

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

Semantic Segmentation



GRASS, CAT,
TREE, SKY

No objects, just pixels

**Classification
+ Localization**



CAT

Single Object

**Object
Detection**



DOG, DOG, CAT

Multiple Object

**Instance
Segmentation**

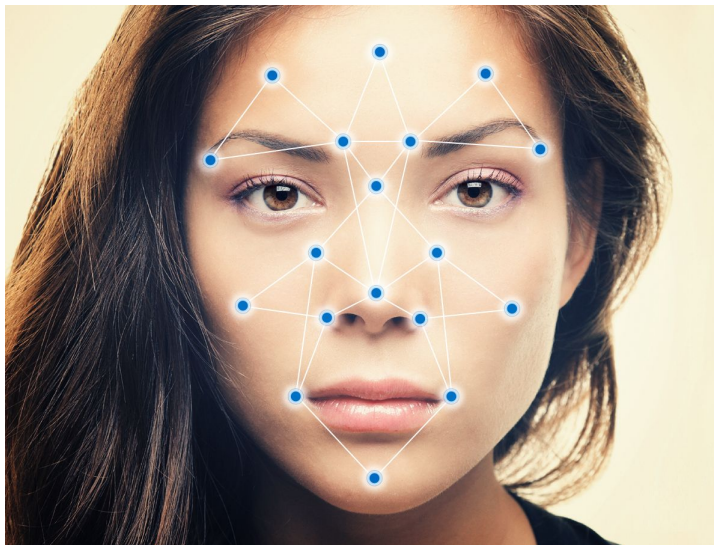


DOG, DOG, CAT

This image is CC0 public domain

Machine Learning Overview

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



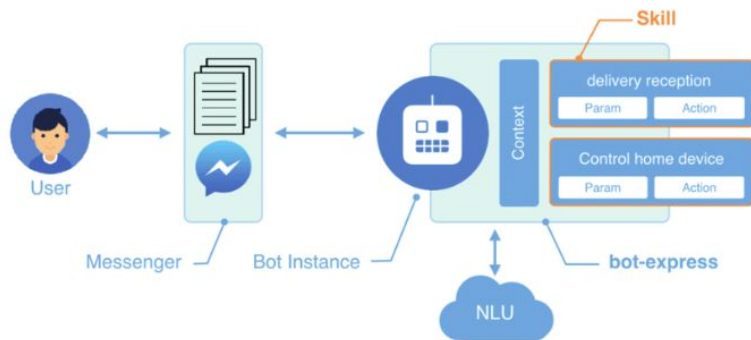
Machine Learning Overview

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

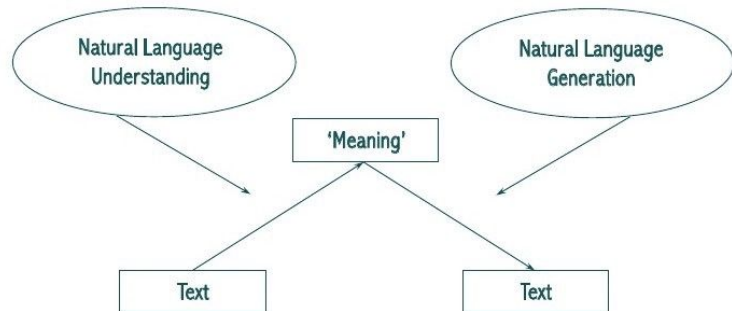


Machine Learning Overview

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

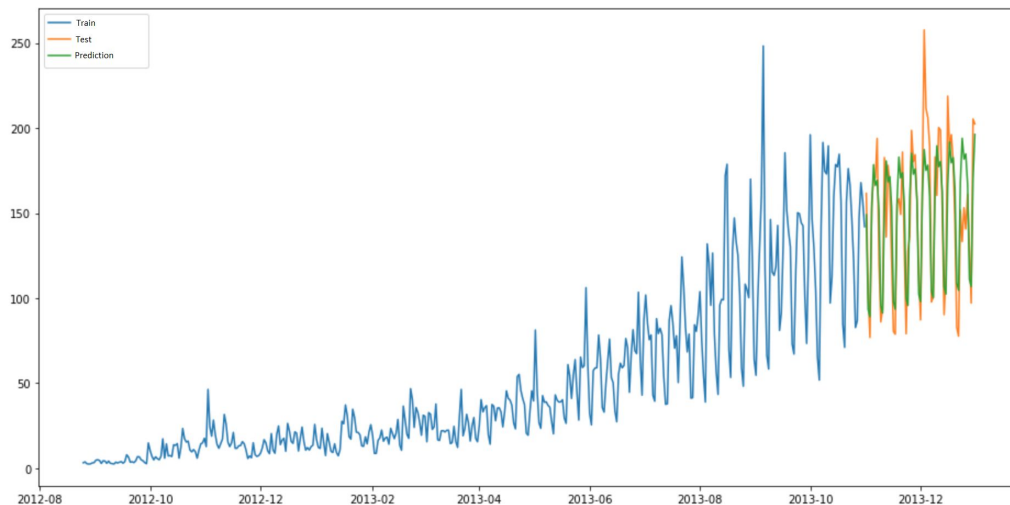
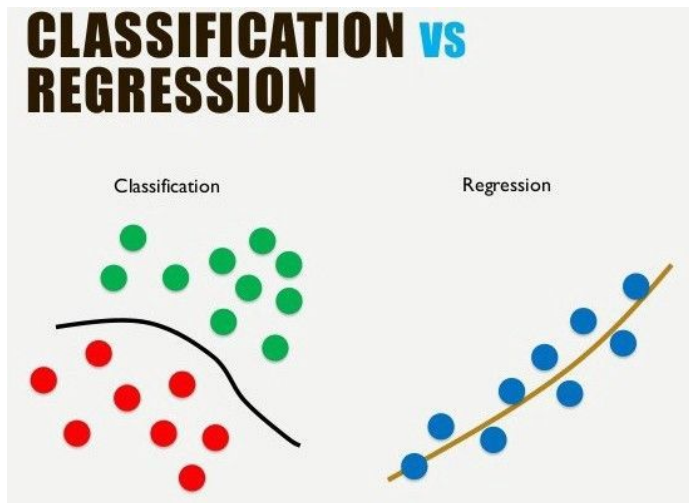


NLP = NLU + NLG



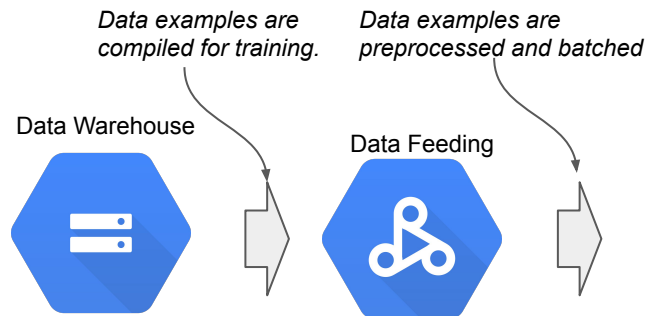
Machine Learning Overview

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



MLOps Overview

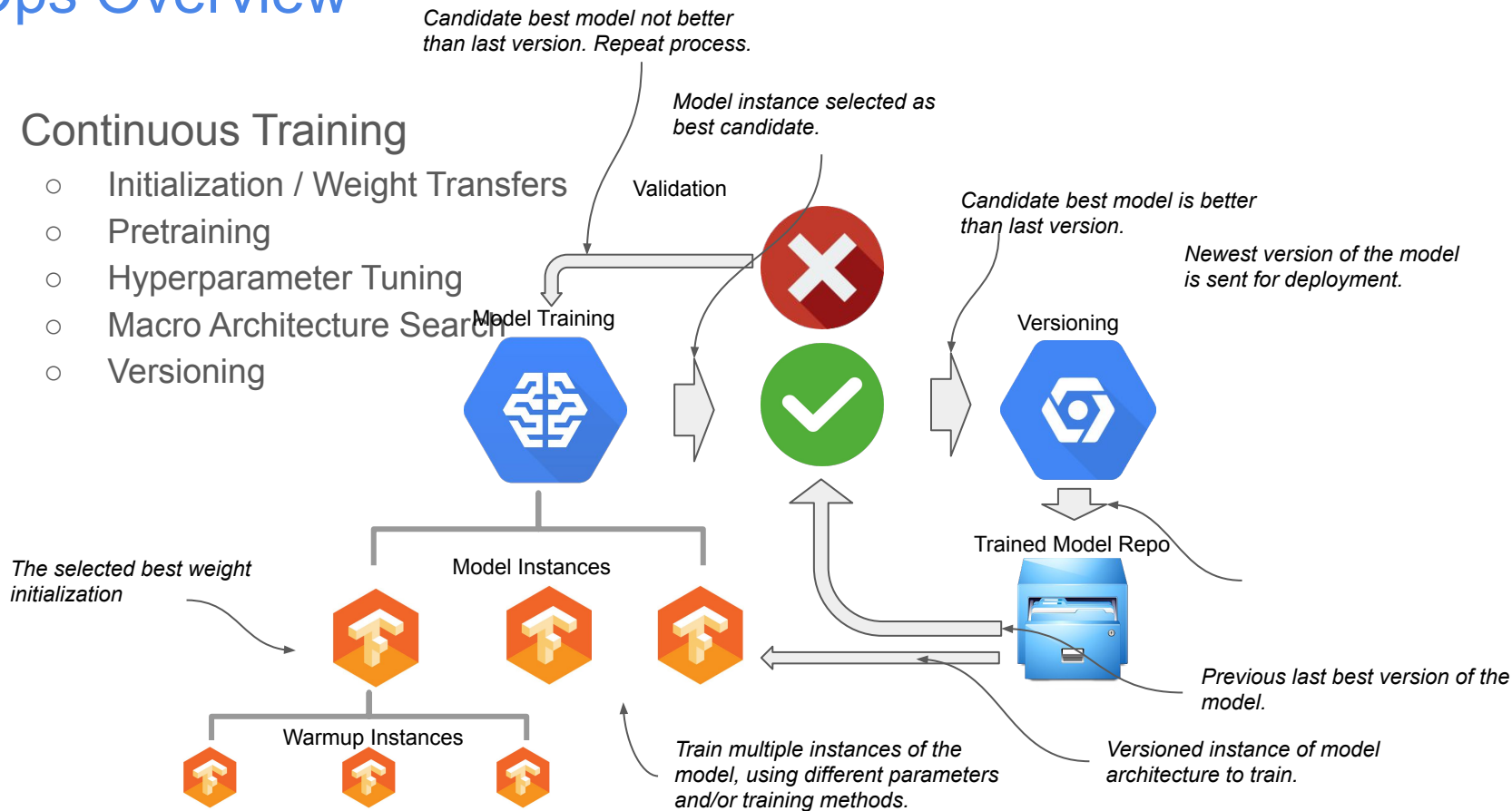
- Data Warehousing
 - Storage
 - Retrieval (I/O)
 - Feeding
 - Search / Query



MLOps Overview

- Continuous Training

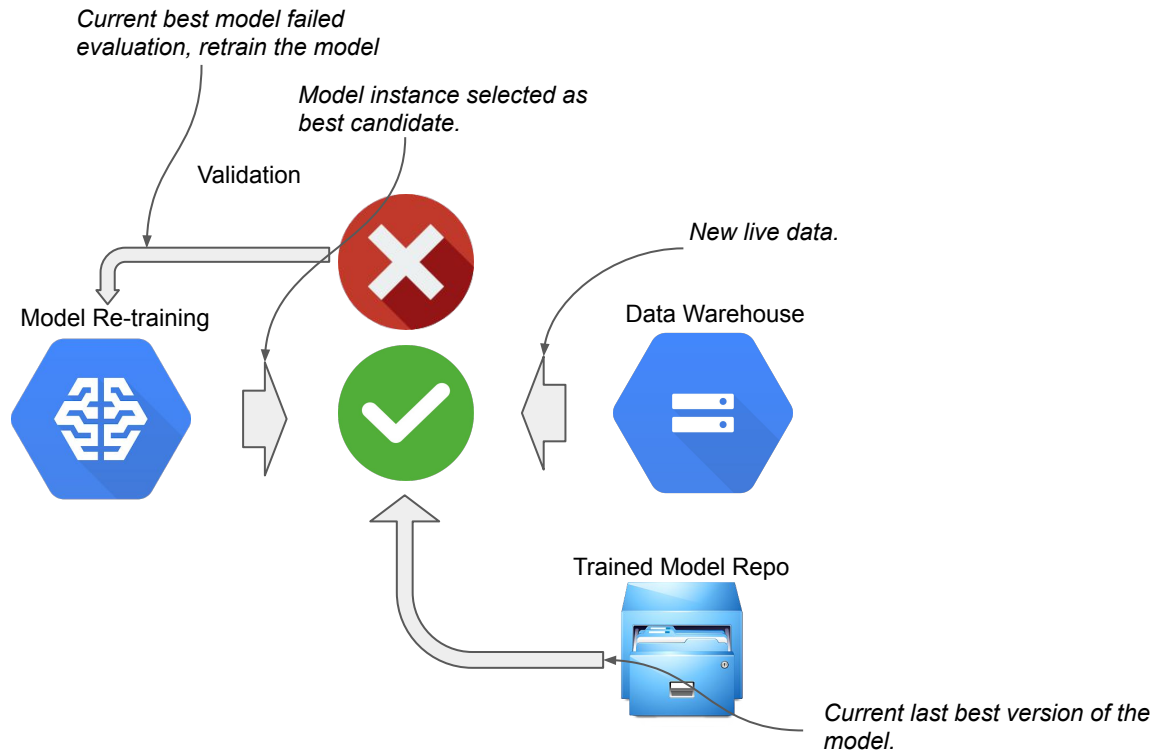
- Initialization / Weight Transfers
- Pretraining
- Hyperparameter Tuning
- Macro Architecture Search
- Versioning



MLOps Overview

- Continuous Evaluation

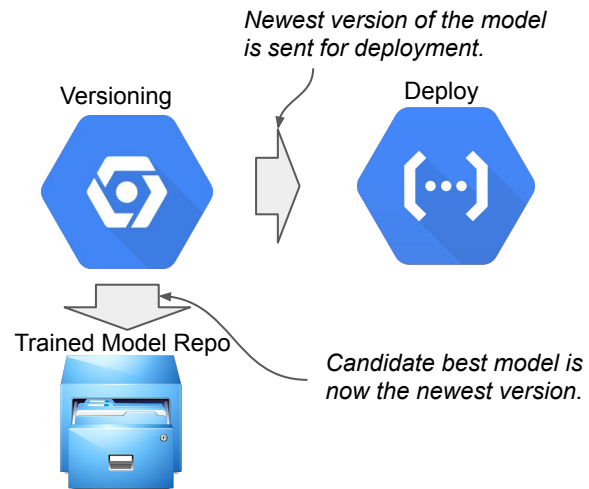
- Training Distribution
- Serving Skew
- Data Drift
- A/B Testing



MLOps Overview

- Deployment

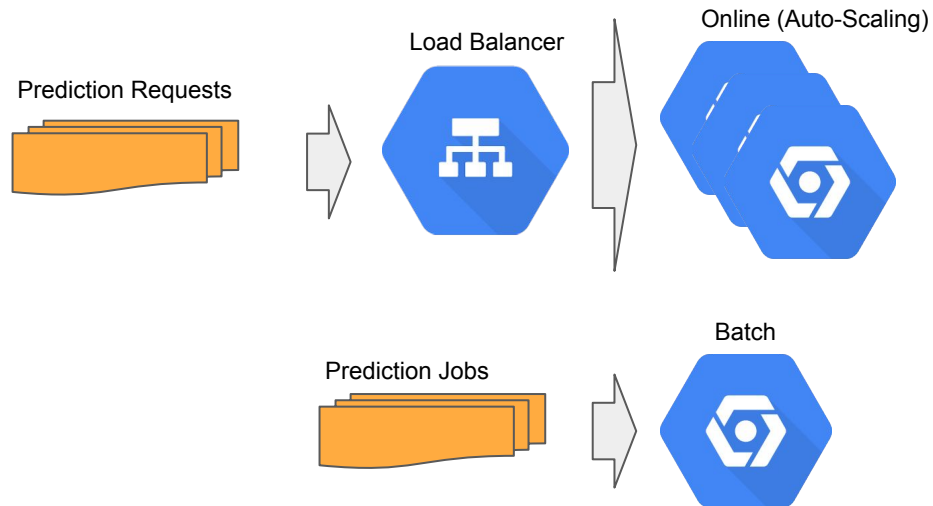
- Scaling
- Load Balancing
- Latency
- Edge



MLOps Overview

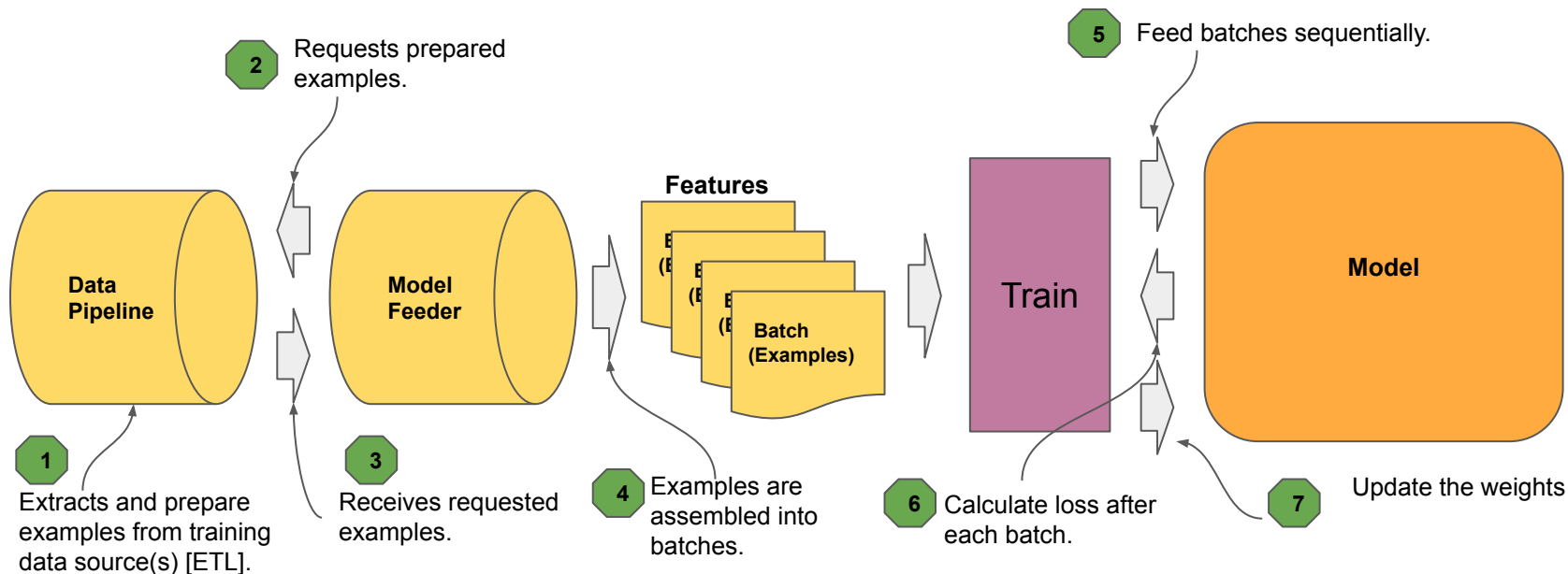
- Serving

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



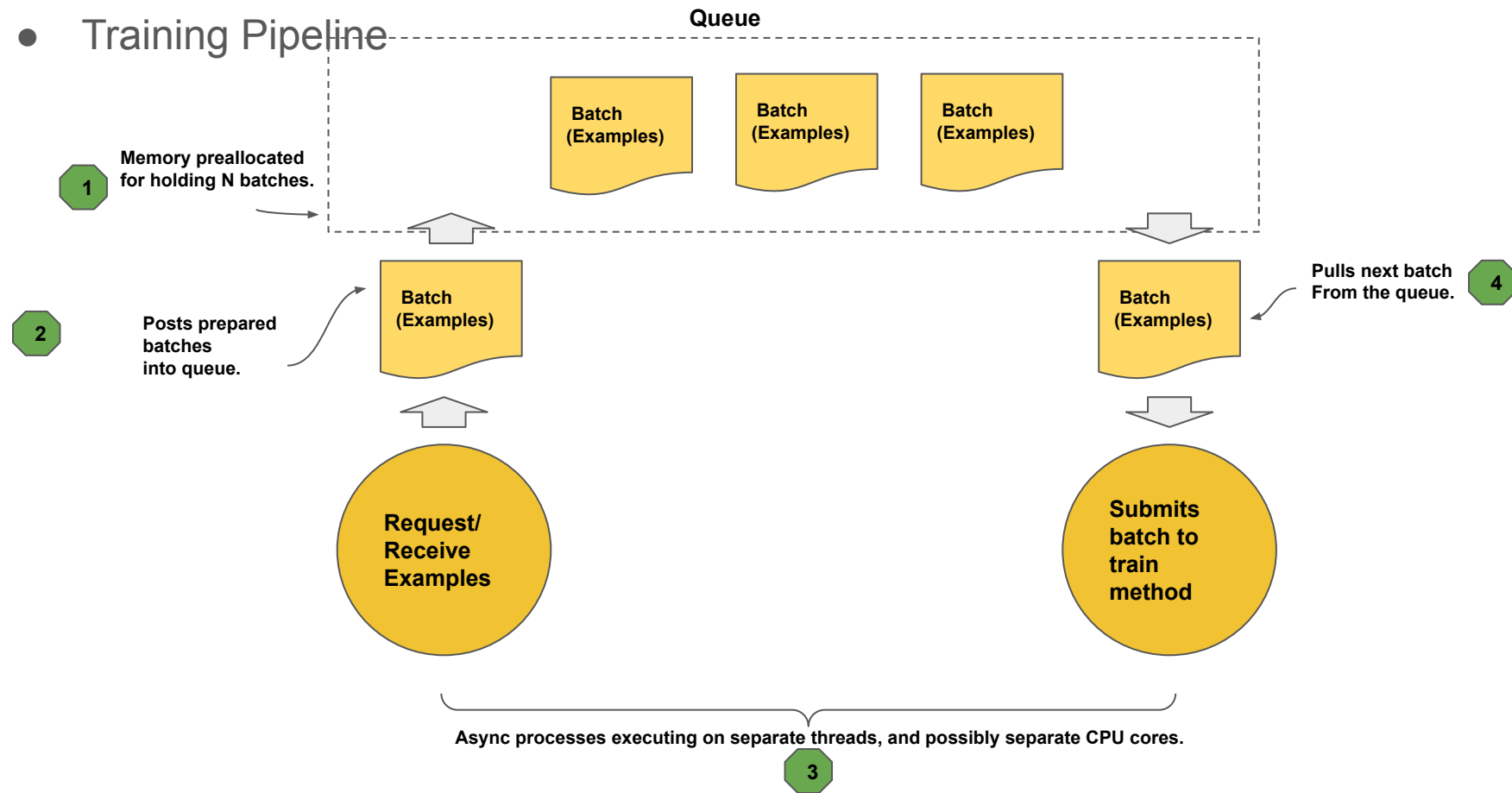
ML end-2-end production pipeline

- Data Pipeline



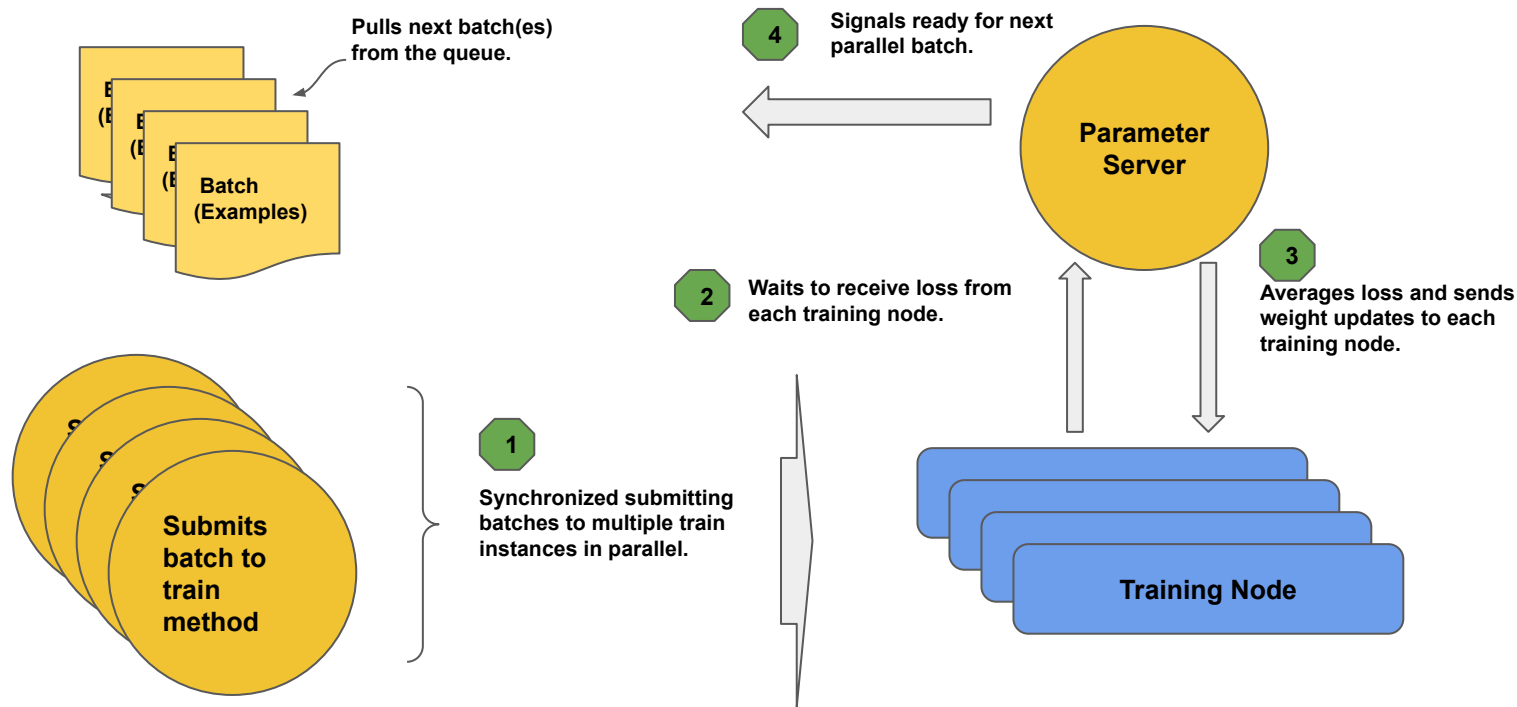
ML end-2-end production pipeline

- Training Pipeline



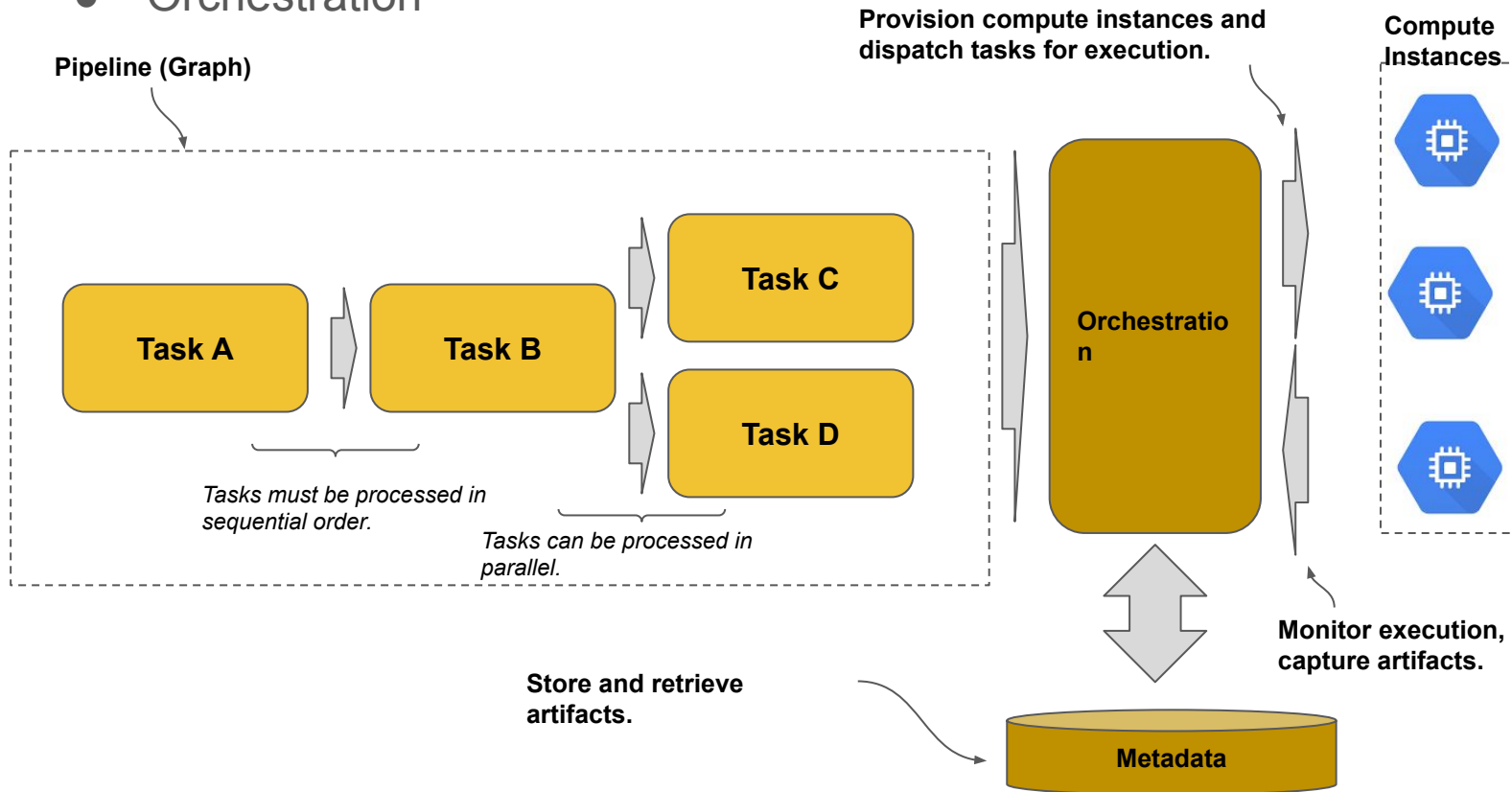
ML end-2-end production pipeline

- Training Pipeline



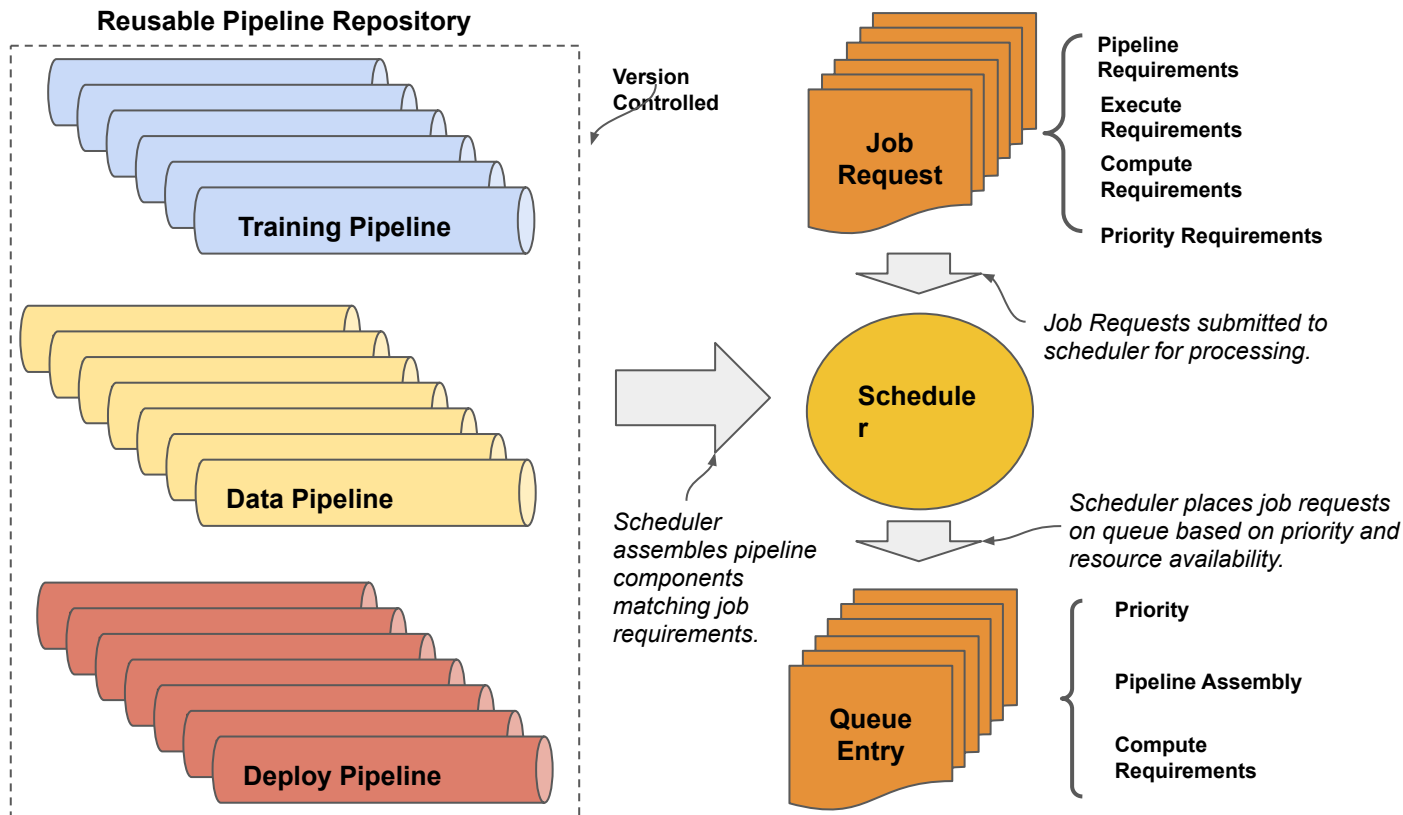
ML end-2-end production pipeline

- Orchestration



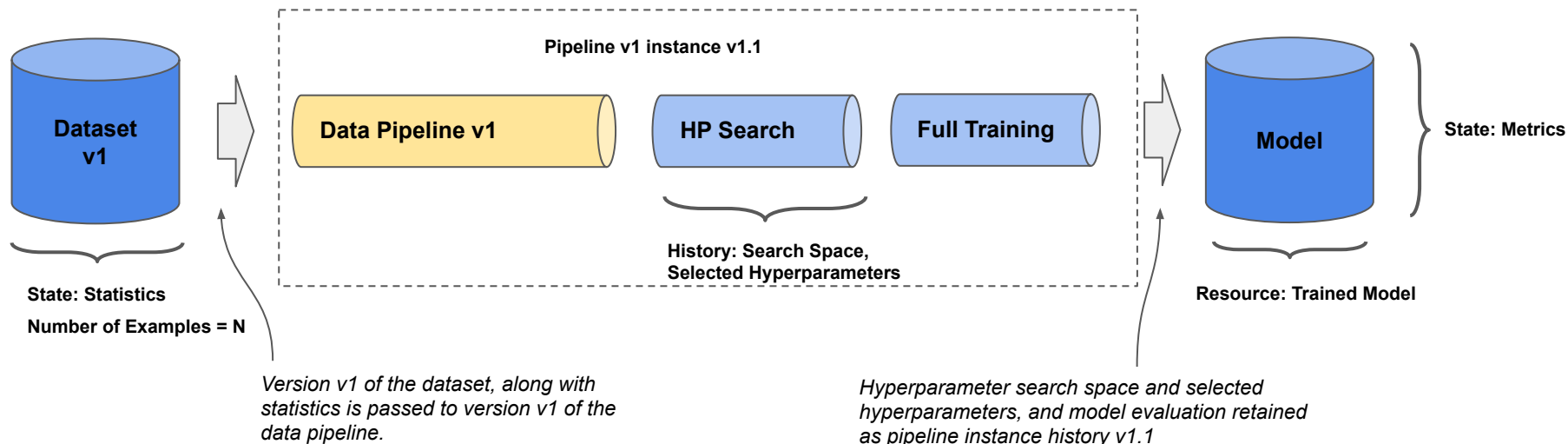
ML end-2-end production pipeline

- Pipeline Components



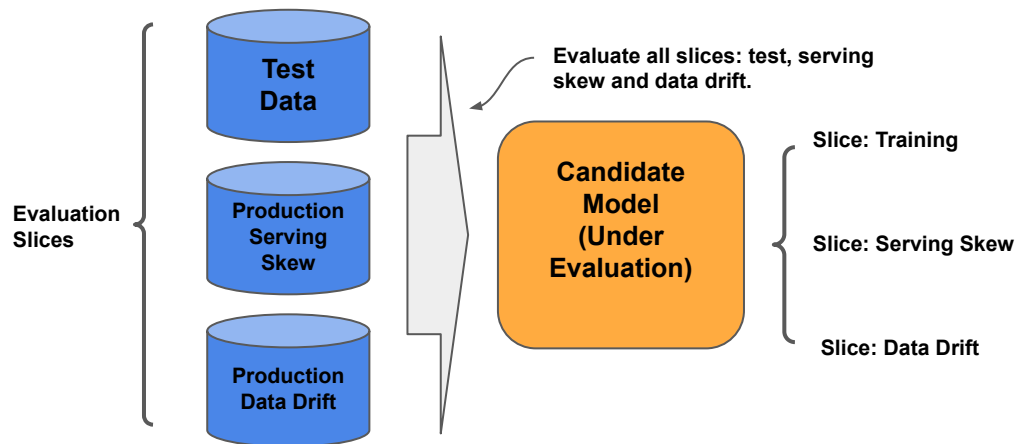
ML end-2-end production pipeline

- Heuristics



ML end-2-end production pipeline

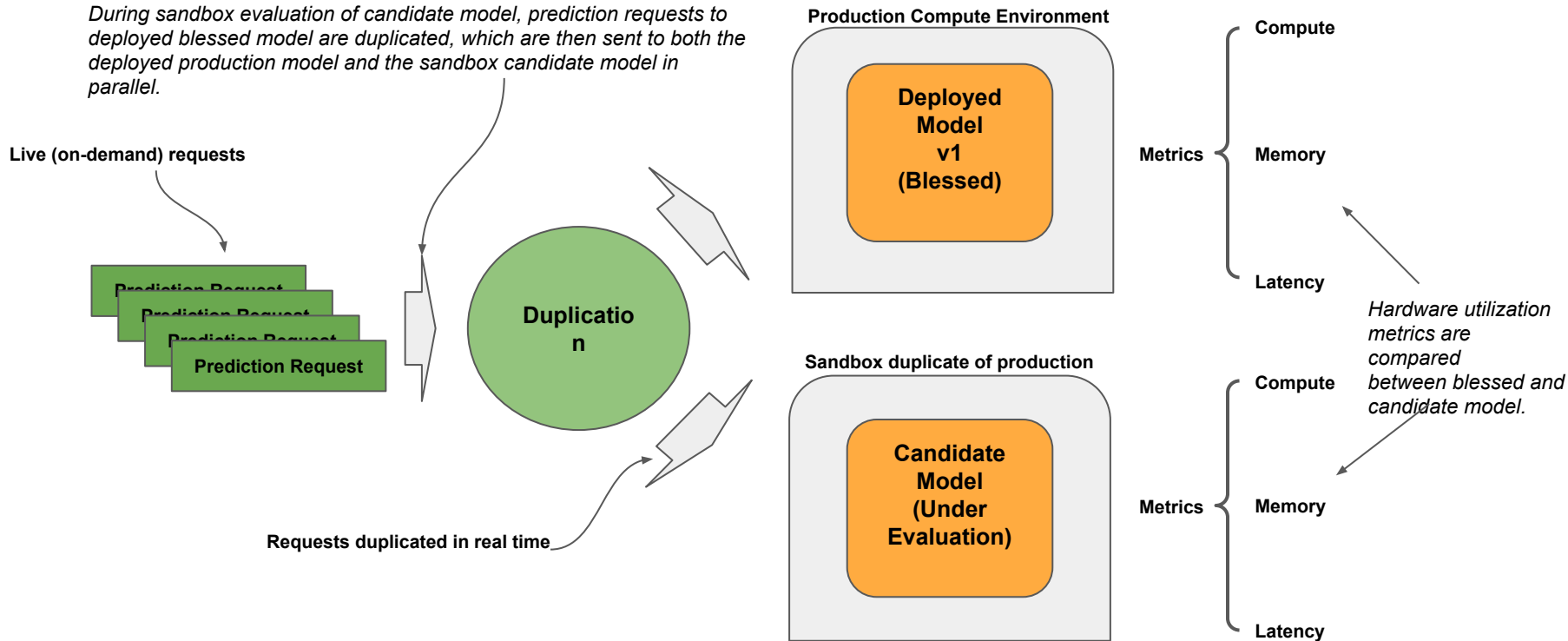
- Evaluation Slicing



ML end-2-end production pipeline

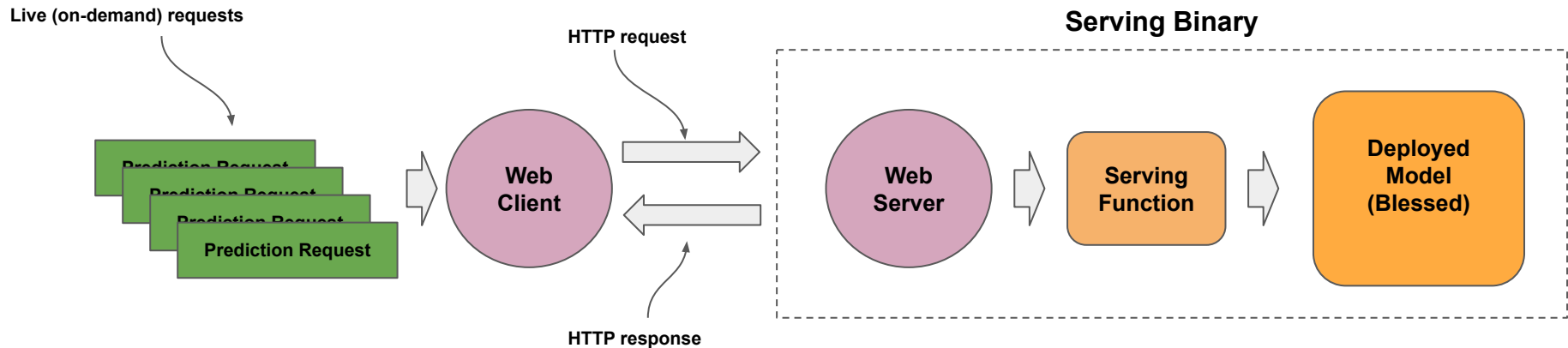
- Sandboxing

During sandbox evaluation of candidate model, prediction requests to deployed blessed model are duplicated, which are then sent to both the deployed production model and the sandbox candidate model in parallel.



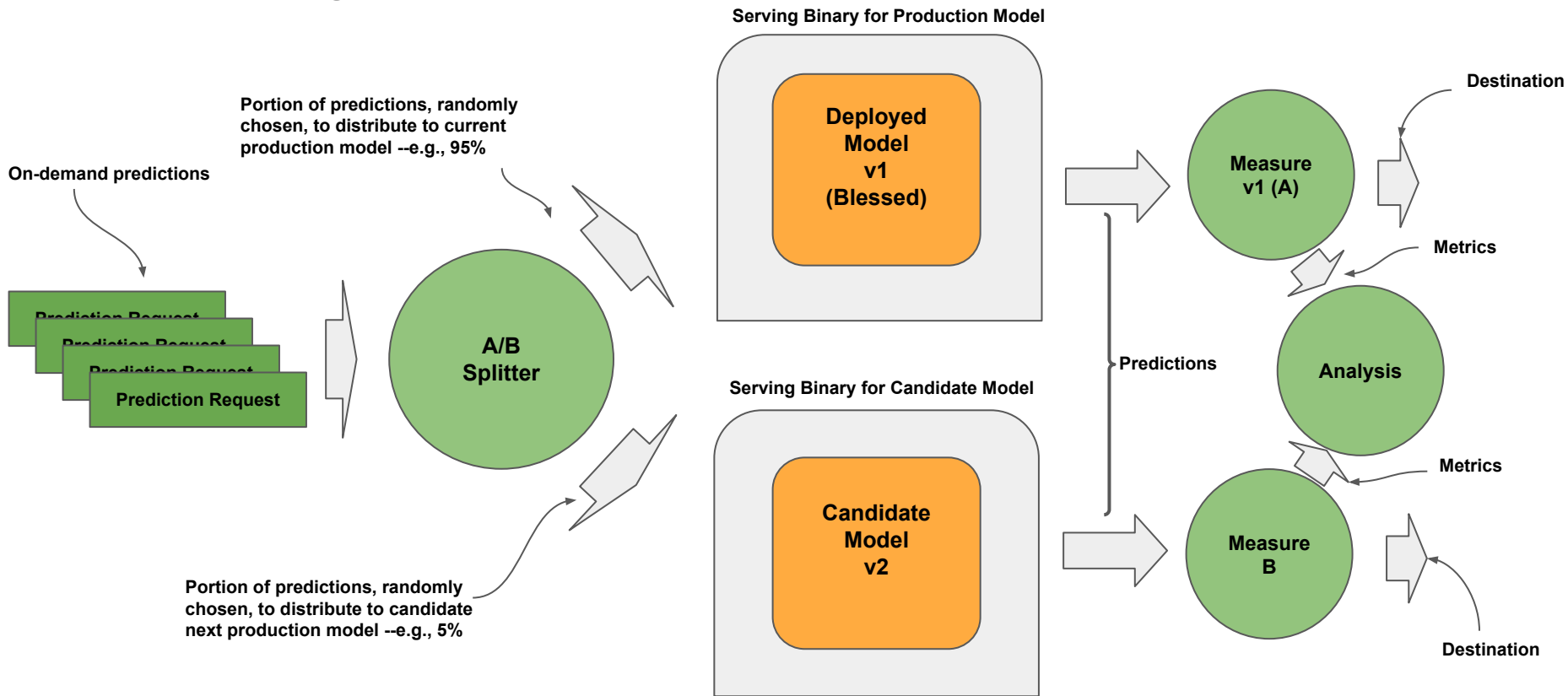
ML end-2-end production pipeline

- Serving Containers



ML end-2-end production pipeline

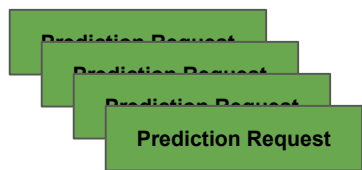
- A/B Testing



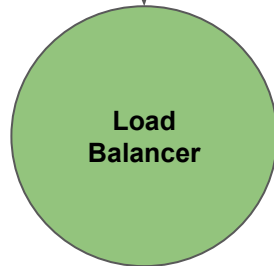
ML end-2-end production pipeline

- Load Balancing

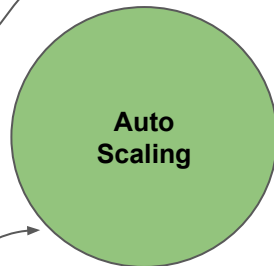
On-demand predictions



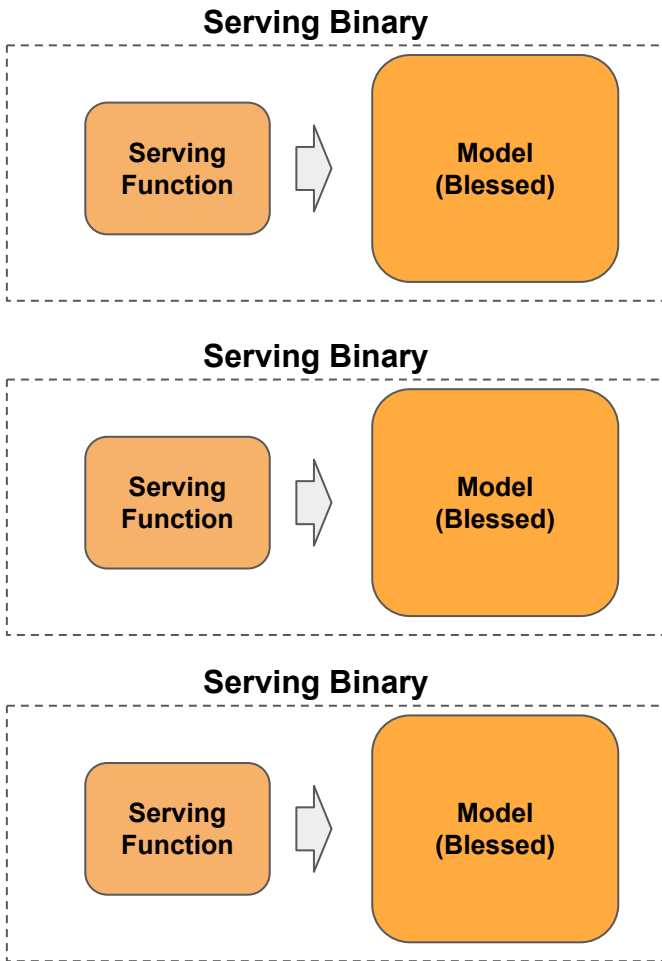
Distributes requests across serving binaries.



Request Frequency,
Response Latency

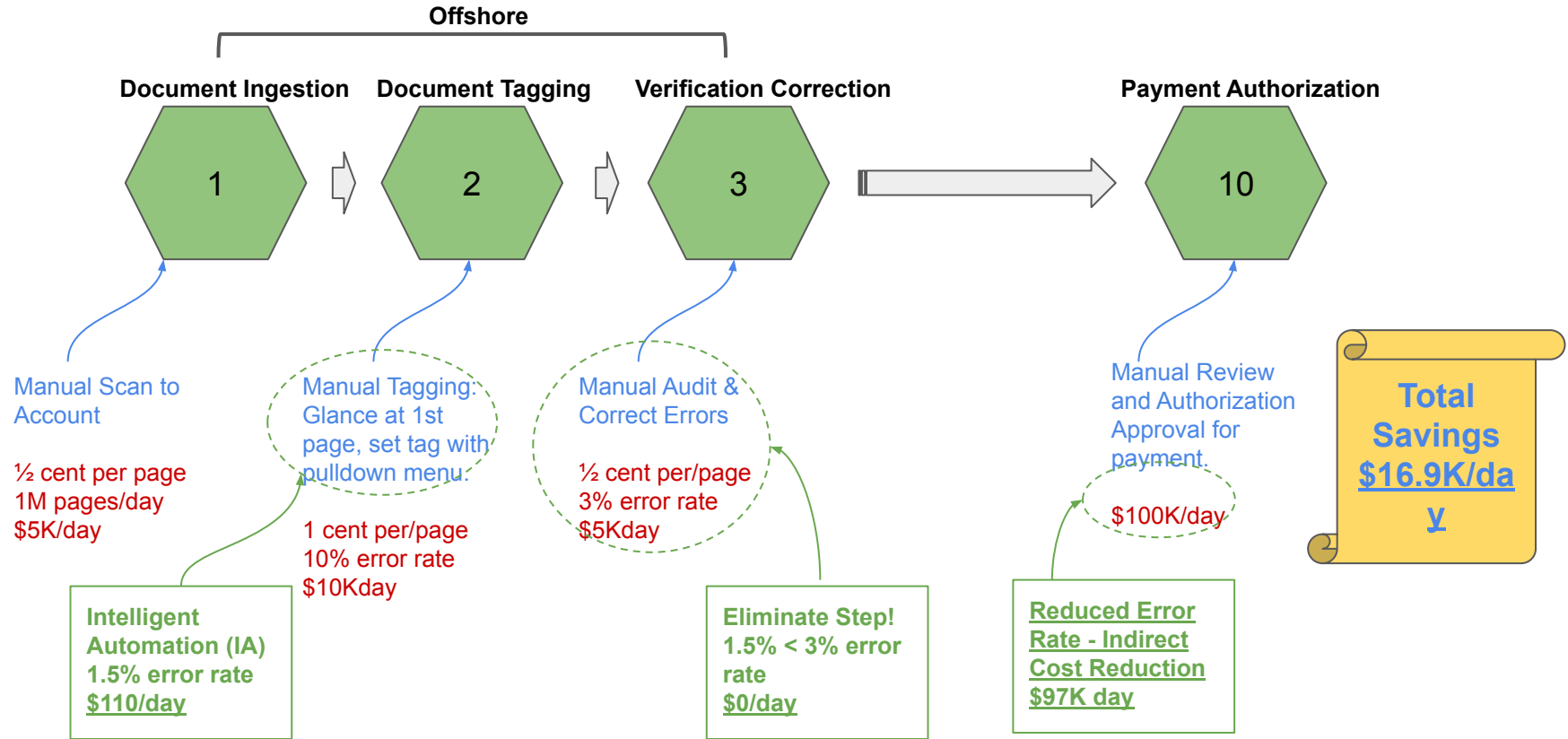


Auto provision and deprovision
(scaling) serving binary
instances.



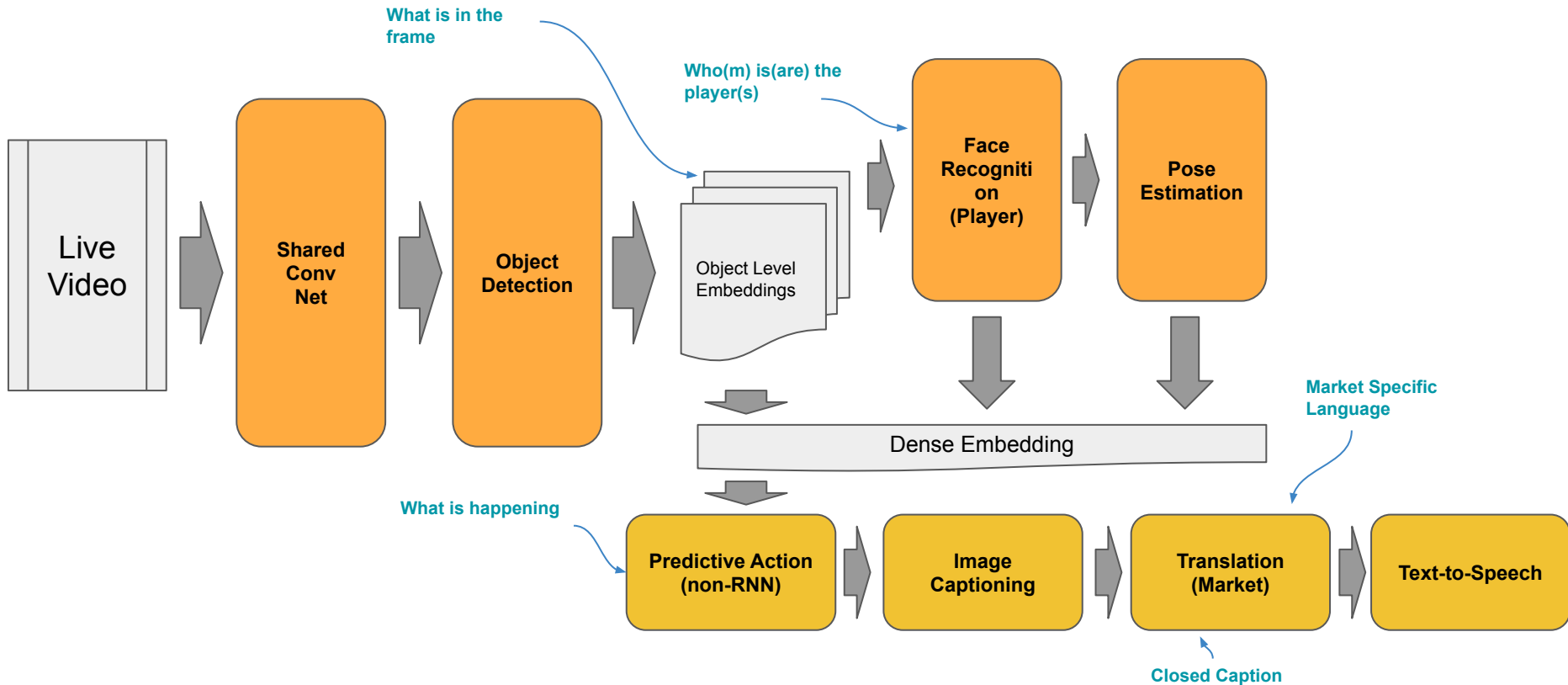
Framing a Business Problem into an e2e Pipeline

Intelligent Automation (IA) Applied to Claim Processing



Framing a Business Problem into an e2e Pipeline

Model Amalgamation Sports Broadcasting



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

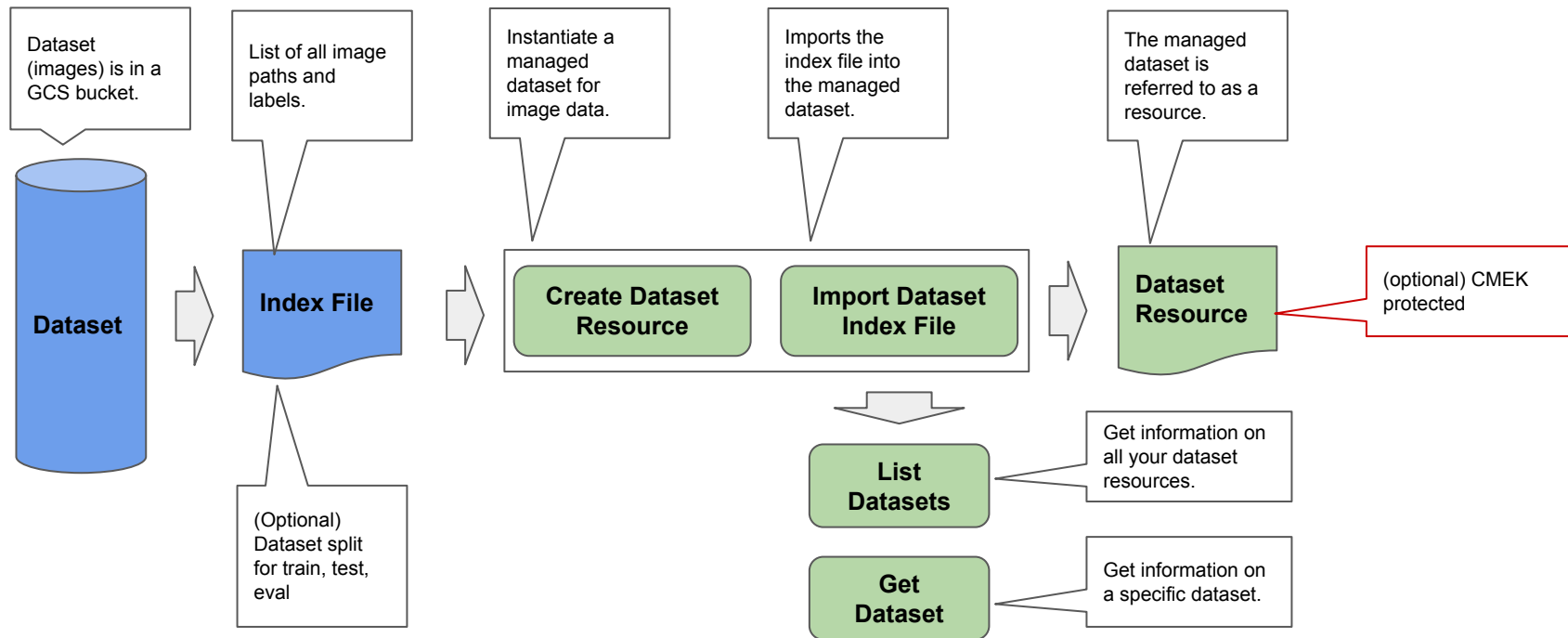
- Strategy for workshop notebooks
 - AutoML
 - follow along (execute) upto training
 - From training on, read only
 - Custom Jobs
 - Execute entire notebook

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 a Dataset



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)

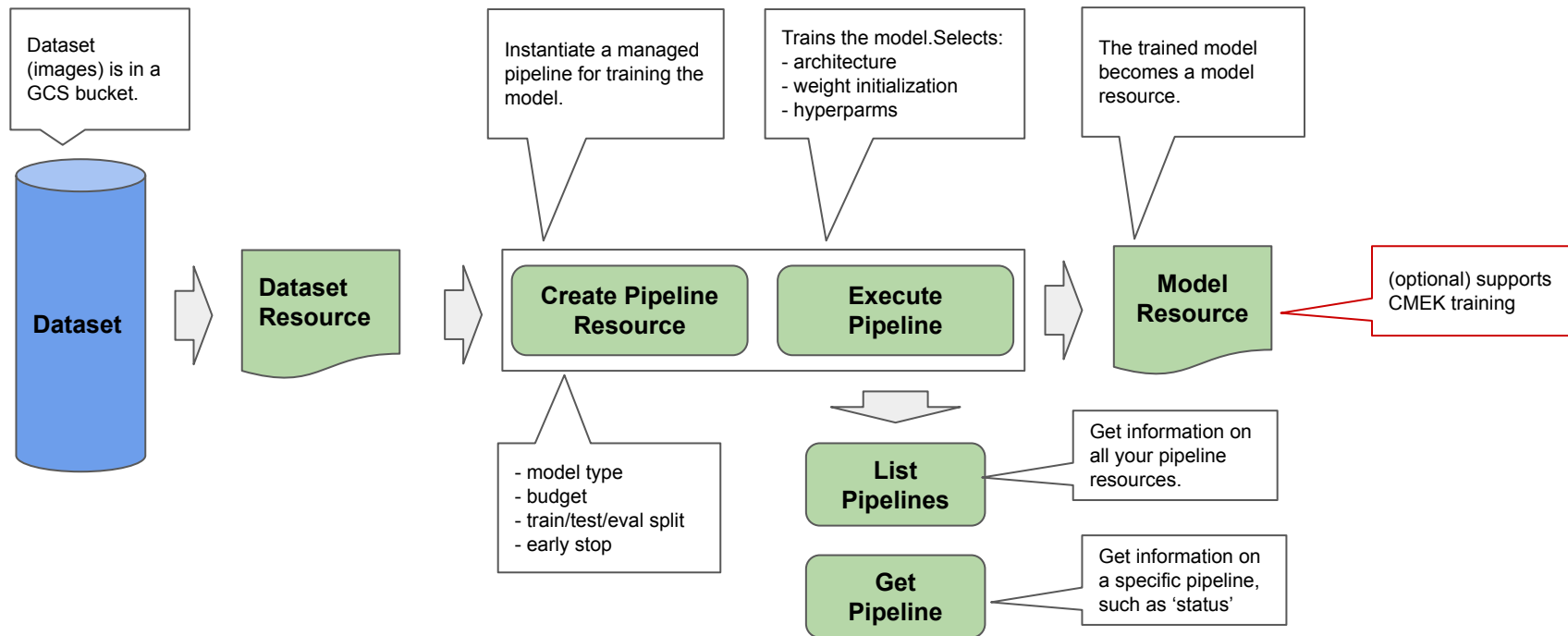
        result = operation.result()
        print("result:", result)
        print("time:", int(time.time() - start_time), "secs")
        print("error:", operation.exception())
        print("meta :", operation.metadata)
        print("after: running:", operation.running(),
            "done:", operation.done(),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

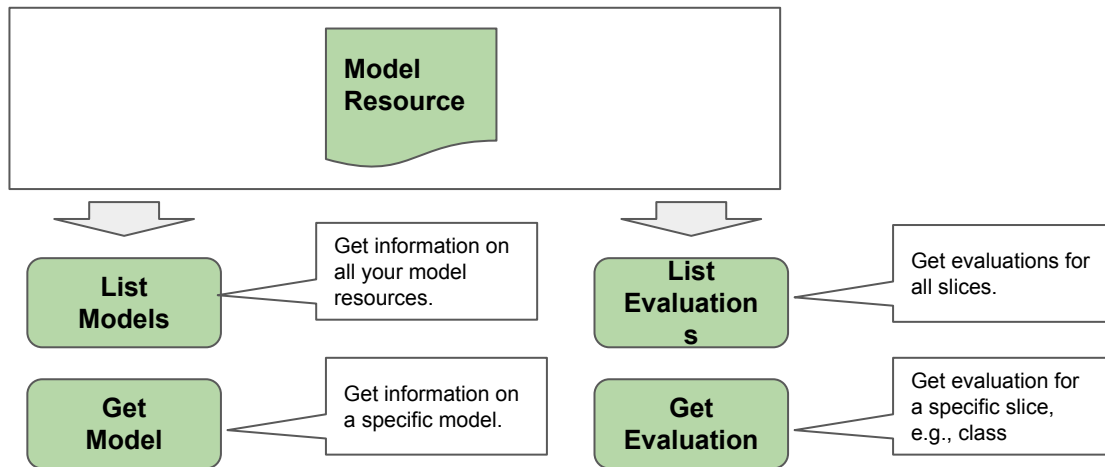
    print("pipeline")
    print(" name:", response.name)
    print(" display_name:", response.display_name)
    print(" state:", response.state)
    print(" training_task_definition:", response.training_task_definition)
    print(" training_task_inputs:", dict(response.training_task_inputs))
    print(" 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

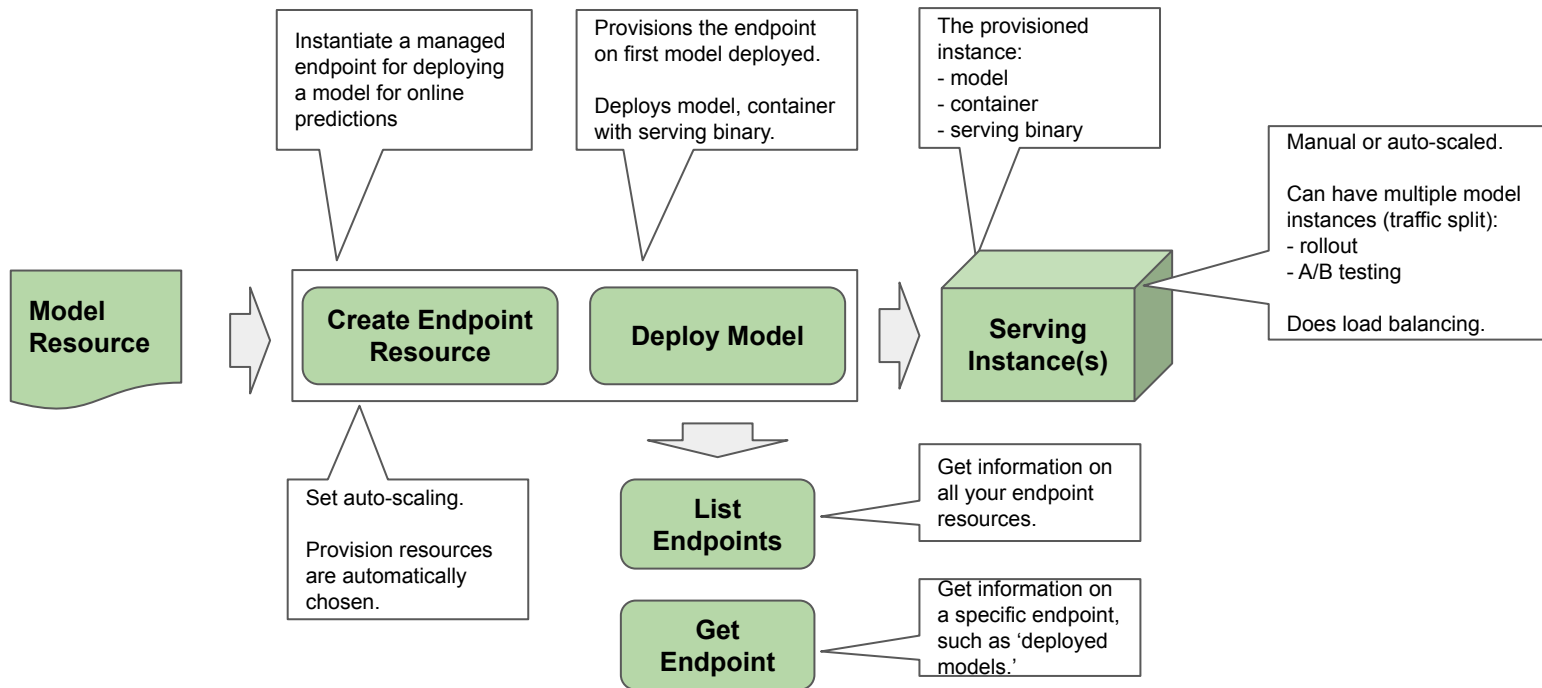
Step 1: Query for evaluations on all slices of the test/eval data (e.g., by class)

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

Deploy for Serving



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
        },
    },
}

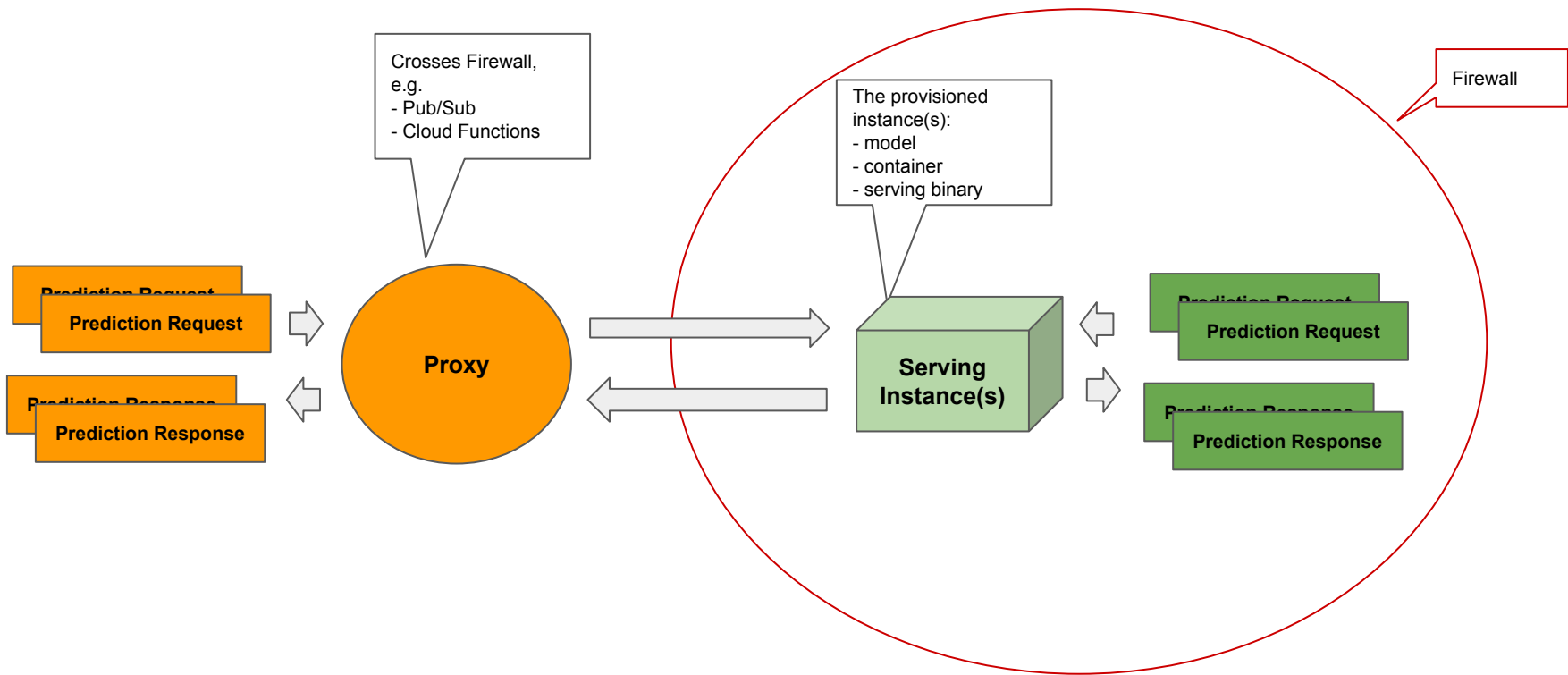
    response = clients['endpoint'].deploy_model(
        endpoint=endpoint, deployed_model=deployed_model, traffic_split=traffic_split)

    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.

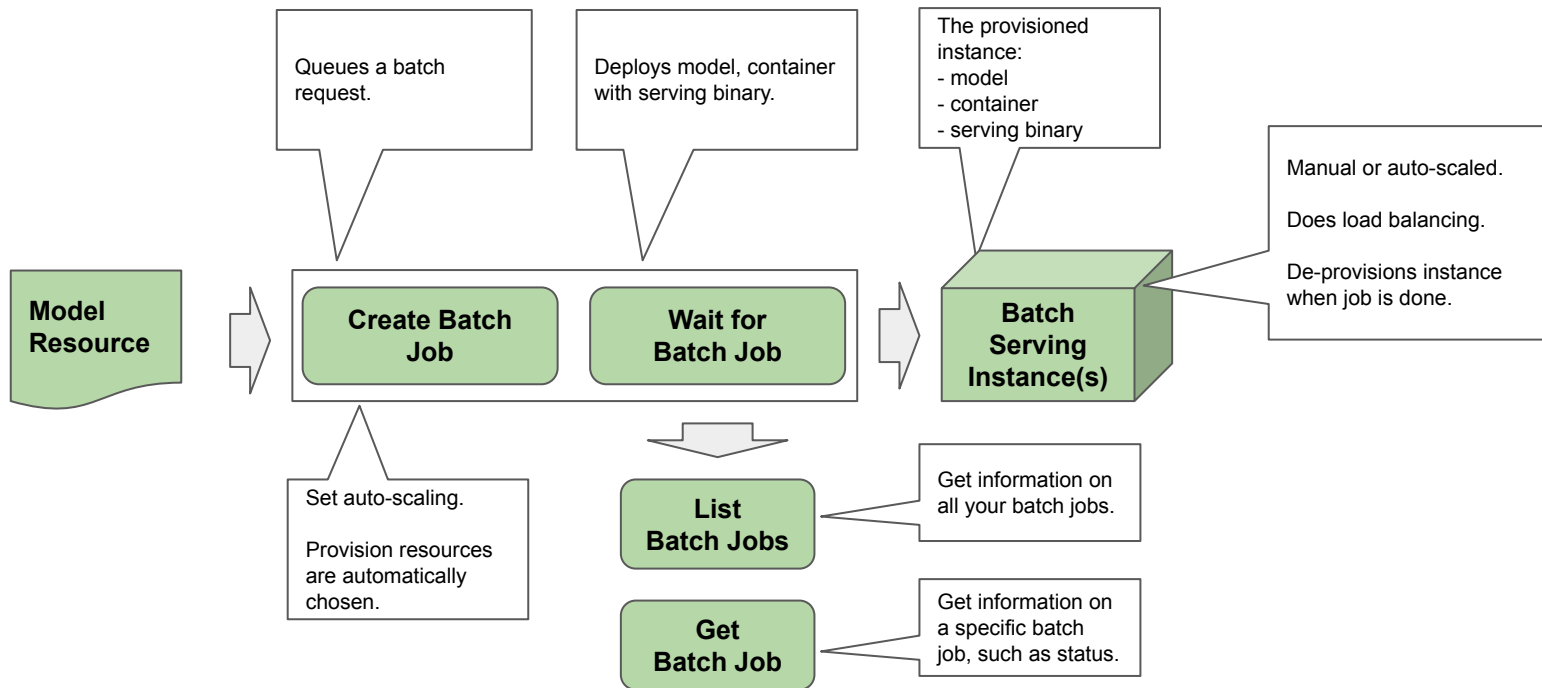
```
test_item_1 = BUCKET_NAME + "/" + file_1
test_item_2 = BUCKET_NAME + "/" + file_2

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

Make Batch Request - No Endpoint/Deployed Model



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 = 'json'
OUT_FORMAT = 'json' # [json]

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).

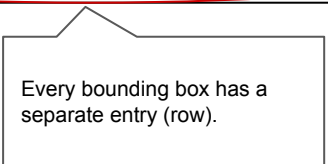
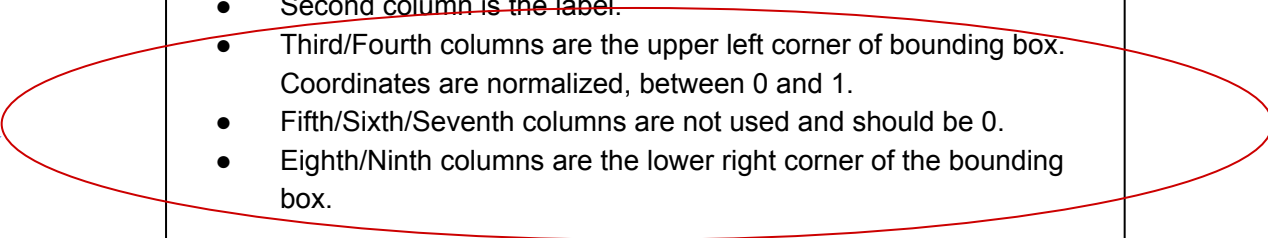


Image Object Detection (IOD) - Task Requirements

```
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 (bbboxes).

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




Image Object Detection (IOD) - Batch Prediction

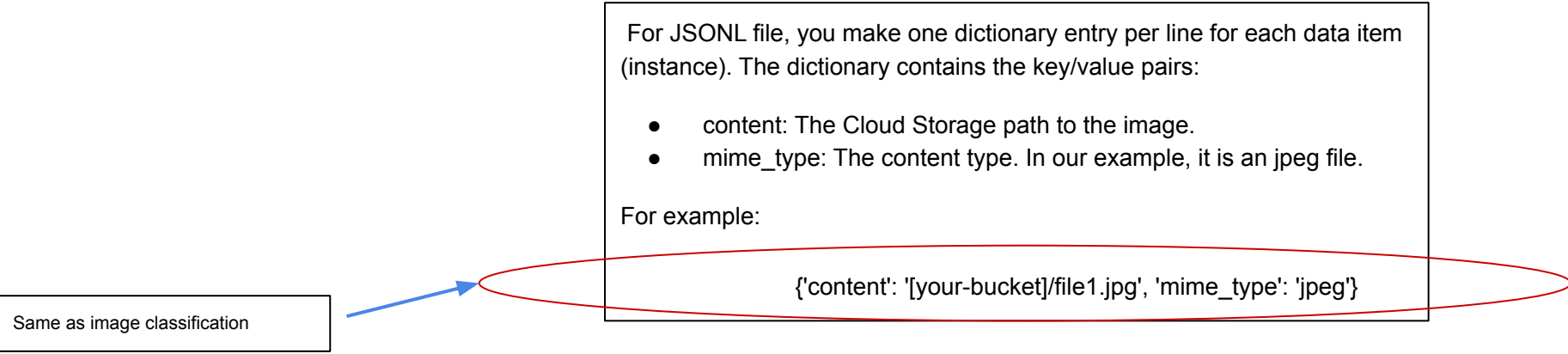
For JSONL file, you make one dictionary entry per line for each data item (instance). The dictionary contains the key/value pairs:

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

For example:

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

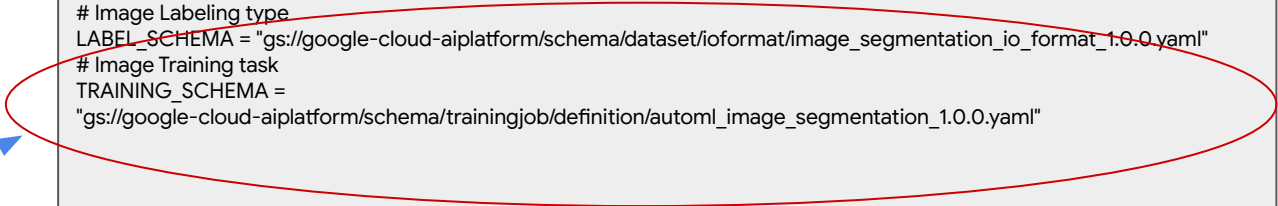
Same as image classification



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.

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.
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

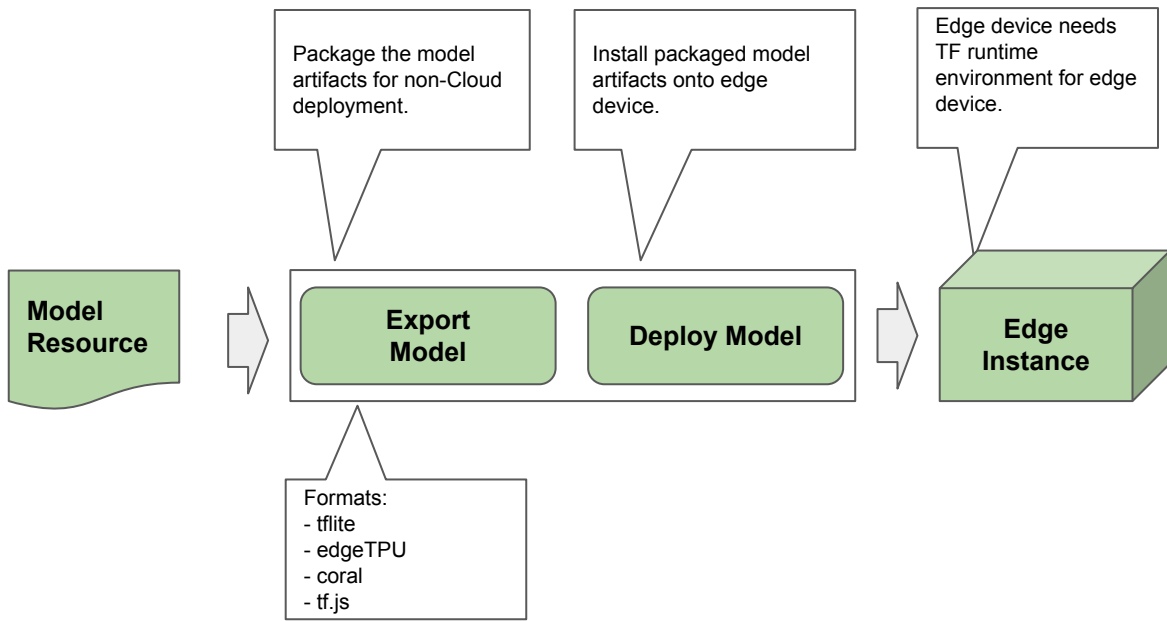


Image Model Exported to Edge - Training

```
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=tfllite_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)

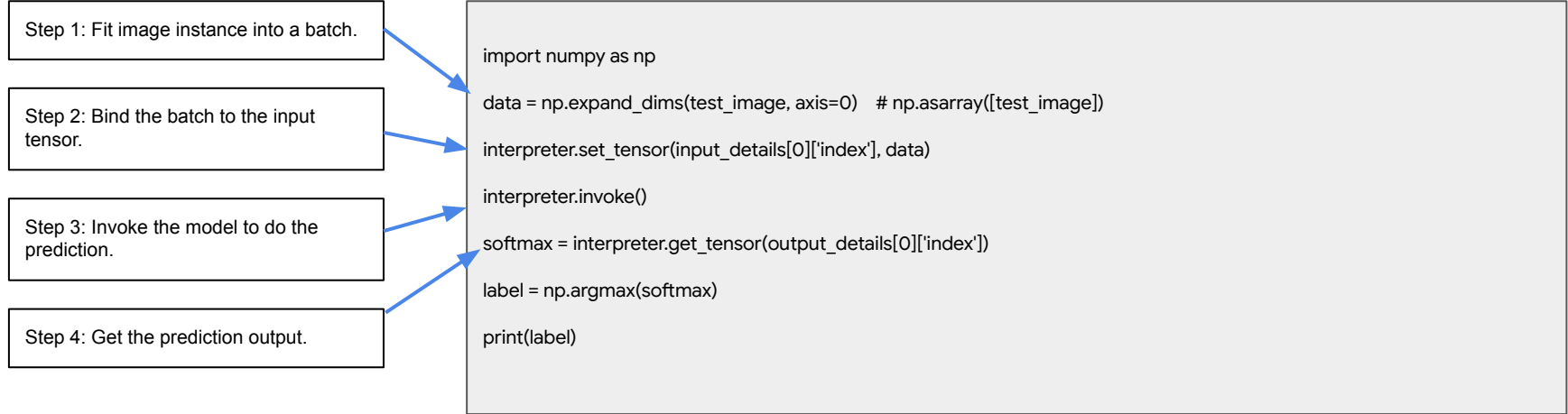
test_image = tf.image.resize(test_image, (224, 224))
print("test image shape", test_image.shape, test_image.dtype)

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



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

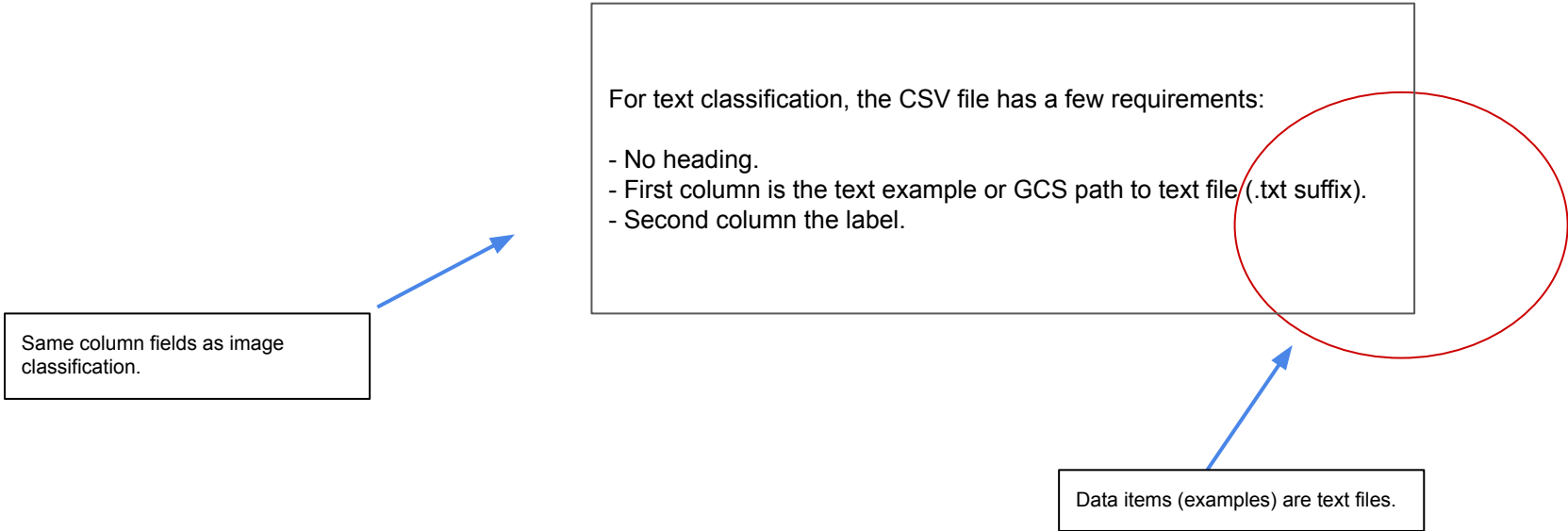


Text Classification - Labeling

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.



```
graph LR; A[Same column fields as image classification.] --> B[For text classification, the CSV file has a few requirements:]; C[Data items (examples) are text files.] --> B; B --- D(( )); D --- E[First column is the text example or GCS path to text file (.txt suffix).]; D --- F[Second column the label.];
```

Data items (examples) are text files.

Text Classification (TCN) - Task Requirements

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

```
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)
```

Text Classification (TCN) - Prediction

Either text example, or GCS path to text file.

Same as image classification

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`).

Text Classification - Batch Prediction

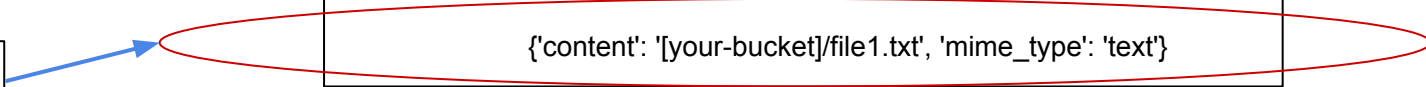
For JSONL file, you make one dictionary entry per line for each data item (instance). The dictionary contains the key/value pairs:

- `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'}`

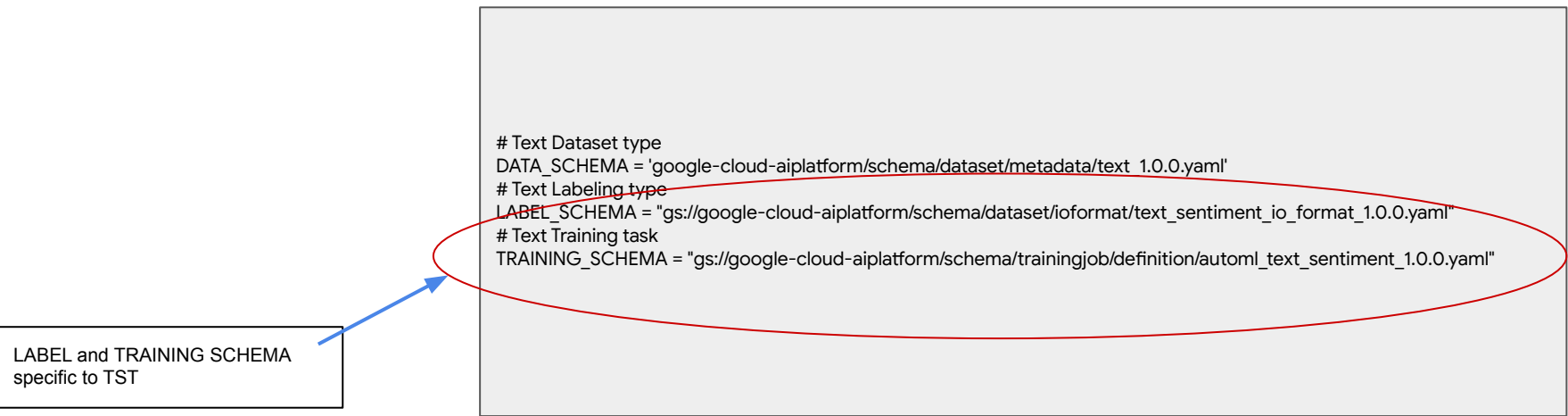
Same as image model, except
content is text file.



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"
```



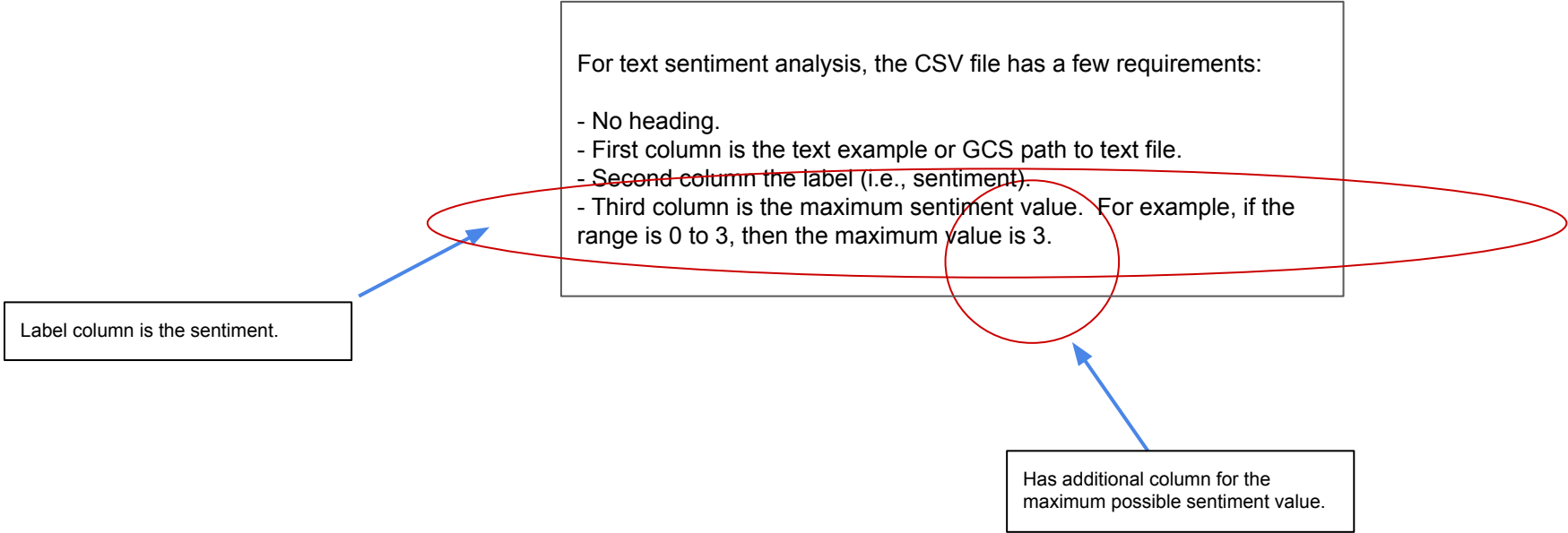
LABEL and TRAINING SCHEMA
specific to TST

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

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

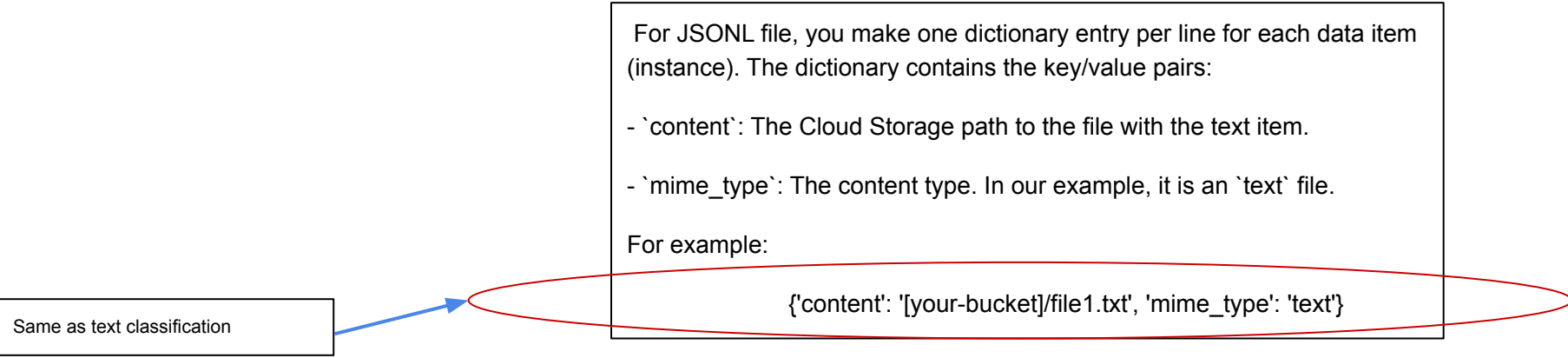
For JSONL file, you make one dictionary entry per line for each data item (instance). The dictionary contains the key/value pairs:

- `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'}`

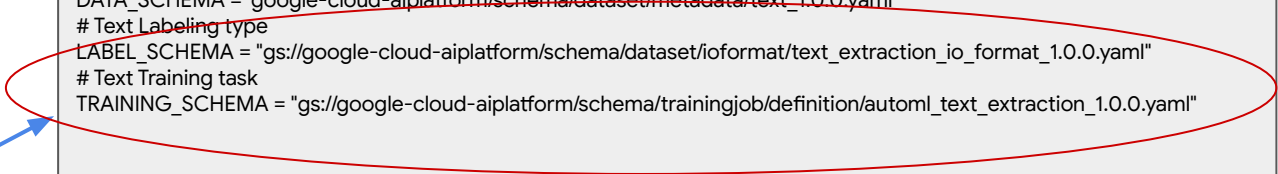
Same as text classification



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

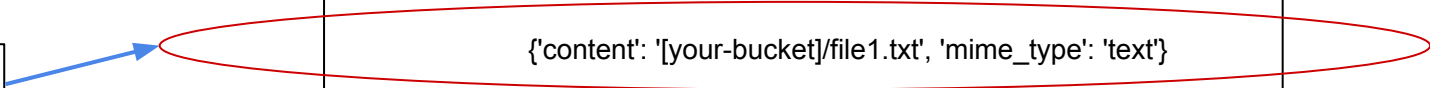
For JSONL file, you make one dictionary entry per line for each data item (instance). The dictionary contains the key/value pairs:

- `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'}`

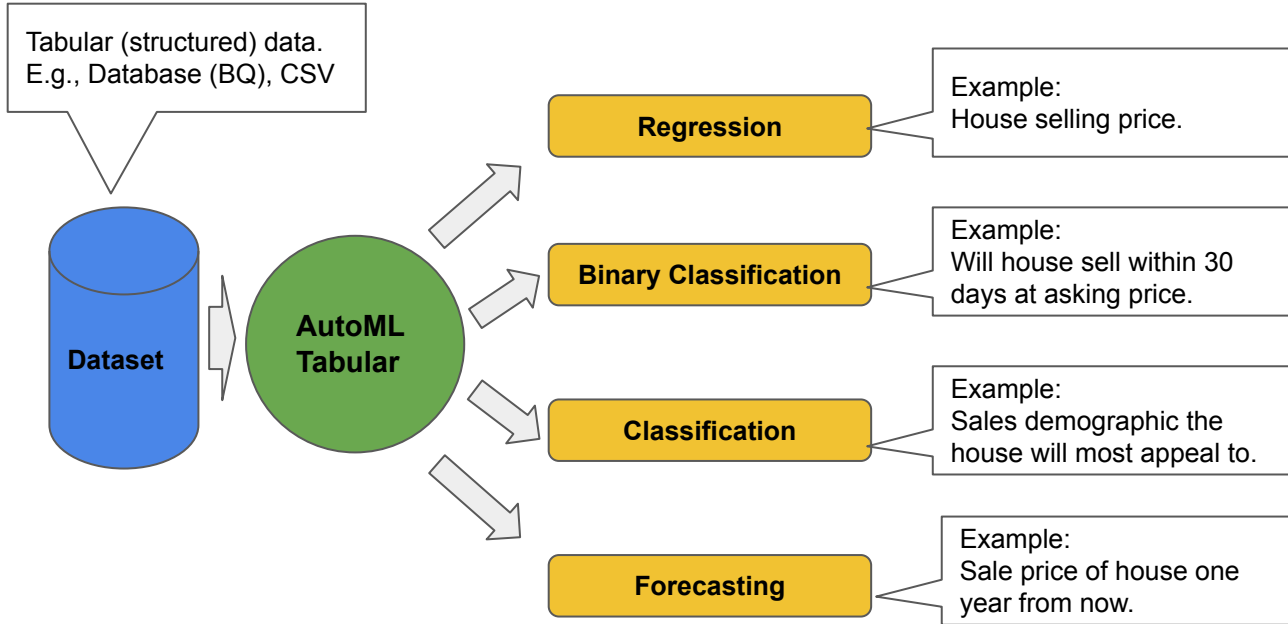
Same as text classification and sentiment analysis.



Workshop 4: Tabular Models

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

Structured Data Models



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),  
  })  
))
```

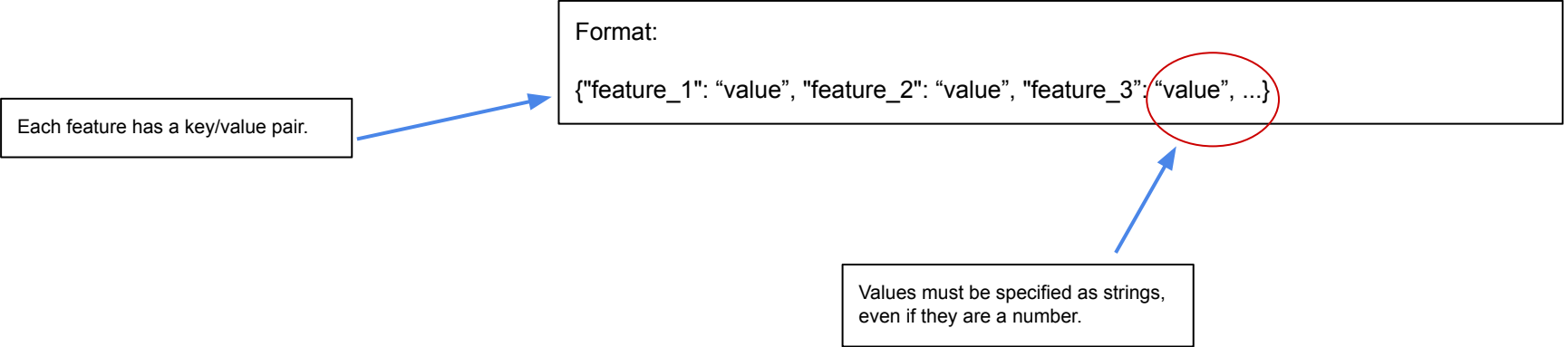
Must specific transformation
(feature engineering) for each
feature – even if defaulting to
automatic.

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

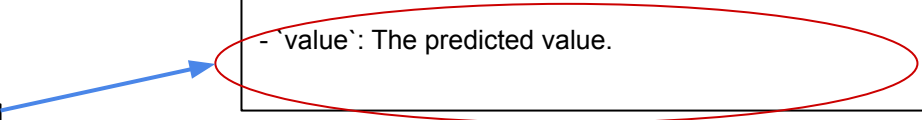


Tabular (LRG) - Prediction Response

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

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.

Each instance is a CSV row



For example:

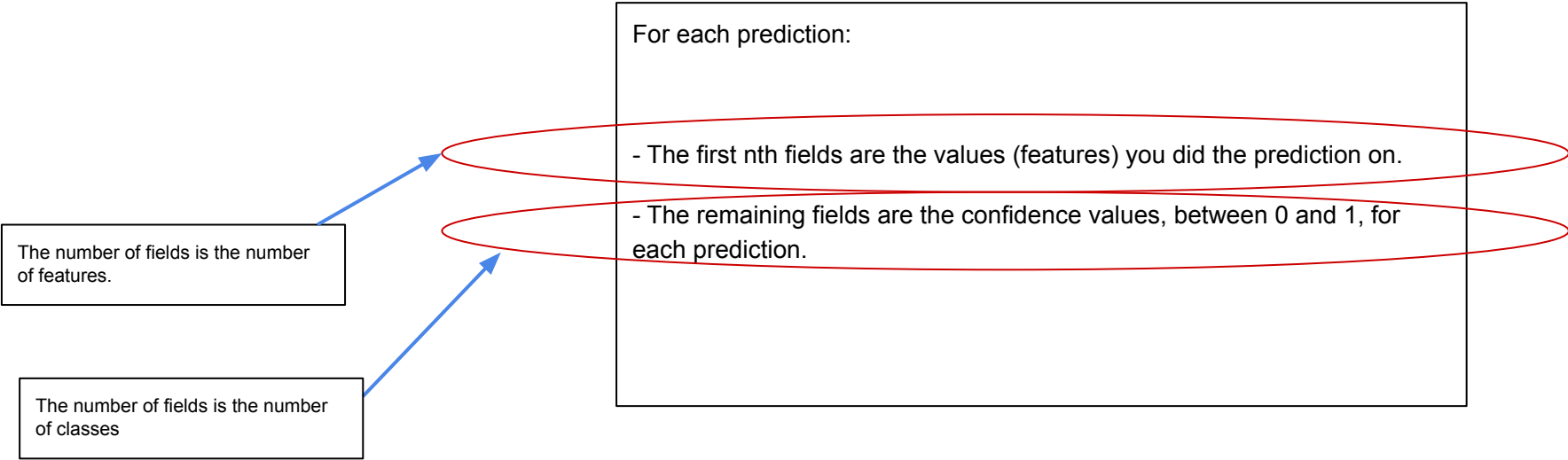
```
"feature_1", "feature_2". ...  
value_1, value_2, ...
```

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.



```
graph LR; A[The number of fields is the number of features.] --> B[The first nth fields are the values (features) you did the prediction on.]; C[The number of fields is the number of classes] --> D[The remaining fields are the confidence values, between 0 and 1, for each prediction.];
```

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

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.

Same as other tabular models

```
graph LR; A[Same as other tabular models] --> B[Requirements]; C[Specific to forecasting] --> B;
```

Specific to forecasting

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.

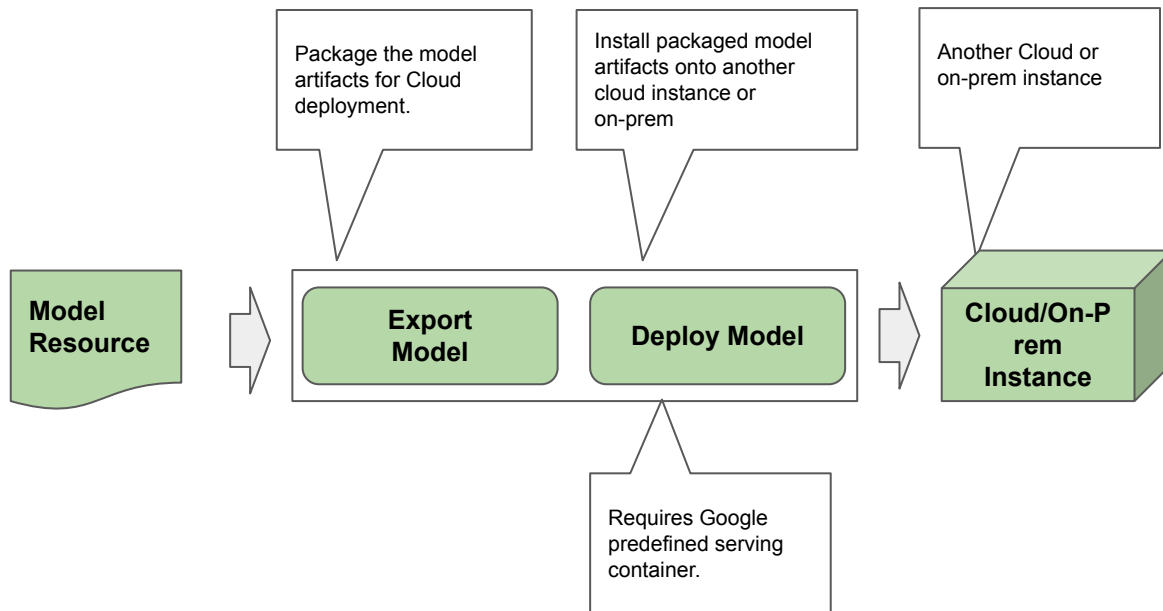
Format for specifying a BigQuery dataset.

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

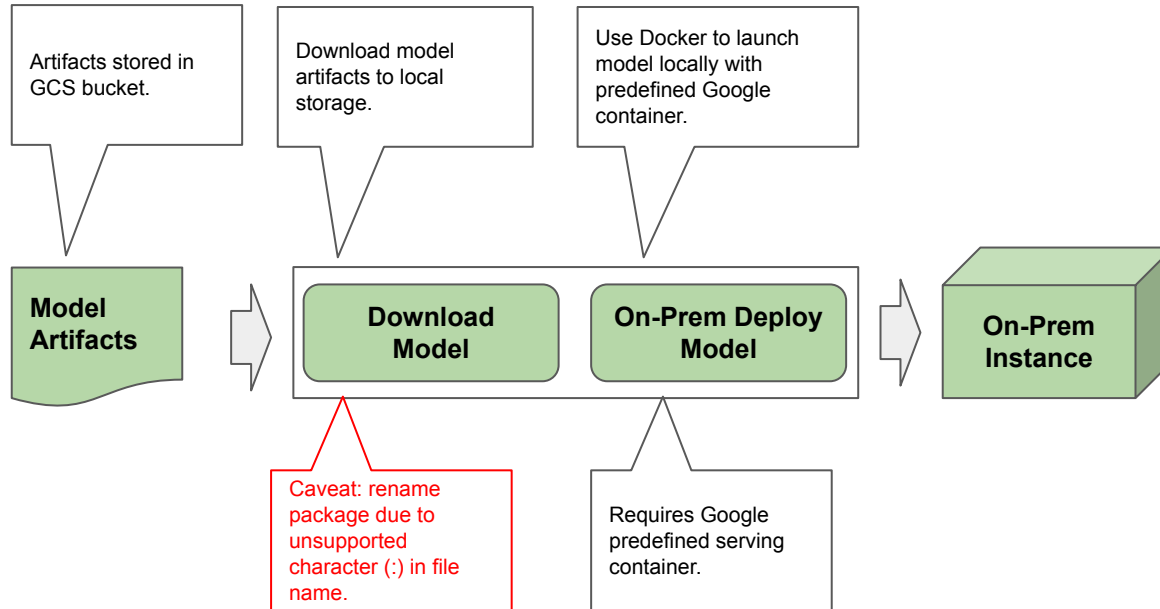
```
def export_model(name, format, gcs_dest):  
    output_config = {  
        "artifact_destination": {"output_uri_prefix": gcs_dest},  
        "export_format_id": format,  
    }  
    response = clients['model!'].export_model(name=name, output_config=output_config)  
    print("Long running operation:", 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 Serving



On-Prem Deploy Model

Tabular Model Exported to on-Prem - Docker Startup

Google predefined serving container.

Run locally the container with the export model.

```
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
```

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

The 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 send the prediction request using CURL:

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

Format of prediction request.

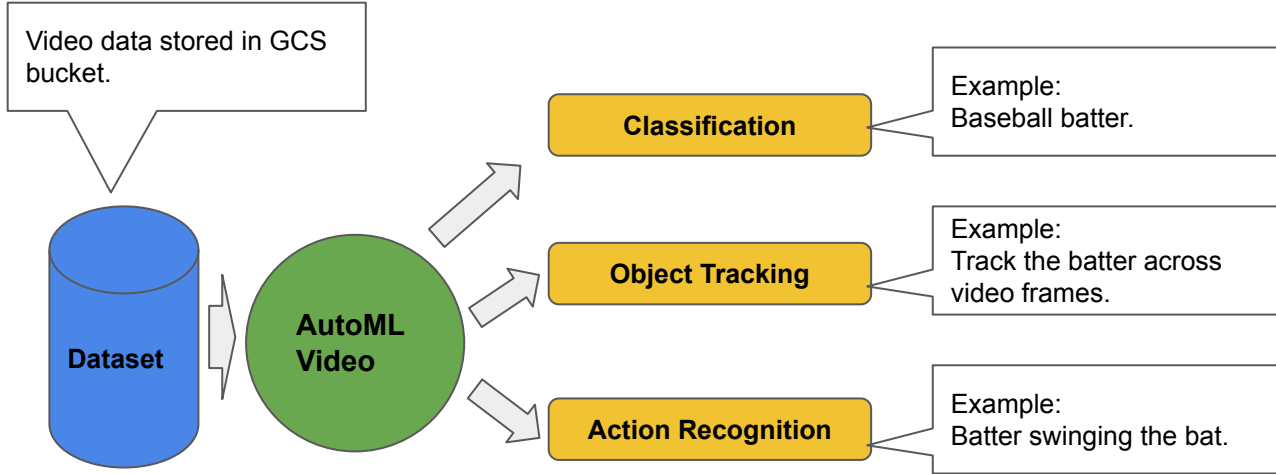
Predefined URL path /predict

Workshop 5: AutoML Wrap Up

- AutoML Video
- AutoML Explainability

Workshop 5: AutoML Video

Video Models



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"
```

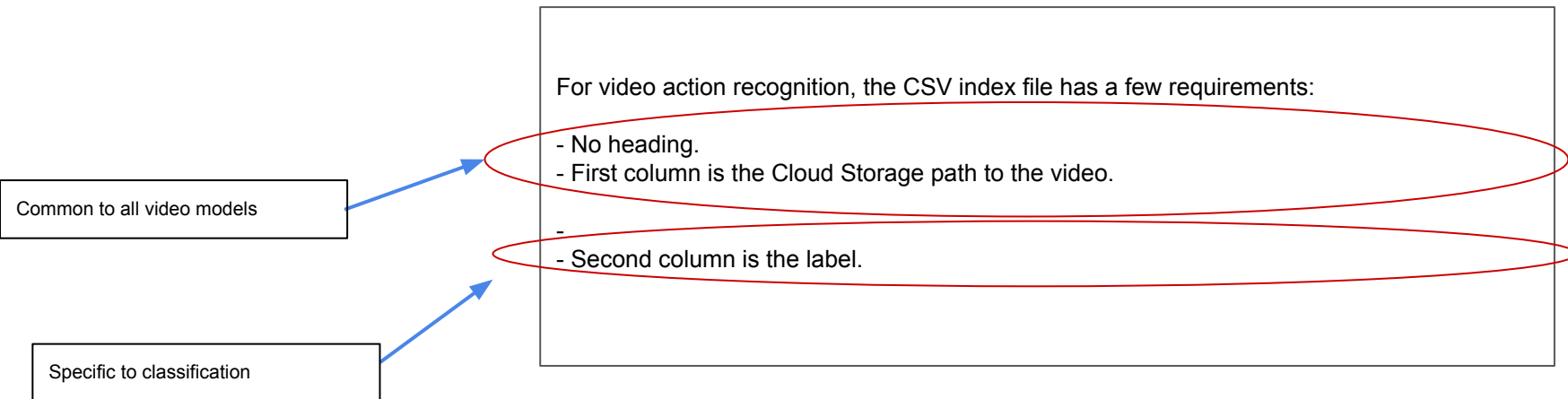
LABEL and TRAINING schemas
specific to model type.

Video Classification (VCN) - Labeling

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.

Common to all video models




Specific to classification

Video Object Tracking (VOT) - 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 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/Eighth/Ninth columns are not used and should be 0.
- Tenth/Eleventh columns are the lower right corner of the bounding box.

Specific to object tracking




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

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.

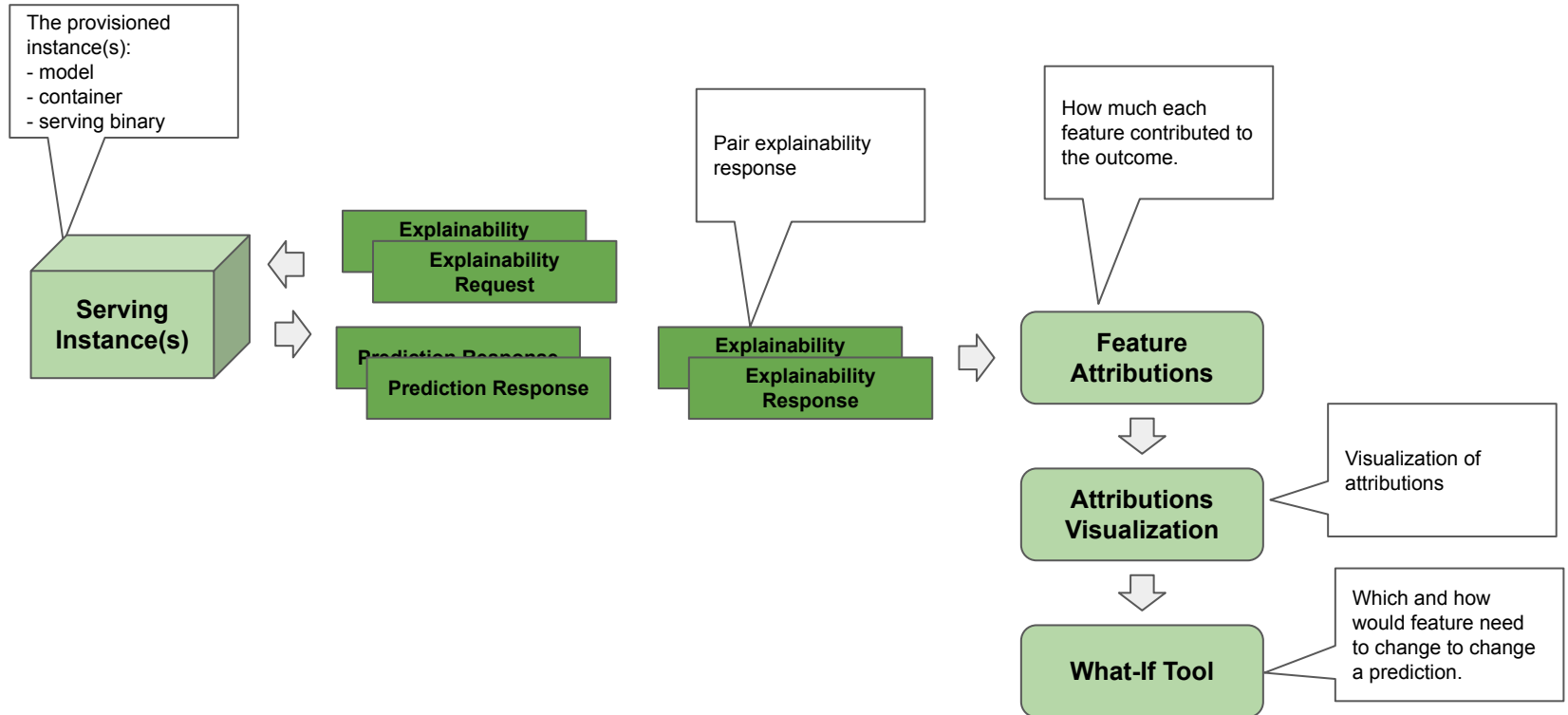
Same as for image models

For all video models

Workshop 5: AutoML Explainability

Workshop 5: AutoML Explainability

Do Online Explainability



Explainability Request

Tabular Explanation Call/Attributions

Call explain method instead of predict

```
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)
```

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']))
```

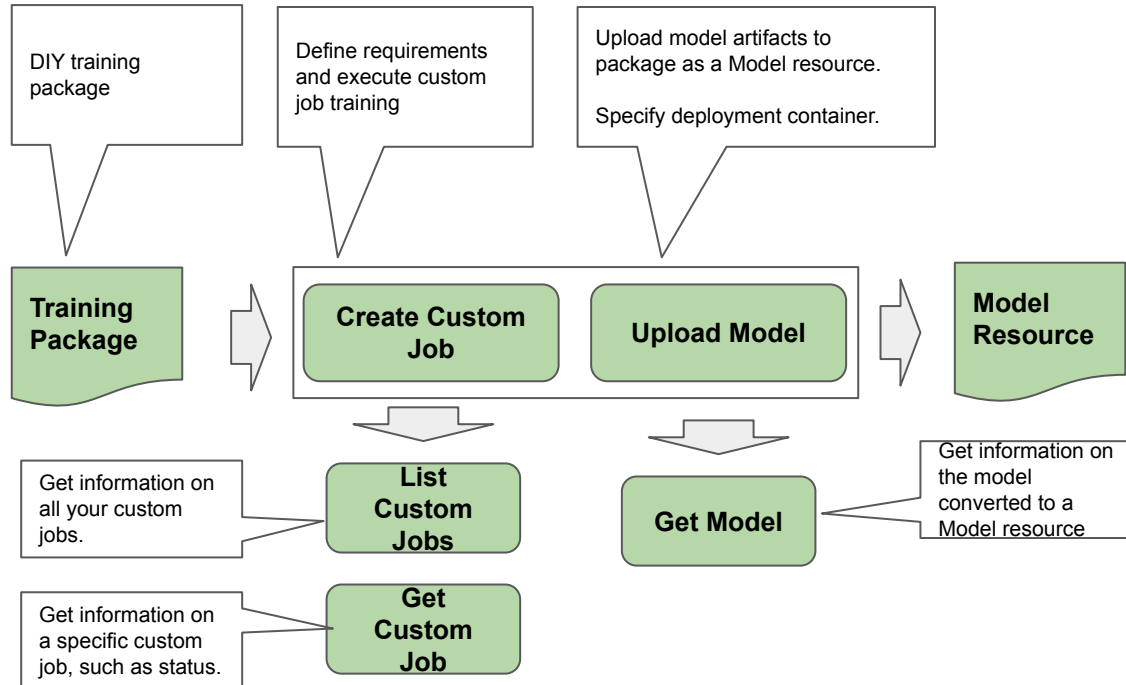
Workshop 6: Custom Jobs

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

Workshop 6: Custom Job Fundamentals

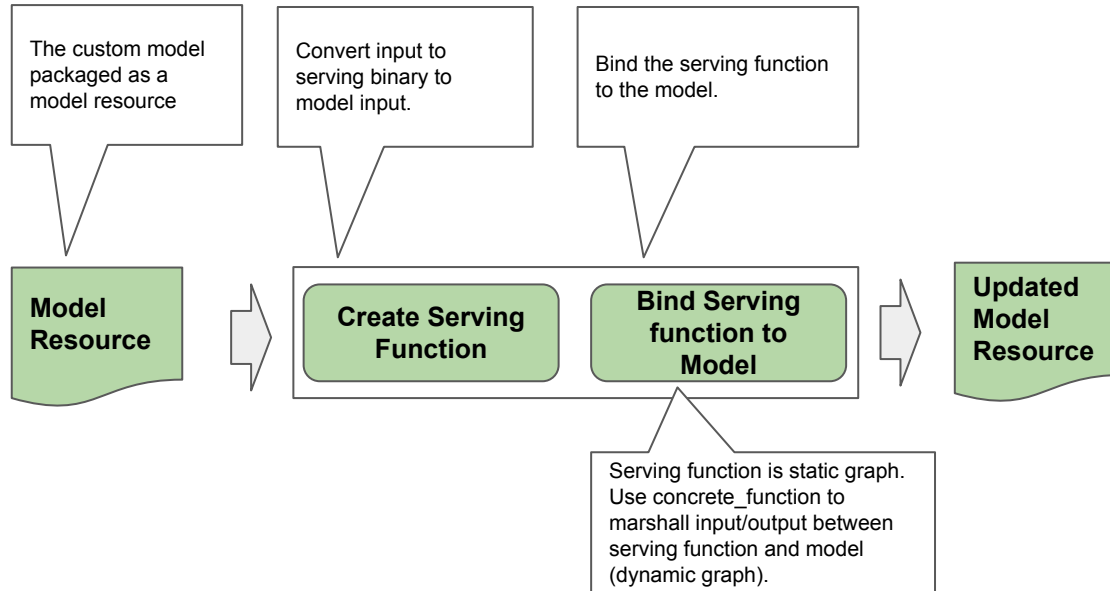
Custom Jobs for all Model Types

Custom Training

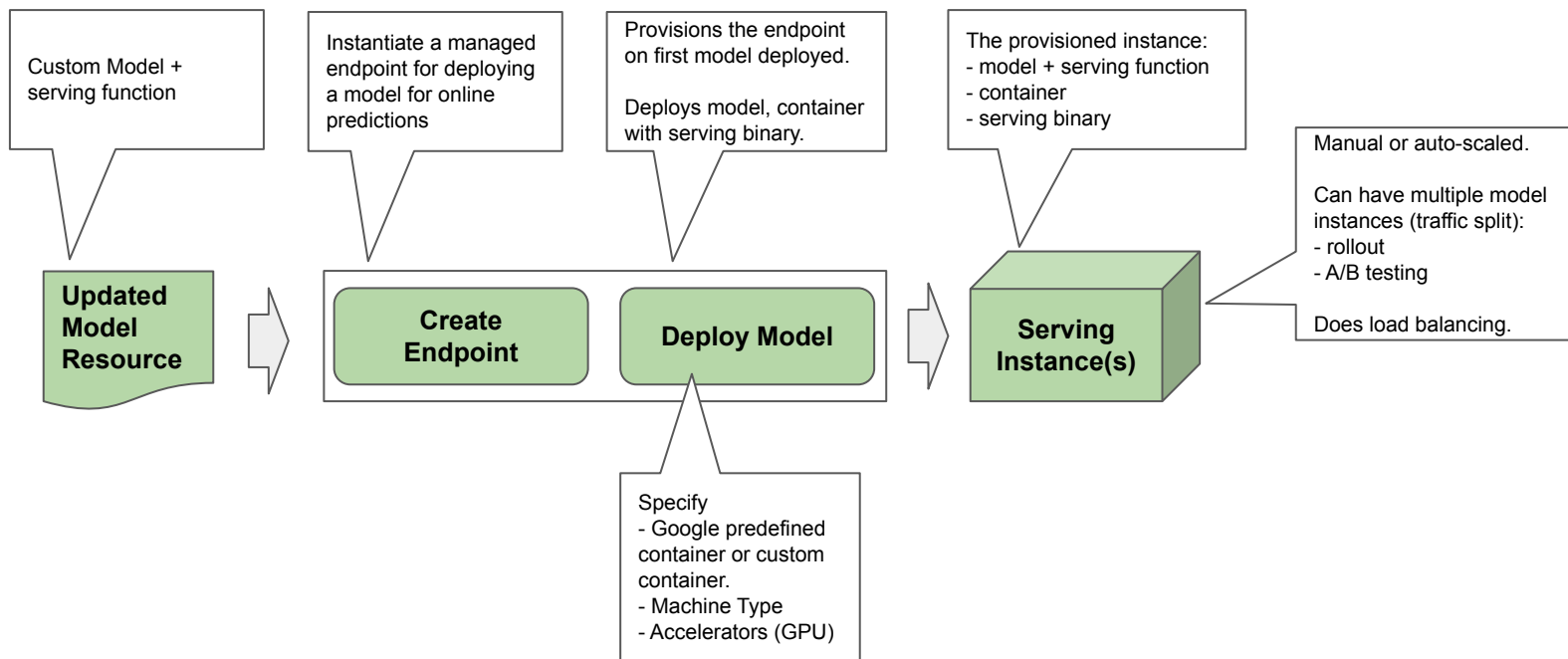


Custom Serving Function - Model data type specific

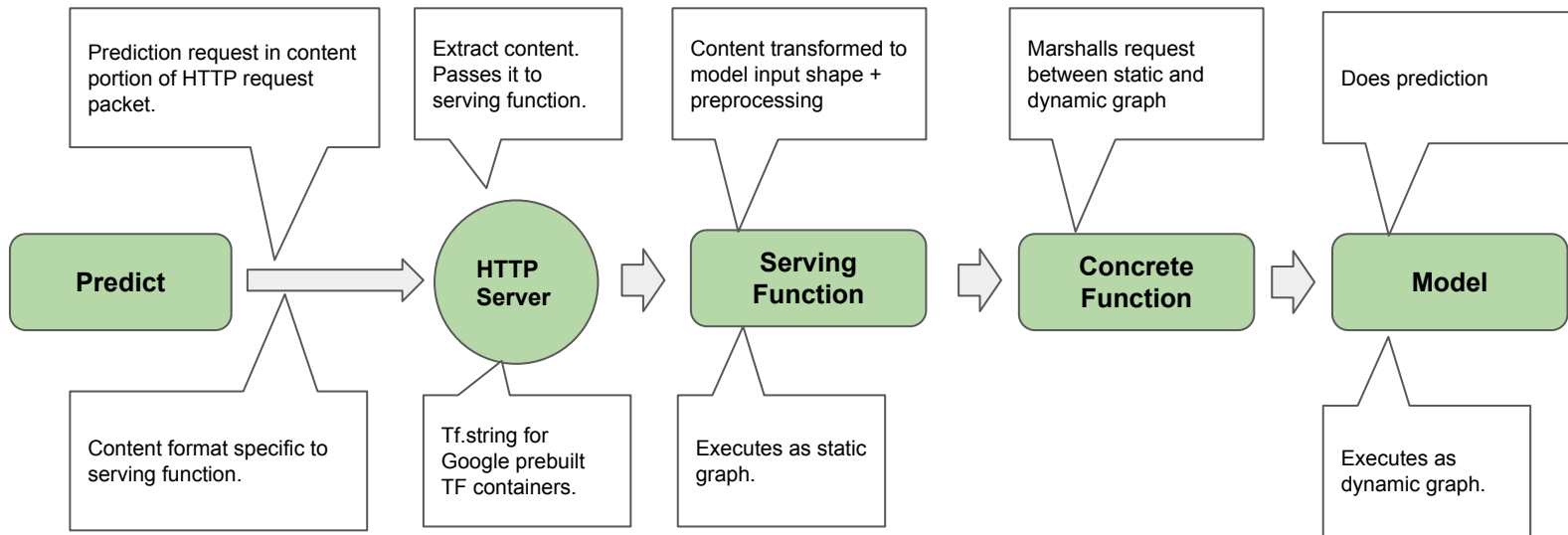
Custom Serving Function



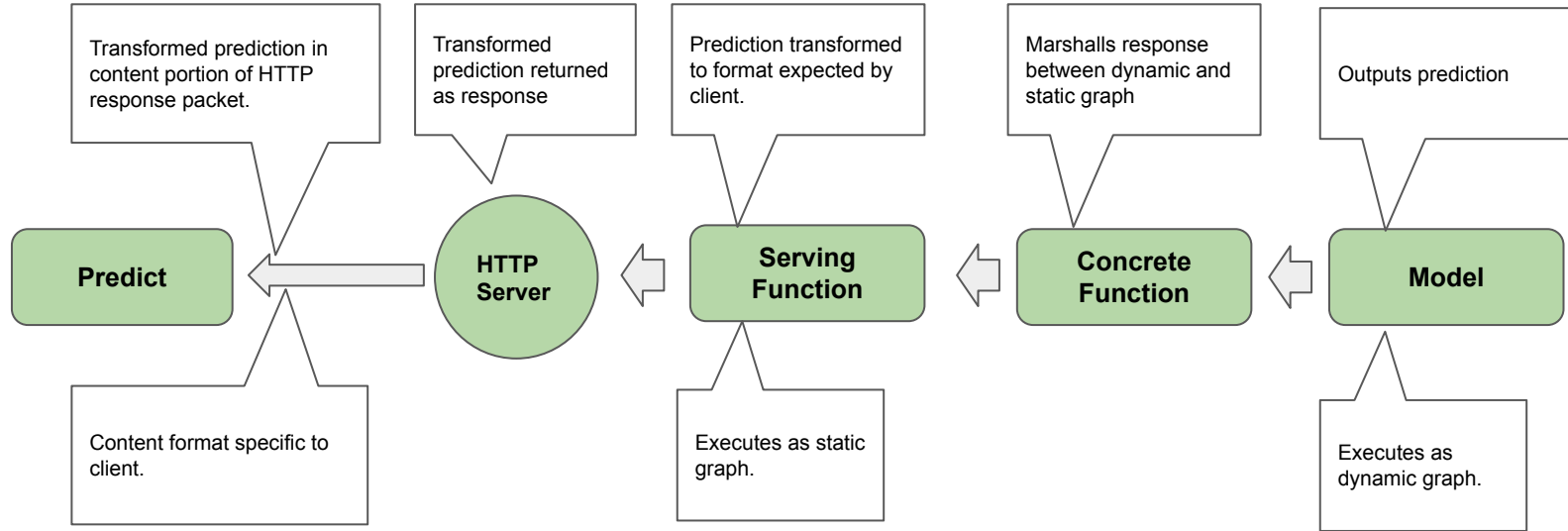
Deploy Custom Model



Make an Online Prediction - Request



Make an Online Prediction - Response



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'

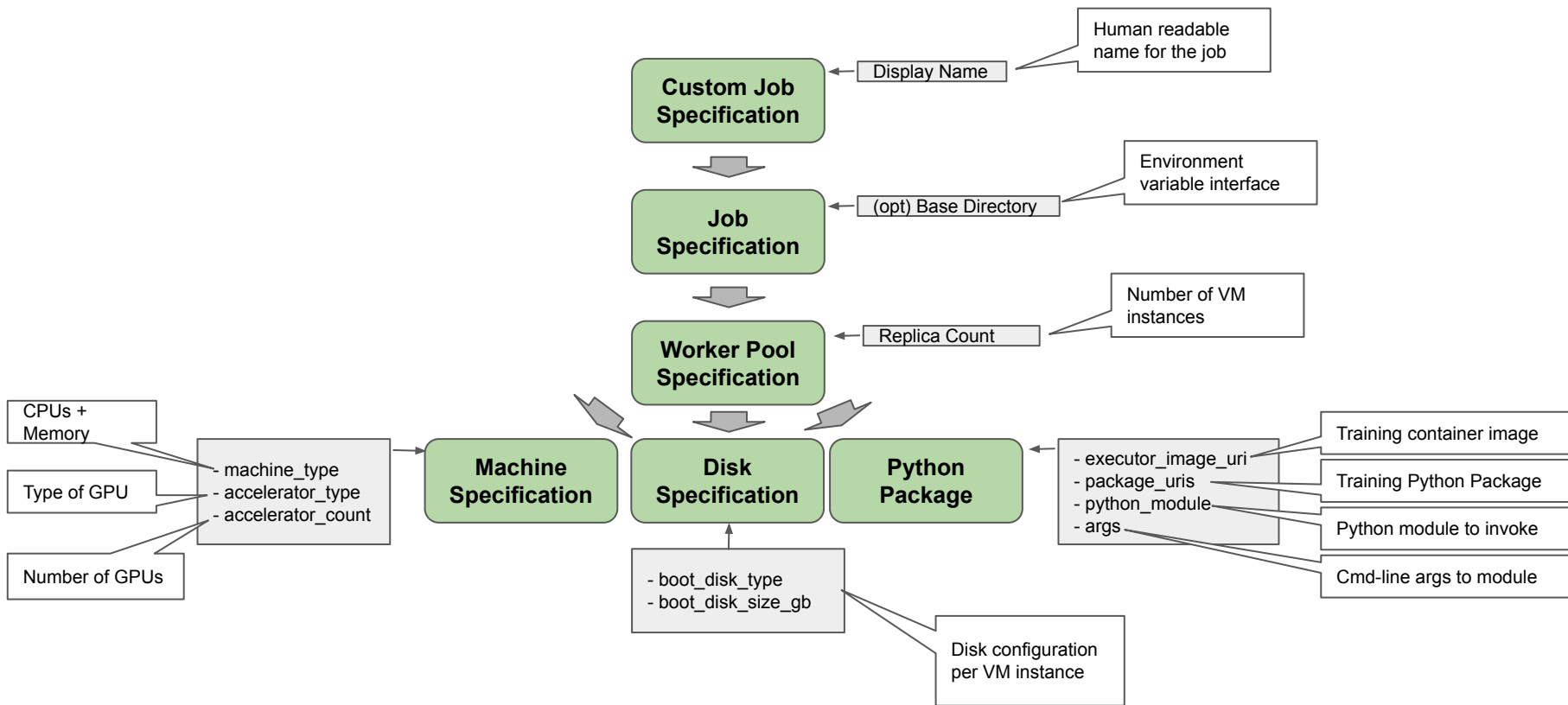
if TF[0] == '2':
    if TRAIN_GPU:
        TRAIN_VERSION = 'tf-gpu.{}'.format(TF)
    else:
        TRAIN_VERSION = 'tf-cpu.{}'.format(TF)
    if DEPLOY_GPU:
        DEPLOY_VERSION = 'tf2-gpu.{}'.format(TF)
    else:
        DEPLOY_VERSION = 'tf2-cpu.{}'.format(TF)
else:
    if TRAIN_GPU:
        TRAIN_VERSION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

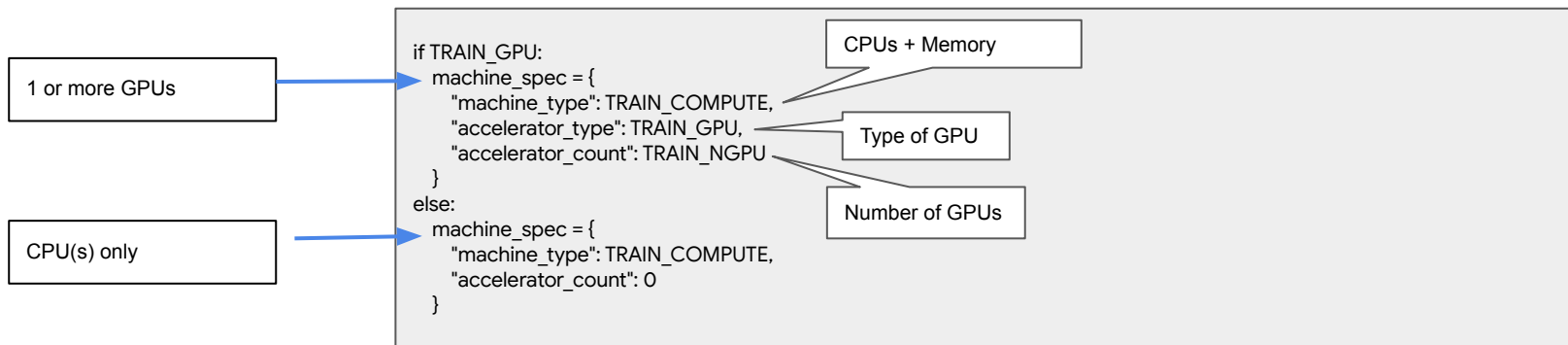
Worker Pool Specification Parameters

```
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



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

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

(Optional) Describes the package.

(Optional) Commentary

Tells the service environment/module requirements for the container.

Folder for the training files.

Entry file that is invoked by the service, along with cmd-line args.

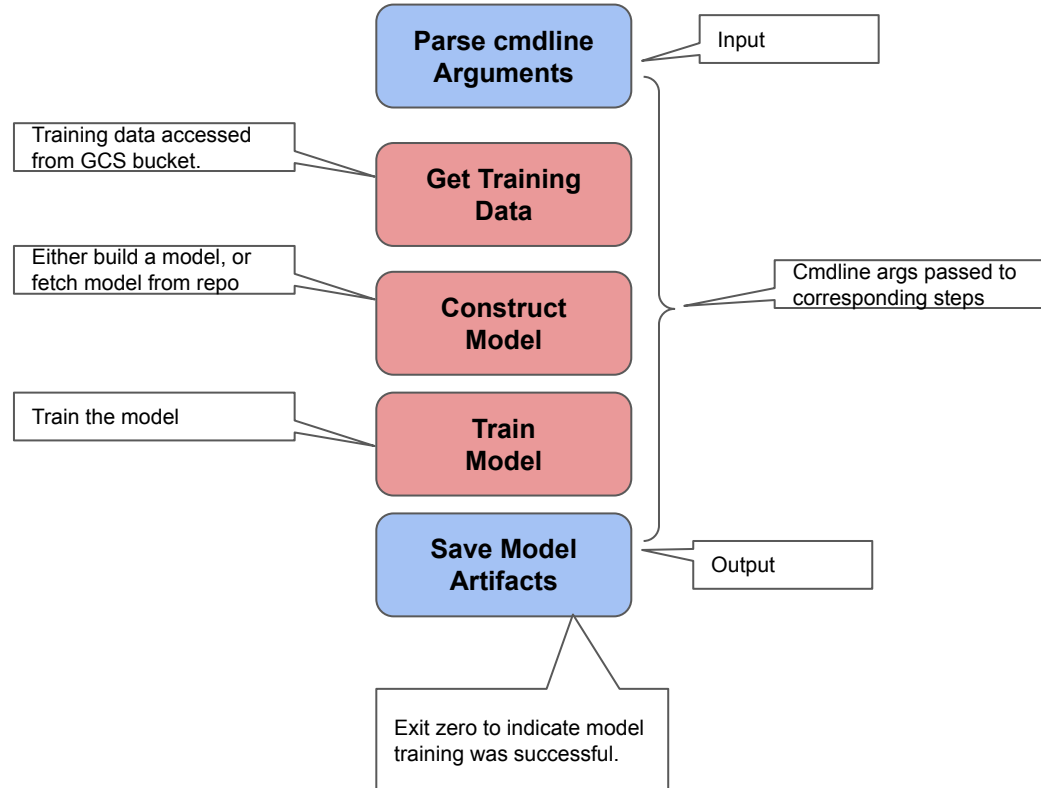
PKG-INFO
README.md
setup.cfg
setup.py
trainer

- `__init__.py`
- `task.py`
- `foobar.py`, etc ...

Makes files in folder an importable package.

Additional files that make up the package.

(Basic) Training File Contents



Parse cmdline Arguments

Parse the Cmd-Line Arguments

Get the location to store the model artifacts.

Hyperparameters

Distribution strategy.

```
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()
```

Otherwise get them from service's environment variable.

Default to single instance.

Save Model Artifacts

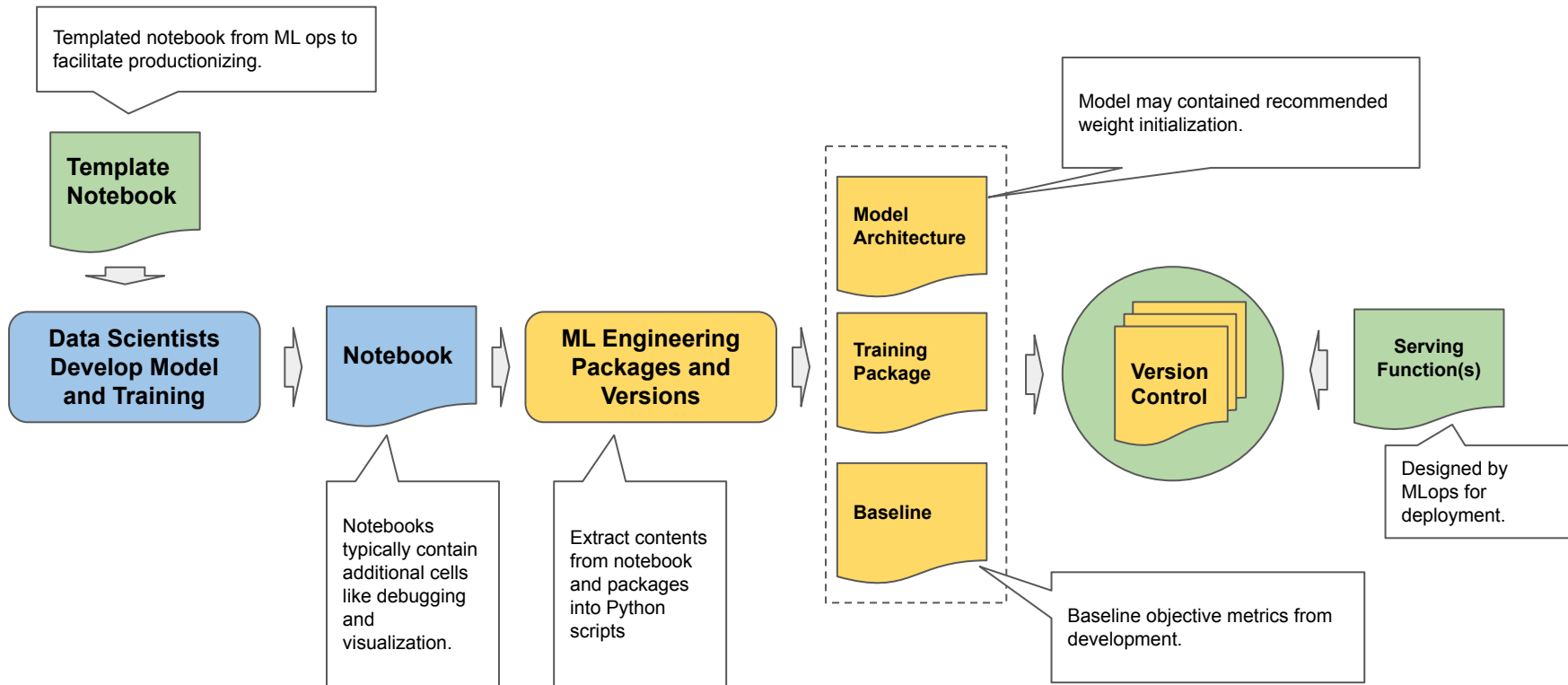
Save the Model Artifacts to GCS

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

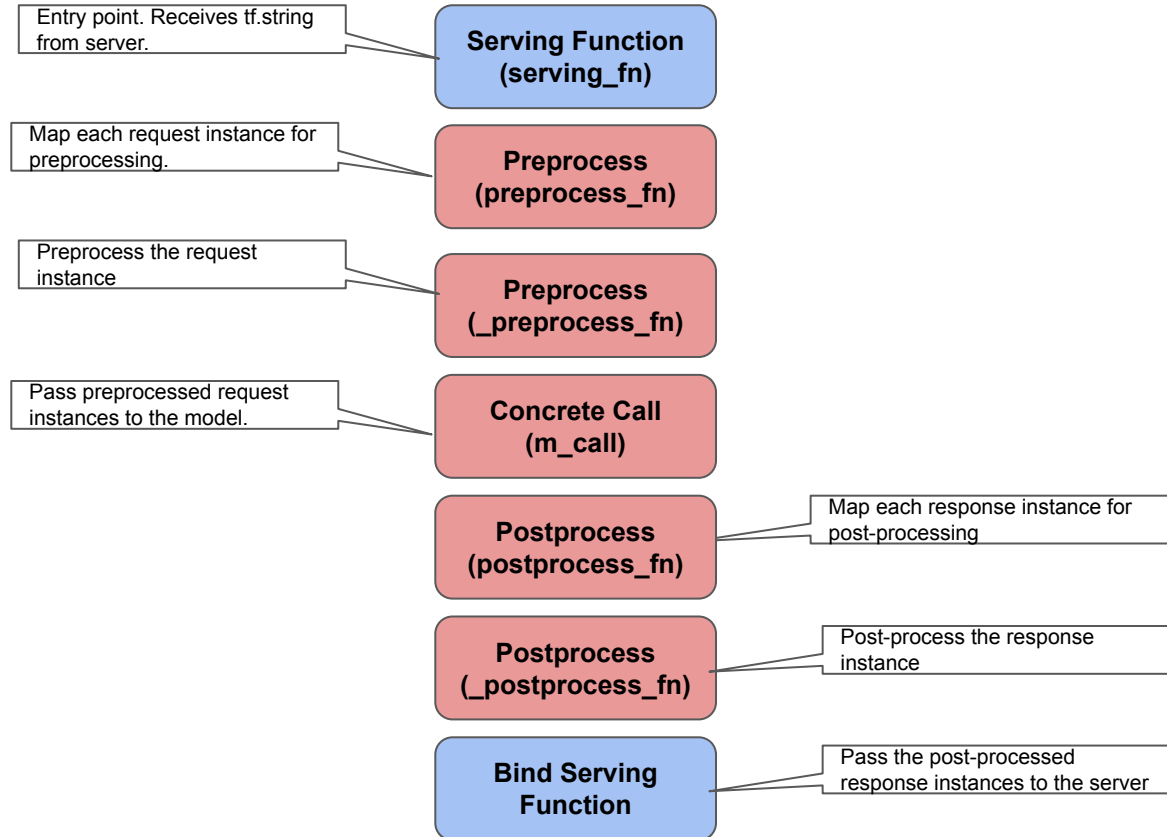
Save mode artifacts after
training.

Saved to a GCS
bucket.

Typical Delivery Process to ML ops



Serving Function Flow



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).

```
@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
```

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

1) Request content received as tf.string

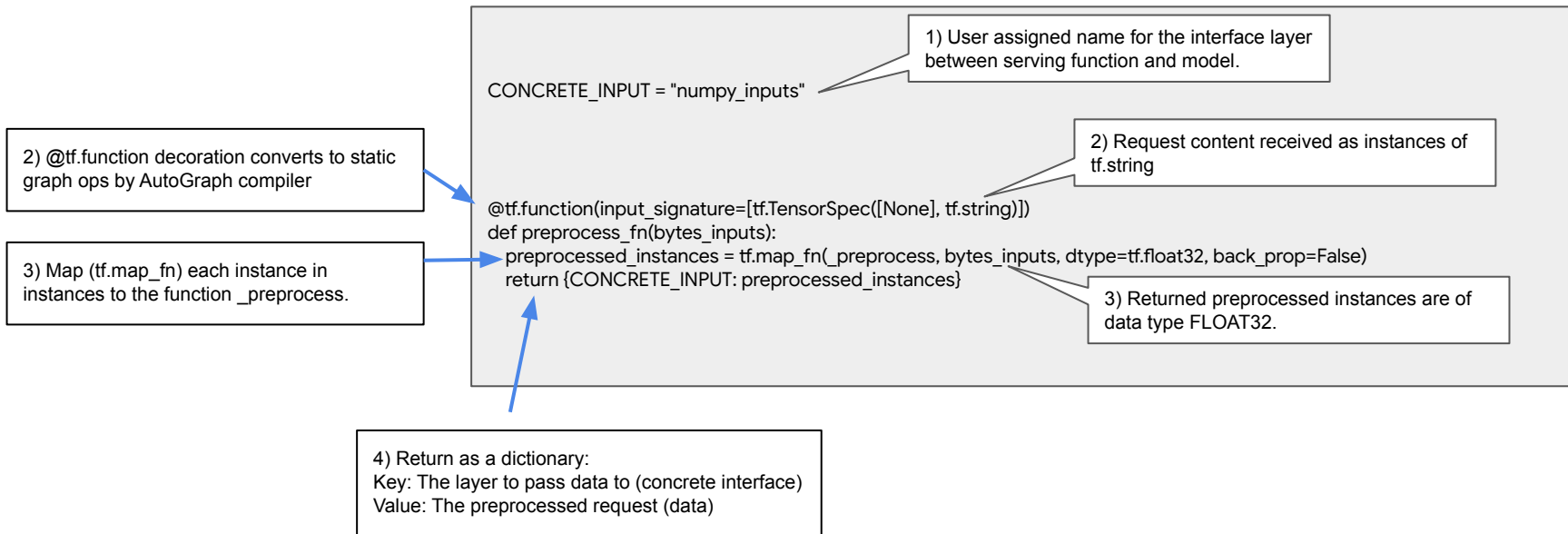
1) Placeholder for instances (number of) size

3) Call the model with the preprocessed instances

6) Will bind serving function to model.

Preprocess (preprocess_fn)

Serving Preprocessing Function



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.

```
@tf.function(input_signature=[tf.TensorSpec([None, <?>], tf.float32)])  
def postprocess_fn(bytes_outputs):  
    postproces_instances = tf.map_fn(_postprocess, bytes_outputs, dtype=tf.<?>, back_prop=False)  
    return postprocess_instances
```

1) Replace with output shape of model.

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

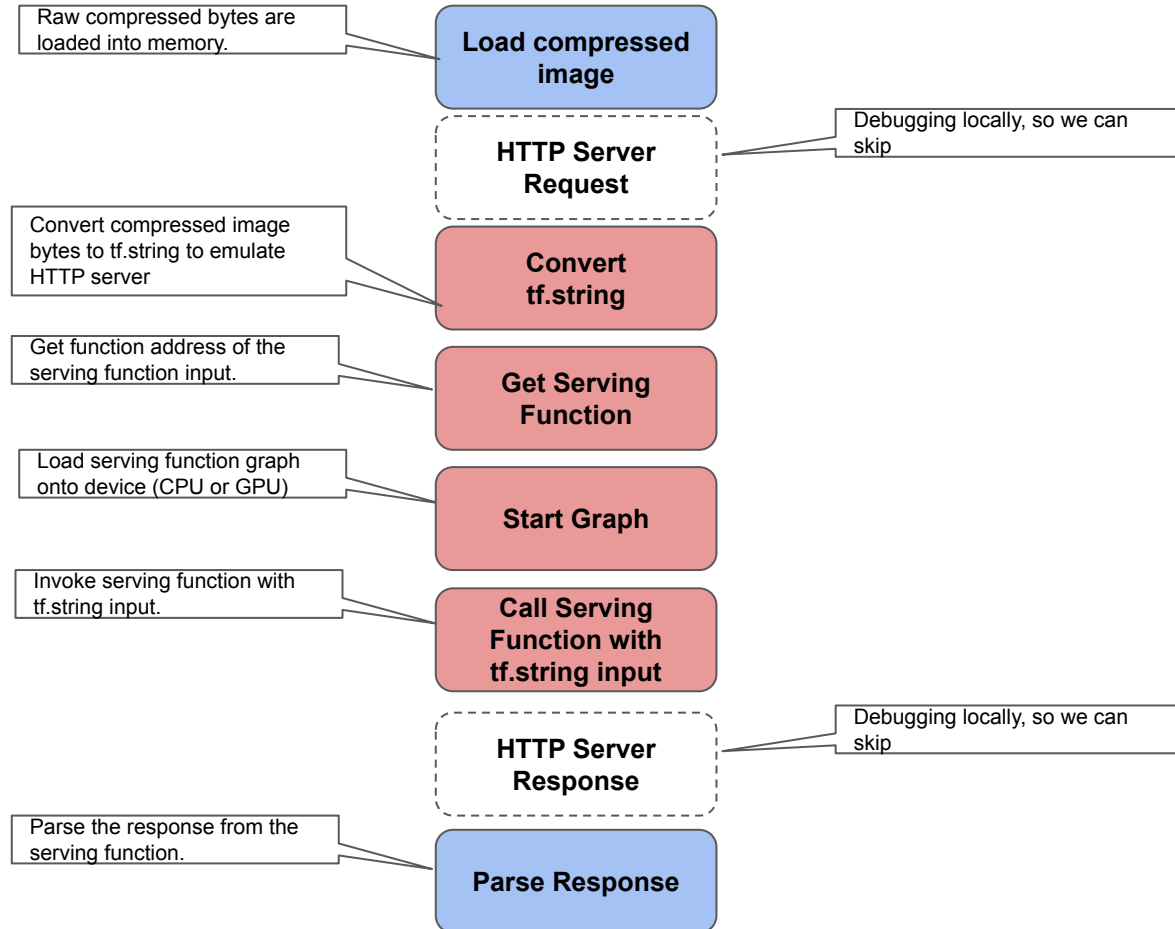
4) Rescale (normalization) pixel data between
0 and 1.

```
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
```

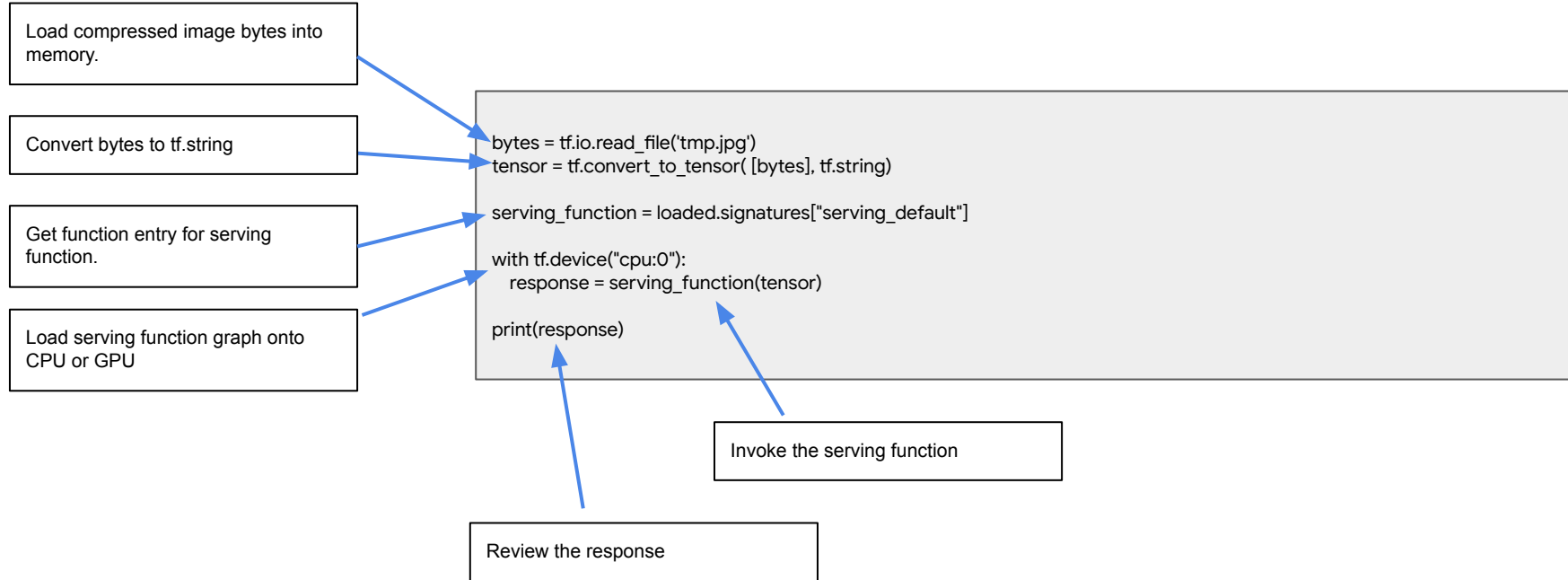
1) Compressed image as tf.string

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

Local Debugging a Serving Function



Debug Serving Function - Image



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 upload_model_response.model
```

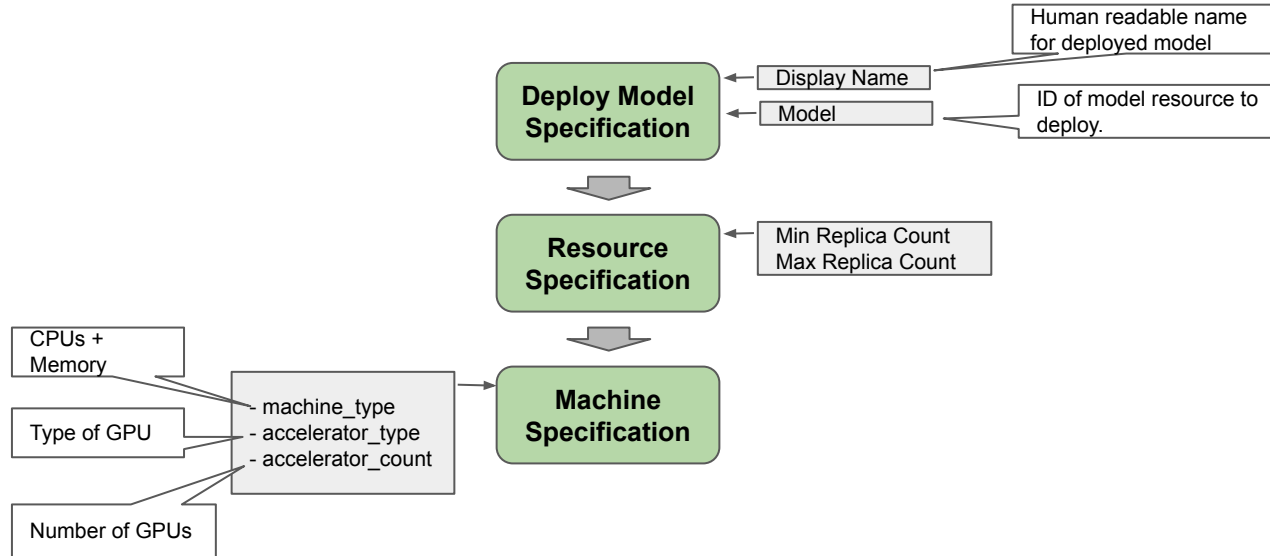
```
model_to_deploy_id = upload_model("mymodel" + TIMESTAMP, IMAGE_URI, model_path_to_deploy)
```

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()
```

Designates 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)

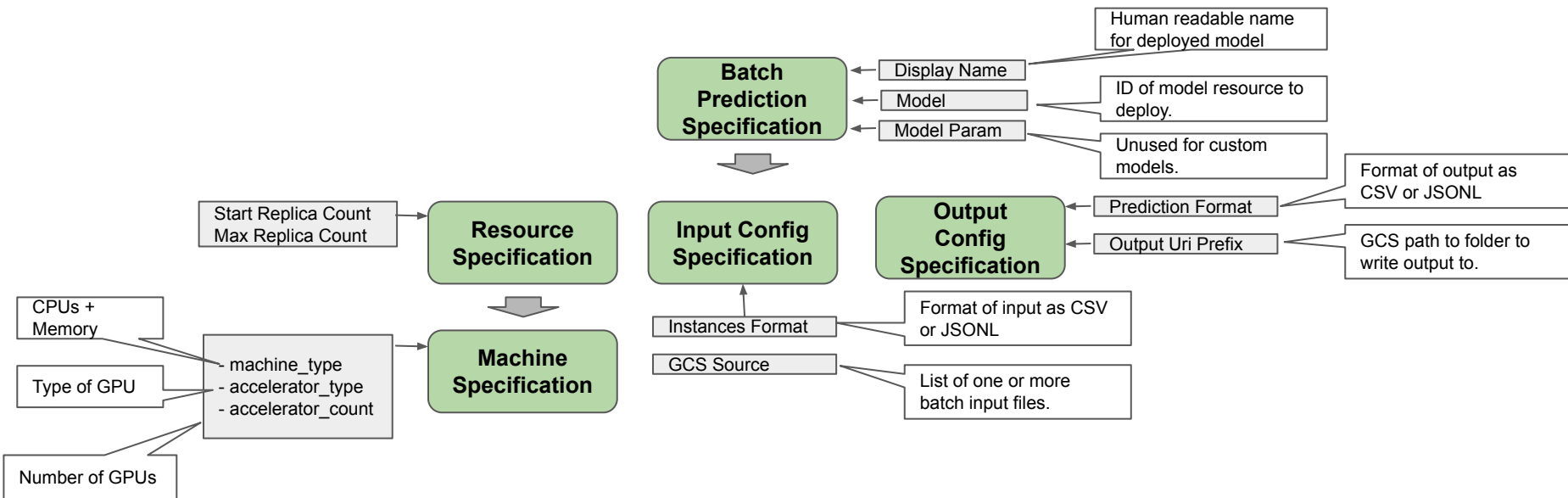
predict_image(b64str, endpoint_id)
```

Which model on endpoint
did the prediction.

No builtin post-processing
for custom models.

Create Batch Prediction Job

Create Batch Prediction Job Specification



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

```
lgcs_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 = {'input_name': {'b64': b64str}}
    f.write(json.dumps(data) + '\n')
    data = {'input_name': {'b64': b64str}}
    f.write(json.dumps(data) + '\n')
```

Workshop 7: Custom Jobs (cont)

- Custom Jobs Text Models
- Custom Jobs Tabular Models
- Hyperparameter Tuning Jobs

Workshop 7: Custom Job Text Models

Predict

Prediction for Custom Text Models

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

Convert text string into an array of bytes

Convert array of bytes to list of characters.

```
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)
```

Which model on endpoint did the prediction.

```
predict_data(test_data, endpoint_name, None)
```

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

Serving input
layer name.

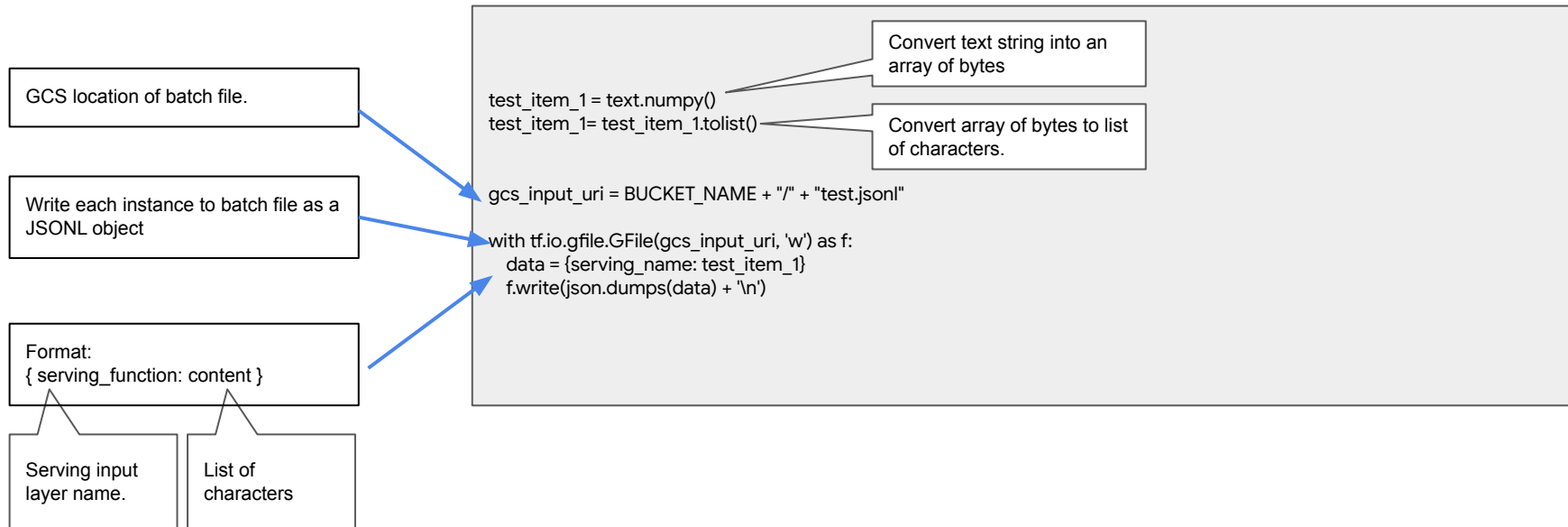
Text as list of
characters

API call for prediction.

JSON object (dictionary) prediction
response for each instance

Batch Predict

Batch Request File Format - Text



Workshop 7: Custom Job Tabular Models

Predict

Prediction for Custom Tabular Models

```
data = data.tolist()
```

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

E.g., [1, 2.0, 3.1, 2.2]

```
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)
```

Which model on endpoint did the prediction.

```
predict_data(test_data, endpoint_name, None)
```

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

Batch Predict

Batch Request File Format - Tabular

