



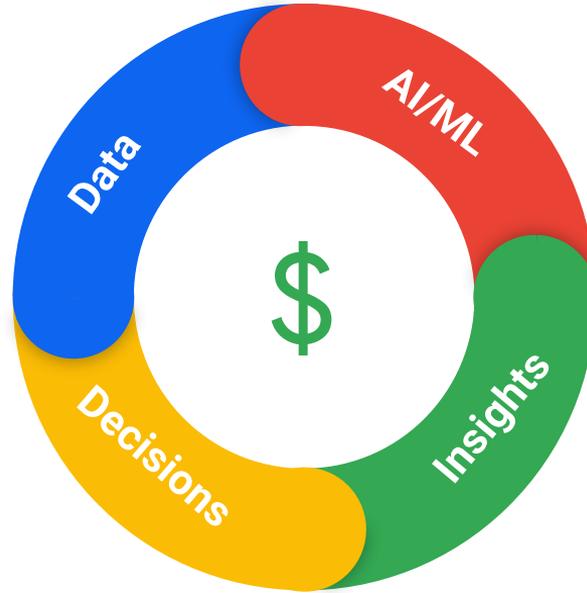
# Economics and best practices of running AI/ML workloads on Kubernetes

**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



# 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

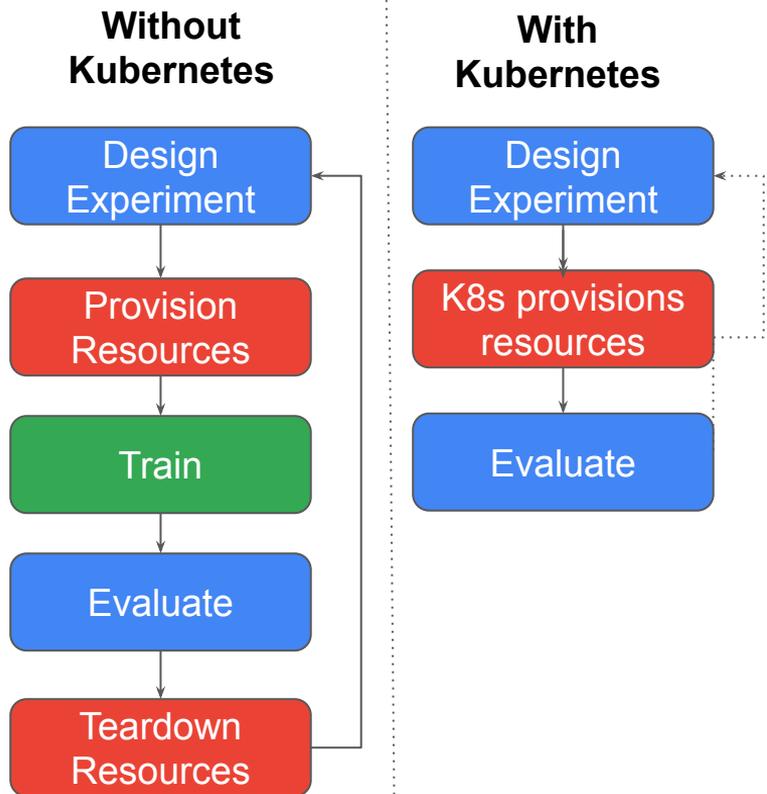


Hardware Accelerators



# 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

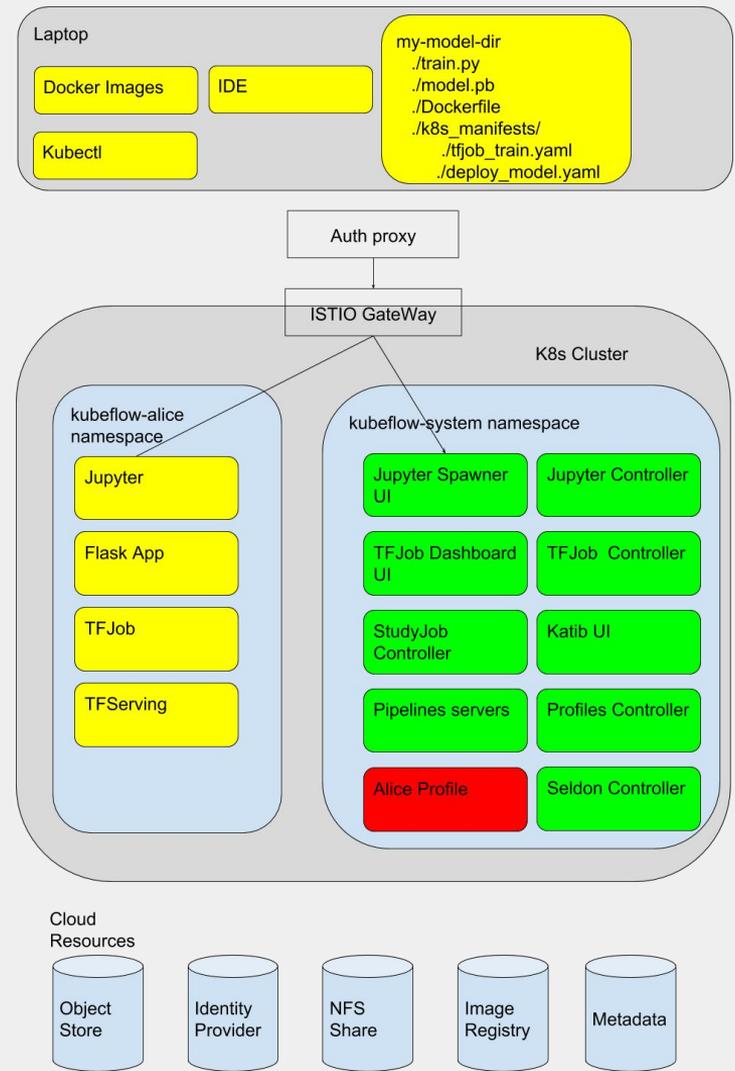


# Kubeflow



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

<https://kubeflow.org>



# 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:
```

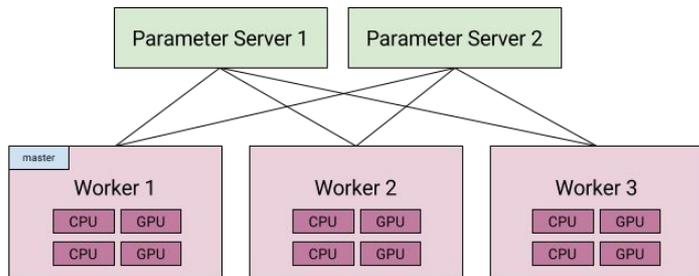
```
      replicas: 3
```

```
      template:
```

```
        spec:
```

```
          containers:
```

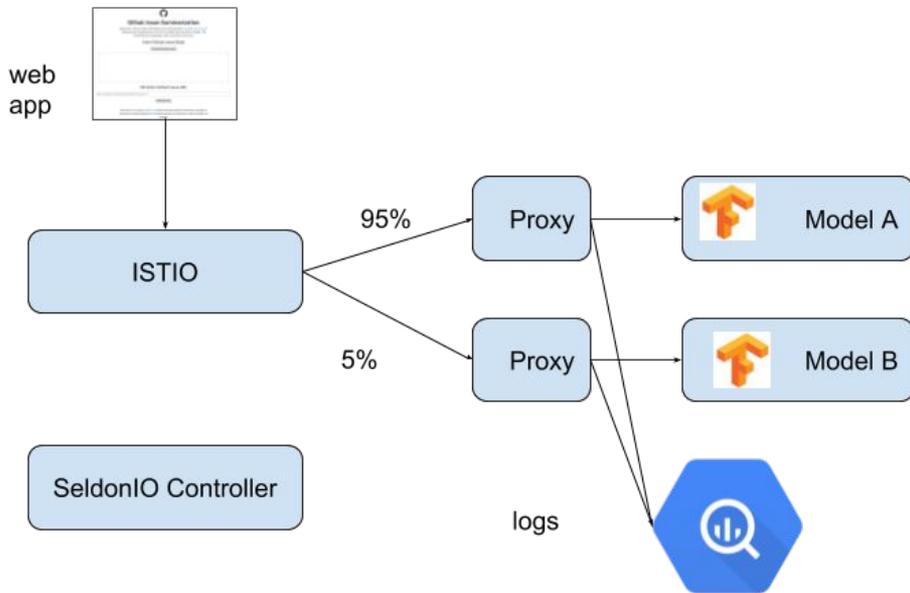
```
            - image: acme/myjob
```



# TensorFlow serving

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

model push  $\neq$  binary push



# Get started right

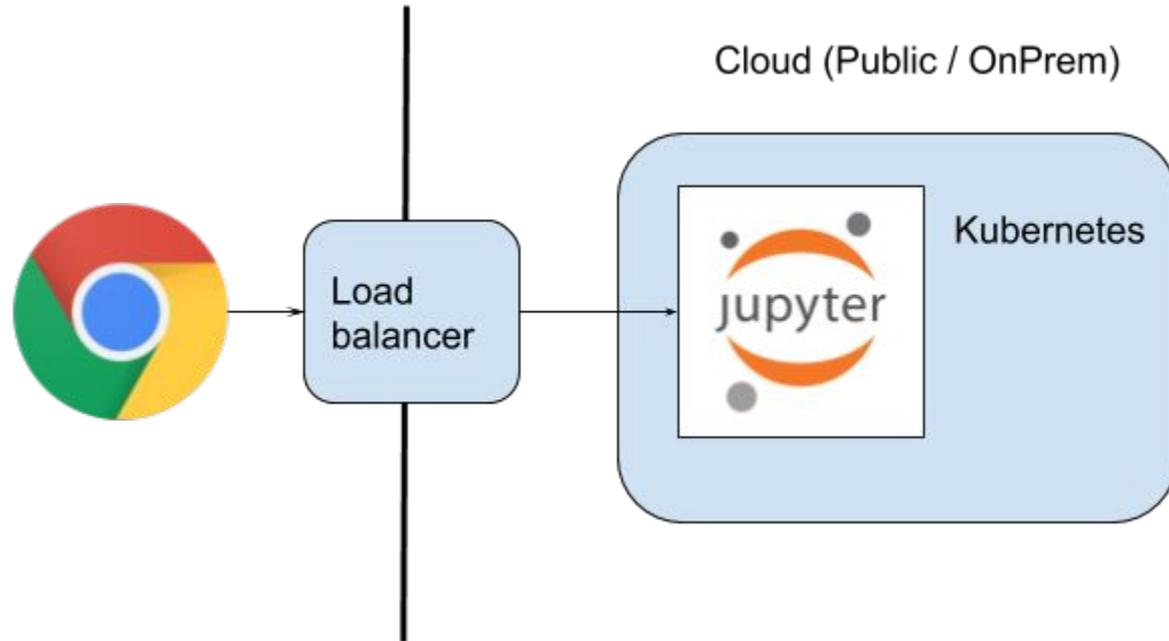
- Day 0 start with the infrastructure (Notebook, Kubernetes, ISTIO, etc...)
- Day 0 focus on model development
  - Use UIs to launch notebooks
  - Python SDK (fairing) for training / deploying models
- Day N leverage K8s to scale
  - Use the same infrastructure as non-ML applications
  - Build a single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



The image shows two screenshots of the Kubeflow interface. The top screenshot displays the 'Notebook Servers' page for the 'kubeflow' namespace. It features a table with columns for Status, Name, Created, Image, CPU, Memory, Volumes, and Actions. A single server named 'jlewi' is listed with a green checkmark status, created 1 day ago, using the 'tensorflow-1.13.1-notebook-cpu' image, with 32 CPU and 32Gi Memory. A '+ NEW SERVER' button is visible in the top right.

The bottom screenshot shows the 'New Notebook Server' configuration form. It includes fields for Name, Namespace, Image (with a dropdown menu), CPU, Memory, Workspace Volume, and Data Volumes. Each section has a brief description and an example configuration. At the bottom, there are 'SHOW' and 'CANCEL' buttons.

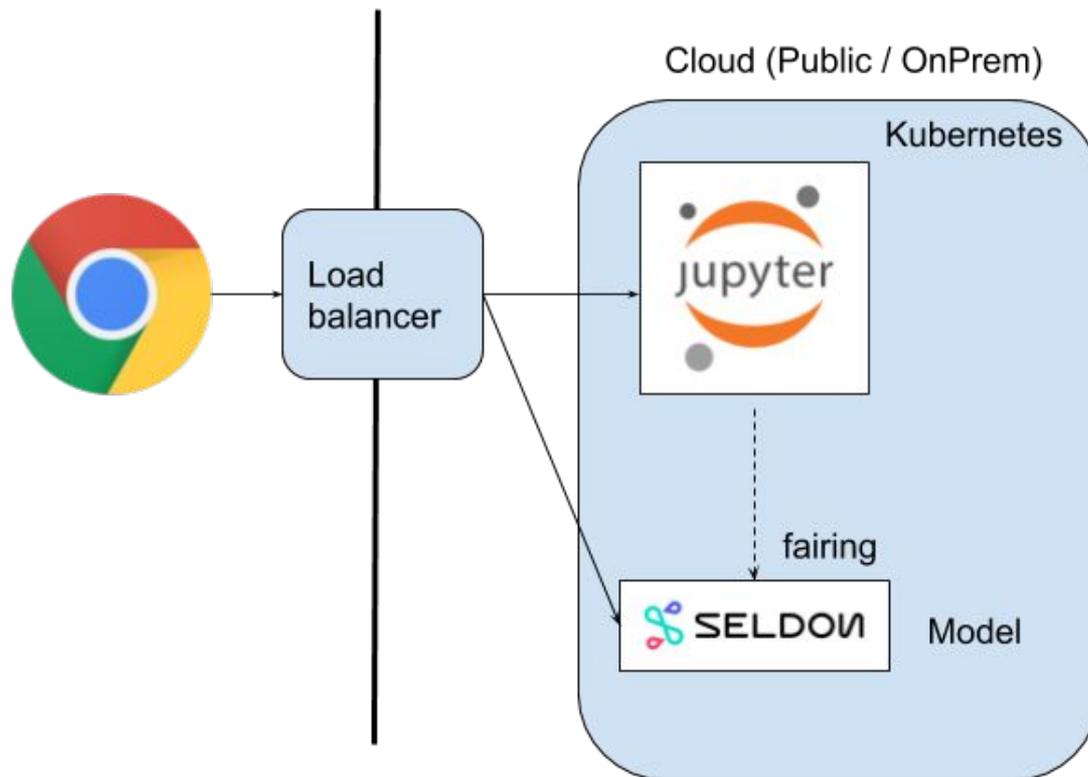
# Day 0: Data scientist friendly Notebooks



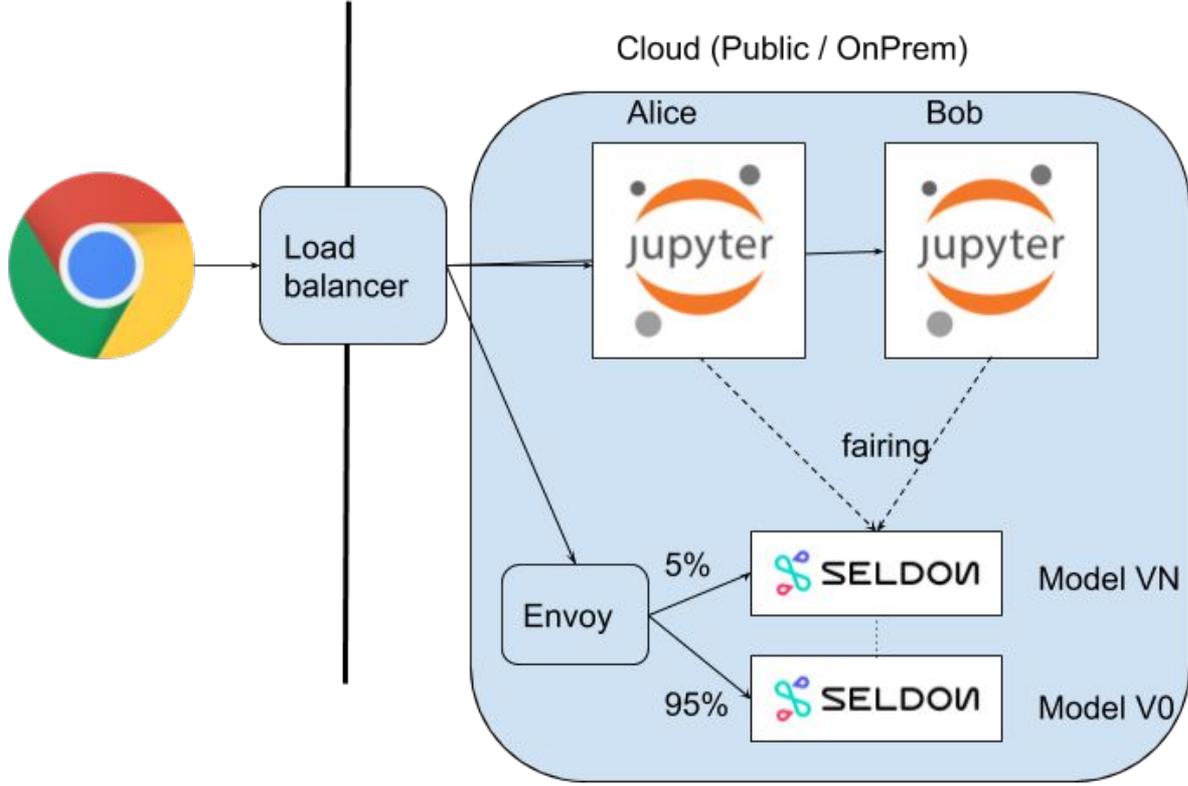
- 1 Connect to data, machines
- 2 Build models
- 3 Train
- 4 Deploy



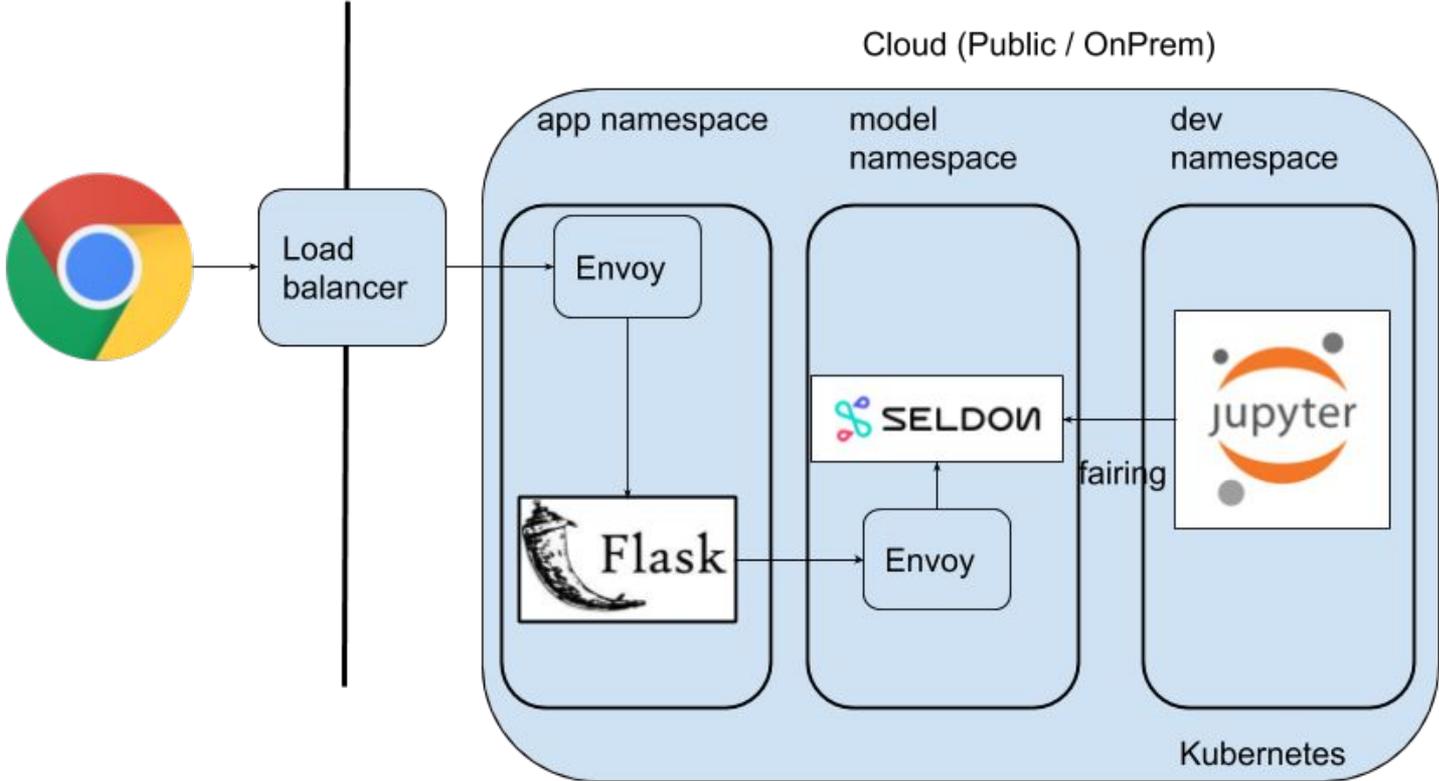
# Deploy Model

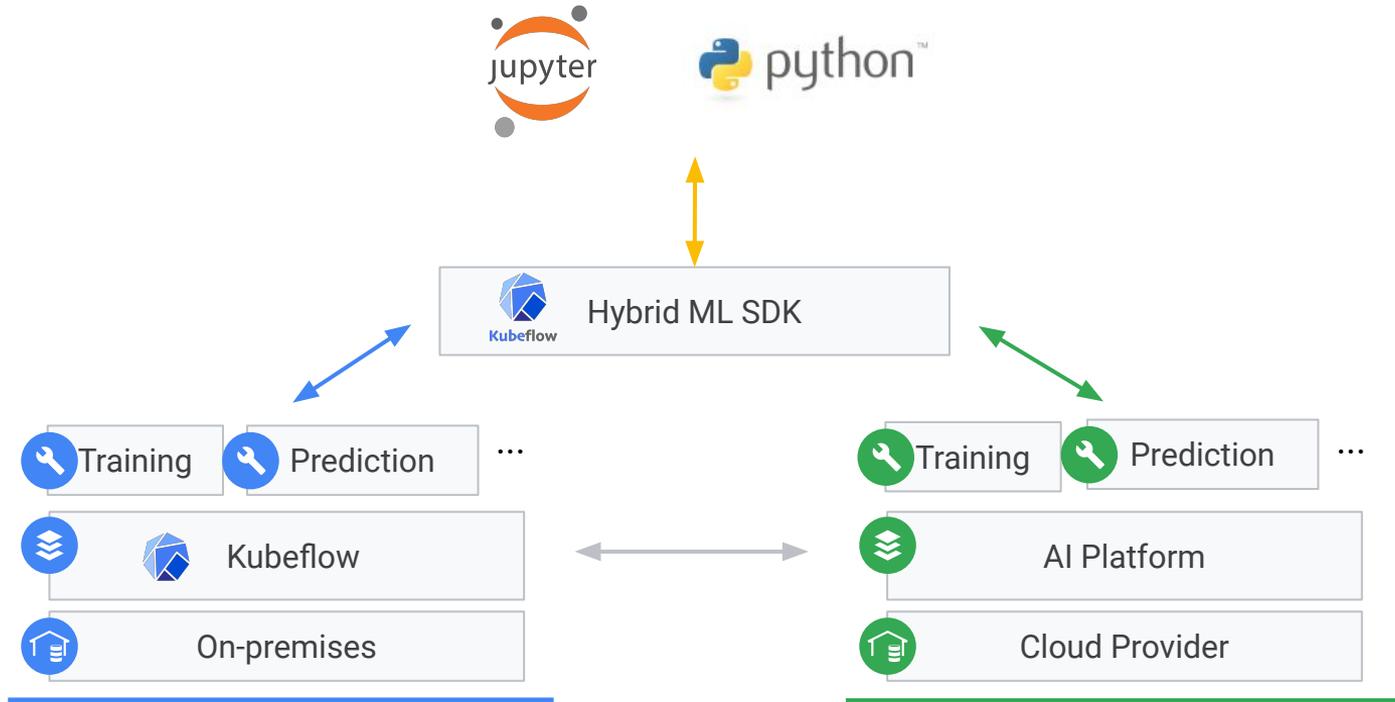


# Experimentation by multiple data scientists

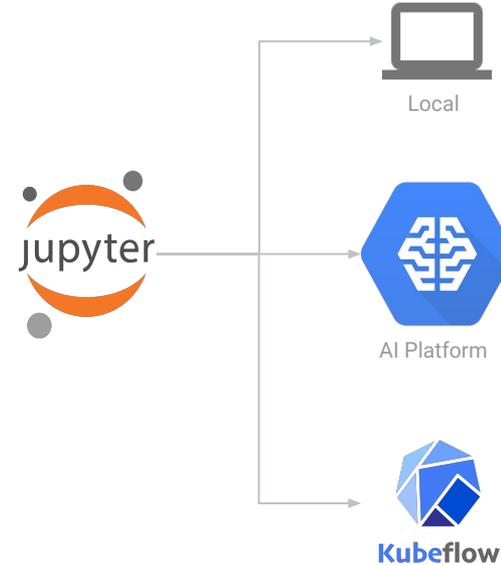


# Kubernetes can handle the complete stack





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

<b>Local</b>	<b>Build &amp; Deploy to AI Platform</b>	<b>Build &amp; Deploy to Kubeflow</b>
<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,</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,</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,</pre>
<pre>    backend=Backend("Local",                     "fairing.config"))</pre>	<pre>    backend=Backend("ai_platform",                     "fairing.config"))</pre>	<pre>    backend=Backend("Kubeflow",                     "fairing.config"))</pre>
<pre>job.submit()  endpoint = PredictionEndpoint(MyModel,                               backend=Backend("Local",   "fairing.config")) endpoint.create()</pre>	<pre>job.submit()  endpoint = PredictionEndpoint(MyModel,                               backend=Backend("ai_platform",   "fairing.config")) endpoint.create()</pre>	<pre>job.submit()  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 Kubleflow, 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).

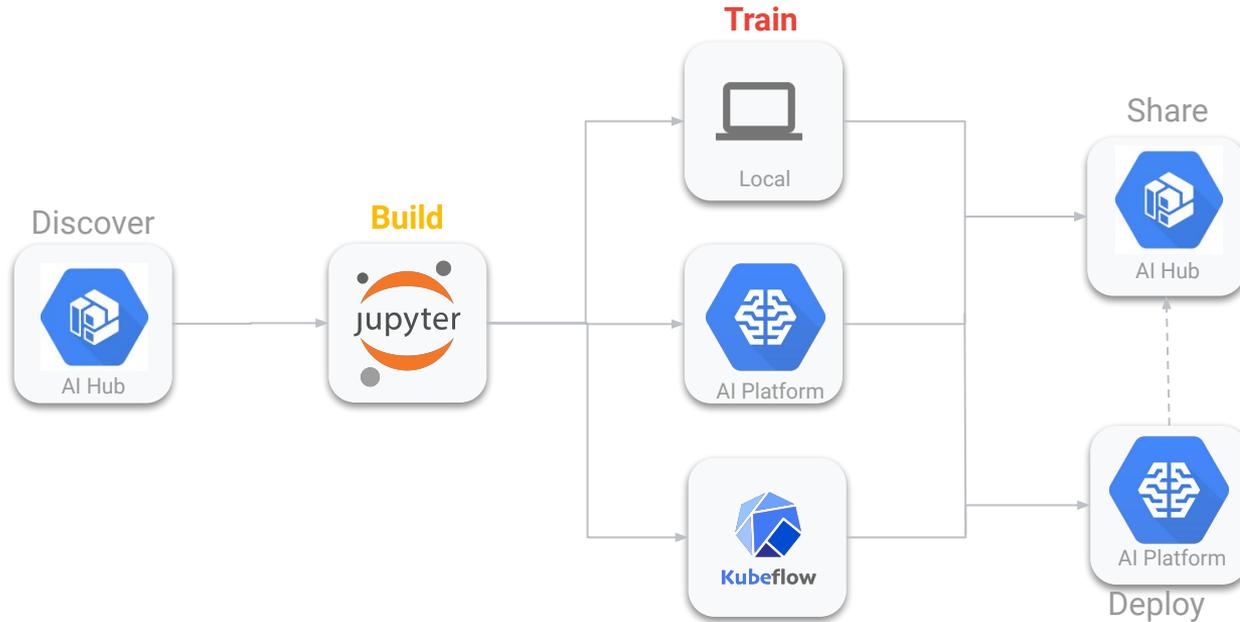


**Easily Train, Tune and Deploy models:** Supports the full ML lifecycle.



**Multi-Framework:** Supports XGBoost, TensorFlow (single node), and Pytorch (single node).

# Demo: Hybrid E2E ML with Kubeflow Fairing



# DEMO

```
File Edit View Run Kernel Git Tabs Settings Help
Cancer detection-fairing-os
Code Python 3

Kubeflow Training

[+] Import fairing

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

train_job = fairing.TrainJob(train,
                             base_docker_image='gcr.io/caip-dexter-dev/jaas:py3.5.3',
                             docker_registry=DOCKER_REGISTRY,
                             backend=fairing.backends.KubeflowK8SBackend())

train_job.submit()

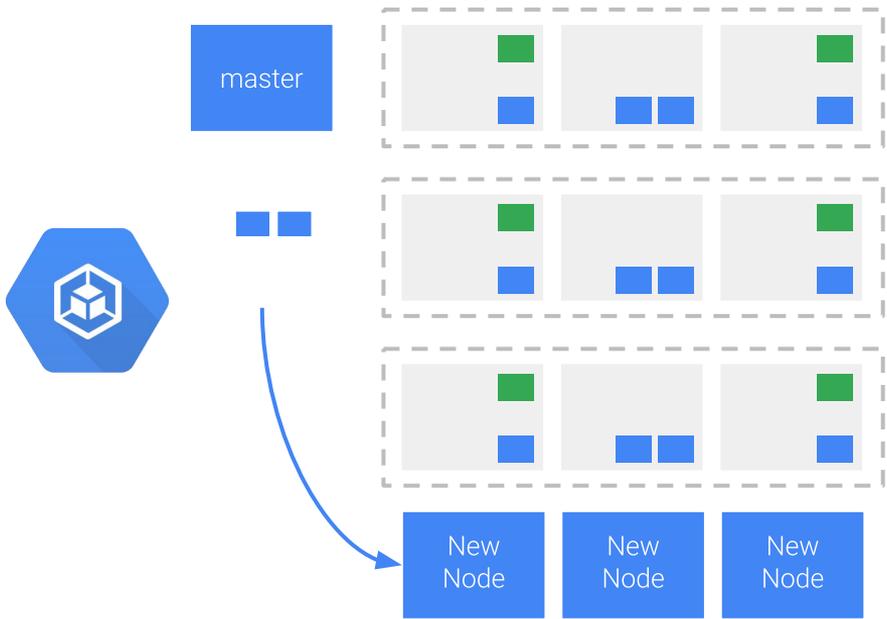
W0485 01:26:52.381327 140428655855360 tasks.py:291 Using preprocessor: <class 'fairing.preprocessors.function.FunctionPreProcessor'>
W0485 01:26:52.382992 140428655855360 tasks.py:321 Using docker registry: gcr.io/caip-dexter-dev/bugbash/fairing-job
W0485 01:26:52.333953 140428655855360 tasks.py:371 Using builder: <class 'fairing.builders.docker.docker.DockerBuilder'>
W0485 01:26:52.339234 140428655855360 docker.py:391 Docker command: ['python', '/app/function_shim.py', '--serialized_fn_file',
'/app/pickled_fn.p']
W0485 01:26:52.320324 140428655855360 base.py:82] /home/jupyter/.local/lib/python3.5/site-packages/fairing/__init__.py already exists in Fairing context, skipping...
W0485 01:26:52.322819 140428655855360 base.py:82] /home/jupyter/.local/lib/python3.5/site-packages/fairing/__init__.py already exists in Fairing context, skipping...
W0485 01:26:52.333553 140428655855360 docker.py:521 Building docker image gcr.io/caip-dexter-dev/bugbash/fairing-job/fairing-job:FE85523...
I0485 01:26:54.421213 140428655855360 docker.py:921 Build output: Step 1/6 : FROM gcr.io/caip-dexter-dev/jaas:py3.5.3
I0485 01:26:54.423281 140428655855360 docker.py:921 Build output:
I0485 01:26:54.424487 140428655855360 docker.py:921 Build output: ---- 12f859e44641
I0485 01:26:54.425575 140428655855360 docker.py:921 Build output: Step 2/6 : WORKDIR /app/
I0485 01:26:54.426416 140428655855360 docker.py:921 Build output:
I0485 01:26:54.427379 140428655855360 docker.py:921 Build output: ---- Using cache
I0485 01:26:54.428562 140428655855360 docker.py:921 Build output: ---- 48879163de88
I0485 01:26:54.429543 140428655855360 docker.py:921 Build output: Step 3/6 : ENV FAIRING_RUNTIME 1
I0485 01:26:54.430258 140428655855360 docker.py:921 Build output:
I0485 01:26:54.431242 140428655855360 docker.py:921 Build output: ---- Using cache
I0485 01:26:54.432182 140428655855360 docker.py:921 Build output: ---- 5221099883ba
I0485 01:26:54.432812 140428655855360 docker.py:921 Build output: Step 4/6 : RUN if [ -e requirements.txt ];then pip install --no-cache --r requirements.txt; fi
I0485 01:26:54.433627 140428655855360 docker.py:921 Build output:
I0485 01:26:54.437316 140428655855360 docker.py:921 Build output: ---- Using cache
I0485 01:26:54.438269 140428655855360 docker.py:921 Build output: ---- 35c74e1cc599
I0485 01:26:54.439148 140428655855360 docker.py:921 Build output: Step 5/6 : COPY /app/ /app/
I0485 01:26:54.439933 140428655855360 docker.py:921 Build output:
I0485 01:26:54.732542 140428655855360 docker.py:921 Build output: ---- e54ce6b21a9a
I0485 01:26:54.734255 140428655855360 docker.py:921 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

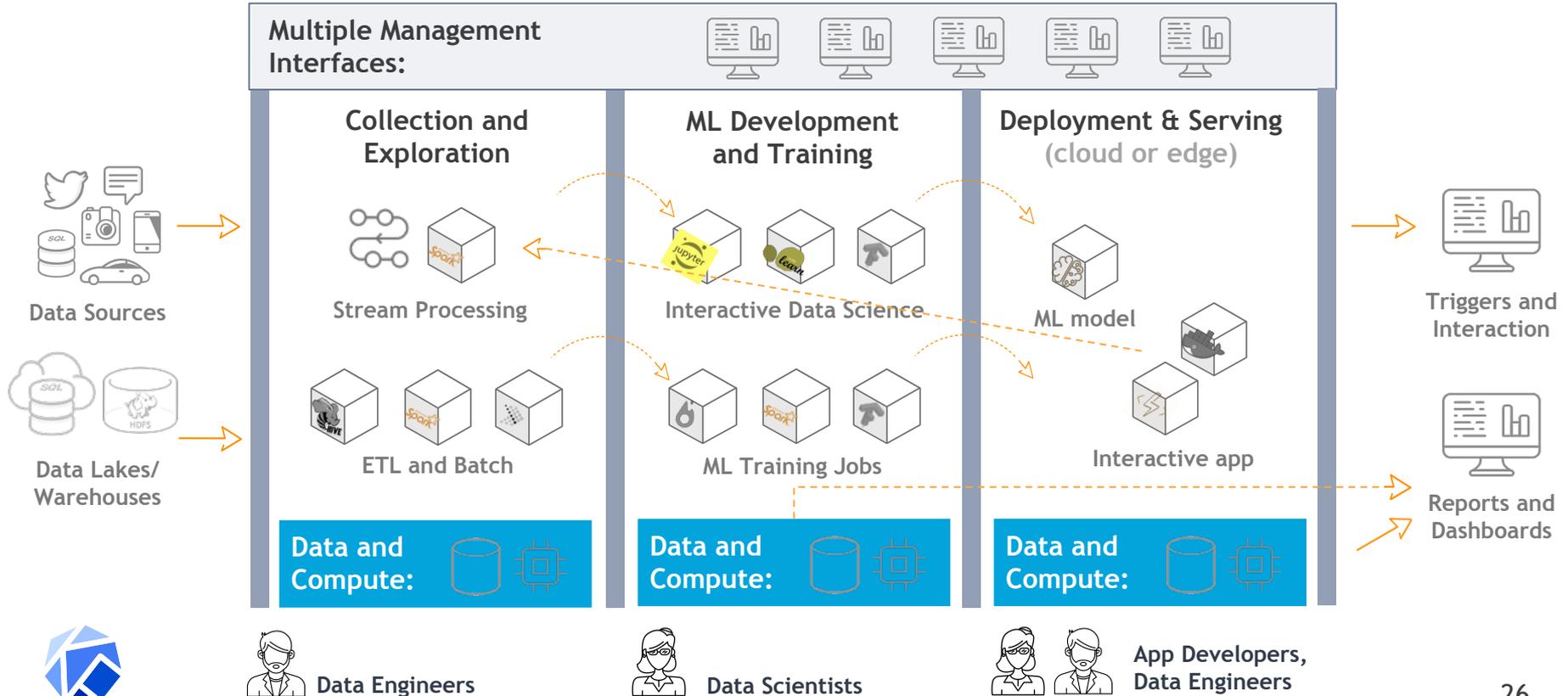
- 1 Standardized API Enforces Best Practices
- 2 Open Source SDK --> No Lock-in
- 3 Easy 'Remoting' & Bursting to the Cloud

# Cluster autoscaler with GPUs and TPUs

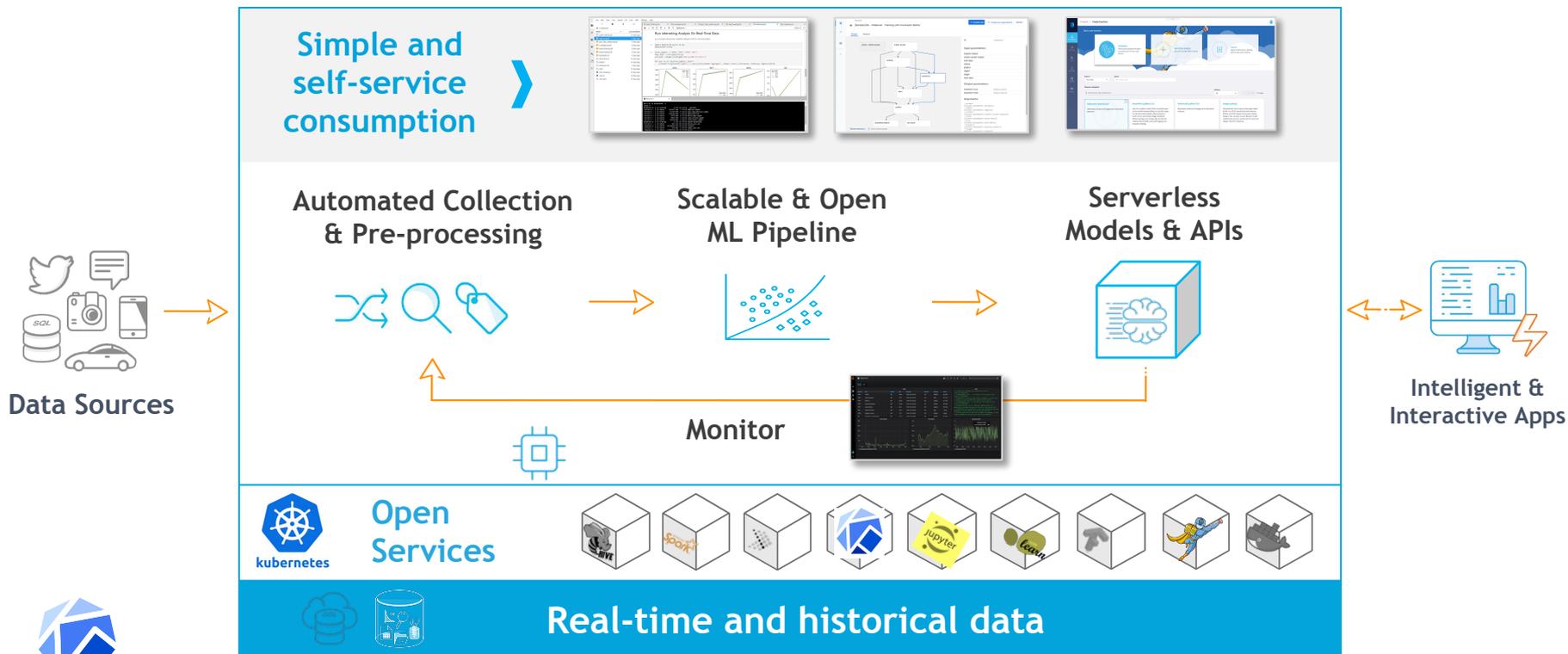
- Automatically **scale up/down** the cluster for the **best performance over cost**
- Nodes with **GPUs/TPUs** get **created** when a cluster needs **more capacity**
- Nodes with **GPUs/TPUs** get **deleted** when they're **idle**



# Today: ML Pipeline is Complex and Siloed



# Kubernetes: One Platform, Complete ML Lifecycle



# Open-Source ML Pipeline Components By Category

Dev Tools  
<..>



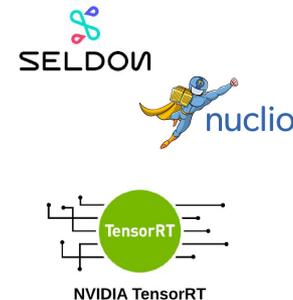
Data Ingest  
& Prep



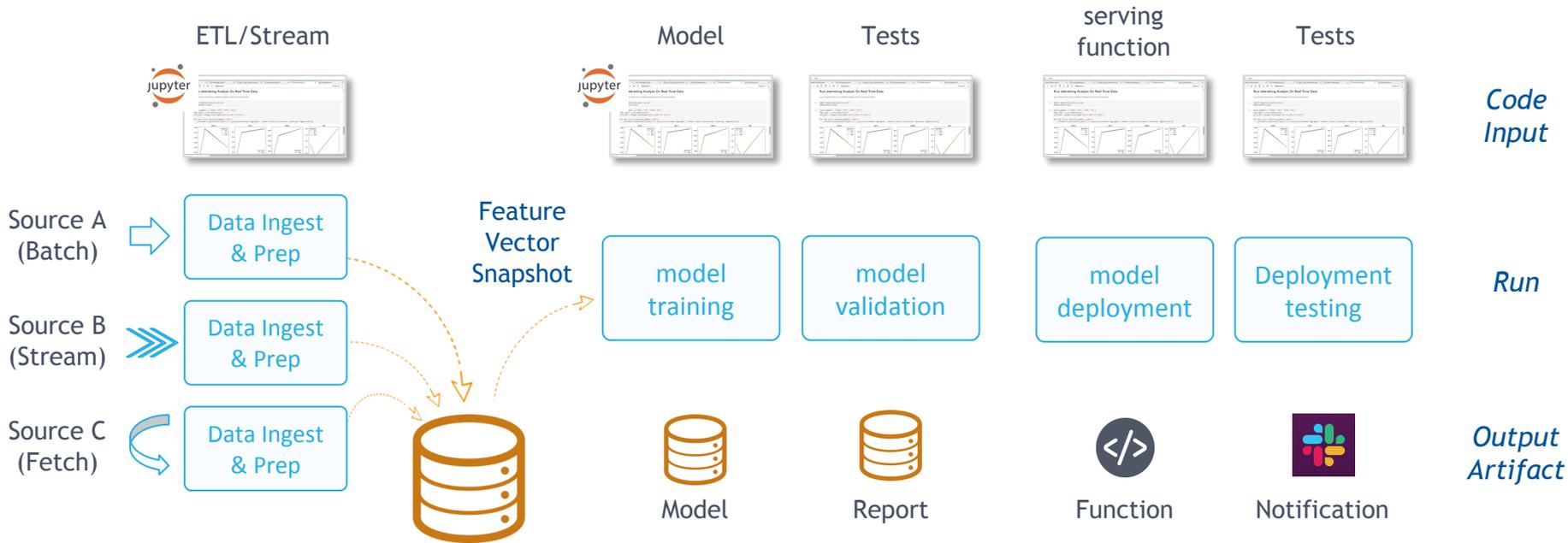
Training  
& Validation



model  
serving



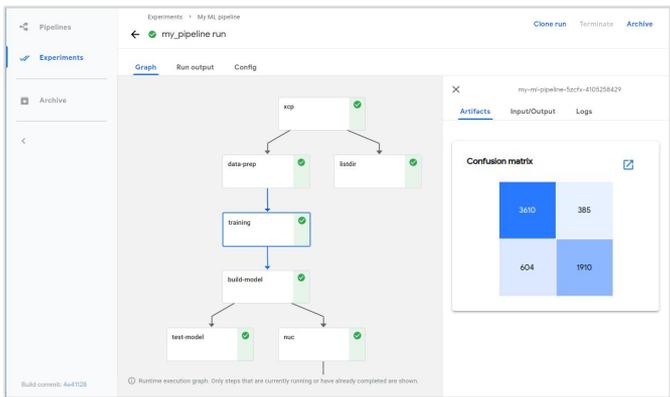
# Typical Data Science Pipeline



**Pipeline must be automated !**



# KubeFlow Pipeline



The screenshot displays the 'Compare runs' interface. It shows a table of runs with the following columns: Run name, Status, Duration, Experiment, Pipeline, Start time, accuracy-score, and loss. Two runs are listed, both with a status of 'Completed' and an accuracy score of 8.000.

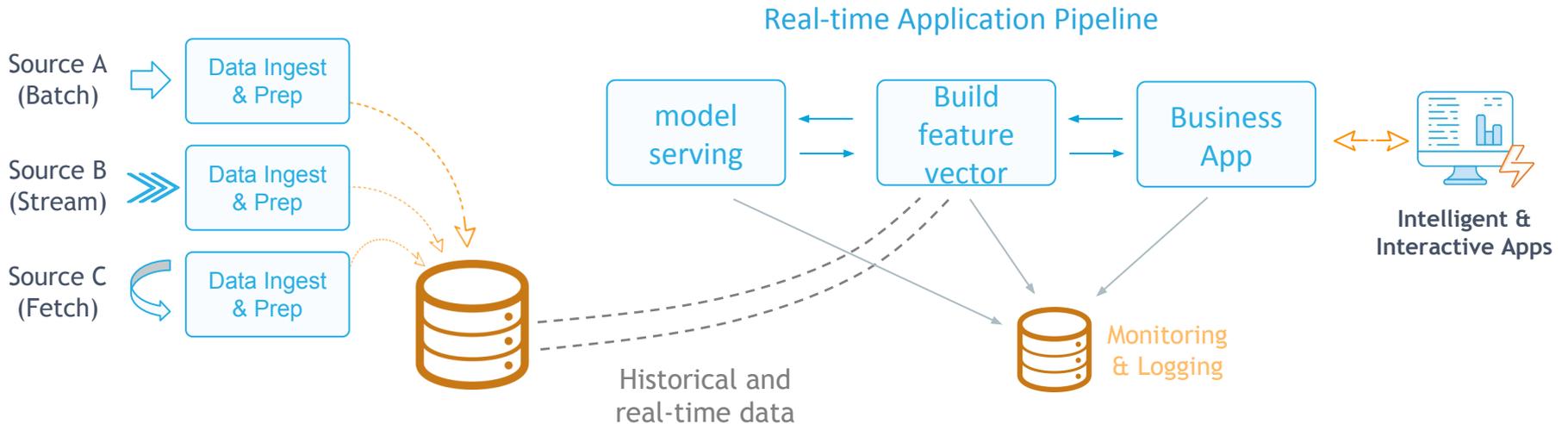
Run name	Status	Duration	Experiment	Pipeline	Start time	accuracy-score	loss
my_pipeline run	Completed	0:00:30	My ML pipeline	[View pipeline]	5/1/2019, 10...	7.000	7.000
my_pipeline run	Completed	0:00:28	My ML pipeline	[View pipeline]	5/1/2019, 10...	8.000	8.000

Below the table, there are sections for 'Parameters' and 'Markdown'. The 'Parameters' section shows a table with columns for 'my\_pipeline run' and 'my\_pipeline run', and rows for 'txt' and 'val'. The 'Markdown' section shows two boxes for 'my\_pipeline run' with 'Results' and '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



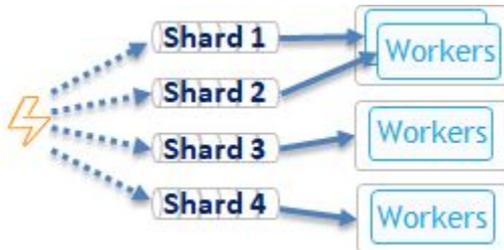
# Nuclio: Taking Serverless to Data Intensive Apps

## Extreme Performance



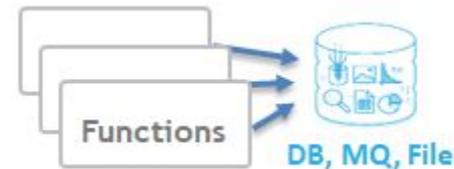
- Non-blocking, parallel
- Zero copy, buffer reuse
- Up to 400K events/sec/proc
- GPU optimizations

## 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







# 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