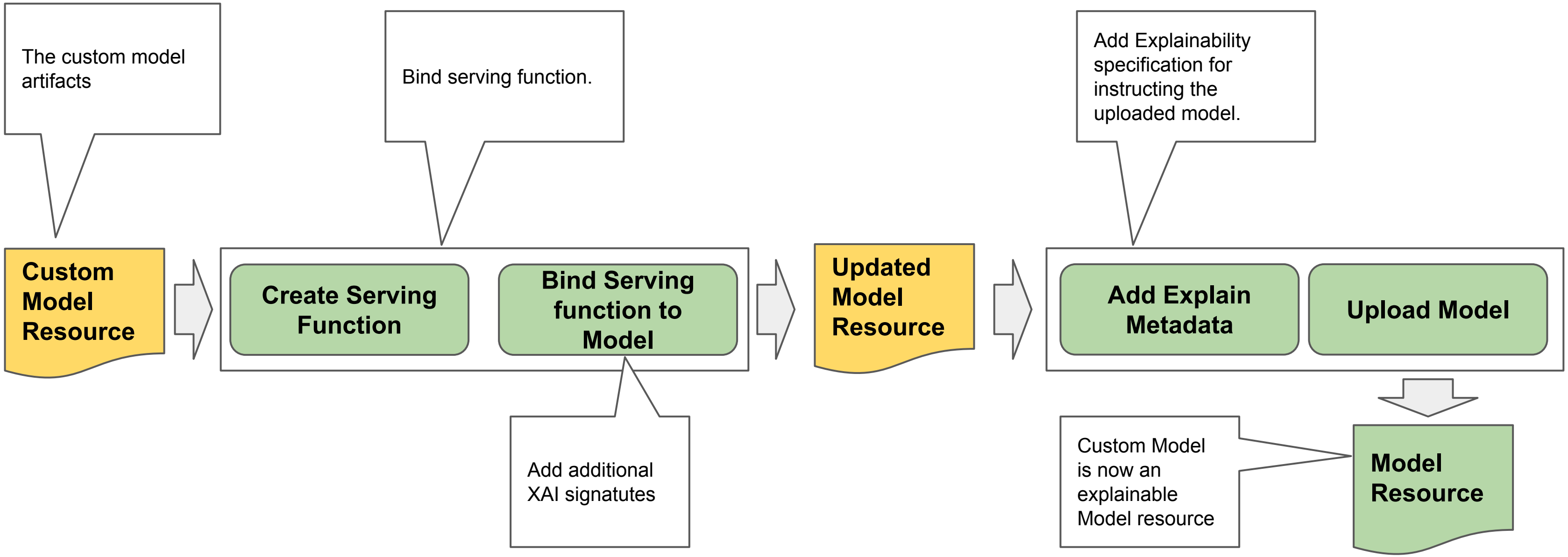
Specify the health and predict routes

The HTTP port the server will listen to for REST calls.

```
def upload_model(display_name, image_uri, model_uri):
  model = {
    "display_name": display_name,
    "metadata_schema_uri": "",
    "artifact_uri": model_uri,
    "container_spec": {
      "image_uri": image_uri,
      "command": ["/usr/bin/tensorflow_model_server"],
      "args": [
        "--model_name=" + MODEL_NAME,
        "--model_base_path=" + "$(AIP_STORAGE_URI)",
        "--rest_api_port=8080",
        "--port=8500",
        "--file_system_poll_wait_seconds=31540000"
      ],
      "health_route": "/v1/models/" + MODEL_NAME,
      "predict_route": "/v1/models/" + MODEL_NAME + ":predict",
    },
  }
  response = clients['model'].upload_model(parent=PARENT, model=model)
  print("Long running operation:", response.operation.name)
  upload_model_response = response.result(timeout=180)
```

Workshop 8: Explainability for Custom Models

Custom Models with Explainability



Bind Serving function to Model

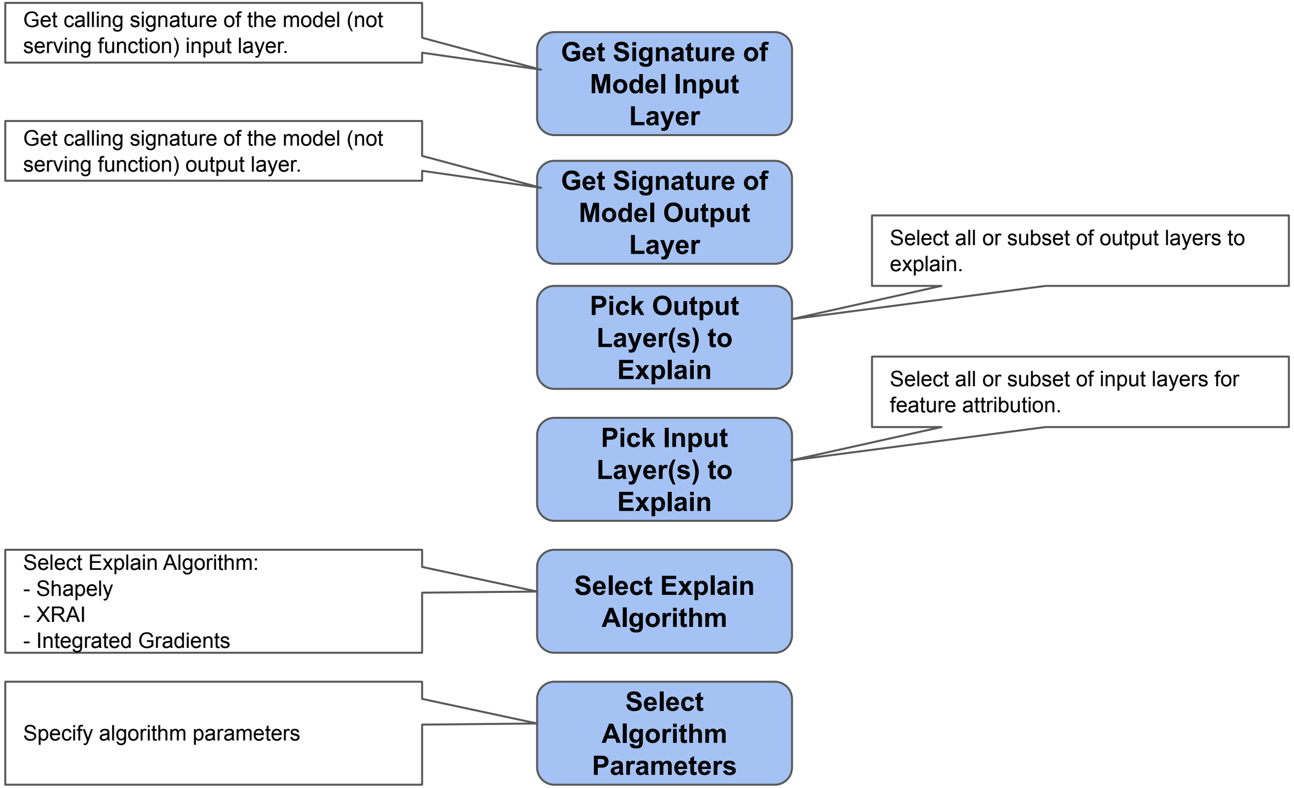
Need to add 2 additional signatures when binding the serving function for XAI

The preprocessing layer in the serving function

The concrete function (interface) between serving function and model

```
tf.saved_model.save(model, model_path_to_deploy, signatures={
    'serving_default': serving_fn,
    # Required for XAI
    'xai_preprocess': preprocess_fn,
    'xai_model': m_call
})
```

Construct Explain Data



Get Signature of Model Input Layer

Get Signature of Model Output Layer

Get signatures of model input and output layers

The model input layers

The model output layers

```
loaded = tf.saved_model.load(model_path_to_deploy)

serving_input = list(loaded.signatures['serving_default'].structured_input_signature[1].keys())[0]
print('Serving function input:', serving_input)
serving_output = list(loaded.signatures['serving_default'].structured_outputs.keys())[0]
print('Serving function output:', serving_output)

input_name = model.input.name
print('Model input name:', input_name)
output_name = model.output.name
print('Model output name:', output_name)
```

Output Layers to Explain

Single Output

For example, in a probability output \[0.1, 0.2, 0.7\] for classification, one wants an explanation for 0.7.

Consider the following formulae, where the output is `y` and that is what we want to explain.

$$y = f(x)$$

Explain ‘y’

Multiple Outputs

Consider the following formulae, where the outputs are `y` and `z`.

Since we can only do attribution for one scalar output, we have to pick whether we want to explain the output `y` or `z`.

Assume in this example the model is object detection and y and z are the bounding box and the object classification. You would want to pick which of the two outputs to explain.

Explain ‘y’

$$y, z = f(x)$$

Not explain ‘z’

Input Layers to Attribute

Single Input

For example, in a classification model, one wants to determine how the input contributed (attribution) to the output being explained.

Consider the following formulae, where the output is `y` to explain and ‘a’ is the feature to do attribution.

$$y = f(a)$$

Explain ‘y’

Multiple Outputs

Consider the following formulae, where `a` and `b` are the features. We have to pick which features to explain how the contributed.

Assume that this model is deployed for A/B testing, where `a` are the data_items for the prediction and `b` identifies whether the model instance is A or B. You would want to pick `a` (or some subset of) for the features, and not `b` since it does not contribute to the prediction.

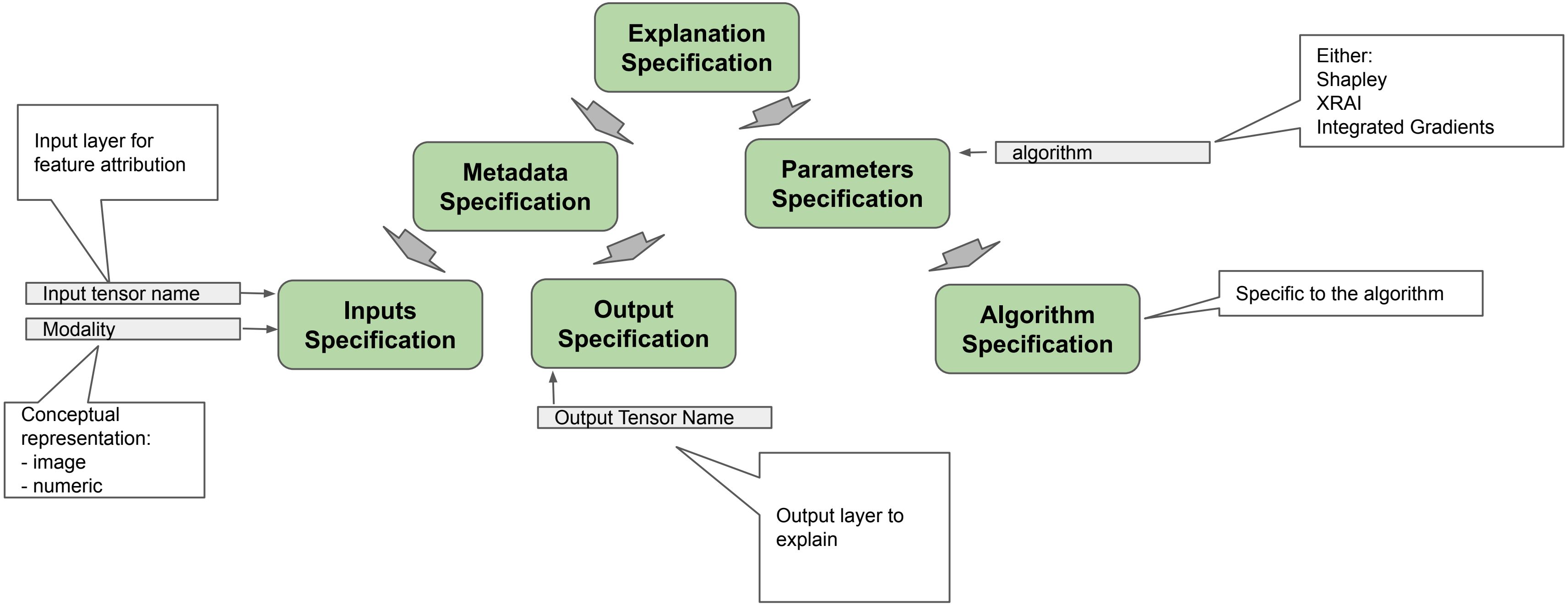
Attribute ‘a’

$$y = f(a,b)$$

Not attribute ‘b’

Add Explain Metadata

Construct Explanation Specification



Upload Model

Upload Model with Explanation Specification

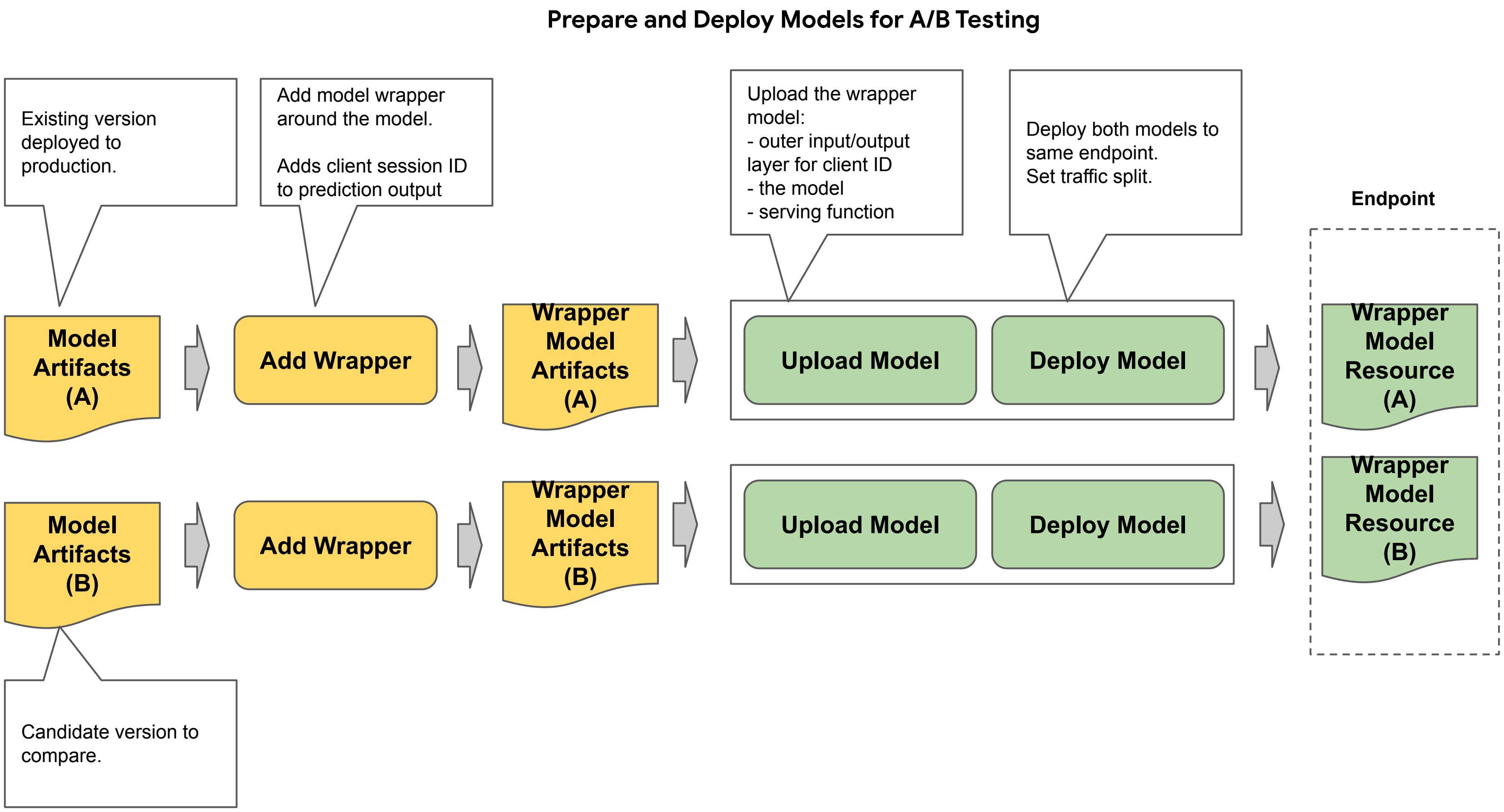
Specify the input for attribution

Specify the output to explain

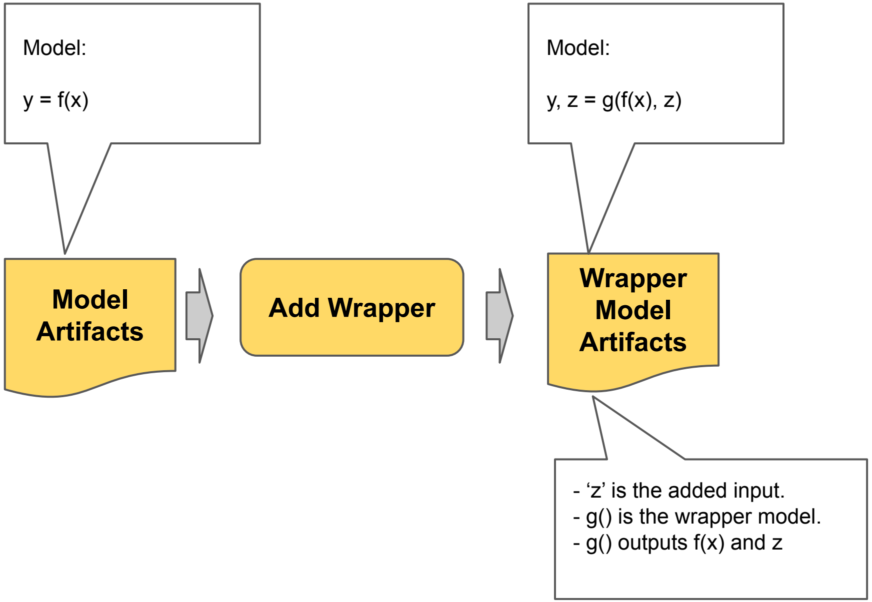
Add explanation spec to upload model.

```
INPUT_METADATA = {'input_tensor_name': CONCRETE_INPUT,
     "modality": "image"
    }

OUTPUT_METADATA = {'output_tensor_name': serving_output}

input_metadata = aip.ExplanationMetadata.InputMetadata(INPUT_METADATA)
output_metadata = aip.ExplanationMetadata.OutputMetadata(OUTPUT_METADATA)

metadata=aip.ExplanationMetadata(
    inputs={'image': input_metadata},
    outputs={'class' : output_metadata}
)

explanation_spec = aip.ExplanationSpec(
  metadata=metadata,
  parameters=parameters
)
def upload_model(display_name, image_uri, model_uri):

  model = aip.Model(display_name=display_name,
         artifact_uri=model_uri,
         metadata_schema_uri="",
         explanation_spec=explanation_spec,
         container_spec={
          "image_uri": image_uri
         })

  response = clients['model'].upload_model(parent=PARENT, model=model)
  print("Long running operation:", response.operation.name)
  upload_model_response = response.result(timeout=180)
```

Workshop 8: A/B Testing


Prepare and Deploy Models for A/B Testing
Existing version deployed to production.
Add model wrapper around the model.
Adds client session ID to prediction output
Upload the wrapper model:
- outer input/output layer for client ID
- the model
- serving function
Deploy both models to same endpoint. Set traffic split.
Endpoint
Model Artifacts (A)
Add Wrapper
Wrapper Model Artifacts (A)
Upload Model
Deploy Model
Wrapper Model Resource (A)
Model Artifacts (B)
Add Wrapper
Wrapper Model Artifacts (B)
Upload Model
Deploy Model
Wrapper Model Resource (B)
Candidate version to compare.

Constructing a Wrapper Model



Add Wrapper

Construct a Wrapper Model

The model's input and output layer.
y= f(x)

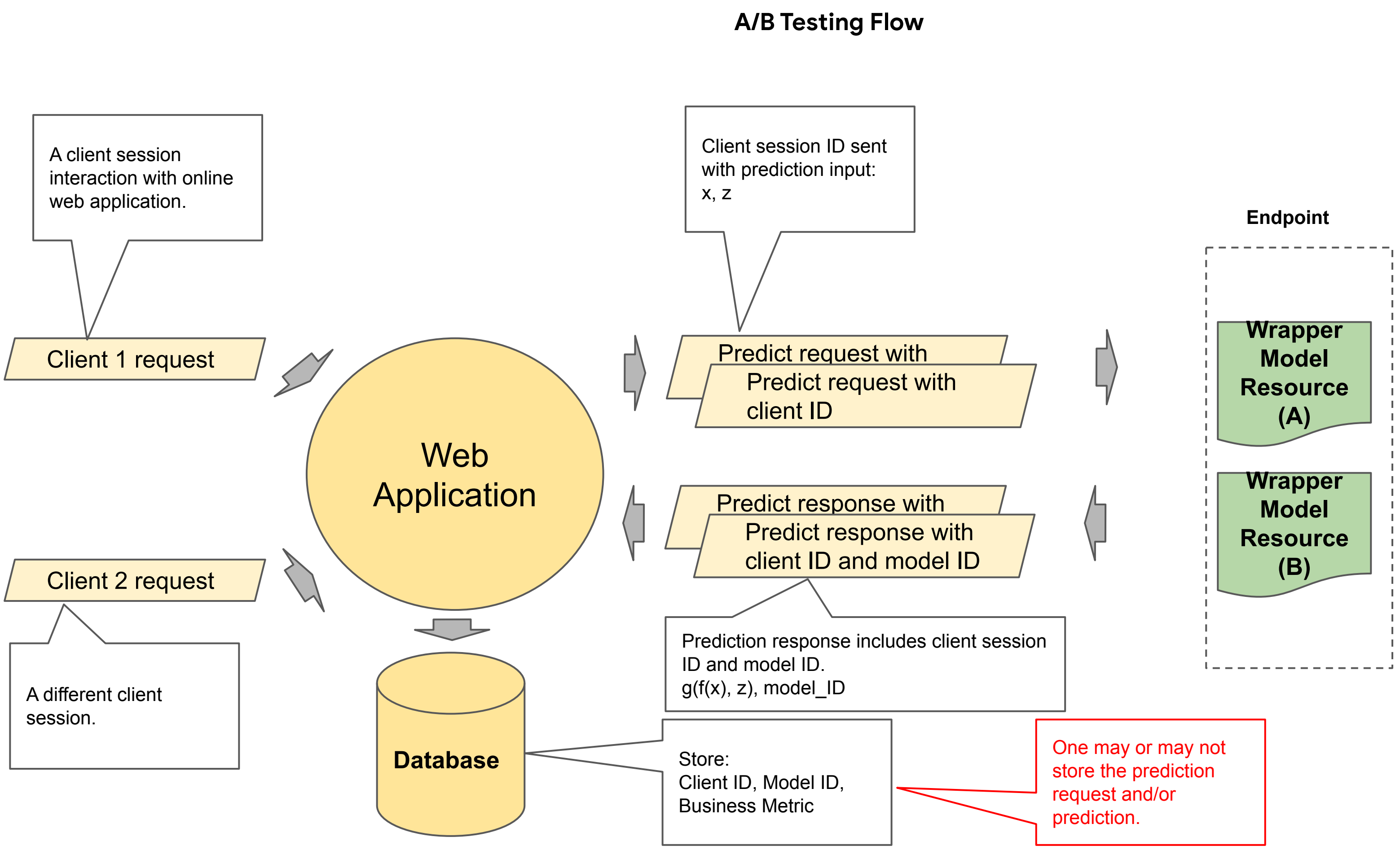
The new input layer (client ID)
z

The wrapper input layers:
x, z

The wrapper output layers:
f(x), z

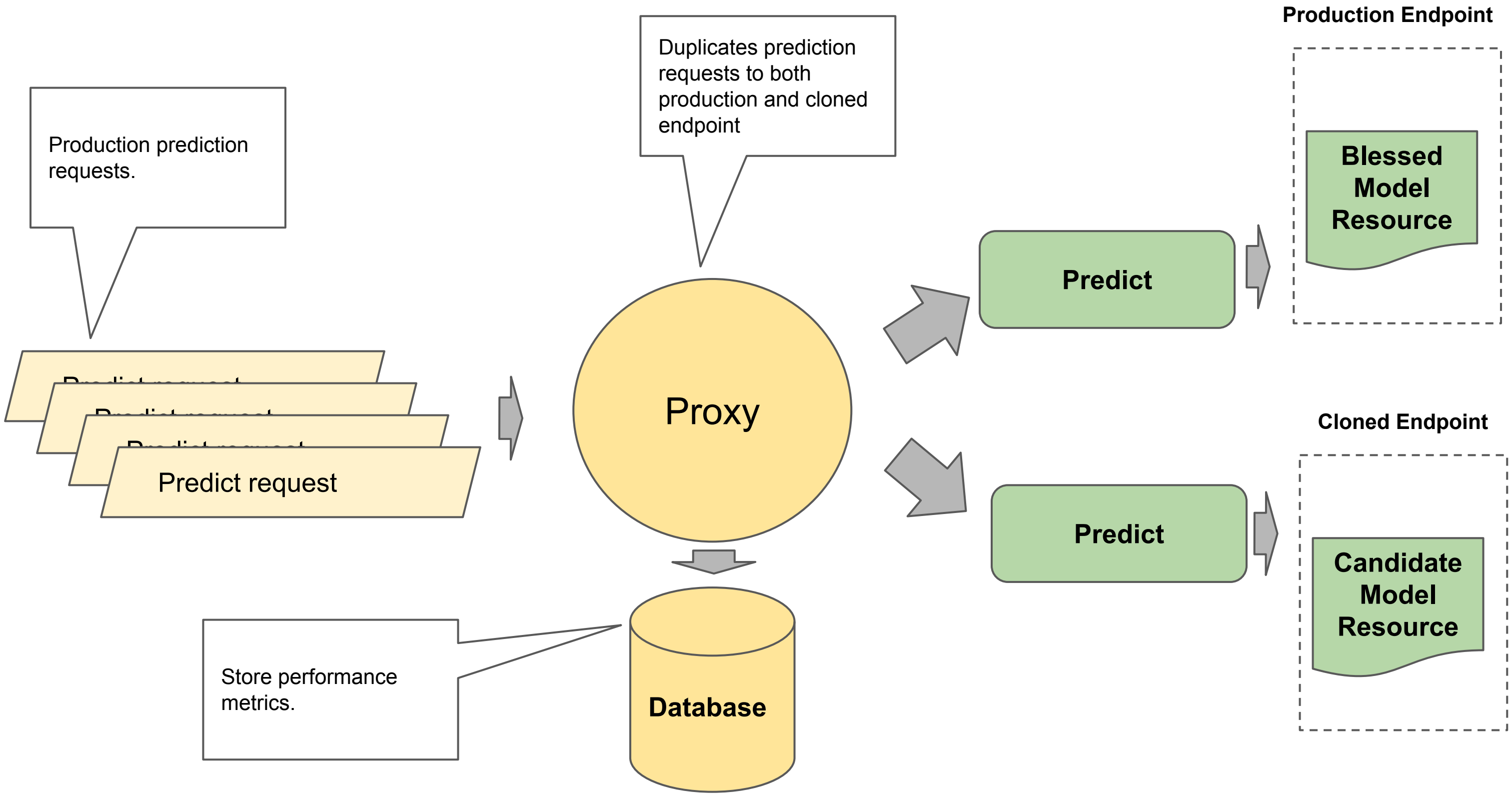
The wrapper model:
y, z = g(f(x), z)

```
from tensorflow.keras import Input, Model

input = model.inputs[0]
output = model.outputs[0]

client_input = Input((1,))

inputs = [input, client_input]
outputs  = [output, client_input]

wrapper_model = Model(inputs, outputs)
```


A/B Testing Flow
A client session interaction with online web application.
Client 1 request
Client 2 request
A different client session.
Web Application
Database
Client session ID sent with prediction input: x, z
Predict request with
Predict request with client ID
Predict response with
Predict response with client ID and model ID
Prediction response includes client session ID and model ID. g(f(x), z), model_ID
Store: Client ID, Model ID, Business Metric
One may or may not store the prediction request and/or prediction.
Endpoint
Wrapper Model Resource (A)
Wrapper Model Resource (B)

Workshop 8: Sandboxing

Sandboxing


Production prediction requests.
Predict request
Duplicates prediction requests to both production and cloned endpoint
Proxy
Store performance metrics.
Database
Predict
Predict
Production Endpoint
Blessed Model Resource
Cloned Endpoint
Candidate Model Resource