MLOps on Google Cloud

Training Course

What this course is **NOT about**

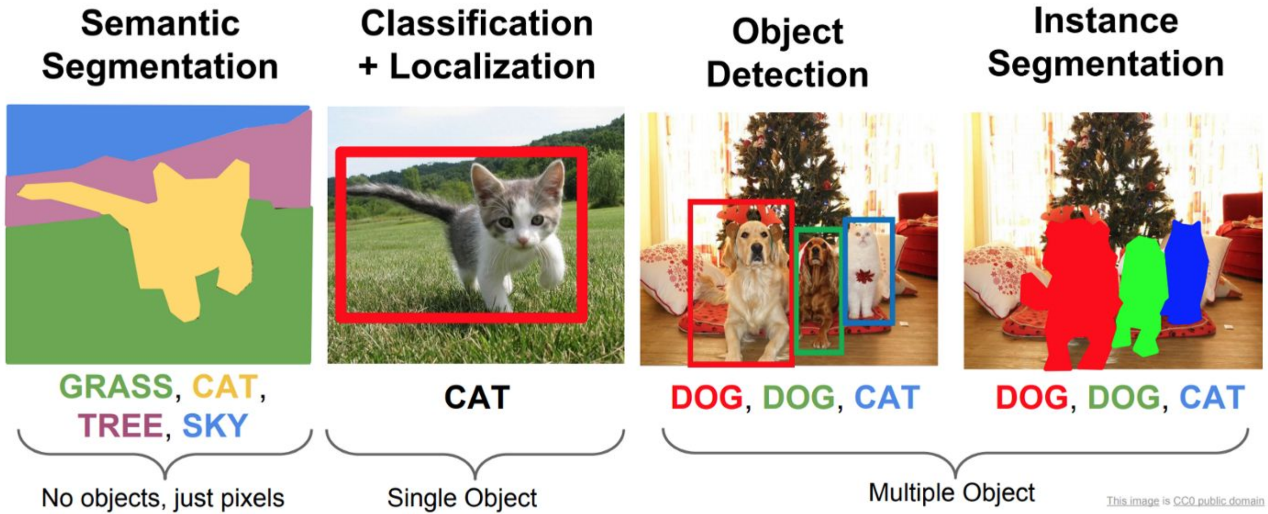
- Data Science
- Designing / Building Models
- Math

What this course is **about**

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

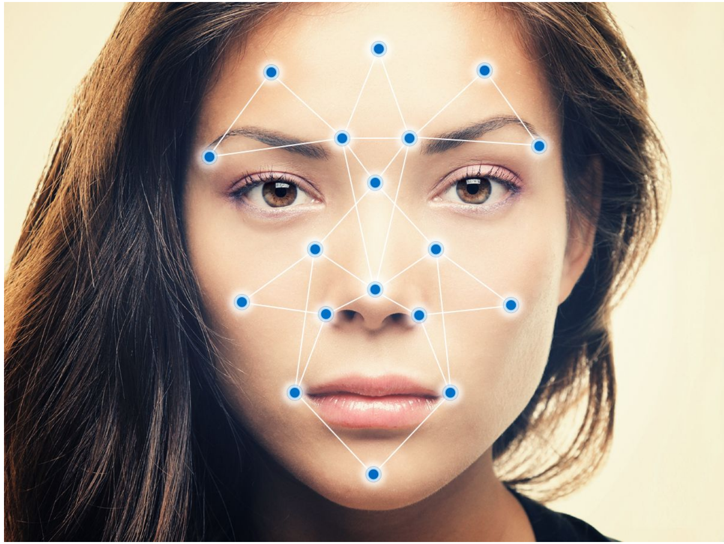
Machine Learning Overview

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



Machine Learning Overview

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

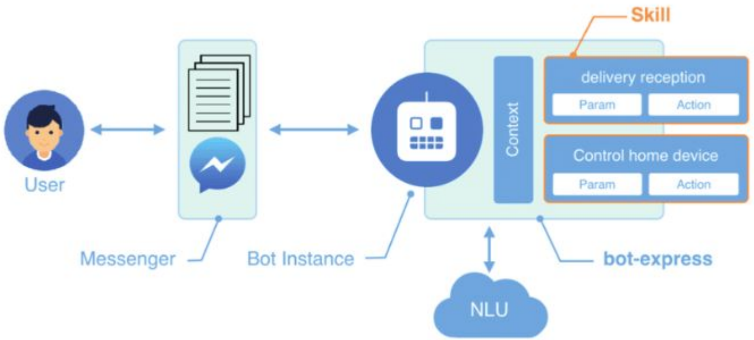


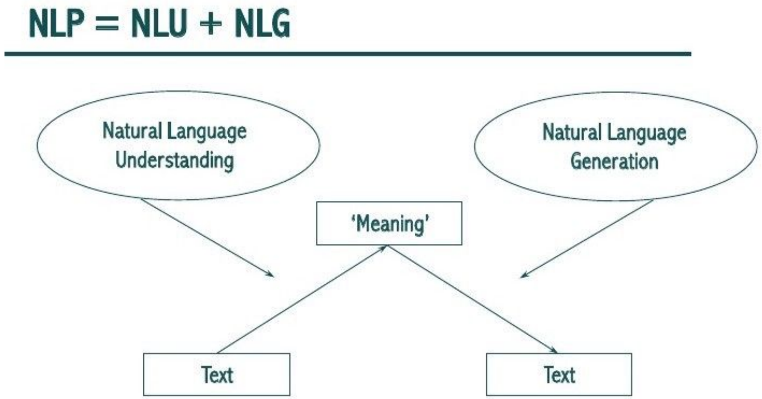
Machine Learning Overview

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

Machine Learning Overview

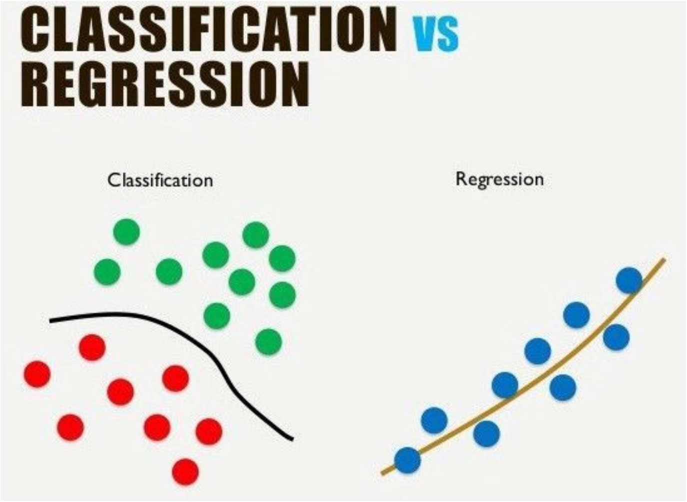
- Natural Language Generation (Text/Audio)
  - Text-2-Speech / Speech-2-Text
  - Summarization
  - Chat

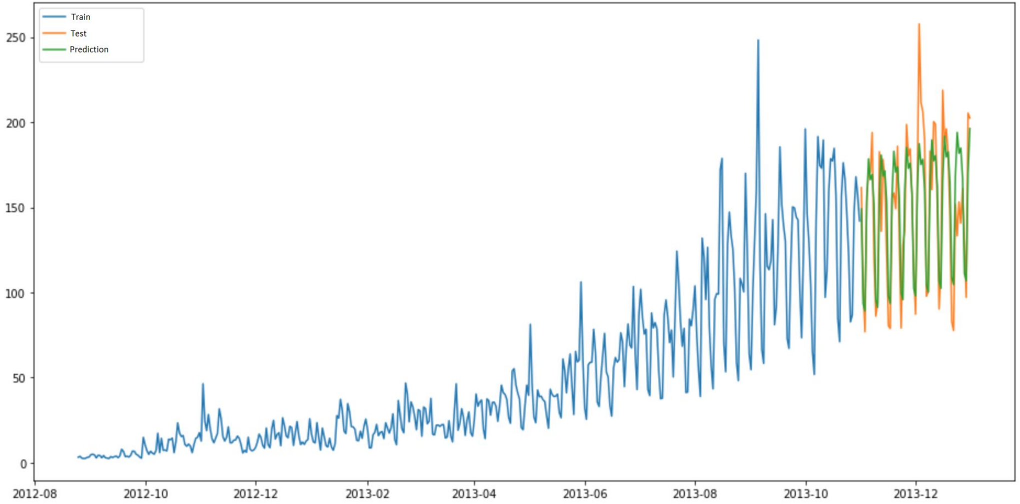




Machine Learning Overview

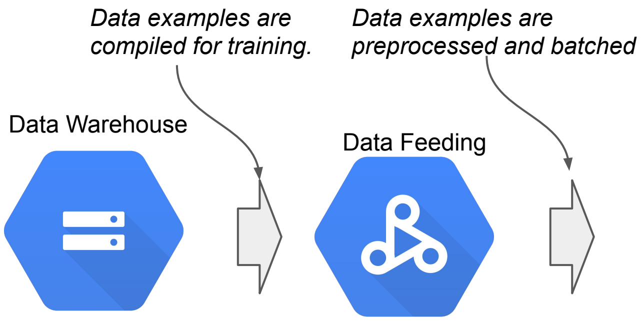
- Structured Data (Tabular, Databases)
  - Classification
  - Regression (Real Number)
  - Forecasting (time-series)





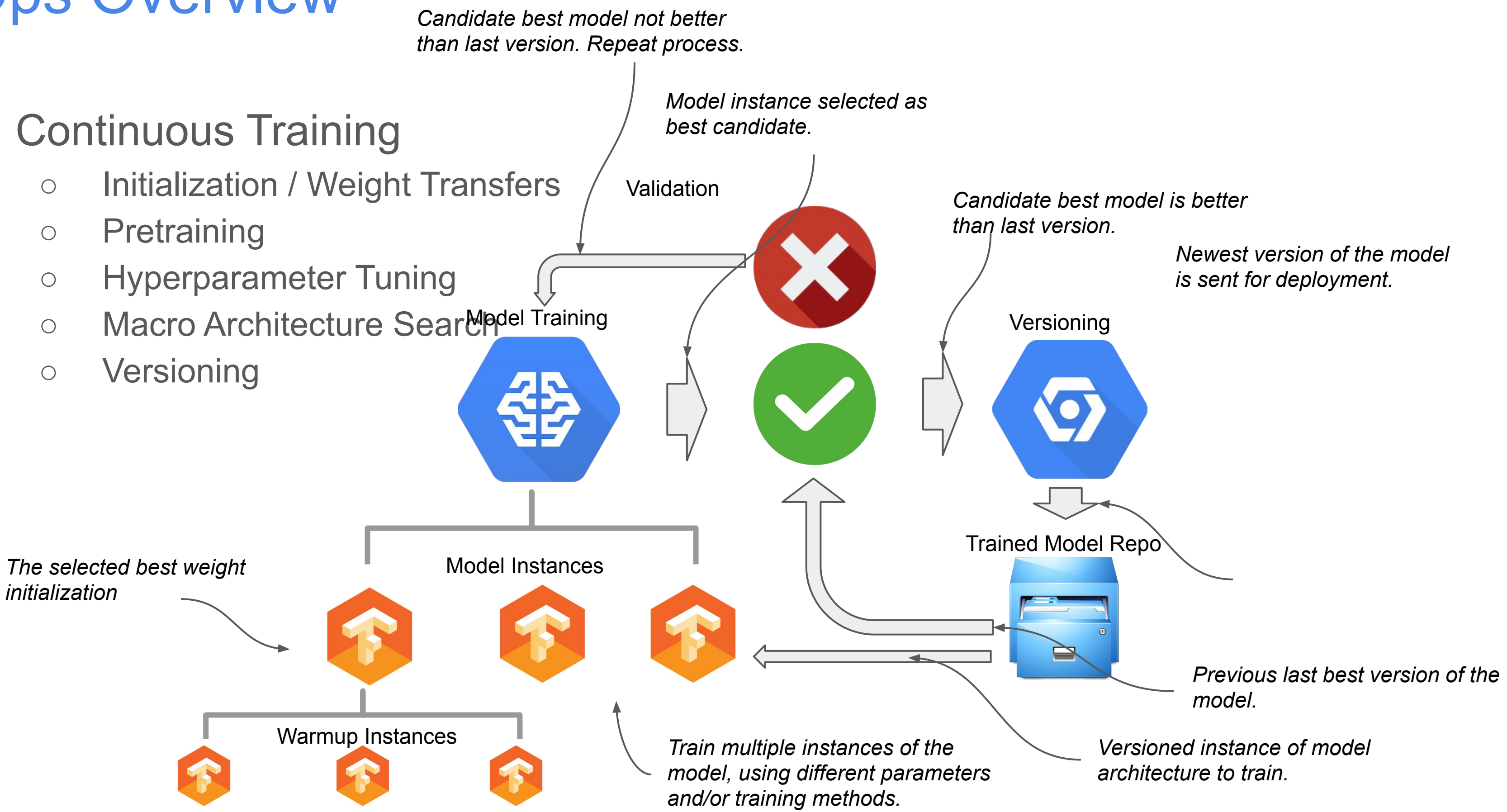
MLOps Overview

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



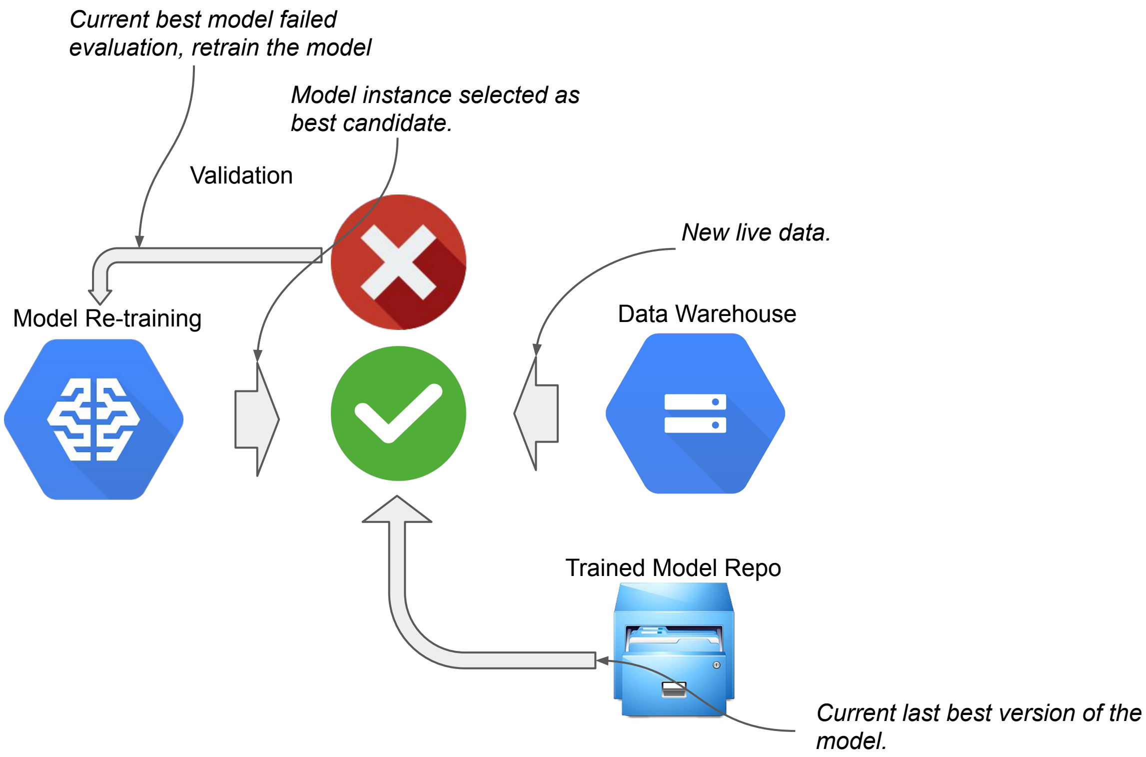
MLOps Overview

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



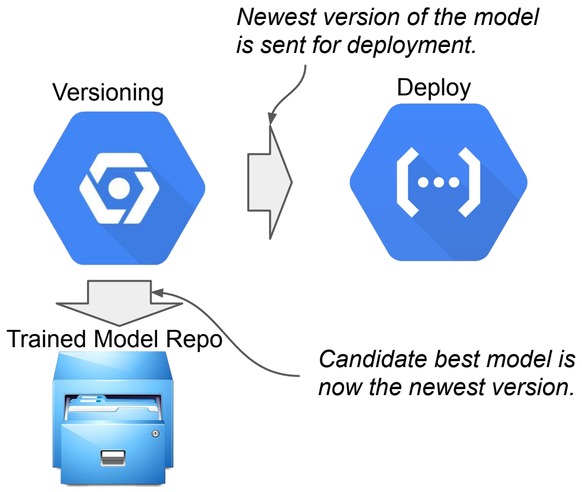
MLOps Overview

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



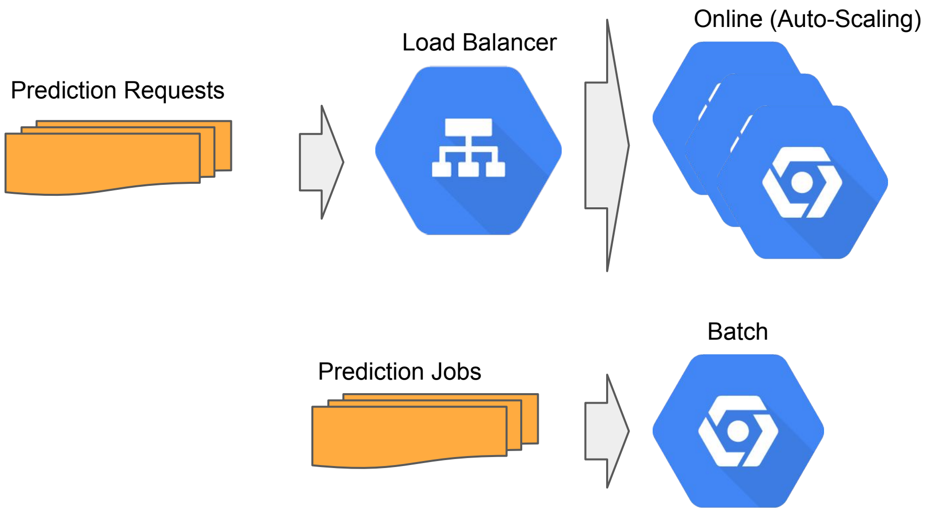
MLOps Overview

- Deployment
  - Scaling
  - Load Balancing
  - Latency
  - Edge



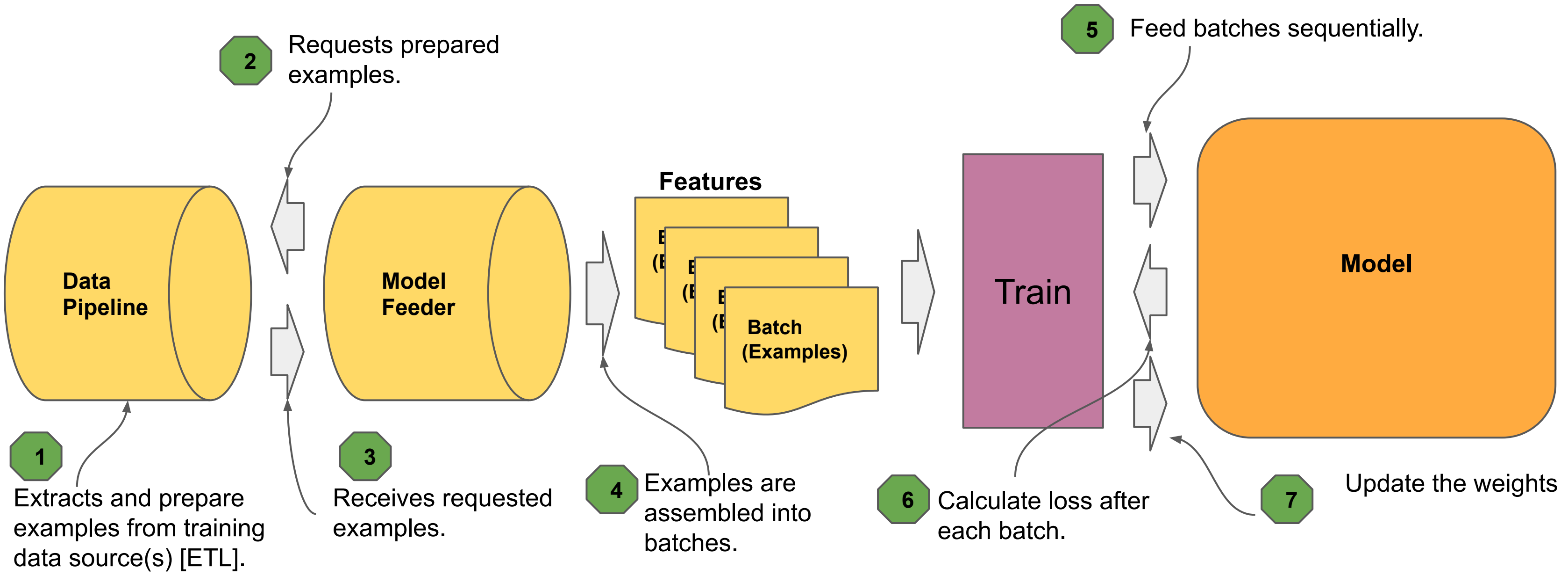
MLOps Overview

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



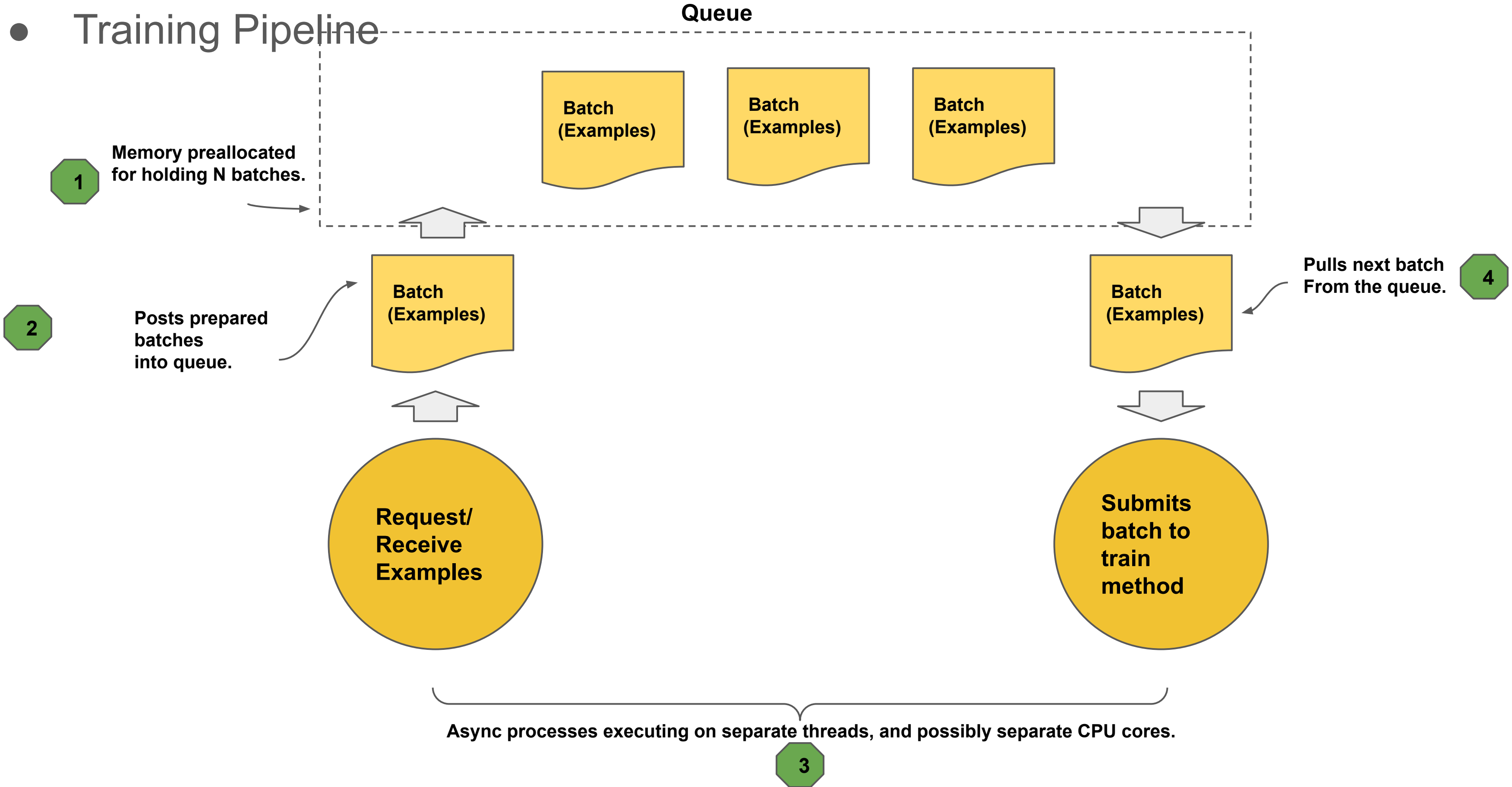
ML end-2-end production pipeline

- Data Pipeline



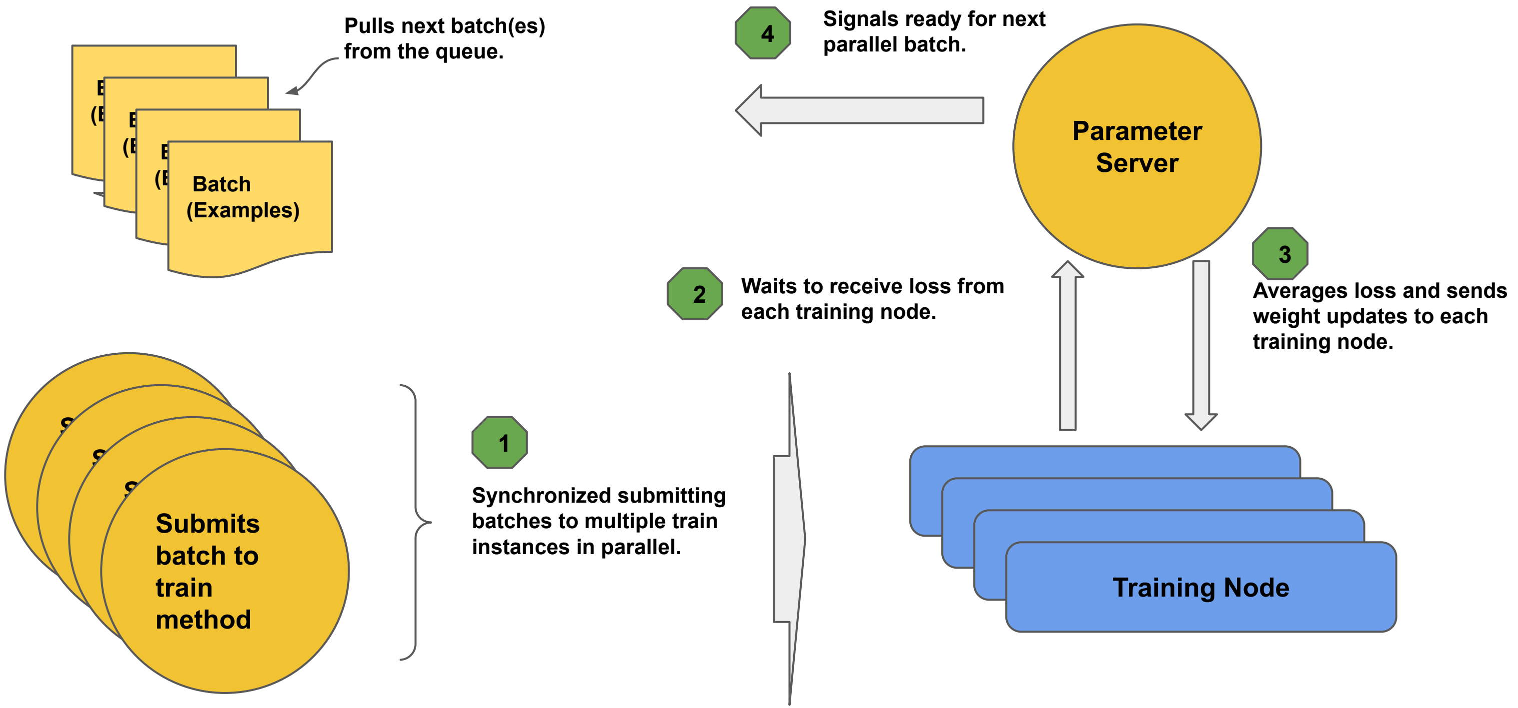
ML end-2-end production pipeline

- Training Pipeline



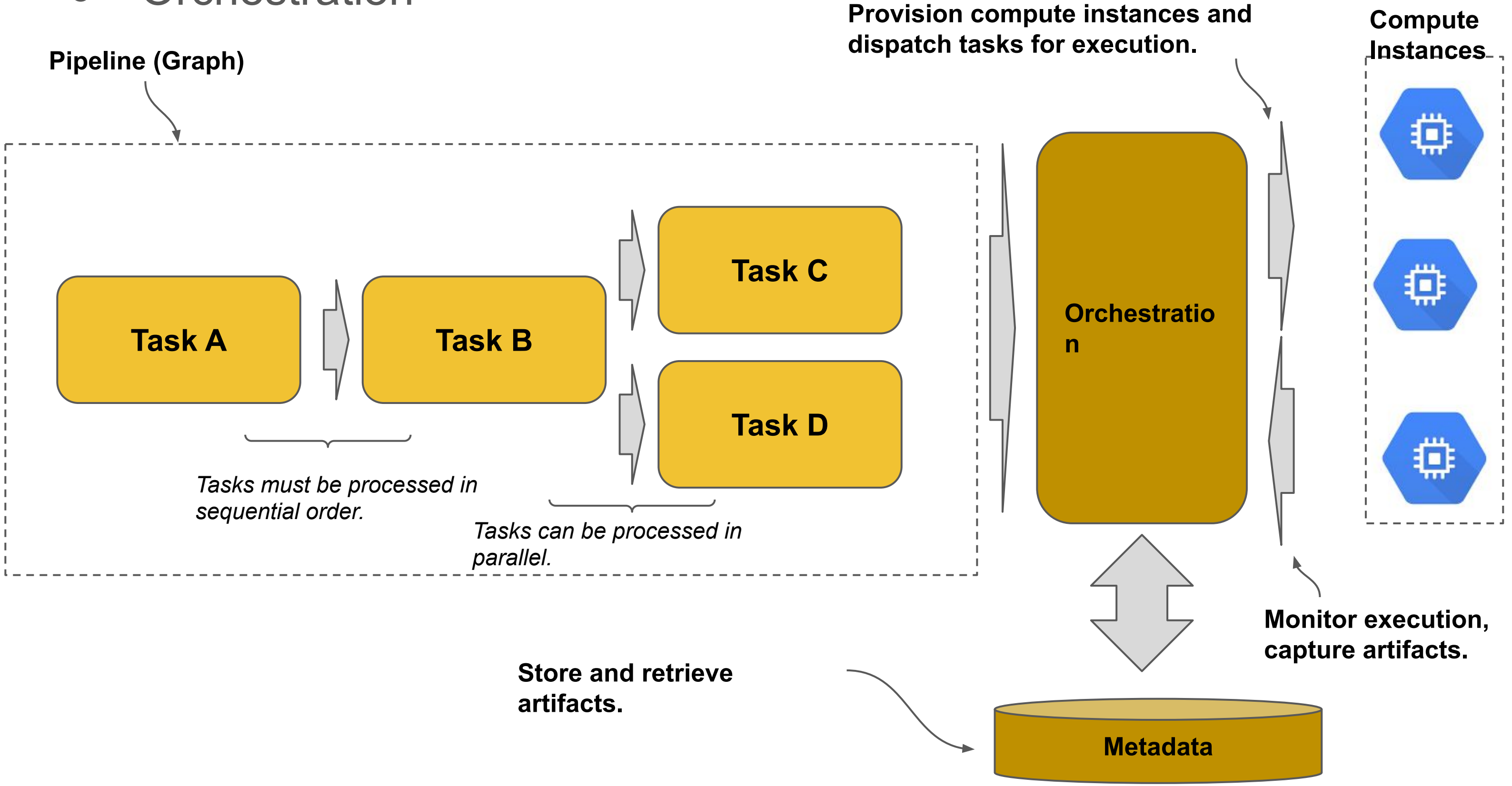
ML end-2-end production pipeline

- Training Pipeline



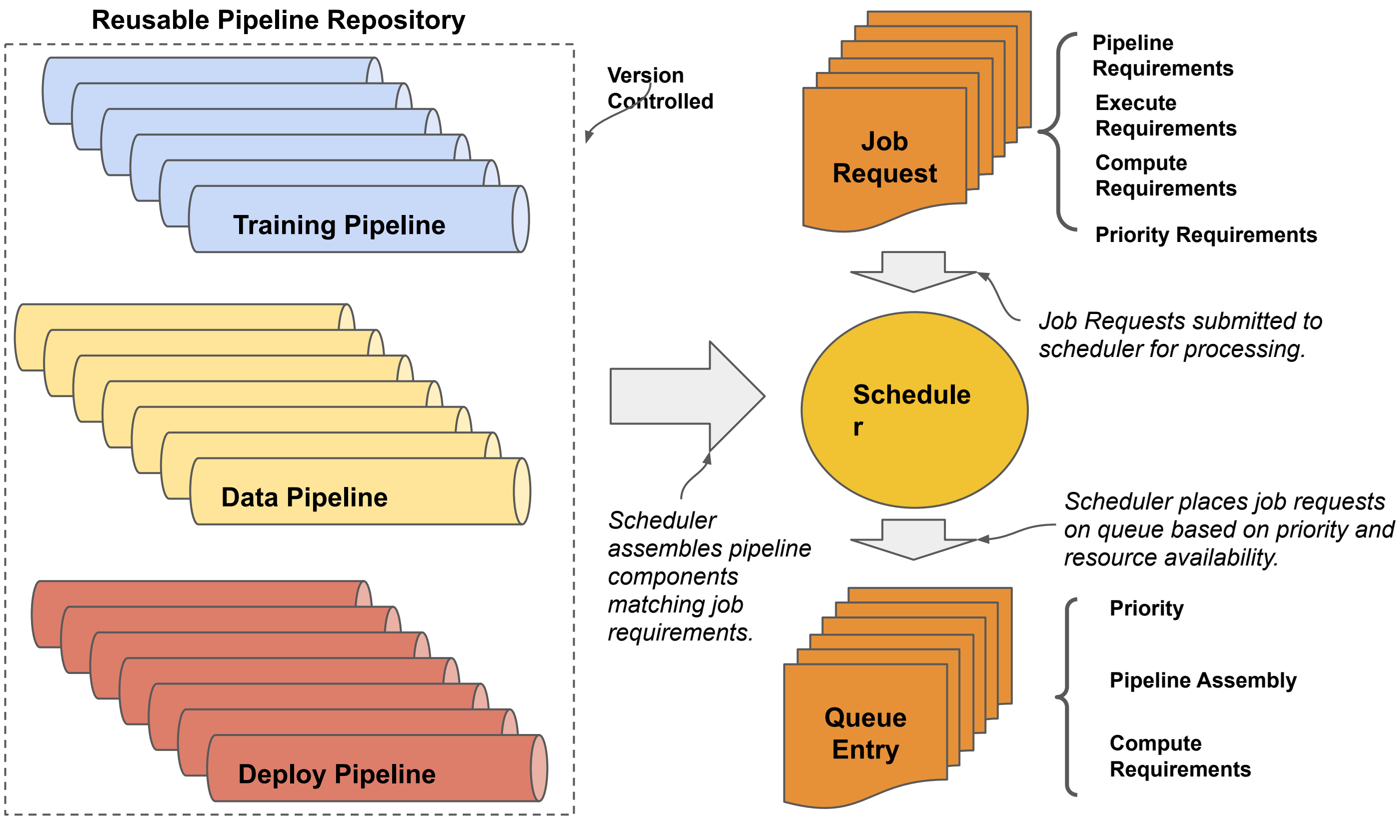
ML end-2-end production pipeline

- Orchestration



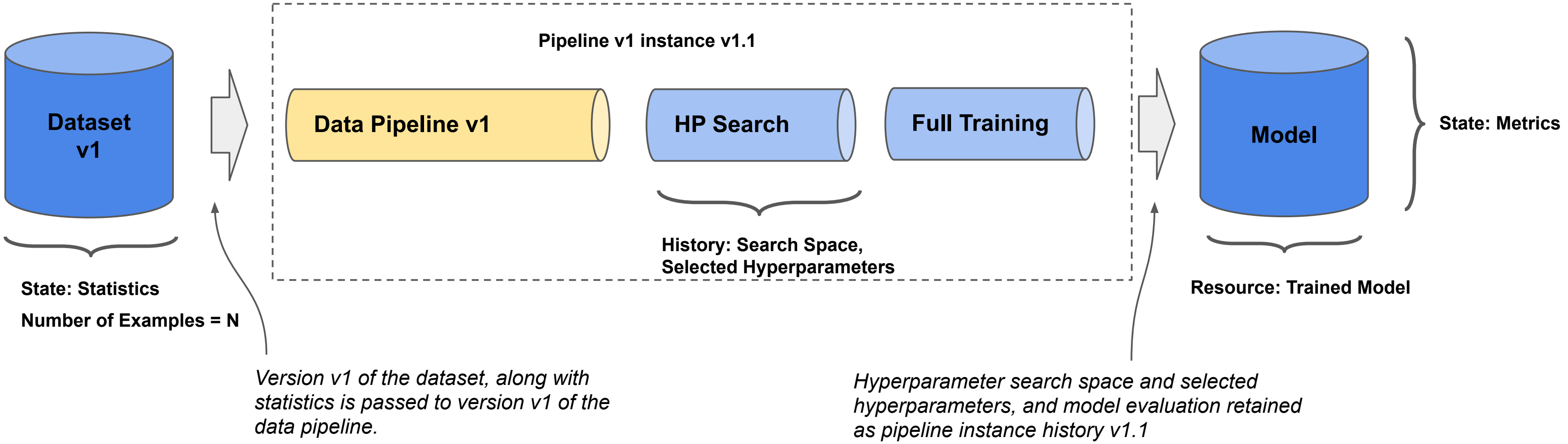
ML end-2-end production pipeline

- Pipeline Components



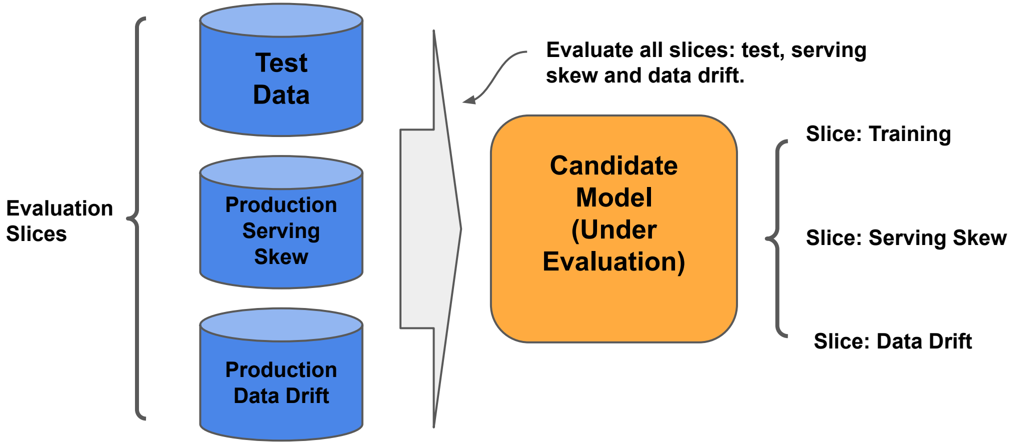
ML end-2-end production pipeline

- Heuristics



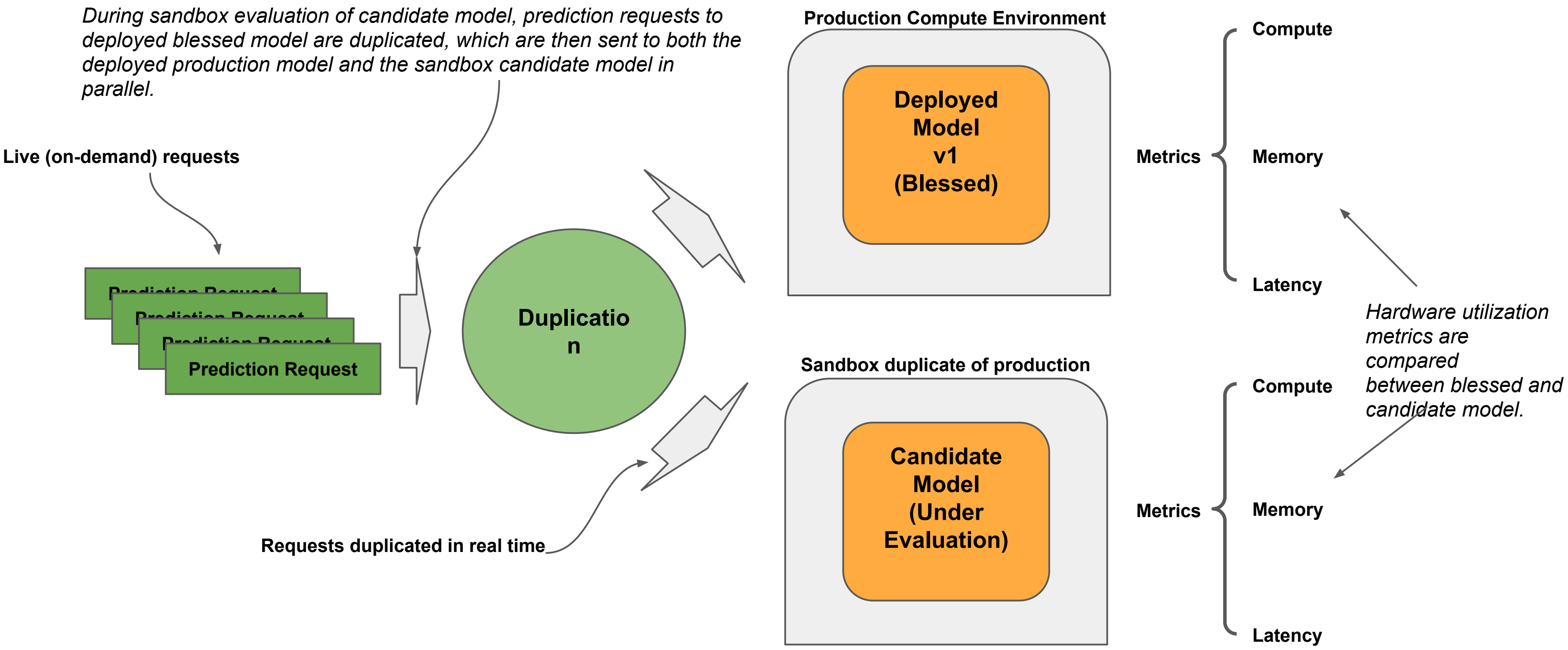
ML end-2-end production pipeline

- Evaluation Slicing



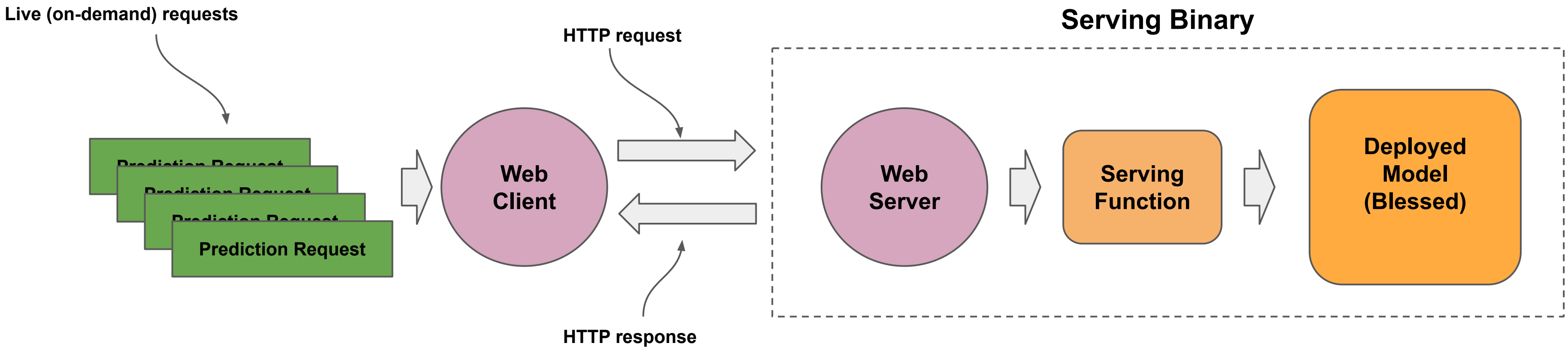
ML end-2-end production pipeline

- Sandboxing



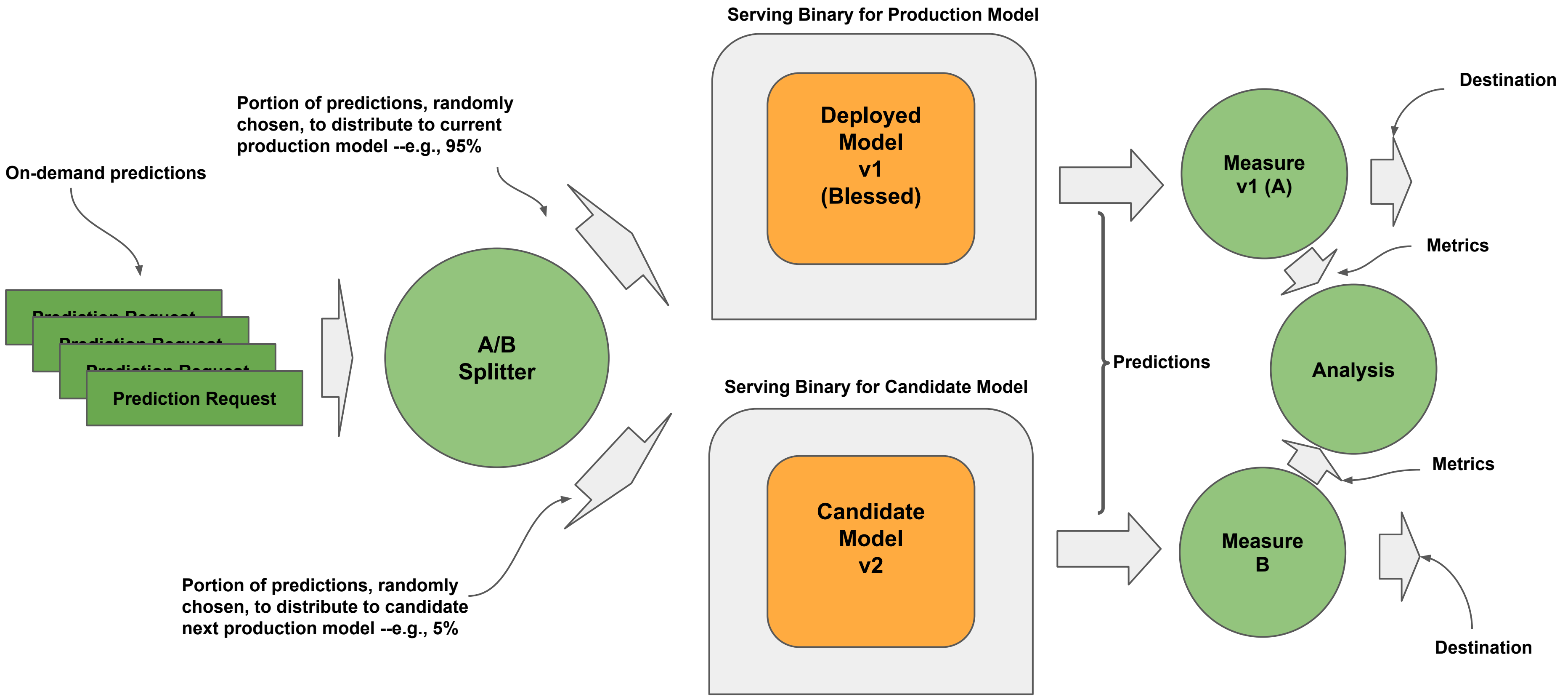
ML end-2-end production pipeline

- Serving Containers



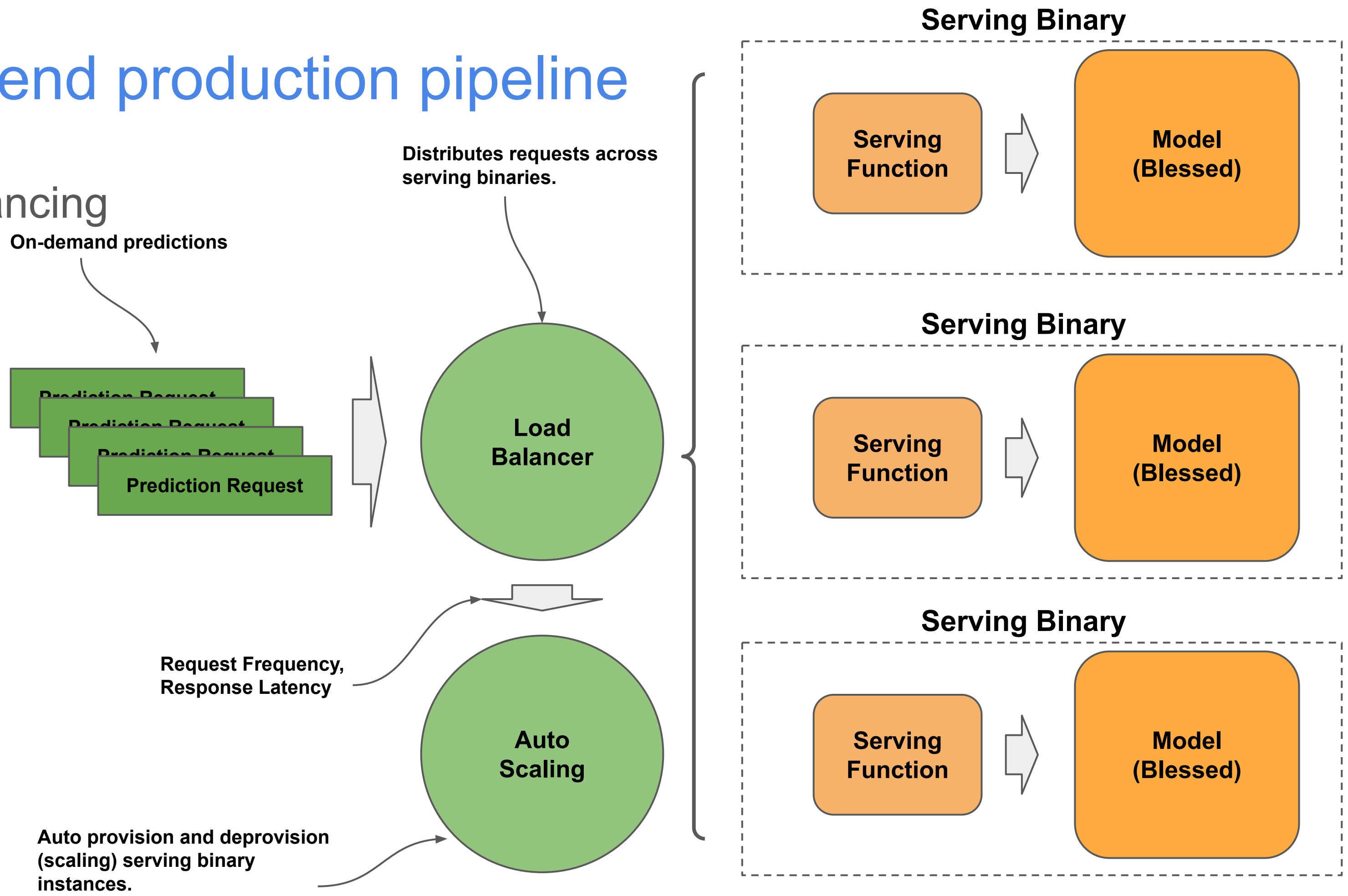
ML end-2-end production pipeline

- A/B Testing



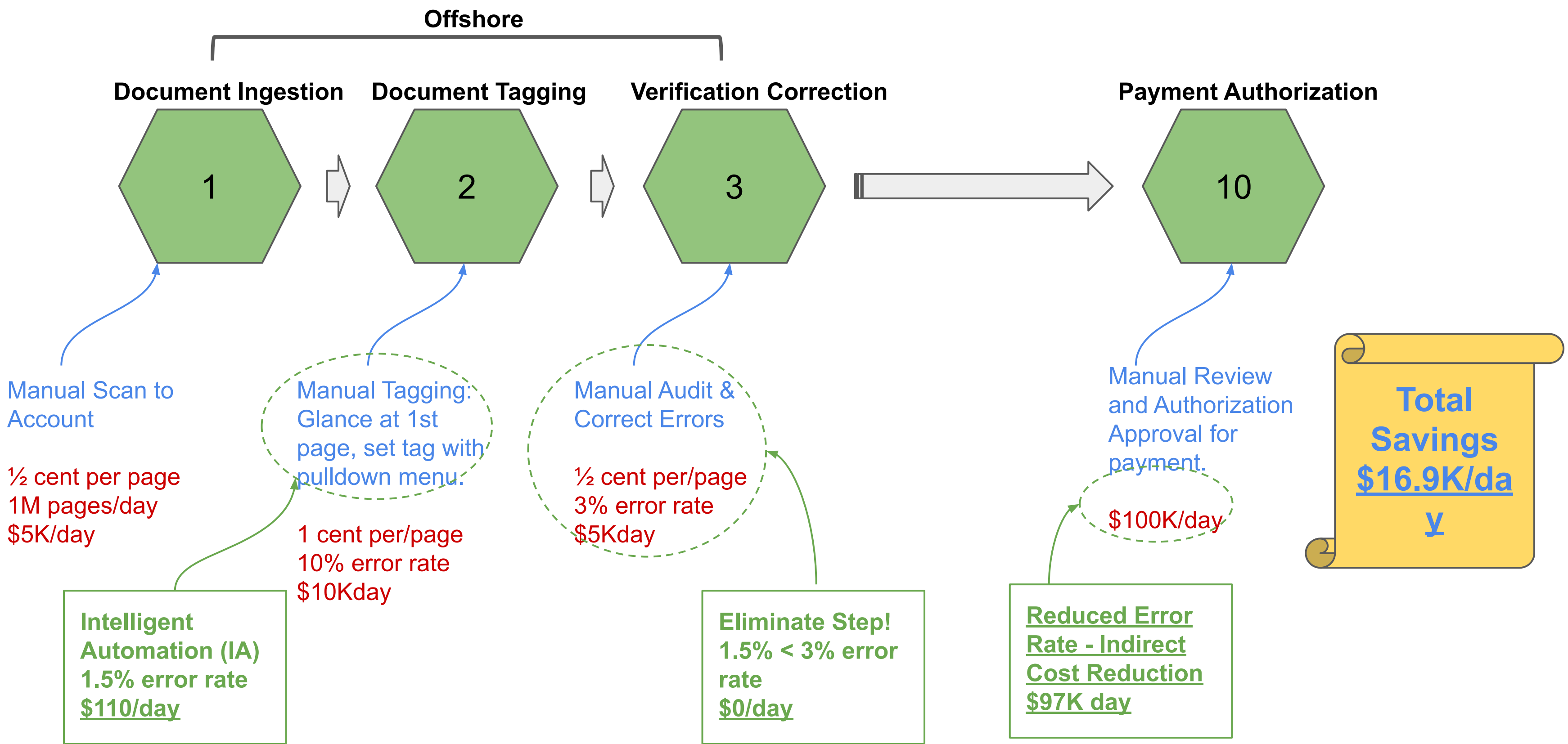
ML end-2-end production pipeline

- Load Balancing

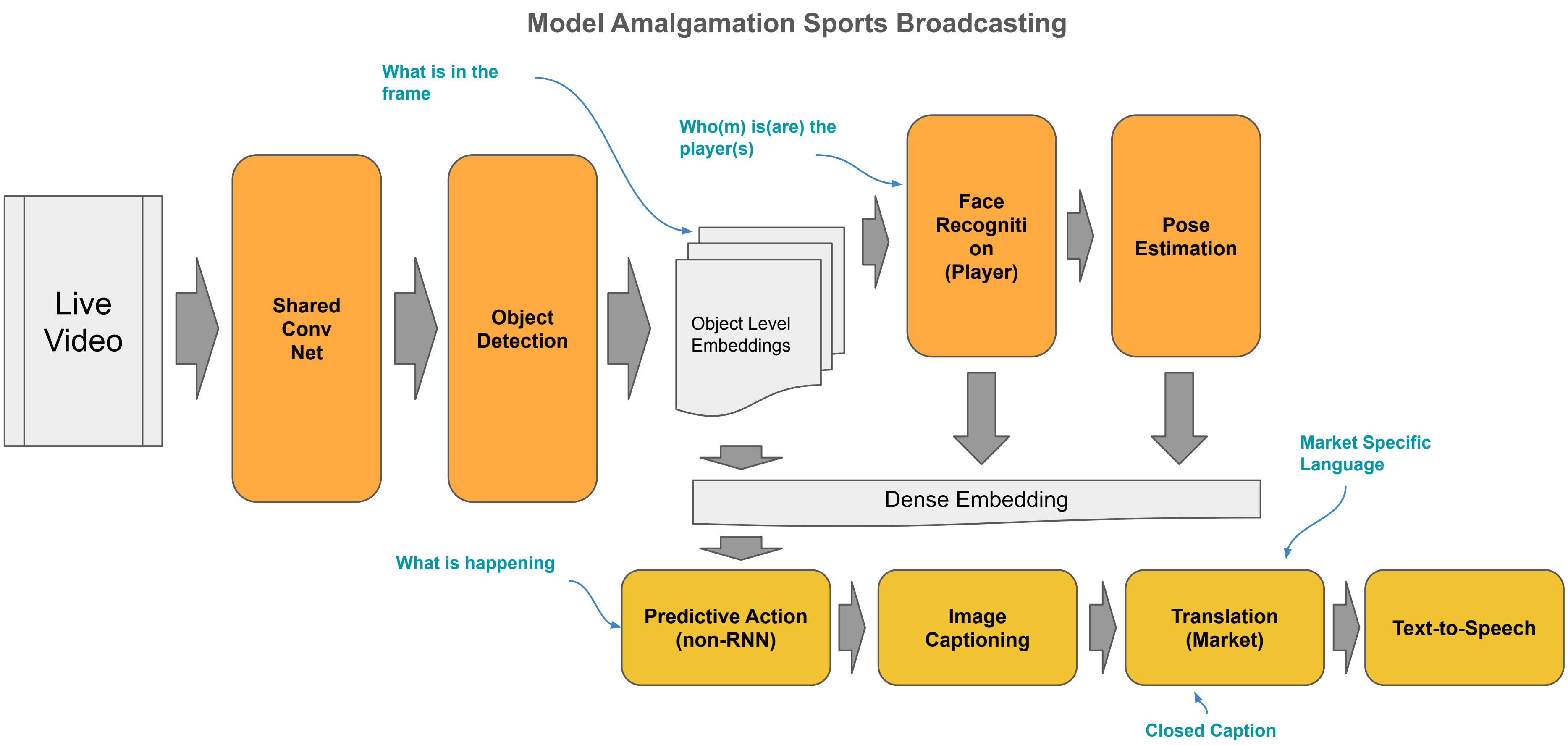


Framing a Business Problem into an e2e Pipeline

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

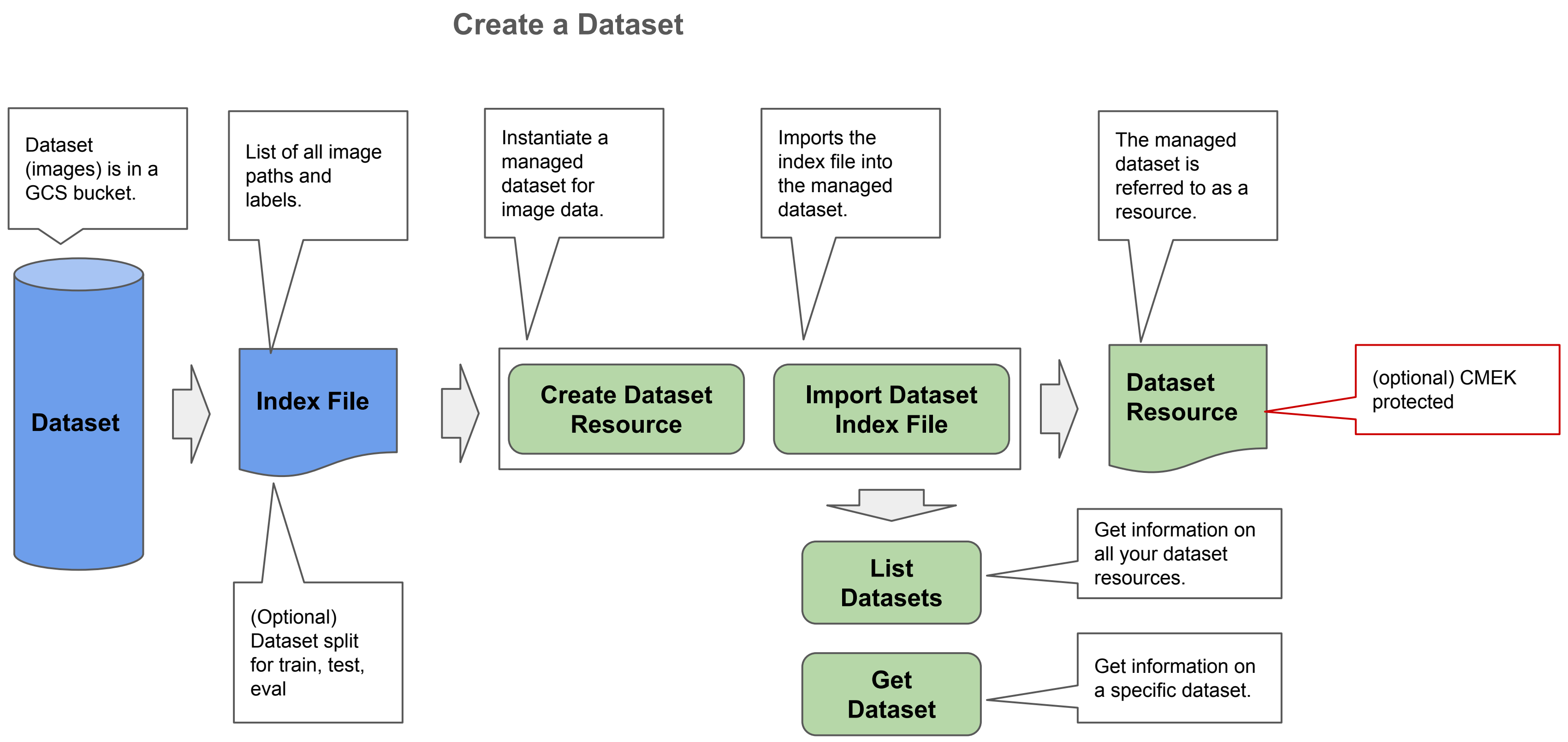
Reducing Costs

- 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.

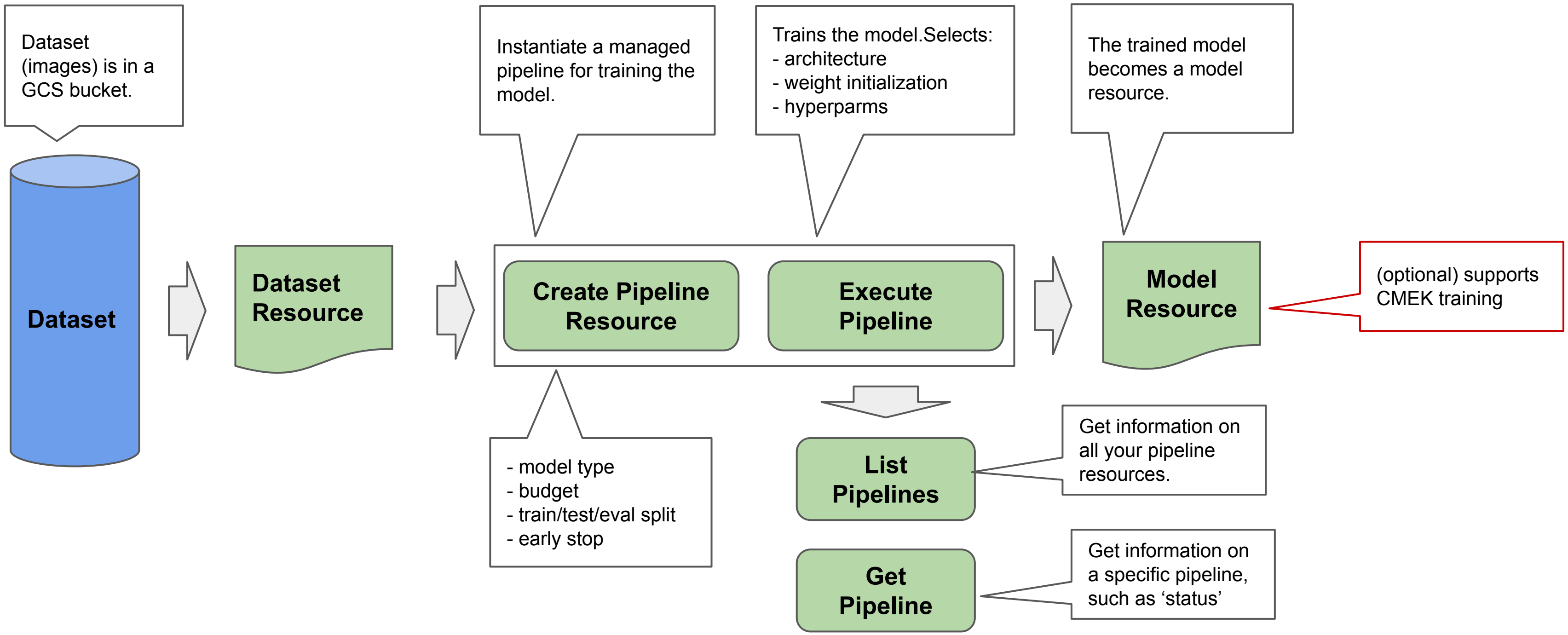
Step 2:
- Import the data.

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

Train a Model



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.

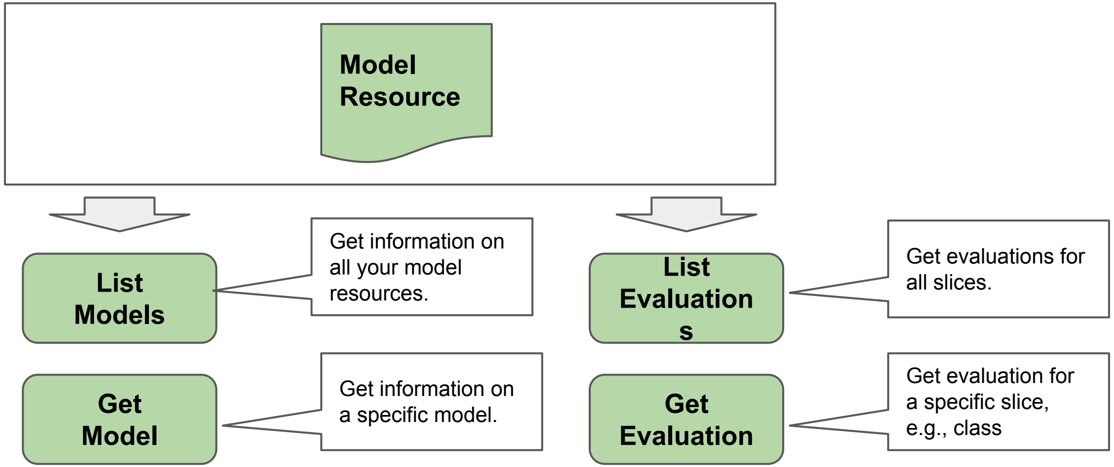
Step 2: return the status

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)
```

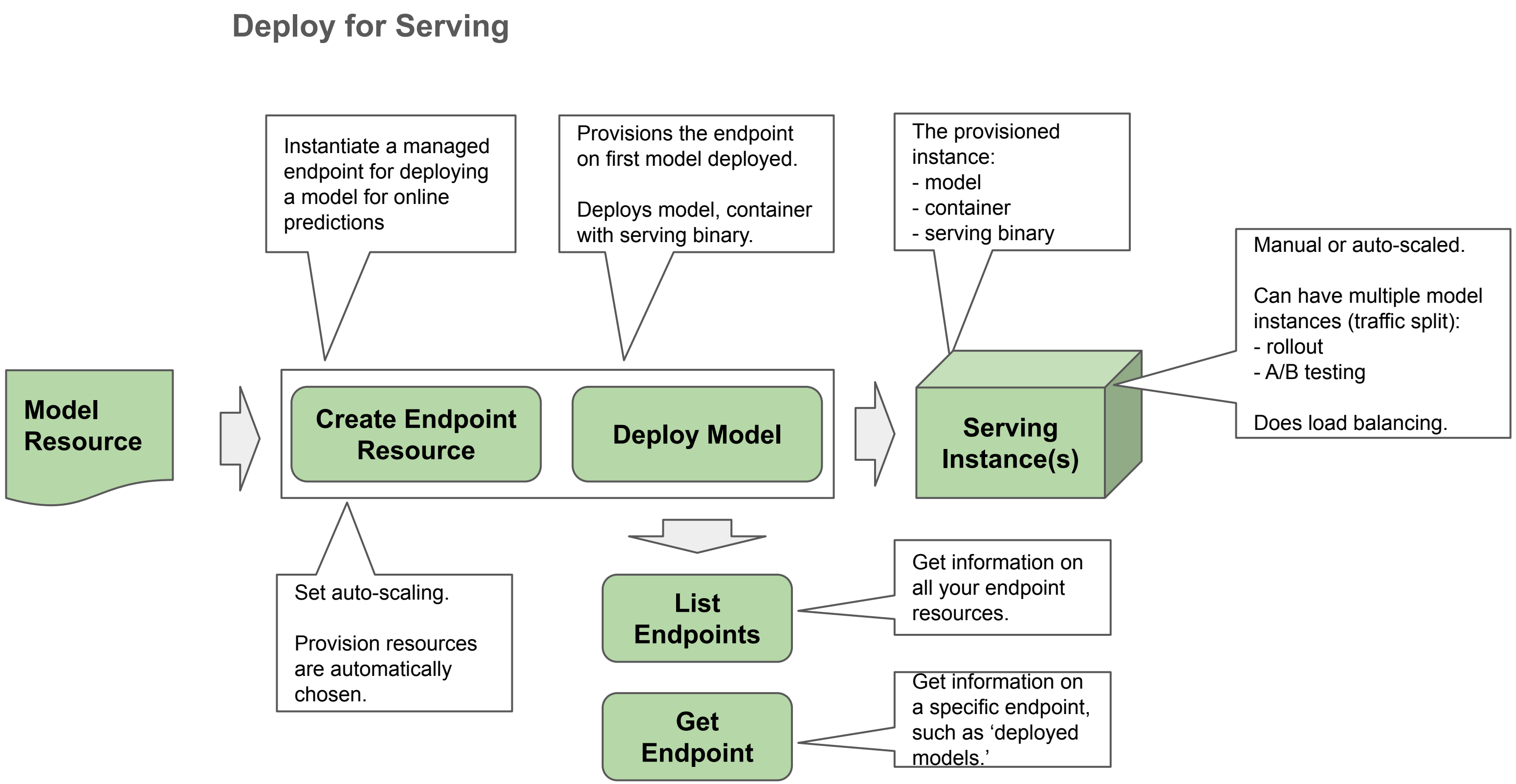
List Evaluations

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.

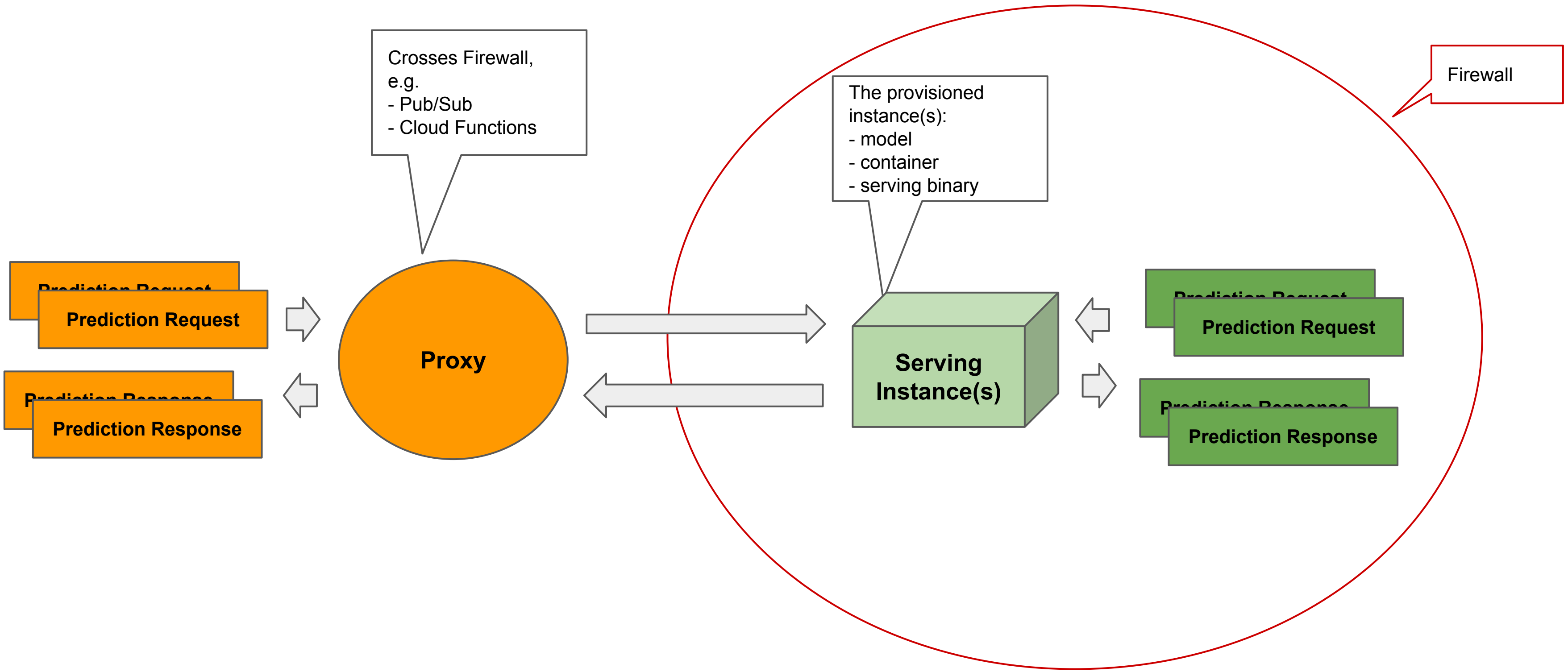
Step 2:
- Specify the traffic split
- Deploy the model

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

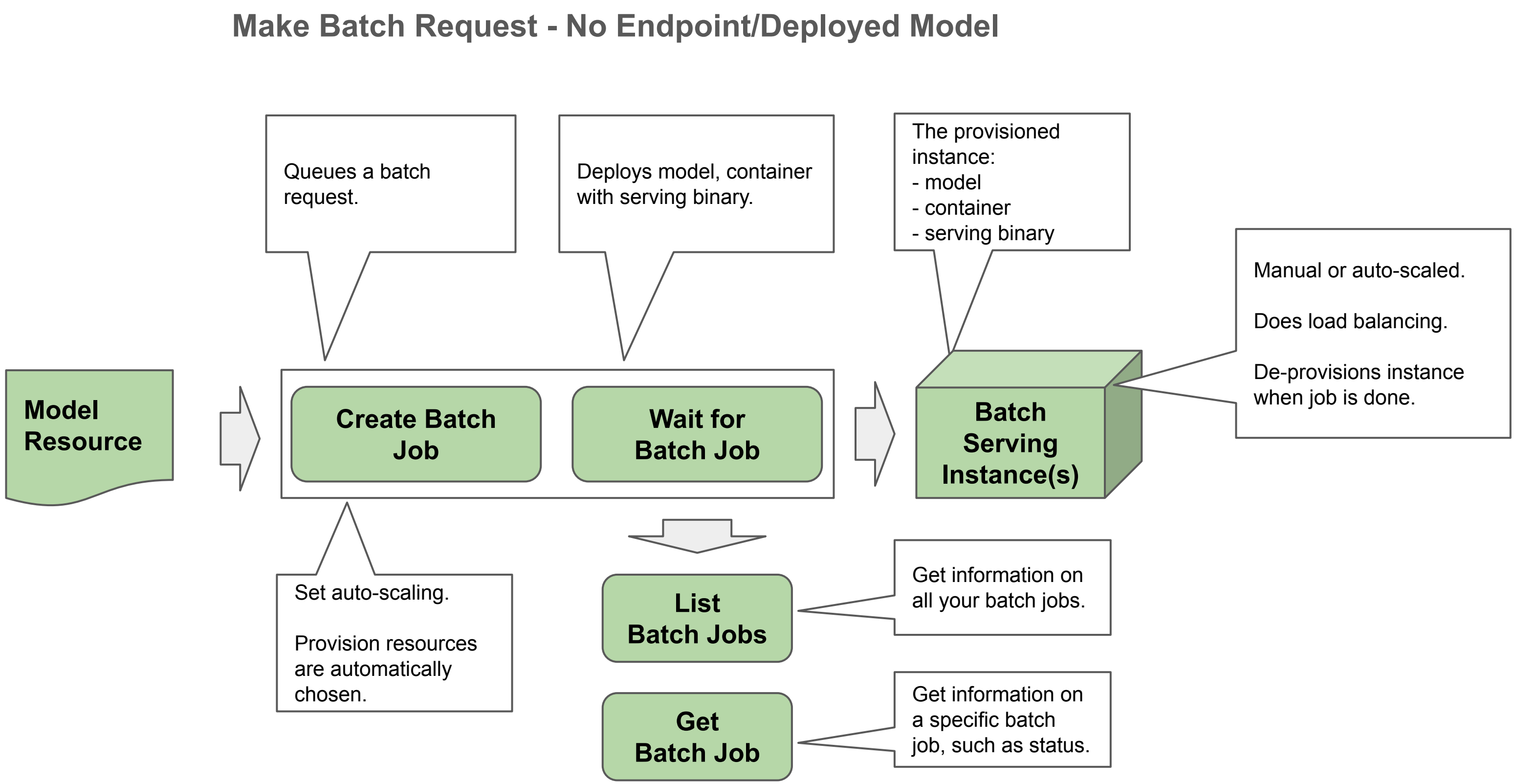
Step 1: Set paths to the images stored in GCS

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) - 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

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:

Same as image classification

{'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 IOD

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.

All fields except for image path are specific to segmentation

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

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.

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

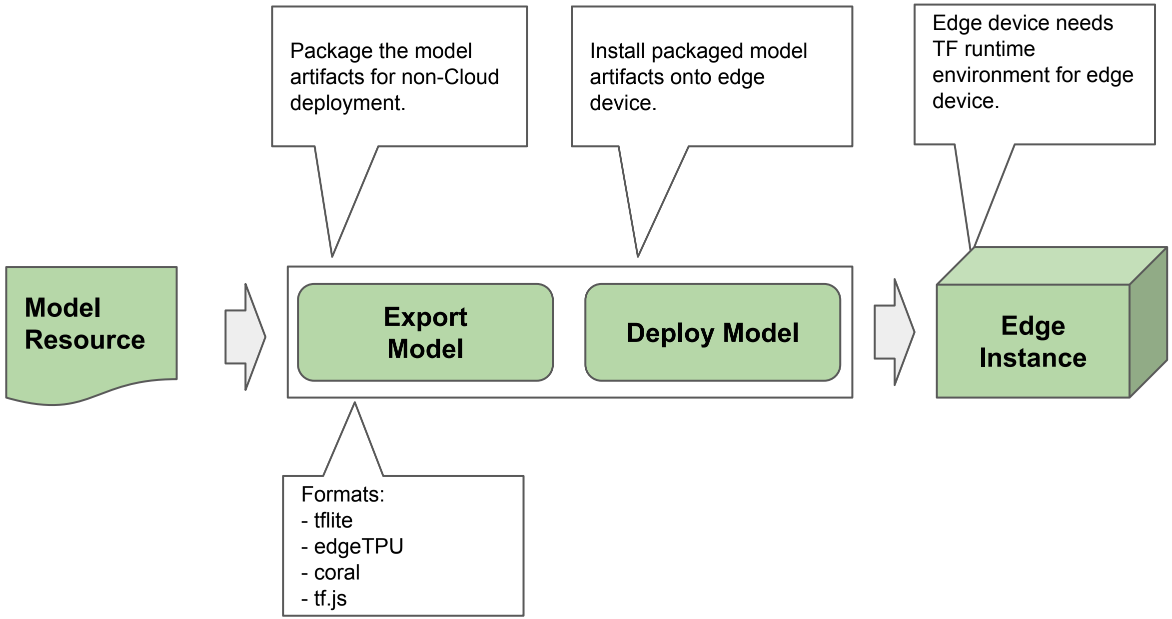


Image Model Exported to Edge - Training

```
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

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

```
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()
```

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)

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 }

Either text example, or GCS path to text file.

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`).

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

Cloud only model.

Specify the maximum sentiment.

```
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)
```

Text Sentiment Analysis - Prediction

Same as text classification

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

The sentiment rating

Text Sentiment Analysis - 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

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

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.

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

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.