



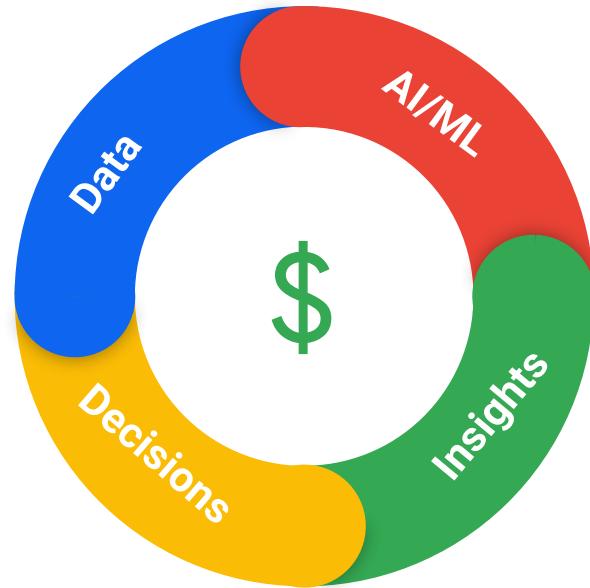
Economics and best practices of running AI/ML workloads on Kubernetes

Maulin Patel, Product Manager, Google
Yaron Haviv, CTO and Founder of Iguazio

Google Cloud



AI/ML driven decisions



Simple

Fast

Cost-effective

MAKING AI/ML

AI/ML is a team sport

ML
Code

How to make AI/ML teams

10x



More Productive

Cloud native AI/ML platform

**ML Framework +
Container +
Kubernetes + HW
Accelerators**

Choose your favorite
ML Framework, pack
models up in
Containers, run on
Kubernetes at scale



ML Framework
Industry-standard & widely adopted

Container
Industry-standard

**Container
Orchestration**
Industry-standard

Hardware Accelerators



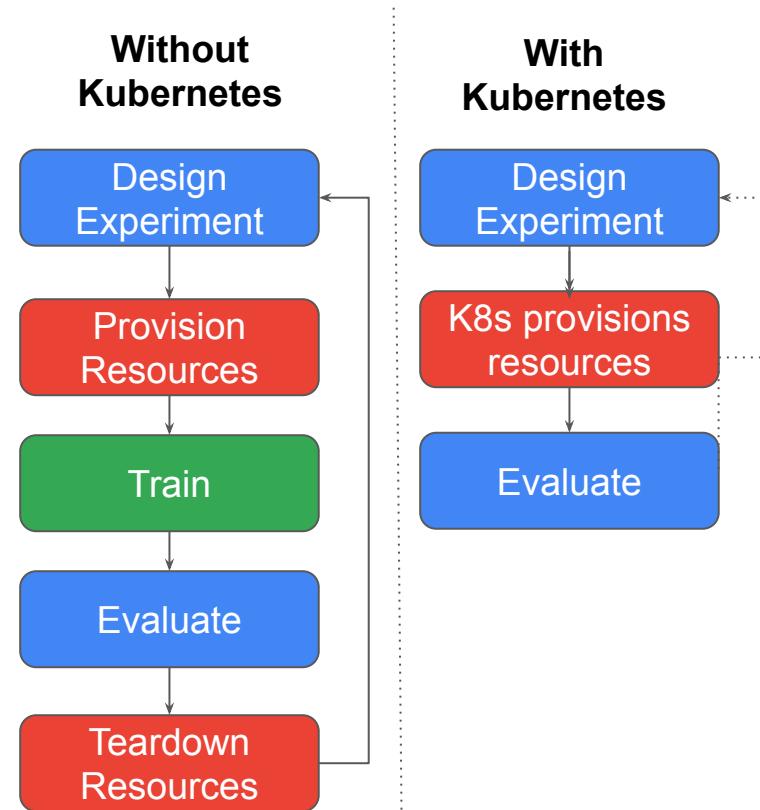
CPUs

GPUs

TPUs

Why Kubernetes for AI/ML?

- Portability
 - Cloud native, open, standard APIs
 - Seamlessly port workloads between Laptop/Cloud
- Scalability
 - Kubernetes scales from a single workstation to thousands of nodes
 - Support for GPU/TPU and distributed computing
- Productivity
 - Frees up users from managing their own workstations, servers and VMs.
 - Lets you focus on model building and training



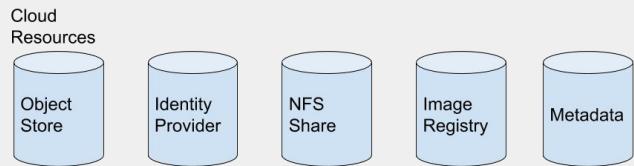
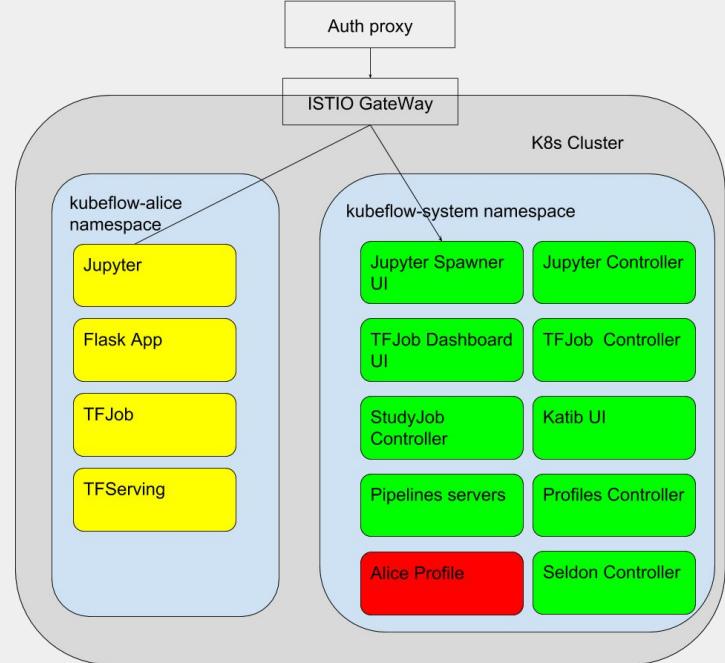
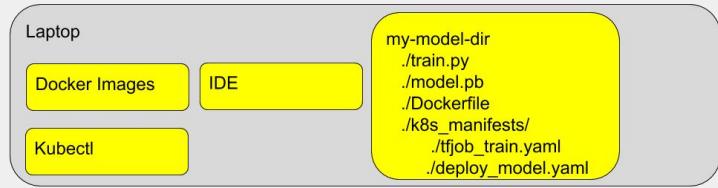
Kubeflow



A Kubernetes-native OSS Platform to **Develop, Deploy** and **Manage, Scalable and End-to-End** ML Workloads



<https://kubeflow.org>

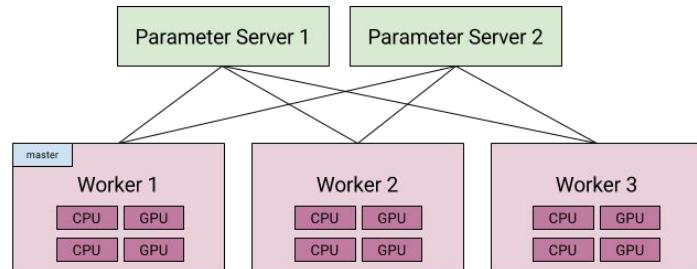


Simple/Fast/Cost-effective

TensorFlow training (TFJob)

- Integrates TensorFlow **distributed training** and estimator API with Kubernetes
- Uses Kubernetes to **scale training** and **leverage** hardware accelerators
- Users benefit from **Kubernetes toolchain**
 - kubectl for CLI
 - Kubernetes dashboard for monitoring

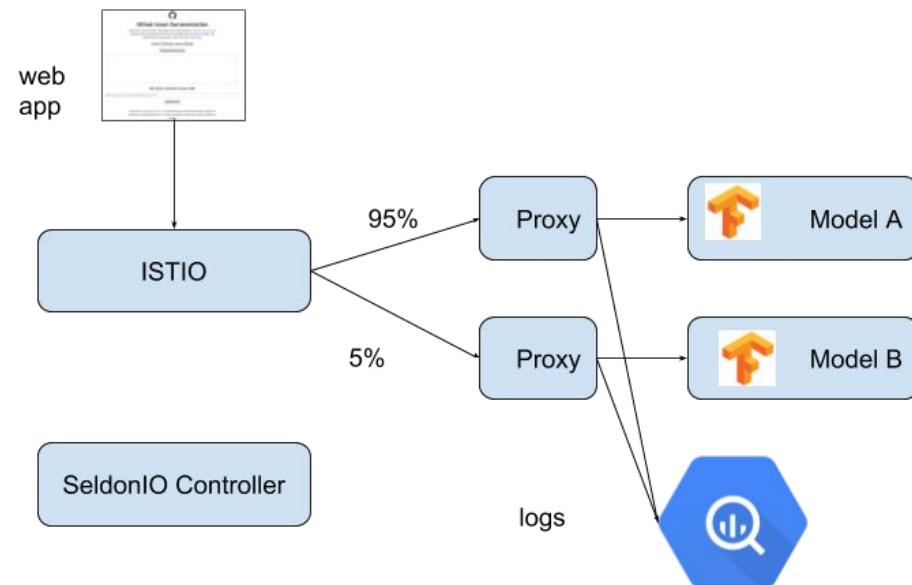
```
apiVersion: kubeflow.org/v1alpha2
kind: TFJob
metadata:
  name: tf-job-simple
  namespace: kubeflow
spec:
  tfReplicaSpecs:
    Workers:
      replicas: 3
      template:
        spec:
          containers:
            - image: acme/myjob
```



TensorFlow serving

model push ≠ binary push

- Kubernetes **native TF Serving**
- Leveraging Kubernetes to **simplify model rollouts**
- **Prometheus exporter for metrics**
- **ISTIO for telemetry and traffic splitting**



Get started right

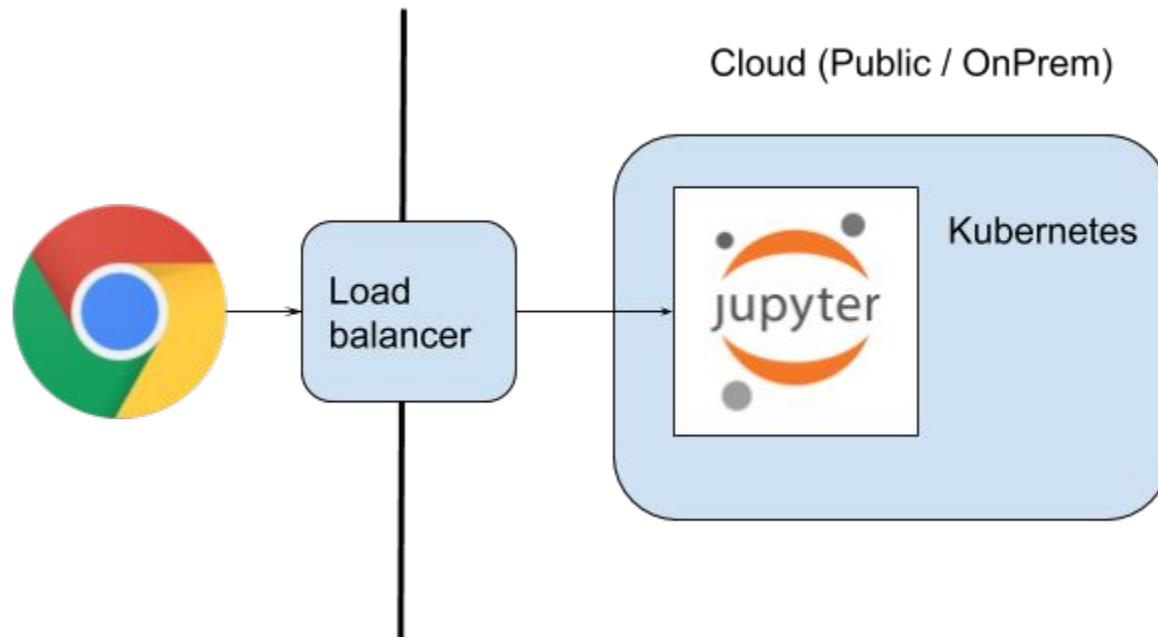
- Day 0 start with the infrastructure (Notebook, Kubernetes, ISTIO, etc...)
- Day 0 focus on model development
 - Use UIs to launch notebooks
 - Python SDK (fairing) for training / deploying models
- Day N leverage K8s to scale
 - Use the same infrastructure as non-ML applications
 - Build a single infrastructure team

The screenshot shows the Kubeflow UI interface. At the top, there's a navigation bar with the Kubeflow logo and a 'Namespace' dropdown set to 'kubeflow'. Below it, the 'Notebook Servers' page lists a single server named 'jlew1' which was created '1 day ago' using the 'tensorflow-1.13.1-notebook-cpu' image. The server has 32 CPU units and 32Gi of memory. A 'CONNECT' button is available for this server.

Below the server list is a 'New Notebook Server' dialog box. It prompts for the name ('Notebook Server's Name') and namespace ('Namespace') of the new server, both set to 'kubeflow'. It also allows specifying the 'Image' (a starting Jupyter Docker Image), 'CPU' (1.5), 'Memory' (1.0G), 'Workspace Volume' (a volume named 'notebook-workspace' with 10 GiB capacity and mounted at '/home/jovyan'), and 'Data Volumes' (an empty volume). The 'Extra Resources' section is currently empty. At the bottom of the dialog are 'SAVE' and 'CANCEL' buttons.



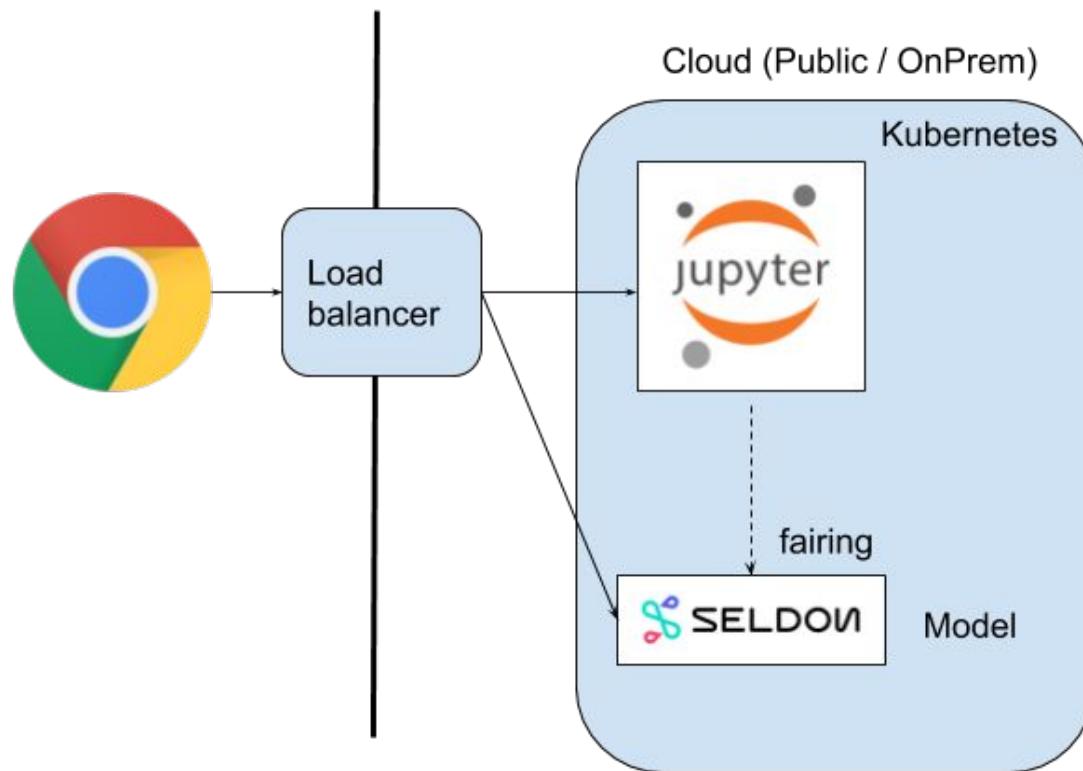
Day 0: Data scientist friendly Notebooks



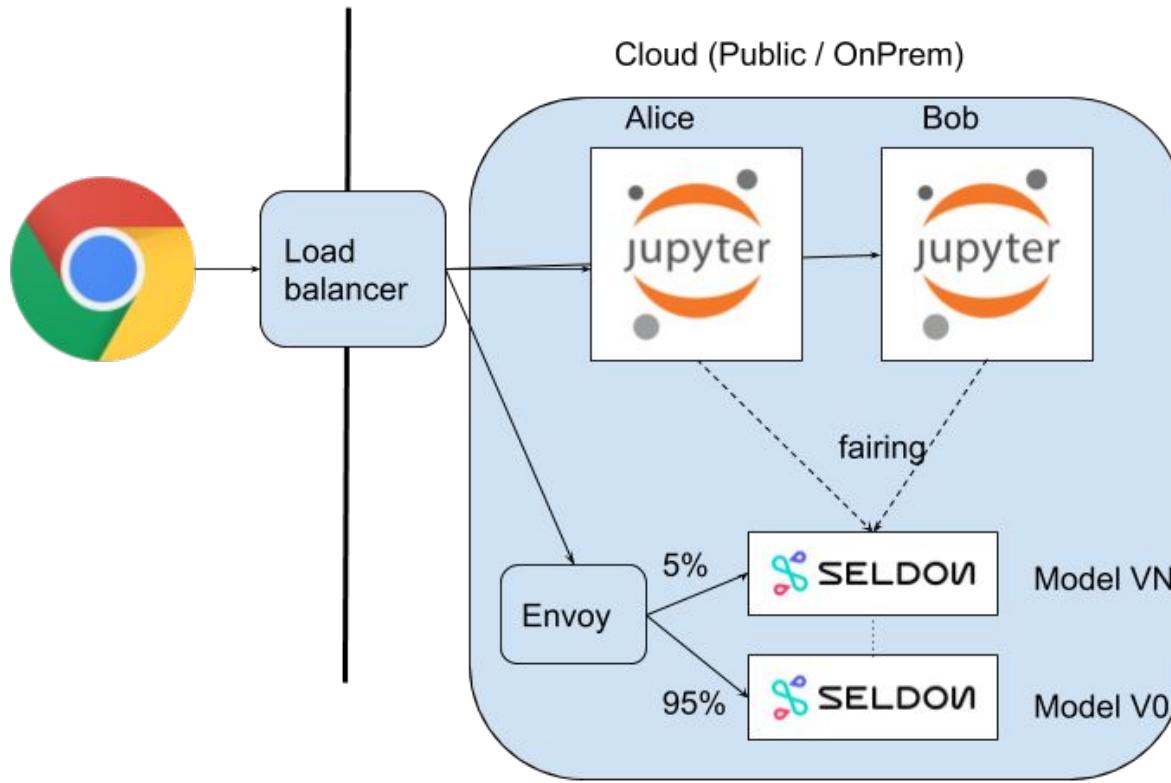
- 1 Connect to data, machines
- 2 Build models
- 3 Train
- 4 Deploy



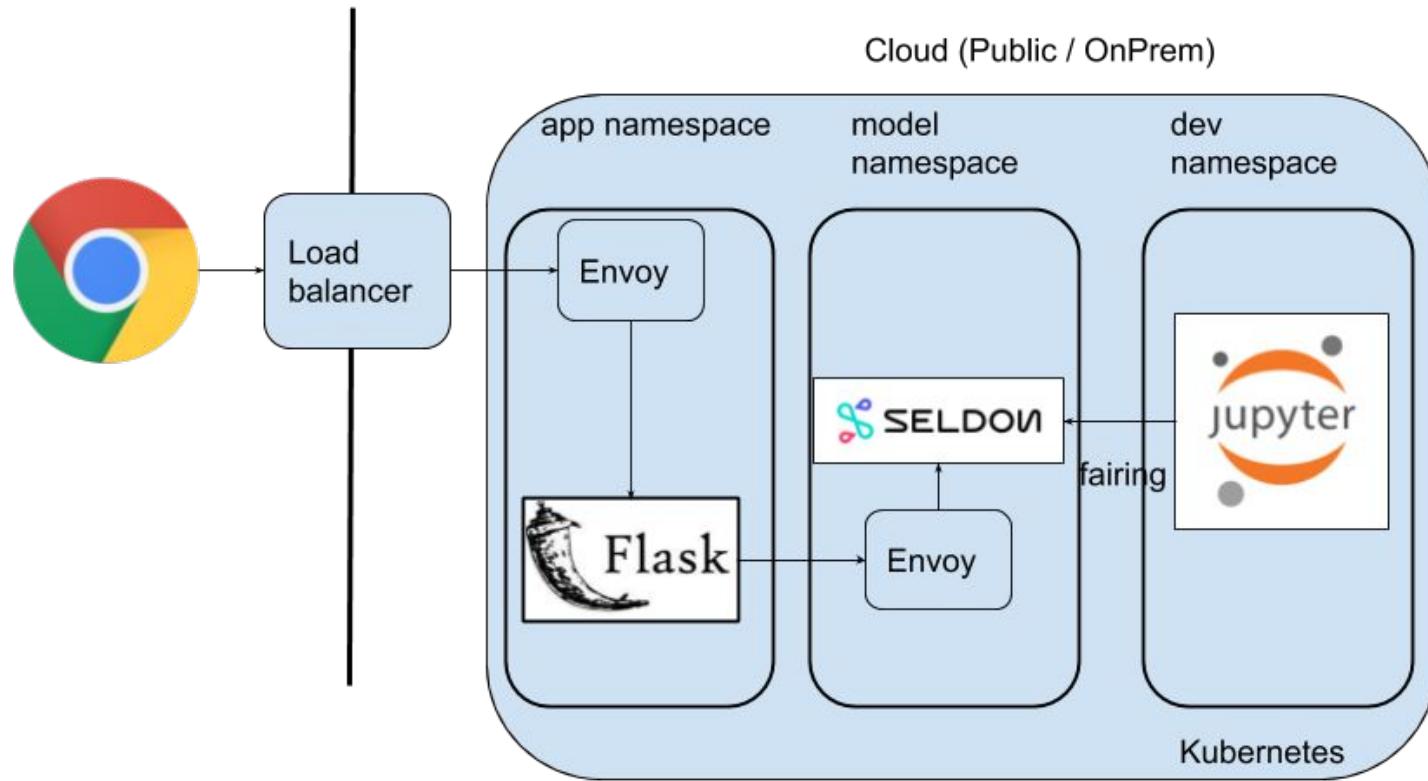
Deploy Model

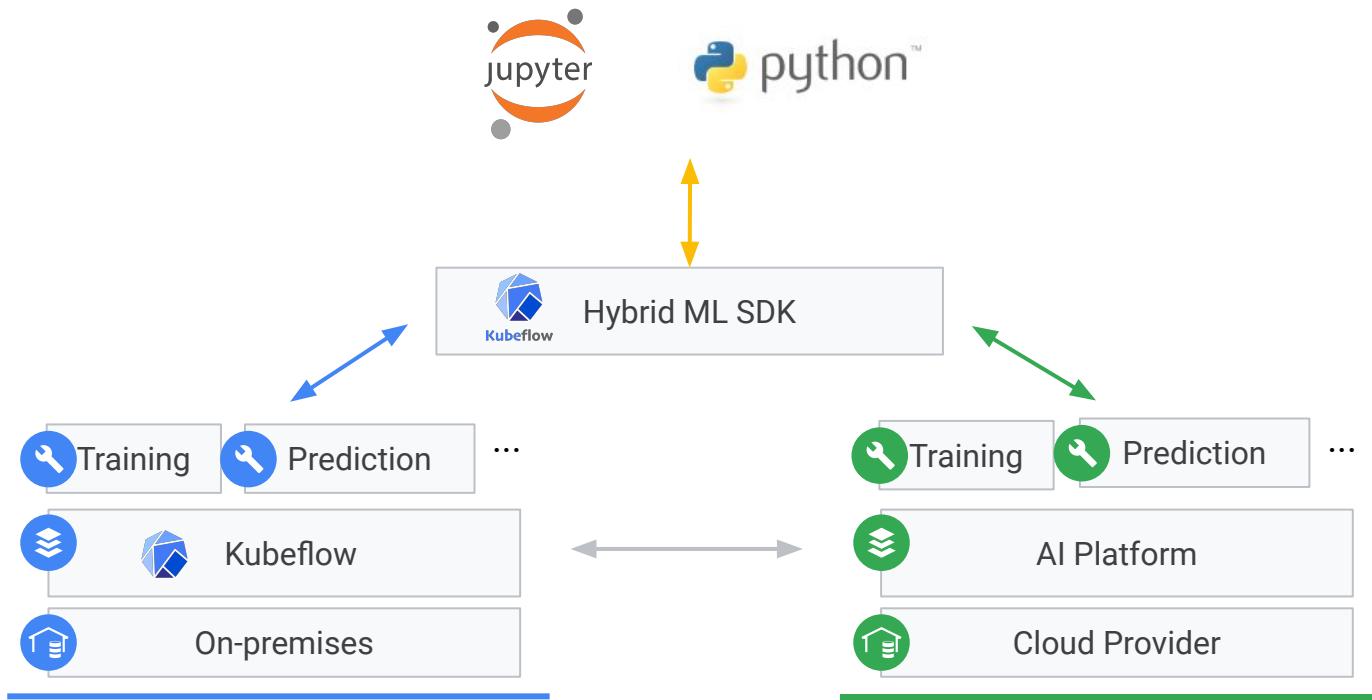


Experimentation by multiple data scientists

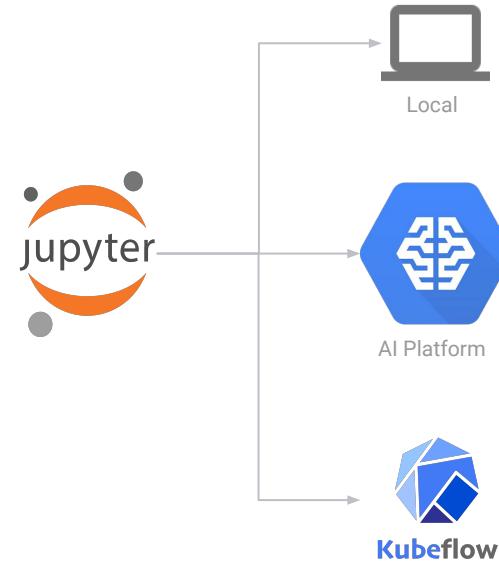


Kubernetes can handle the complete stack





Kubeflow Fairing is an open source Hybrid ML SDK for data scientists to ‘**write ML code once and run anywhere**’.



Code: Today

Local

```

import xgboost

class MyModel(object):
    def train(self):
        # load data
        # do feature engineering
        # train a model

    def predict():
        # prediction logic

if __name__ == '__main__':
    model = MyModel()
    model.train()

```

Build & Deploy to AI Platform

Training

```
gcloud ml-engine jobs submit training my_job \
    --module-name trainer.task \
    --staging-bucket gs://my-bucket \
    --package-path /my/code/path/trainer \
    --packages additional-dep1.tar.gz,dep2.whl
```

Prediction

```
gcloud alpha ml-engine versions create
{VERSION_NAME} --model {MODEL_NAME} \ --origin
gs://{BUCKET}/{MODEL_DIR}/ \ --runtime-version
{RUNTIME_VERSION} \ --package-uris
gs://{BUCKET}/{PACKAGES_DIR}/my_package-0.2.tar.gz \
--model-class=my_model.ModelExample
```

Build & Deploy to Kubeflow

```

apiVersion: kubeflow.org/v1alpha2
kind: TFJob
metadata:
  labels:
    experiment: experiment10
  name: tfjob
  namespace: kubeflow
spec:
  tfReplicaSpecs:
    Ps:
      replicas: 1
      template:
        metadata:
          creationTimestamp: null
        spec:
          containers:
            - args:
              - python
              - - tf_cnn_benchmarks.py
            image:
.
.
```

Code: With Kubeflow Fairing

Local	Build & Deploy to AI Platform	Build & Deploy to Kubeflow
<pre>import xgboost class MyModel(object): def train(self): # load data # train a model def predict(): # prediction logic from fairing import TrainJob from fairing.backends import Backend job = TrainJob(MyModel, backend=Backend("Local", "fairing.config")) job.submit() endpoint = PredictionEndpoint(MyModel, backend=Backend("Local", "fairing.config")) endpoint.create()</pre>	<pre>import xgboost class MyModel(object): def train(self): # load data # train a model def predict(): # prediction logic from fairing import TrainJob from fairing.backends import Backend job = TrainJob(MyModel, backend=Backend("ai_platform", "fairing.config")) job.submit() endpoint = PredictionEndpoint(MyModel, backend=Backend("ai_platform", "fairing.config")) endpoint.create()</pre>	<pre>import xgboost class MyModel(object): def train(self): # load data # train a model def predict(): # prediction logic from fairing import TrainJob from fairing.backends import Backend job = TrainJob(MyModel, backend=Backend("Kubeflow", "fairing.config")) job.submit() endpoint = PredictionEndpoint(MyModel, backend=Backend("Kubeflow", "fairing.config")) endpoint.create()</pre>

Code: With Kubeflow Fairing

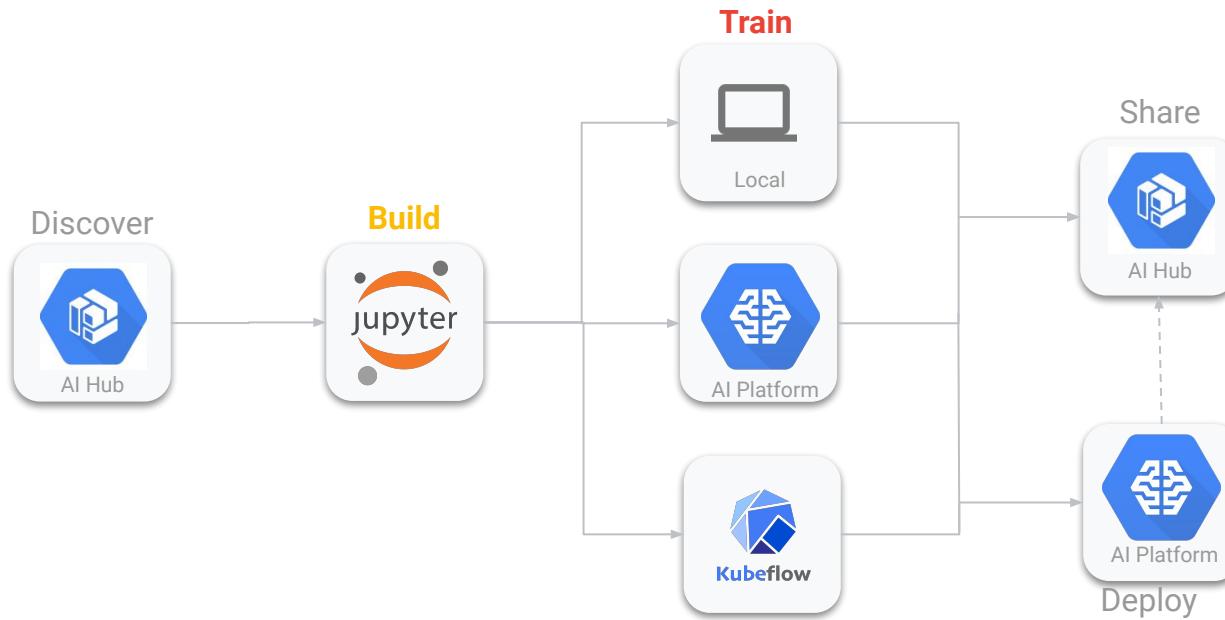
Local	Build & Deploy to AI Platform	Build & Deploy to Kubeflow
<pre>import xgboost class MyModel(object): def train(self): # load data # train a model def predict(): # prediction logic from fairing import TrainJob from fairing.backends import Backend job = TrainJob(MyModel, backend=Backend("Local", "fairing.config")) job.submit()</pre>	<pre>import xgboost class MyModel(object): def train(self): # load data # train a model def predict(): # prediction logic from fairing import TrainJob from fairing.backends import Backend job = TrainJob(MyModel, backend=Backend("ai_platform", "fairing.config")) job.submit()</pre>	<pre>import xgboost class MyModel(object): def train(self): # load data # train a model def predict(): # prediction logic from fairing import TrainJob from fairing.backends import Backend job = TrainJob(MyModel, backend=Backend("Kubeflow", "fairing.config")) job.submit()</pre>
<pre>endpoint = PredictionEndpoint(MyModel, backend=Backend("Local", "fairing.config")) endpoint.create()</pre>	<pre>endpoint = PredictionEndpoint(MyModel, backend=Backend("ai_platform", "fairing.config")) endpoint.create()</pre>	<pre>endpoint = PredictionEndpoint(MyModel, backend=Backend("Kubeflow", "fairing.config")) endpoint.create()</pre>

Kubeflow Fairing

An open source Hybrid ML SDK for data scientists to ‘write ML code once and run anywhere’

- 
- Data Scientist Focused:** Simple and uses language familiar to Data Scientists
 - Multi-Platform:** Supports AI Platform and Kubeflow, making it easy for users to switch between on-prem and GCP.
 - Scalable and Cost Effective:** Data Scientists can easily burst onto GCP when they need more resources (i.e. more machines, GPUs, or TPUs).
 - Easily Train, Tune and Deploy models:** Supports the full ML lifecycle.
 - Multi-Framework:** Supports XGBoost, TensorFlow (single node), and Pytorch (single node).

Demo: Hybrid E2E ML with Kubeflow Fairing



DEMO

The screenshot shows a Jupyter Notebook interface with the title "Cancer detection-fairing-os". The code cell contains Python code for setting up a Fairing job to train a machine learning model. The log output below the code cell shows the build process of a Docker image for the job, including steps for building the Dockerfile, copying files, and running the training script.

```
File Edit View Run Kernel Cell Settings Help
Cancer detection-fairing-os
Code
Kubeflow Training
(*): import fairing

GCP_PROJECT = fairing.cloud.gcp.guess_project_name()
DOCKER_REGISTRY = 'gcr.io/{}/fairing-job'.format(GCP_PROJECT)

train_job = fairing.TrainJobTrain(
    base_docker_image="gcr.io/cailp-dexter-dev/jaas:py3.5.3",
    docker_registry=DOCKER_REGISTRY,
    backend=fairing.backends.KubeFlowGKEBackend())

train_job.submit()

#0485 01:26:52.381327 148428655855368 tasks.py:29] Using preprocessor: <class 'fairing.preprocessors.function.FunctionPreProcessor'>
#0485 01:26:52.382992 148428655855368 tasks.py:32] Using docker registry: gcr.io/cailp-dexter-bugbash/fairing-job
#0485 01:26:52.332953 148428655855368 tasks.py:37] Using builder: <class 'fairing.builders.docker.DockerBuilder'>
#0485 01:26:52.339234 148428655855368 docker.py:191 Docker command: ['python', '/app/function_shim.py', '--serialized_fn_file',
'/app/pickled_fn.p']
#0485 01:26:52.328324 148428655855368 base.py:82] /home/jupyter/.local/lib/python3.5/site-packages/fairing/_init_.py already exists in Fairing context, skipping...
#0485 01:26:52.322819 148428655855368 base.py:82] /home/jupyter/.local/lib/python3.5/site-packages/fairing/_init_.py already exists in Fairing context, skipping...
#0485 01:26:52.333553 148428655855368 docker.py:52] Building docker image gcr.io/cailp-dexter-bugbash/fairing-job:1EEB5D23...
#0485 01:26:54.421213 148428655855368 docker.py:92] Build output: Step 1/6 : FROM gcr.io/cailp-dexter-dev/jaas:py3.5.3
#0485 01:26:54.423281 148428655855368 docker.py:92] Build output:
#0485 01:26:54.424467 148428655855368 docker.py:92] Build output: --> 12fd59e44641
#0485 01:26:54.425575 148428655855368 docker.py:92] Build output: Step 2/6 : WORKDIR /app/
#0485 01:26:54.426418 148428655855368 docker.py:92] Build output:
#0485 01:26:54.427379 148428655855368 docker.py:92] Build output: --> Using cache
#0485 01:26:54.428562 148428655855368 docker.py:92] Build output: --> 48879163de88
#0485 01:26:54.429543 148428655855368 docker.py:92] Build output: Step 3/6 : ENV FAIRING_RUNTIME 1
#0485 01:26:54.430258 148428655855368 docker.py:92] Build output:
#0485 01:26:54.431242 148428655855368 docker.py:92] Build output: --> Using cache
#0485 01:26:54.432182 148428655855368 docker.py:92] Build output: --> 5221999889ba
#0485 01:26:54.432812 148428655855368 docker.py:92] Build output: Step 4/6 : RUN if [ -e requirements.txt ]; then pip install --no-cache -r requirements.txt; fi
#0485 01:26:54.433627 148428655855368 docker.py:92] Build output:
#0485 01:26:54.437318 148428655855368 docker.py:92] Build output: --> Using cache
#0485 01:26:54.438268 148428655855368 docker.py:92] Build output: --> 35c74e1cc599
#0485 01:26:54.439148 148428655855368 docker.py:92] Build output: Step 5/6 : COPY /app/ /app/
#0485 01:26:54.439933 148428655855368 docker.py:92] Build output:
#0485 01:26:54.712542 148428655855368 docker.py:92] Build output: --> e54ce6821a9a
#0485 01:26:54.714255 148428655855368 docker.py:92] Build output: Step 6/6 : CMD python /app/function_shim.py --serialized_fn_file /app/pickled_fn.p
```

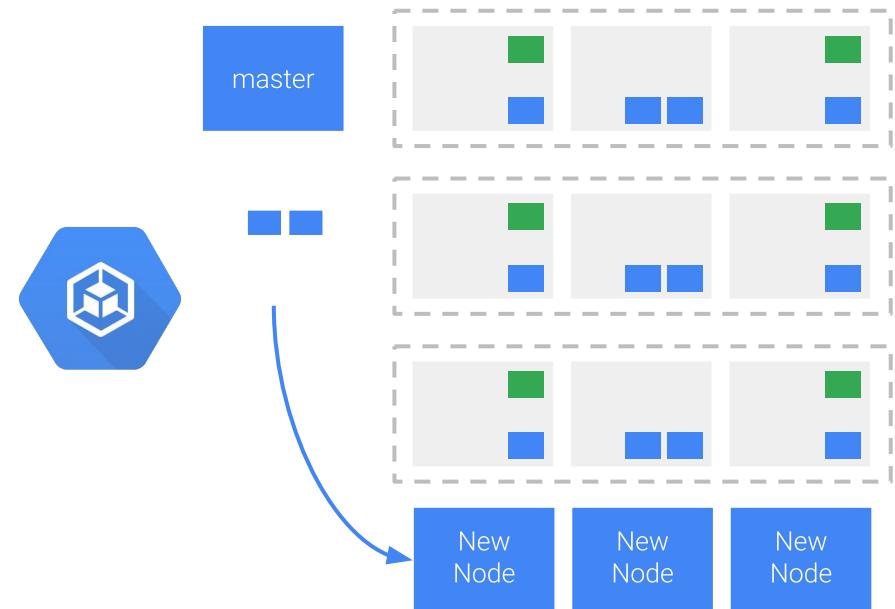


Kubeflow Fairing: Key Benefits for ML Ops Teams

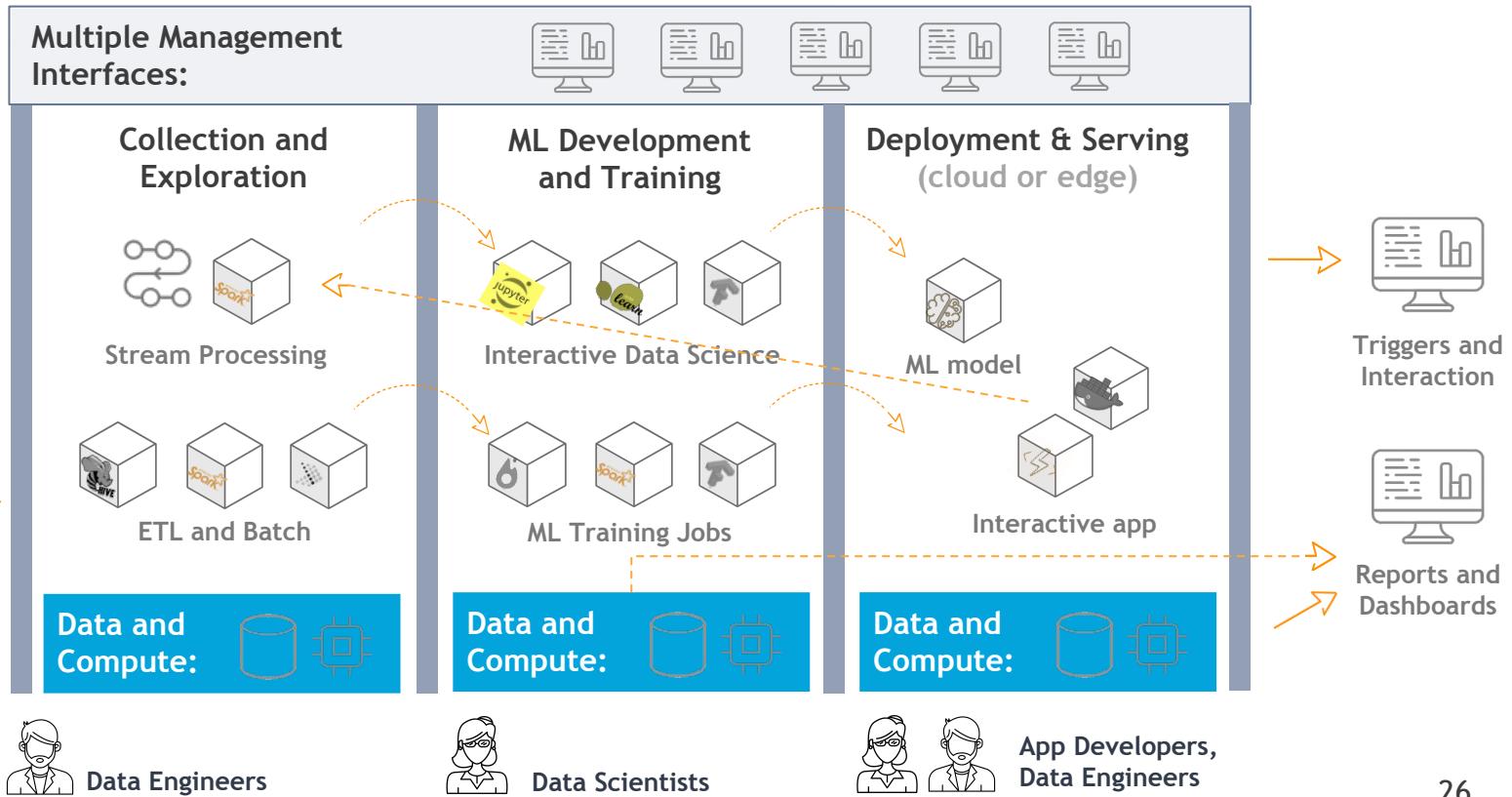
- 1 Standardized API Enforces Best Practices
- 2 Open Source SDK --> No Lock-in
- 3 Easy 'Remoting' & Bursting to the Cloud

Cluster autoscaler with GPUs and TPUs

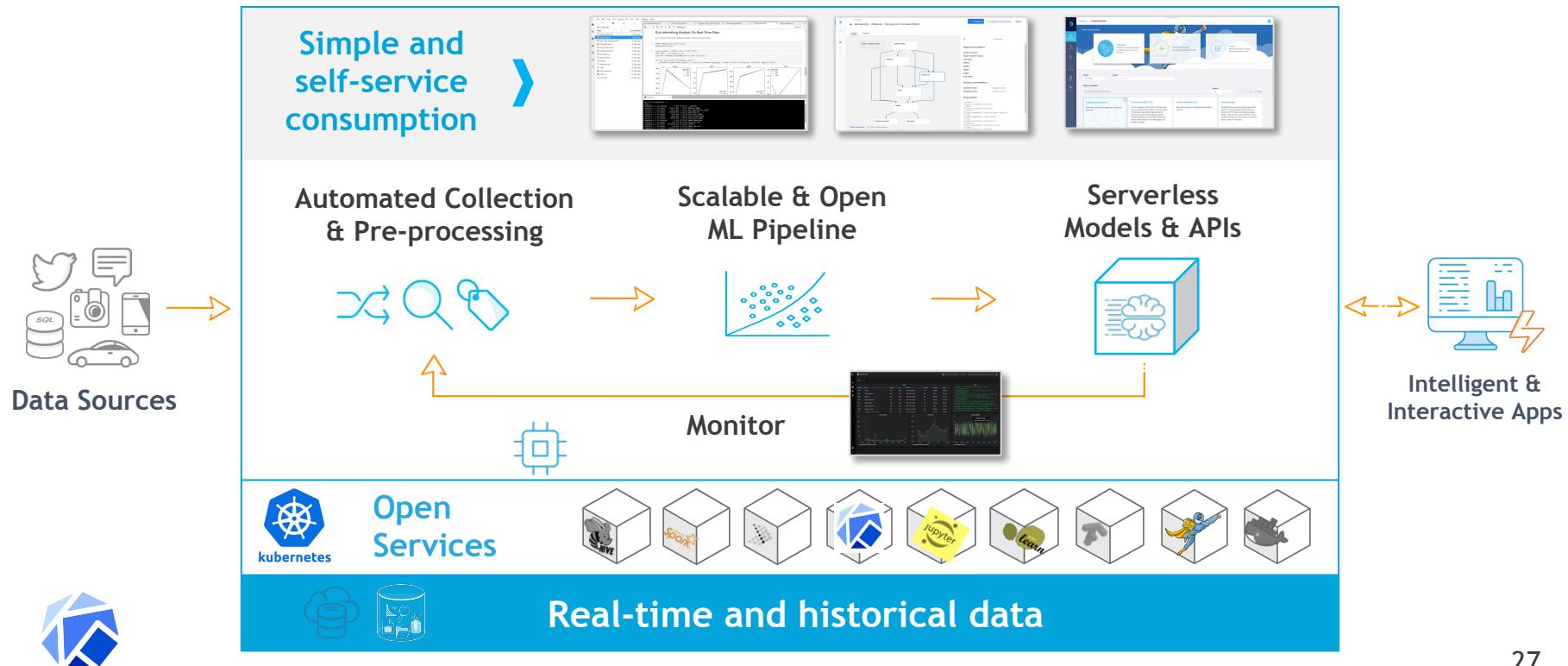
- Automatically **scale up/down** the cluster for the **best performance over cost**
- Nodes with **GPUs/TPUs** get **created** when a cluster needs **more capacity**
- Nodes with **GPUs/TPUs** get **deleted** when they're **idle**



Today: ML Pipeline is Complex and Siloed



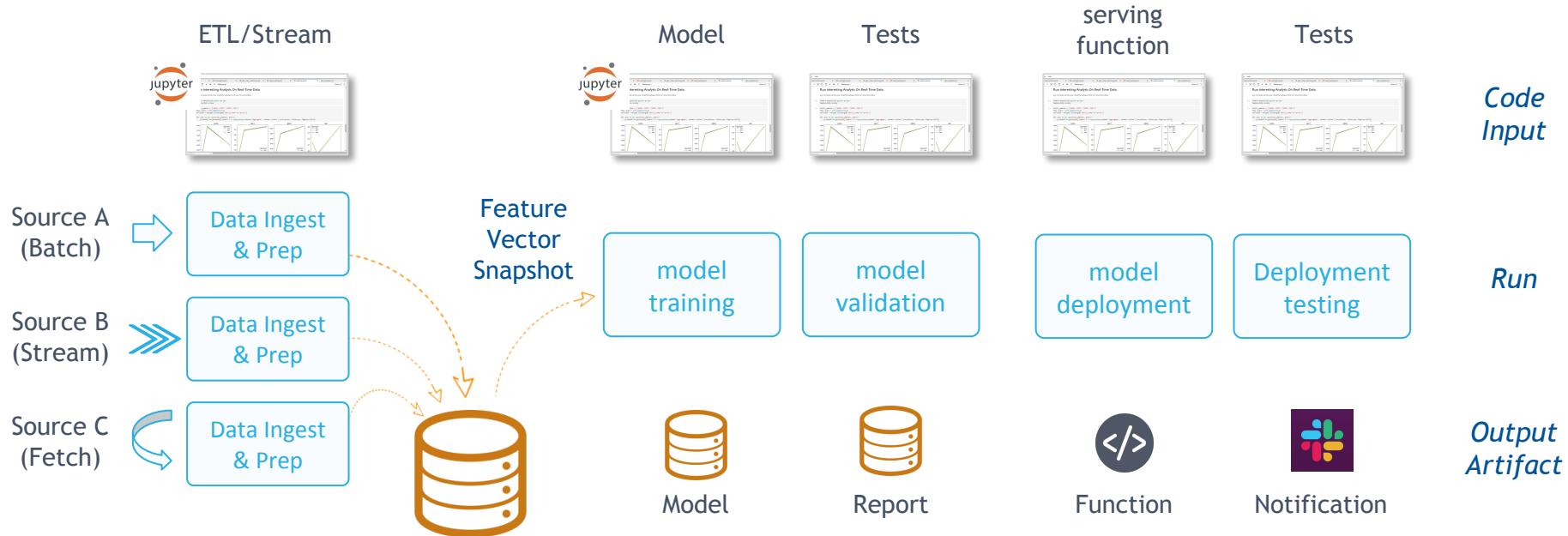
Kubernetes: One Platform, Complete ML Lifecycle



Open-Source ML Pipeline Components By Category



Typical Data Science Pipeline



Pipeline must be automated !



KubeFlow Pipeline

The screenshot displays two main sections of the KubeFlow Pipeline interface.

Top Section: Shows the "Experiments > my_ml_pipeline" page. It features a "Graph" view where a workflow is visualized as a directed acyclic graph (DAG). The steps include "xop", "data-prep", "listdir", "training", "build-model", "test-model", and "nuc". A modal window titled "my-ml-pipeline-5acbf-4105258429" shows a "Confusion matrix" with the following data:

	0	1
0	3610	385
1	604	1910

Bottom Section: Shows the "Experiments > Compare runs" page. It provides a "Run overview" table comparing two runs:

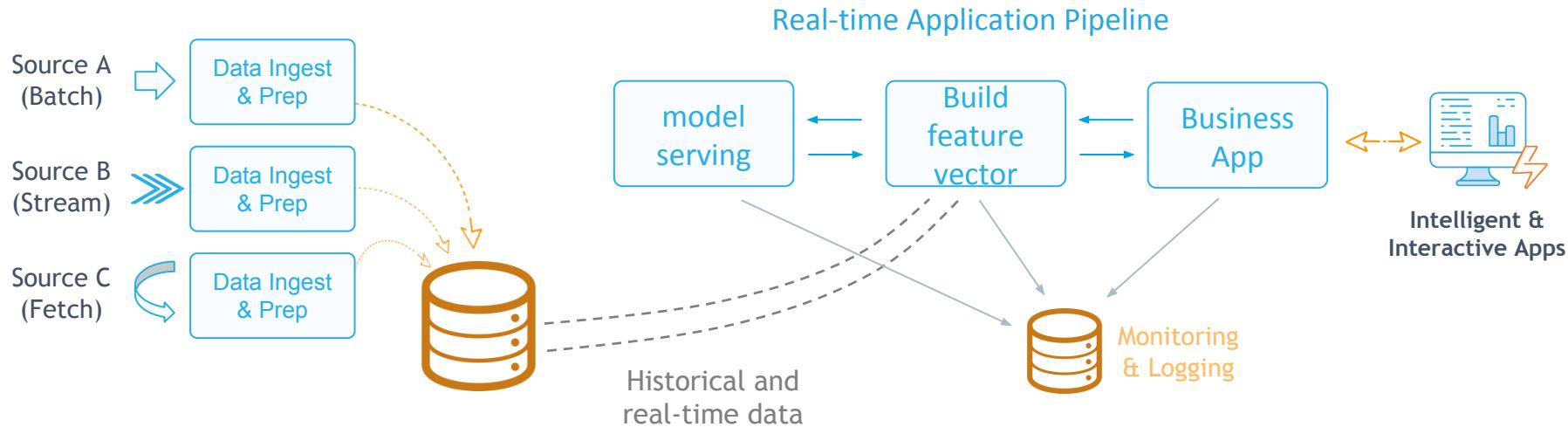
Run name	Status	Duration	Experiment	Pipeline	Start time	accuracy-score	loss
my_pipeline run	Success	0:00:30	My ML pipeline	[View pipeline]	5/1/2019, 10:23:45	7.000	7.000
my_pipeline run	Success	0:00:28	My ML pipeline	[View pipeline]	5/1/2019, 10:23:45	8.000	8.000

Below the table, there are sections for "Parameters" (showing parameters like "txt" with values "good evening" and "good morningf", and "val" with values "7" and "8") and "Markdown" (two separate text boxes labeled "Results" containing "sample results").

- Advanced workflow engine and experiment management in one tool
- Convert python code to workflows
- Reusable component library
- Managing multiple runs, compare artifacts and results between runs
- Steps can be containers, code scripts, CRDs (e.g. TFJob), and now functions



Application Serving Environment, More Challenges



Scale, performance, online updates, monitoring, security...

Serverless A Way To Simplify Data Science

- Automate process from code to container and assigned cluster resources
- Add instrumentation with minimal developer overhead
- Auto scaling, rolling upgrades, ...

Sounds Ideal So Why Not?

- ⌚ Slow performance, lack of concurrency, no GPUs
- ⌚ Stateless, limit application patterns
- ⌚ No stream processing support (mostly HTTP)
- ⌚ Hard to debug and diagnose and build dependencies



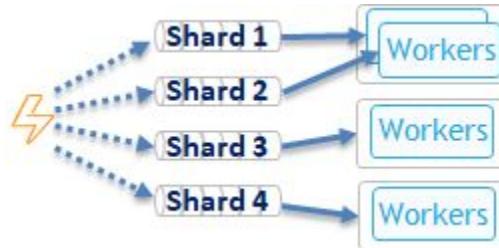
Nuclio: Taking Serverless to Data Intensive Apps

Extreme Performance



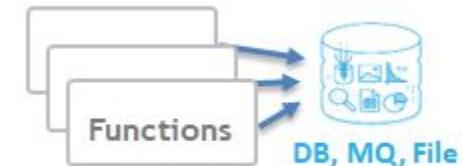
- Non-blocking, parallel
- Zero copy, buffer reuse
- Up to 400K events/sec/proc
- GPU optimizations

Advanced Data & AI Features



- Auto-rebalance, checkpoints
- Any source: Kafka, NATS, Kinesis, event-hub, iguazio, pub/sub, RabbitMQ, Cron, ..
- Jupyter, NVIDIA Rapids integration

Statefulness



- Data bindings
- Shared volumes
- Context cache



Nuclio Automating & Accelerating Data Science

```
stage is not defined use @nuclio.function
vehicle
import os
import json
def handle(event, context):
    print("Received event: %s" % event)
    print("Vehicle ID: %s" % event['vehicle_id'])
    print("Latitude: %s" % event['lat'])
    print("Longitude: %s" % event['lon'])
    print("Speed: %s" % event['speed'])
    print("Timestamp: %s" % event['timestamp'])

    # Process vehicle data
    vehicle = Vehicle()
    vehicle.vehicle_id = event['vehicle_id']
    vehicle.lat = event['lat']
    vehicle.lon = event['lon']
    vehicle.speed = event['speed']
    vehicle.timestamp = event['timestamp']

    # Store vehicle data in database
    vehicle.save()

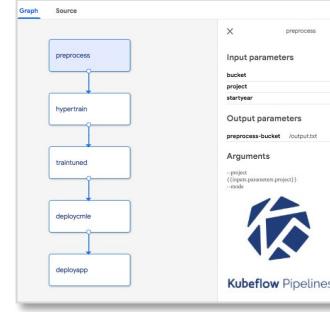
    # Calculate route
    route = calculate_route(vehicle)

    # Create JSON response
    response = {
        "route": route,
        "vehicle_id": vehicle.vehicle_id,
        "lat": vehicle.lat,
        "lon": vehicle.lon,
        "speed": vehicle.speed,
        "timestamp": vehicle.timestamp
    }

    # Return response
    return response
```

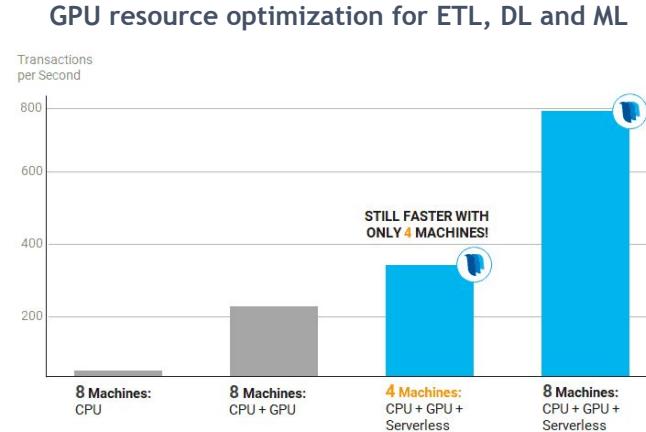


One magic command from notebook to function



Extending Pipelines from batch:

1. Parallel processing steps
2. Code build/deployment steps
3. Stream processing



Automation:

1. Auto-scaling (to zero)
2. Automated logging & monitoring
3. Security hardening
4. Auto-build and CI/CD
5. Workload mobility (cloud/edge/..)



Demo: Building an end to end ML pipeline in minutes with KubeFlow

```
[1] # nucio: ignore
# If the nucio-jupyter package is not installed run: pip install nucio-jupyter
# Install dependencies
[2] Nucioles and pip install tensorflow
Nucioles exec TO_LAMBDA -- python3 -c "import tensorflow; print(tensorflow.__version__)"
Requirement already satisfied: tensorflow in ./python3/lib/python3.6/site-packages (0.15.1)
Requirement already satisfied: numpy>=1.16.0 in ./python3/lib/python3.6/site-packages (from tensorflow) (1.16.0)
Requirement already satisfied: six>=1.11.0 in ./python3/lib/python3.6/site-packages (from tensorflow) (1.11.0)
Requirement already satisfied: singledispatch<3.4.0 in ./python3/lib/python3.6/site-packages (from nucio) (3.4.0.2)
Requirement already satisfied: tensorflow<2.0.0 in ./python3/lib/python3.6/site-packages (from nucio)
Nucio: build step setting spec.buildImage = tensorflow:1.15.0
[3] From tensorflow import TextLine
import tensorflow as tf
def handle(self, event):
    context.logger.info("This is an NLP example!")
    # process and correct the text
    text = event['text']
    corrected_text = self._correct(text, "utf-8")
    context.logger.info(f"corrected: {corrected_text}")
    # debug print the text before and after correction
    context.logger.info(f"before: {text}")
    context.logger.info(f"after: {corrected_text}")
    # indicate sentence
    context.logger.info(f"sentiment: {context.sentiment.polarity}, subjectivity: {context.sentiment.subjectivity}")
    # root target (comes from environment and return translated text
    lang = os.getenv("TO_LANG", "fr")
    event['text'] = self._translate(corrected_text, lang)
[4] # nucio: ignore
event = nucio.event("hey ya! good morning")
# translate to french
Python 2019-05-14 14:19:16,977 [Info]: This is an NLP example!
Python 2019-05-14 14:19:16,979 [Info]: Corrected text: 'good morning', 'via': 'good morning'
Python 2019-05-14 14:19:16,979 [Info]: Sentiment: {'polarity': '-0.1', 'subjectivity': '0.0000000000000001'}
[5] "Good"
[6] Nucioles deploy -n nlp -p nlp -c
Nucioles ['deploy', '-n', 'nlp', '-p', 'nlp', '--', './nlp-example.pytpe']
Nucioles: [nucio.deploy] 2019-05-14 14:19:17,000 [Info]: building processor Image
Nucioles: [nucio.deploy] 2019-05-14 14:19:17,000 [Info]: building artifacts
Nucioles: [nucio.deploy] 2019-05-14 14:19:17,000 [Info]: build complete
Nucioles: [nucio.deploy] 2019-05-14 14:19:17,000 [Info]: build complete
Nucioles: [nucio.deploy] 2019-05-14 14:19:17,000 [Info]: done updating nlp, function address: 3:121,204,269,32182
Nucioles: [nucio.deploy] 2019-05-14 14:19:17,000 [Info]: done updating nlp, function address: 3:121,204,269,32182
[7] curl -X POST -H "Content-type: application/json" http://127.0.0.1:8080/nlp/translate
Content-type: application/json
Content-type: application/json
```

The screenshot shows the KubeFlow interface for an experiment named "my_ml_pipeline". The left sidebar has tabs for Pipelines, Experiments (selected), and Archive. The main area displays the execution graph and a confusion matrix.

Execution Graph:

```
graph TD
    XOP[XOP] --> data-prep
    data-prep --> training
    listdir[listdir] --> training
    training --> build-model
    build-model --> test-model
    build-model --> nuc
```

Confusion matrix:

	0	1
0	3610	385
1	604	1910

Empowering your teams to drive innovation

Simple

- Data Scientist friendly notebooks
- Freedom from managing infrastructure
- TFJob, TFServing, ...

Fast

- On-demand scale up and down
- GPUs and TPUs

Cost-effective

- Making AI/ML teams more productive
- Avoid vendor lock-in with open platform
- Write once run anywhere
- Preemptible GPUs/TPUs