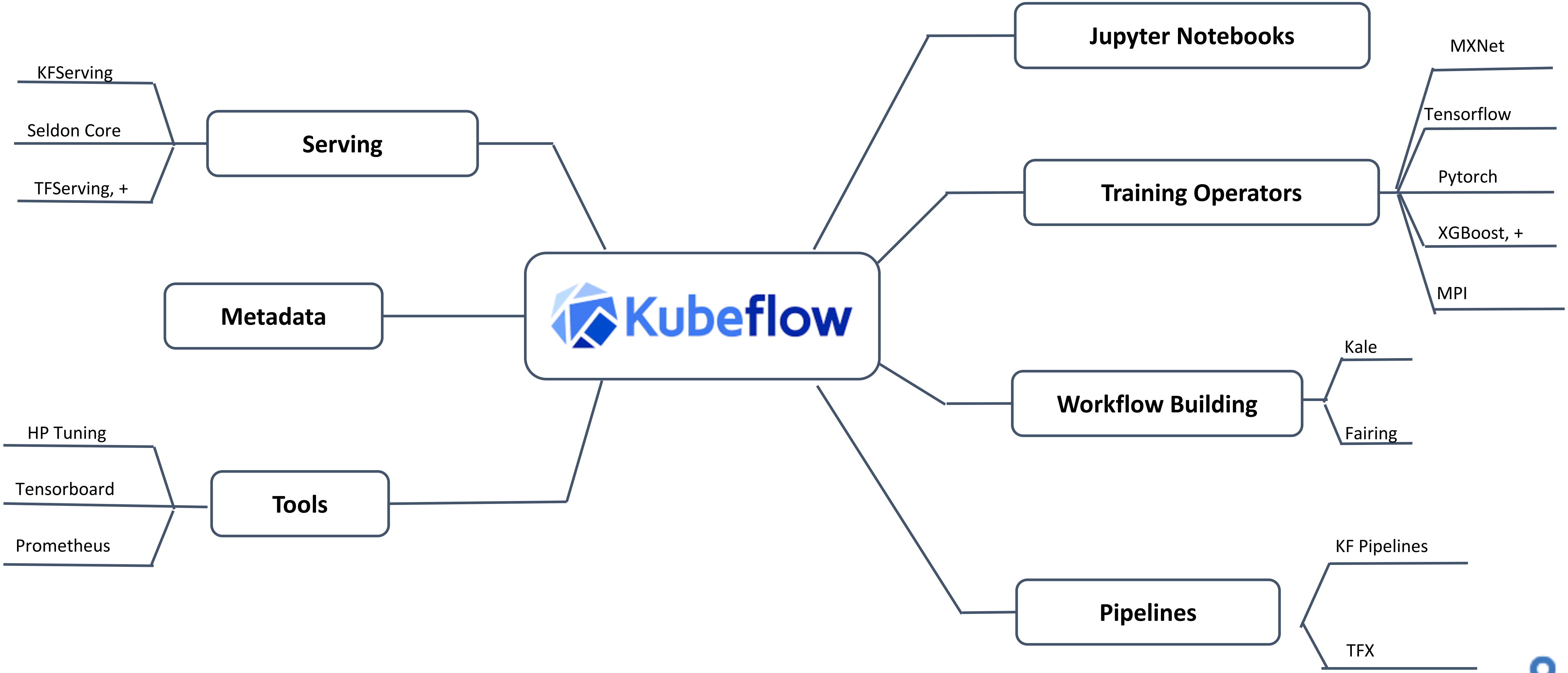
Kubeflow - Distributed Training and HPO

Tommy Li



Distributed Model Training and HPO (TFJob, PyTorch Job, MPI Job, Katib, …)

Kubeflow

- Addresses One of the key goals for model builder persona:

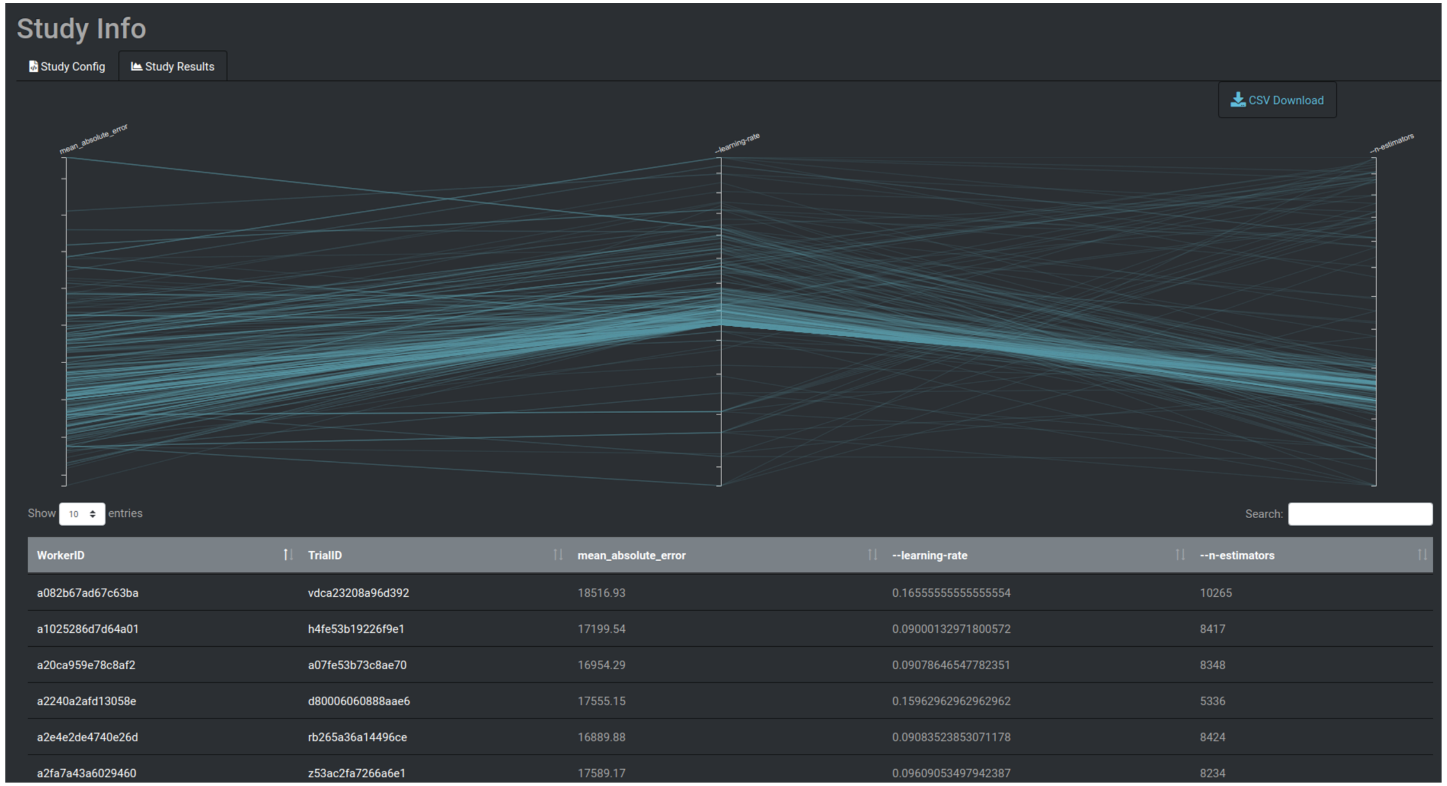
  Distributed Model Training and Hyper parameter optimization for **Tensorflow, PyTorch, XGBoost, MXNet, etc.**

  Common problems in HP optimization
  - Overfitting
  - Wrong metrics
  - Too few hyperparameters

Katib: a fully open source, Kubernetes-native hyperparameter tuning service

- Inspired by Google Vizier
- Framework agnostic
- Extensible algorithms
- Simple integration with other Kubeflow components

Kubeflow also supports distributed MPI based training using Horovod



TensorFlow PYTORCH XGBoost

HOROVOD

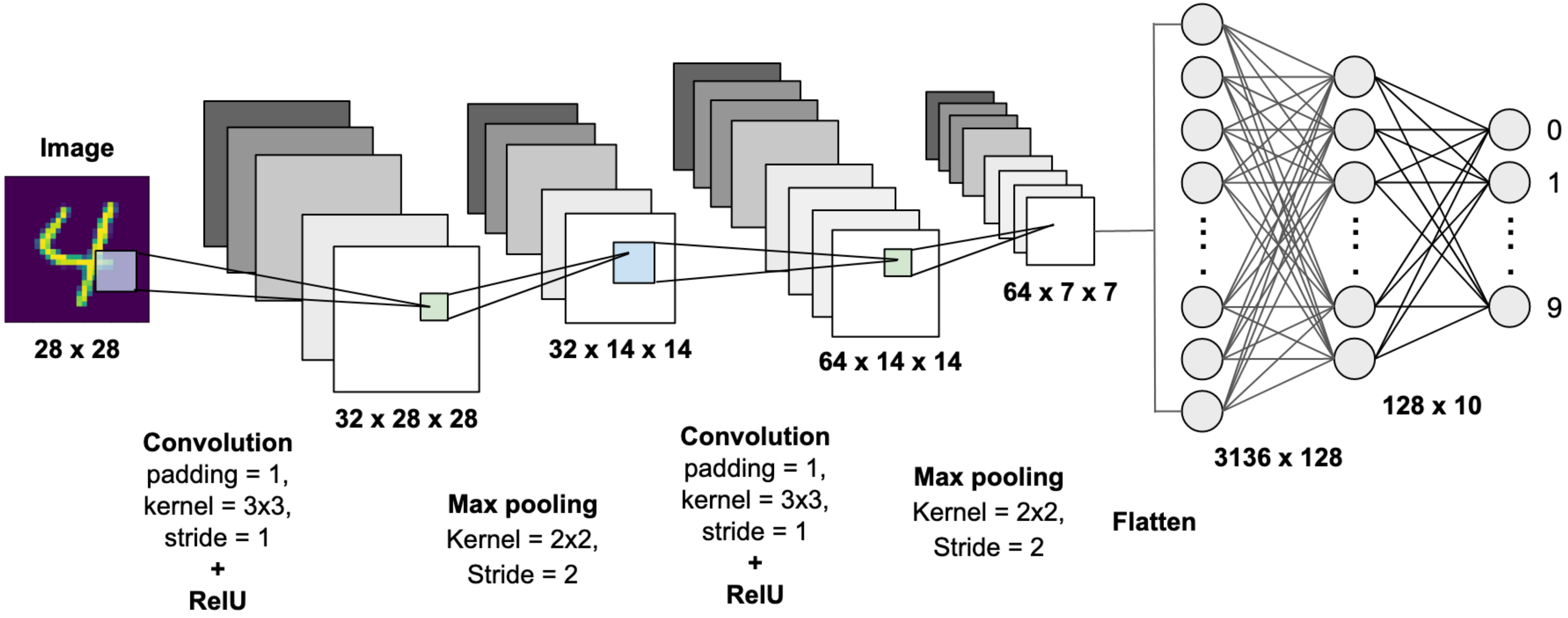
IBM

Distributed Training Operators

	TF Operator	PyTorch Operator	MPI Operator
Framework Support	TensorFlow	PyTorch	HOROVOD TensorFlow/Keras Apache MXNet/PyTorch/OpenMPI
Distribution Strategy & Backend	tf.distribute MPI/NCCL/PS/TPU	torch.distributed Gloo/MPI/NCCL	horovod DistributedOptimizer Gloo/MPI/NCCL

Traditional Model Training

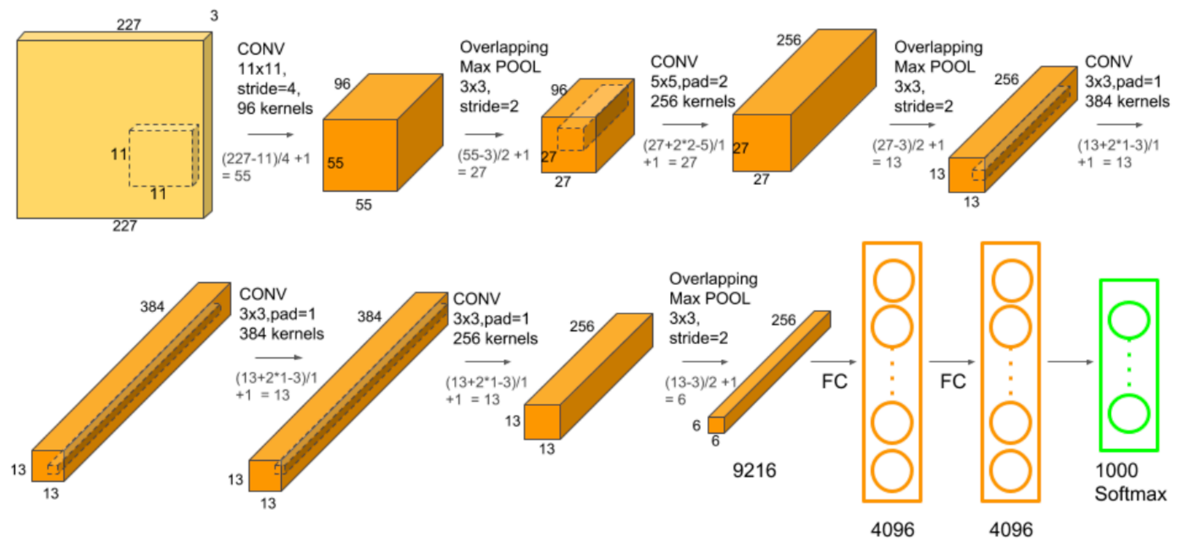
Machine 1



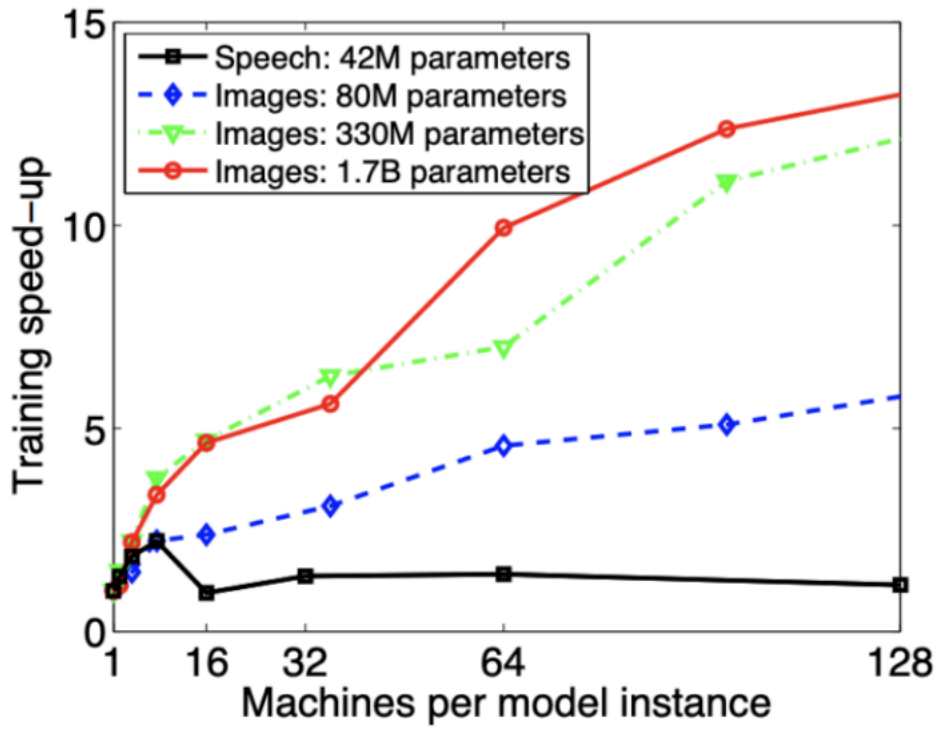
Source: https://towardsdatascience.com/mnist-handwritten-digits-classification-using-a-convolutional-neural-network-cnn-af5fafbc35e9

Need for Distributed Training

- Models that are too large for a single device

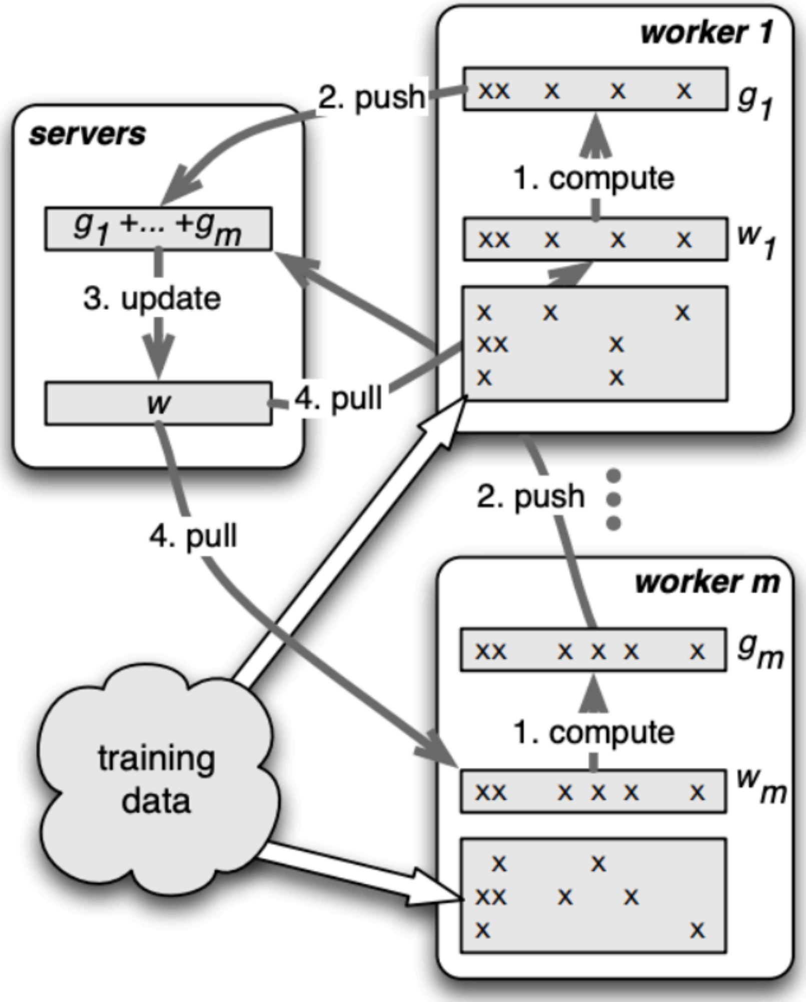


- Improved parallelization



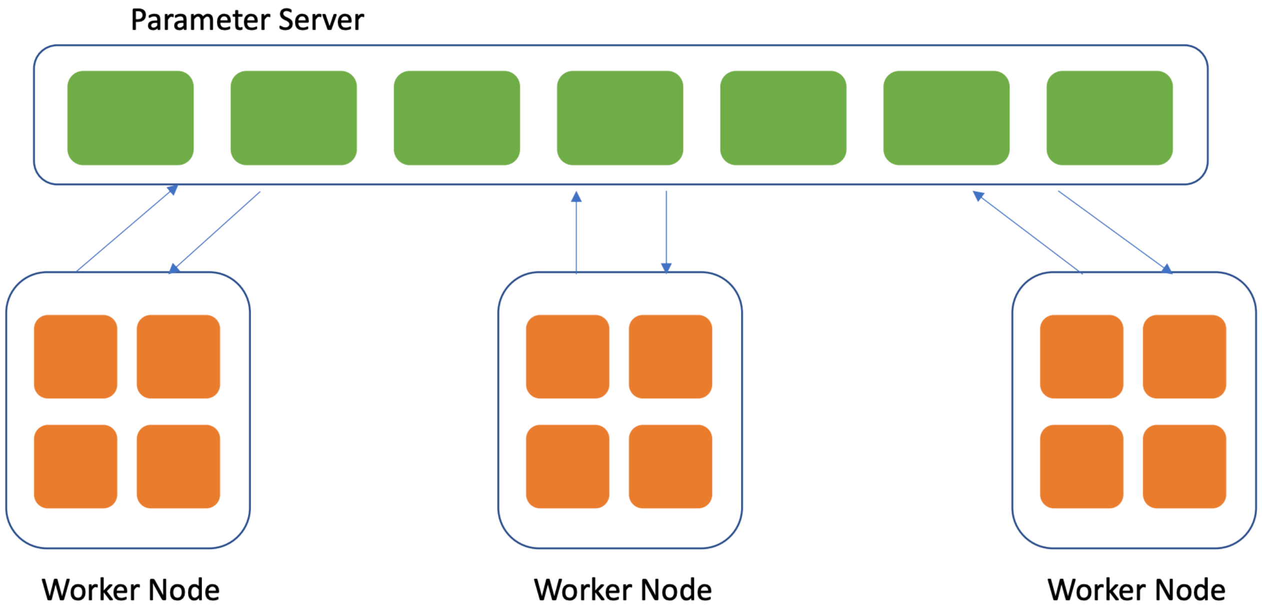
Source: http://www.juyang.co/distributed-model-training-ii-parameter-server-and-allreduce/

Distributed Model Training

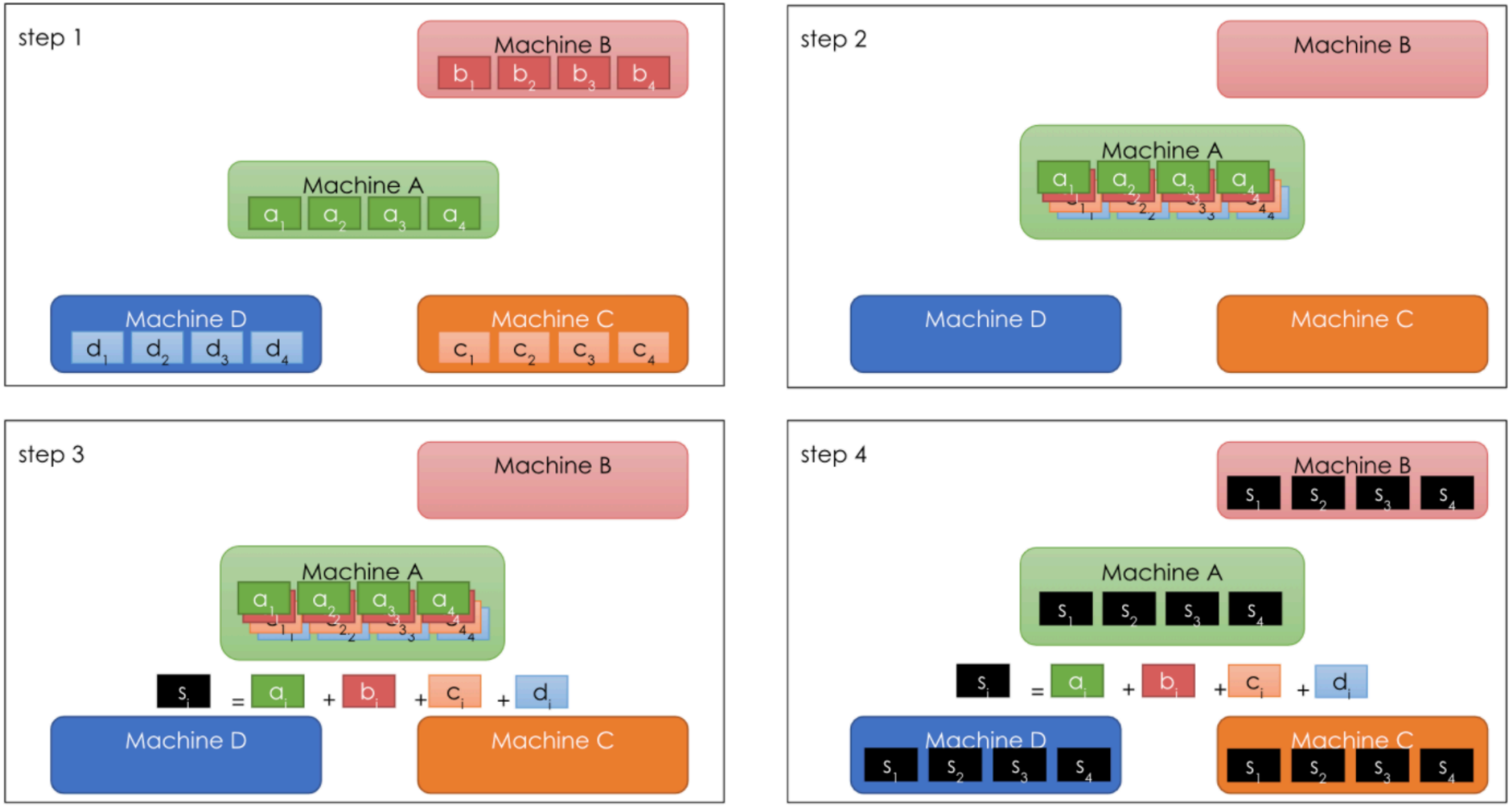


Parameter Servers

- Most simple form of distributed training
- One centralized parameter server does the aggregation job of collecting and redistributing results of each worker node

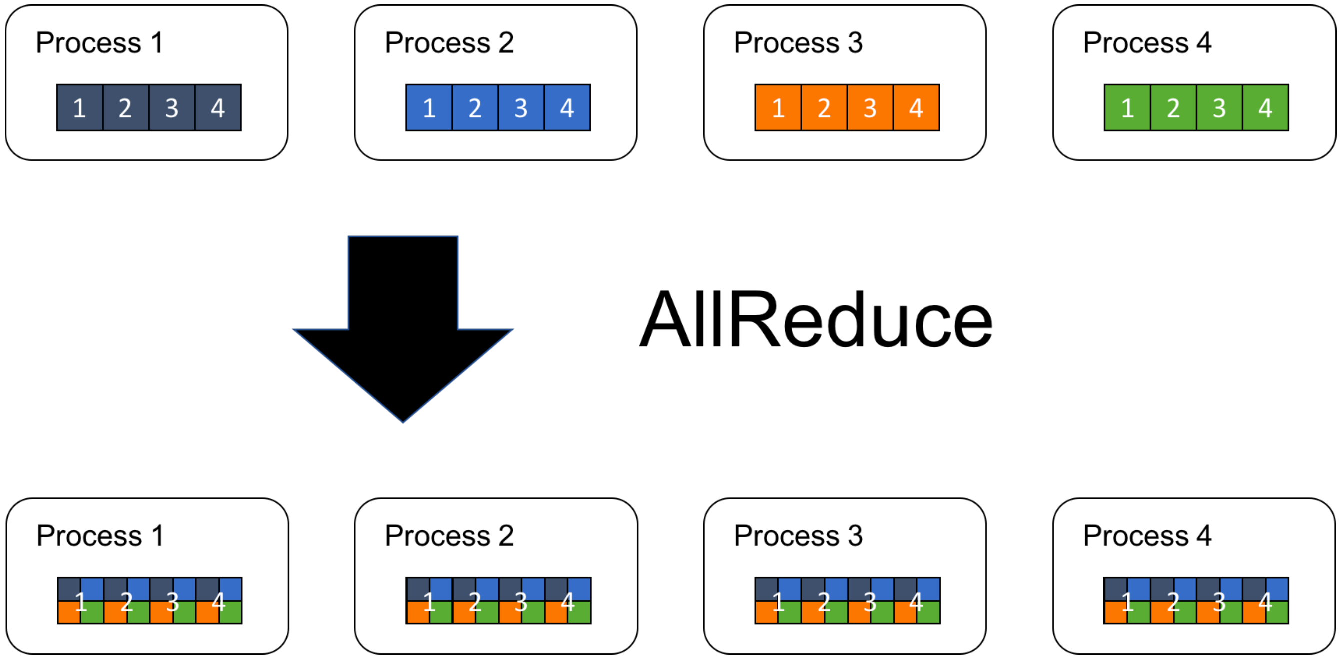


Parameter Servers

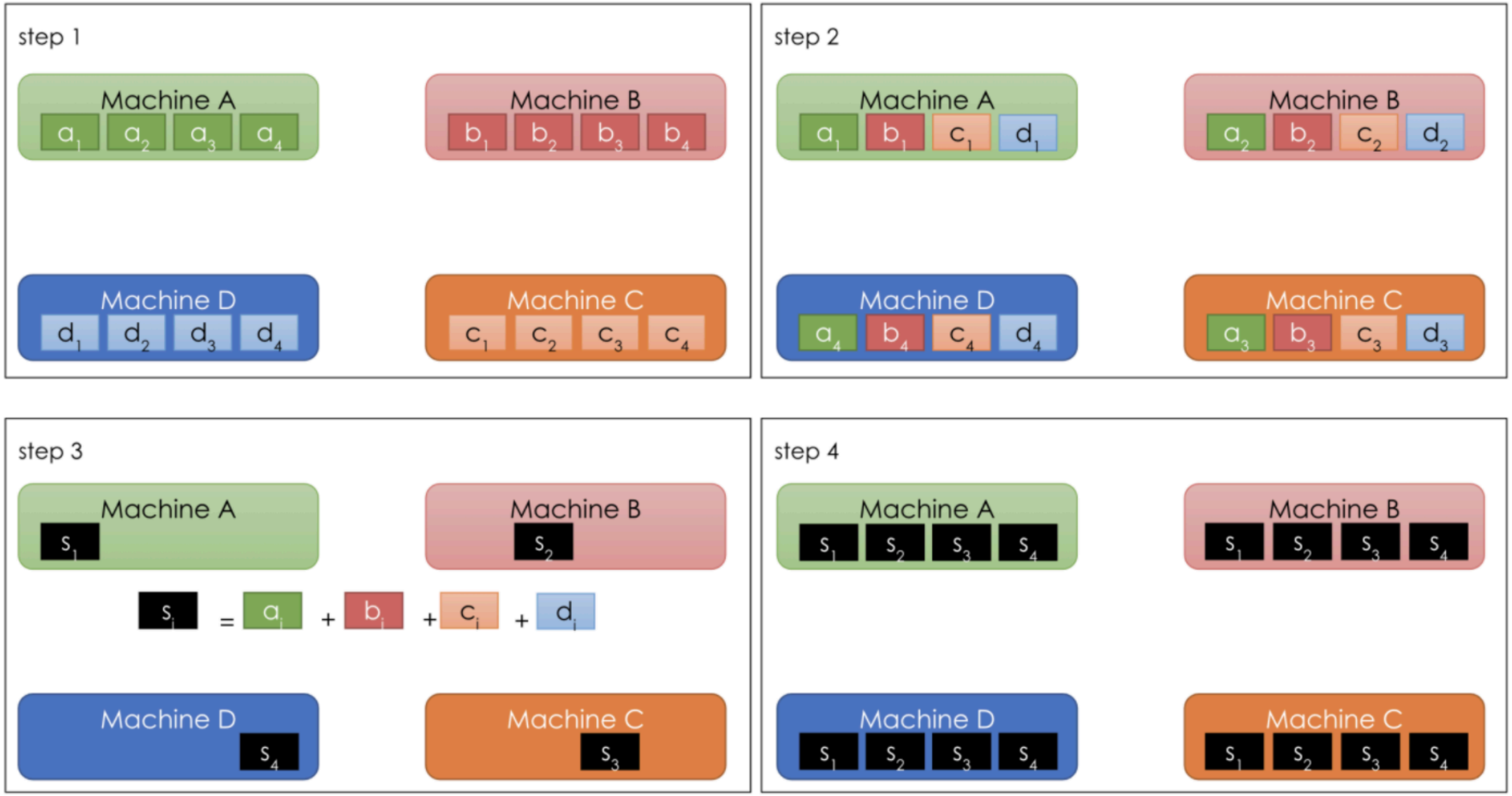


AllReduce

- Most parallelized form of distributed training
- There are many different styles of AllReduce with each having different benefits and costs



AllReduce



step 1
Machine A
a1 a2 a3 a4
Machine B
b1 b2 b3 b4
Machine D
d1 d2 d3 d4
Machine C
c1 c2 c3 c4
step 2
Machine A
a1 b1 c1 d1
Machine B
a2 b2 c2 d2
Machine D
a4 b4 c4 d4
Machine C
a3 b3 c3 d3
step 3
Machine A
s1
Machine B
s2
si = ai + bi + ci + di
Machine D
s4
Machine C
s3
step 4
Machine A
s1 s2 s3 s4
Machine B
s1 s2 s3 s4
Machine D
s1 s2 s3 s4
Machine C
s1 s2 s3 s4

Advantages of allreduce-style training

- Each worker stores a complete set of model parameters, so adding more workers is easy
- Failures among workers can be recovered easily by just restarting the failed worker and loading the model from an existing worker
- Models can be updated more efficiently by leveraging network structure
- Scaling up and down workers only requires reconstructing the underlying allreduce communicator and re-assigning the ranks among the workers

Distributed Training in Kubeflow

Kubernetes → Kubeflow →

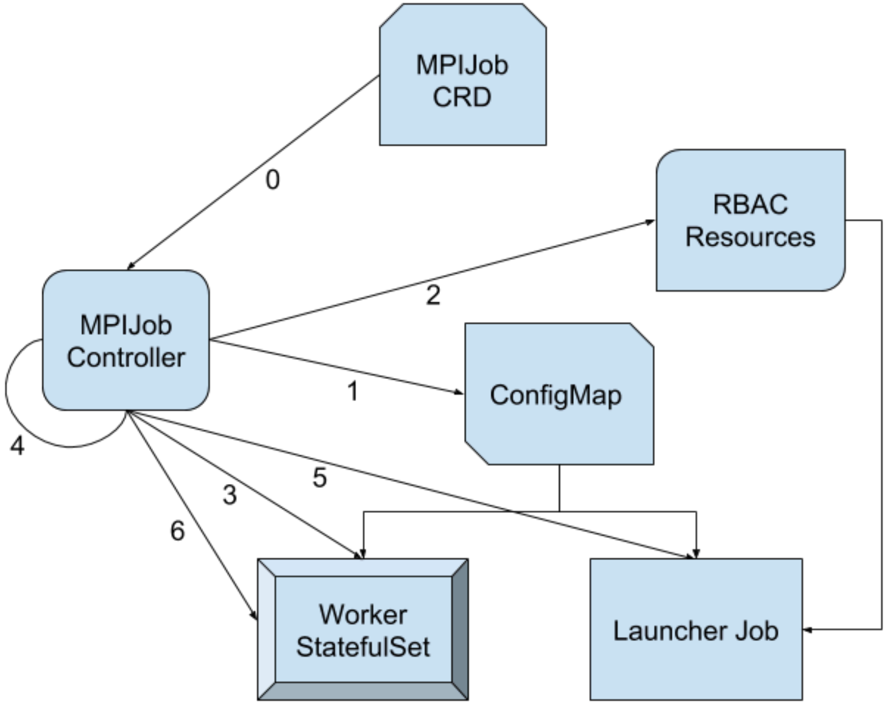
- tf-operator
- pytorch-operator
- mpi-operator

MPI Operator

- The MPI Operator allows for running allreduce-style distributed training on Kubernetes
- Provides common Custom Resource Definition (CRD) for defining training jobs
- Unlike other operators, such as the TF Operator and the Pytorch Operator, the MPI Operator is decoupled from one machine learning framework. This allows the MPI Operator to work with many machine learning frameworks such as Tensorflow, Pytorch, and Apache MXNet

Design

- When a new MPIJob is created the MPIJob Controller goes through a set of steps
- 1. Create a ConfigMap
- 2. Create the RBAC resources (Role, Service Account, Role Binding) to allow remote execution (pods/exec)
- 3. Create the worker StatefulSet
- 4. Wait for worker pods to be ready
- 5. Create the Job which is run under the Service Account (from Step 2)



Example API Spec

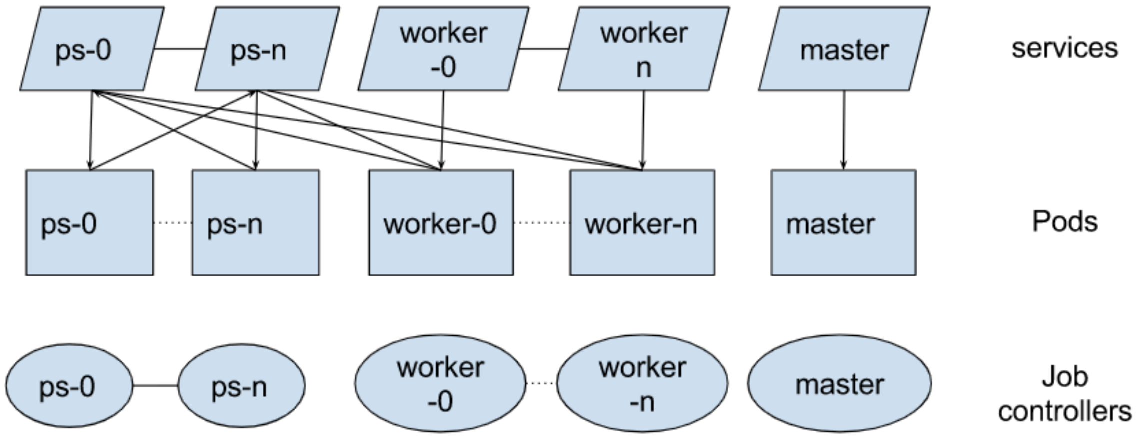
```yaml
apiVersion: kubeflow.org/v1alpha2
kind: MPIJob
metadata:
  name: tensorflow-benchmarks
spec:
  slotsPerWorker: 1
  cleanPodPolicy: Running
  mpiReplicaSpecs:
    Launcher:
      replicas: 1
      template:
         spec:
           containers:
           - image: mpioperator/tensorflow-benchmarks:latest
             name: tensorflow-benchmarks
             command:
             - mpirun
             - python
             - scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py
             - --model=resnet101
             - --batch_size=64
             - --variable_update=horovod
    Worker:
      replicas: 2
      template:
        spec:
          containers:
          - image: mpioperator/tensorflow-benchmarks:latest
            name: tensorflow-benchmarks
            resources:
              limits:
                nvidia.com/gpu: 1
```

TF Operator

- TFJobs are Kubernetes custom resource definitions for running distributed and non-distributed Tensorflow jobs on Kubernetes
- The tf-operator is the Kubeflow implementation of TFJobs
- A TFJob is a collection of TfReplicas where each TfReplica corresponds to a set of Tensorflow processes performing a role in the job

Design

- A distributed Tensorflow Job is collection of the following processes
  - Chief – The chief is responsible for orchestrating training and performing tasks like checkpointing the model
  - Ps – The ps are parameters servers; the servers provide a distributed data store for the model parameters to access
  - Worker – The workers do the actual work of training the model. In some cases, worker 0 might also act as the chief
  - Evaluator - The evaluators can be used to compute evaluation metrics as the model is trained



TFJob vs. MPIJob

```
apiVersion: "kubeflow.org/v1beta1”
kind: TFJob
metadata:
 name: distributed-training
spec:
 tfReplicaSpecs:
   Worker:
     replicas: 4
     template:
       spec:
         containers:
         - name: tensorflow
           image: distributed_training_tf:latest
           resources:
             limits: nvidia.com/gpu: 4
             command: "python tf_benchmarks.py"
```

```
apiVersion: "kubeflow.org/v1alpha2”
kind: MPIJob
metadata:
 name: distributed-training
spec:
 mpiReplicaSpecs:
   Worker:
     replicas: 4
     template:
       spec:
         containers:
         - name: tensorflow
           image: distributed_training_hovorod:latest
           resources:
             limits: nvidia.com/gpu: 4
             command: ”mpirun python hovorod_benchmarks.py"
```

Pytorch Operator

- Similar to TFJobs and MPIJobs, PytorchJobs are Kubernetes custom resource definitions for running distributed and non-distributed PytorchJobs on Kubernetes
- The pytorch-operator is the Kubeflow implementation of PytorchJobs
- There are a number of metrics that can be monitored for each component container of the pytorch-operator by using Prometheus Montoring

Pytorch Operator monitoring

- Prometheus monitoring for pytorch operator makes the many available metrics easy to monitor
- There are metrics for each component container for the pytorch operator, such as CPU usage, GPU usage, Keep-Alive check, and more
- There are also metrics for reporting PytorchJob information such as job creation, successful completions, failed jobs, etc.

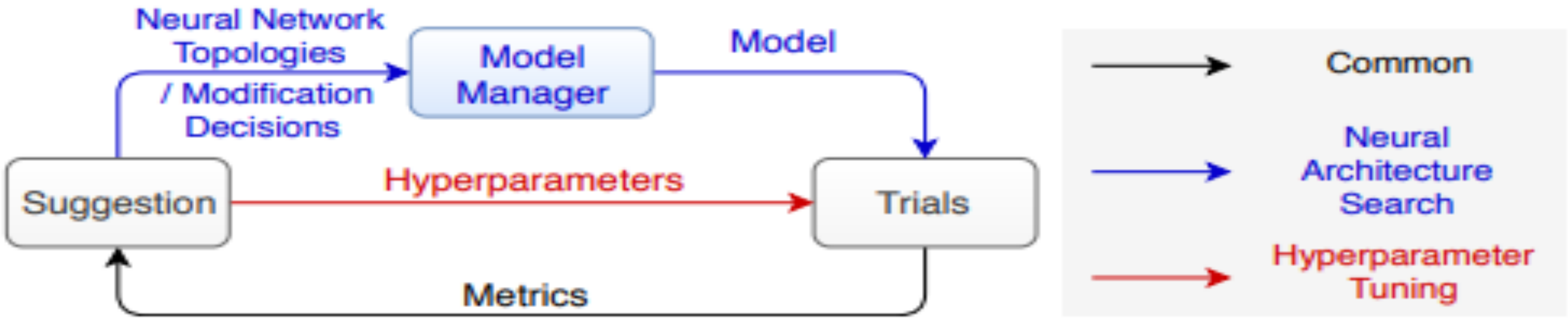
Demo

Introduction to Katib

Kubeflow-Katib

- Motivation: Automated tuning machine learning model's hyperparameters and neural architecture search.
- Major components:
  - katib-db-manager: GRPC API server of Katib which is the DB Interface.
  - katib-mysql: Data storage backend of Katib using mysql.
  - katib-ui: User interface of Katib.
  - katib-controller: Controller for Katib CRDs in Kubernetes.
- Katib: Kubernetes Native System for Hyperparameter Turning and Neural Architecture Search.
- Github Repository: https://github.com/kubeflow/katib

AutoML workflows

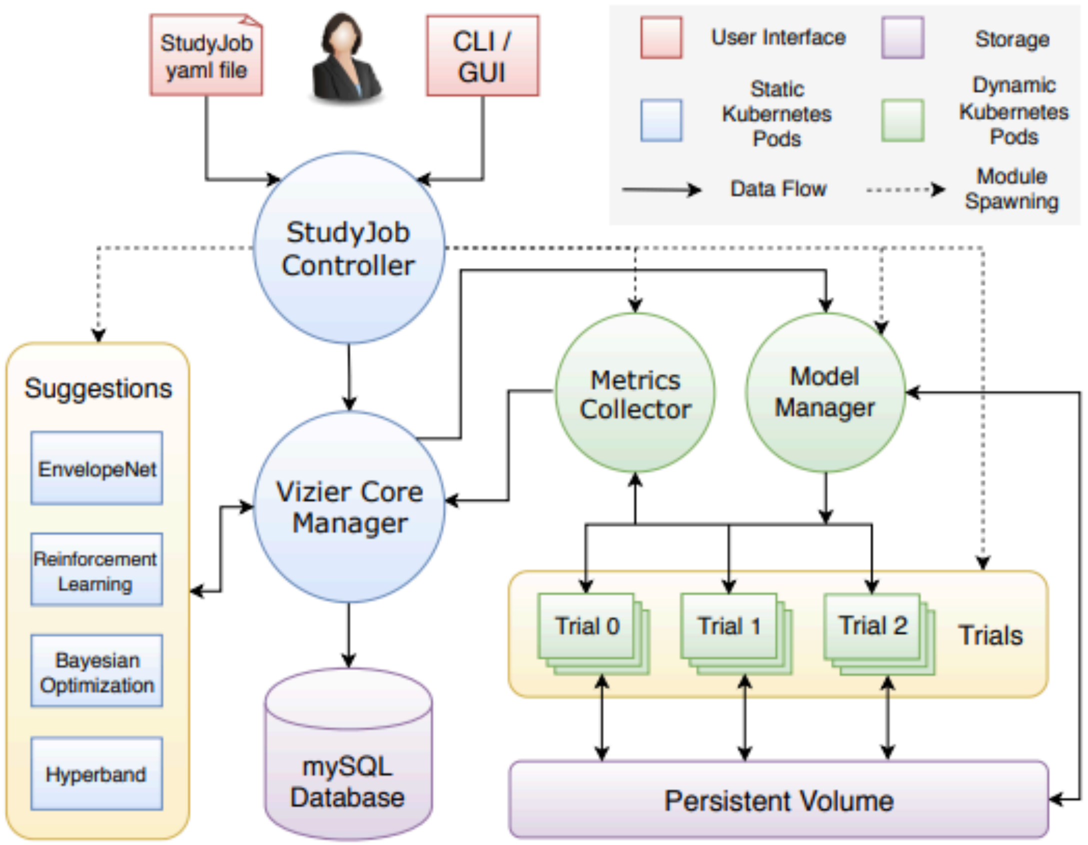


Click to add text

Figure 1: Summary of AutoML workflows

Katib is a scalable Kubernetes-native general AutoML platform.
Katib integrate hyper-parameter turning and NAS into one flexible frame-work.

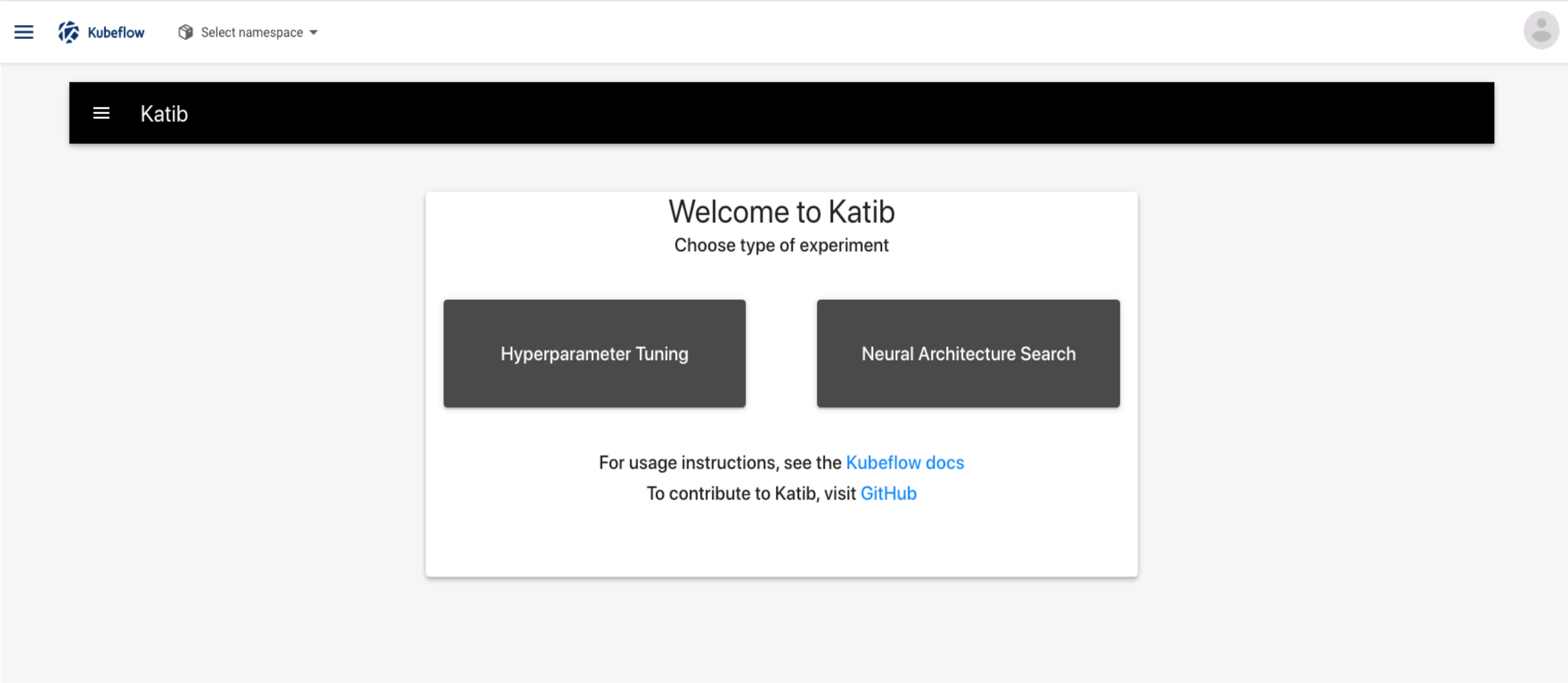
Design of Katib



Note: StudyJob is now called Experiment

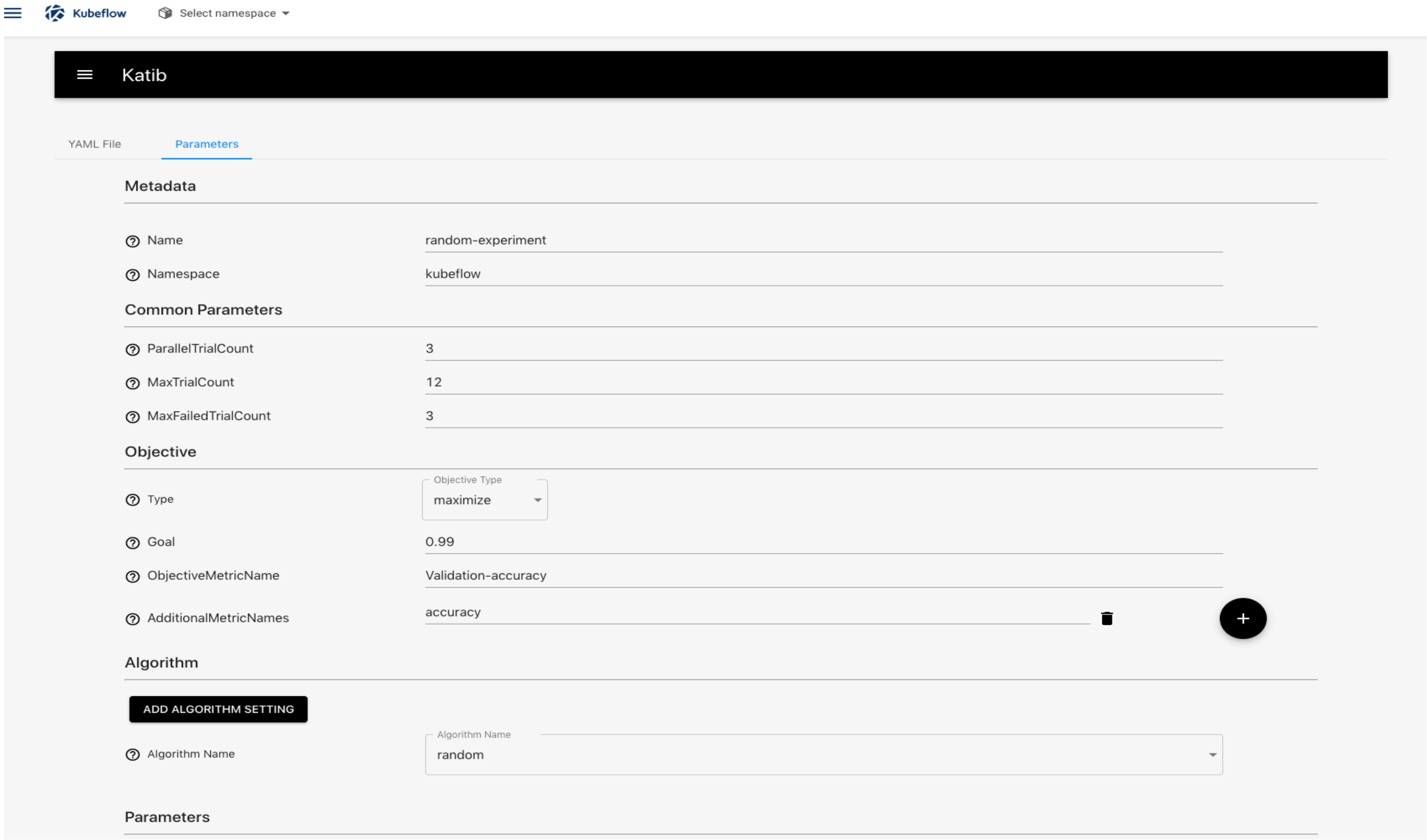
Accessing the katib UI

- Under the Kubeflow web UI, click the Katib on the left side bar.



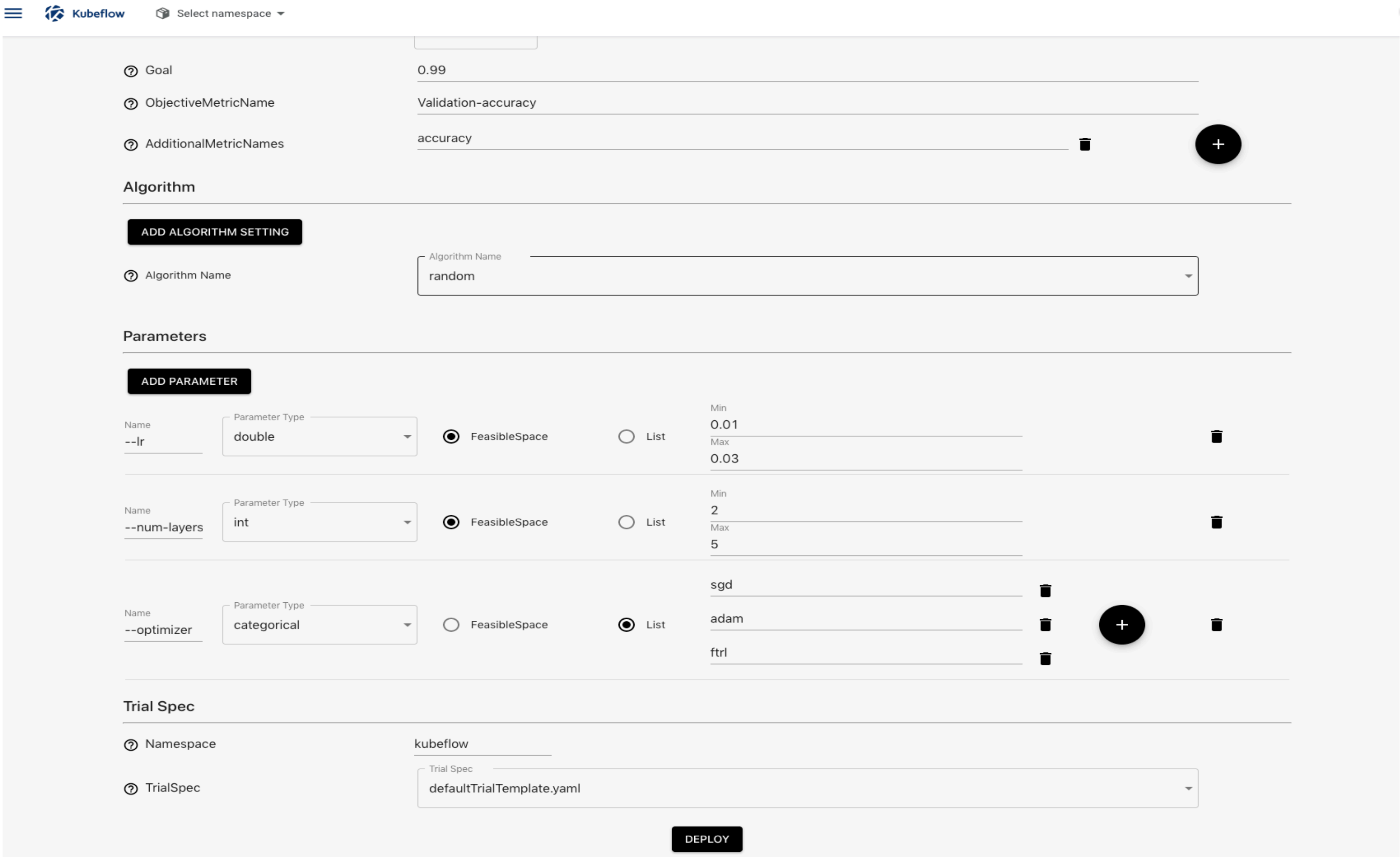
First Example of Katib

- We are using random-example from Hyper-parameter Turning



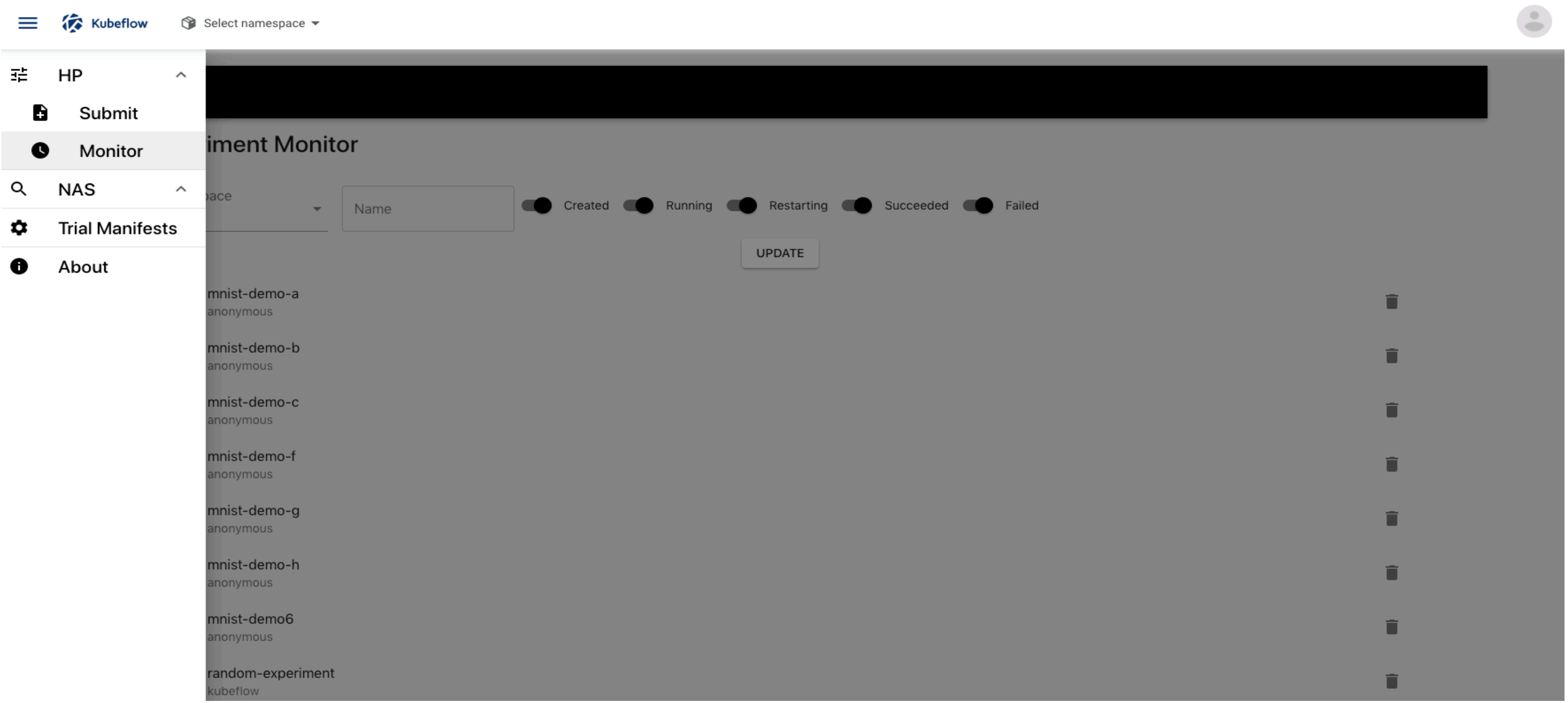
Deploy the random-example

- Click the Deploy



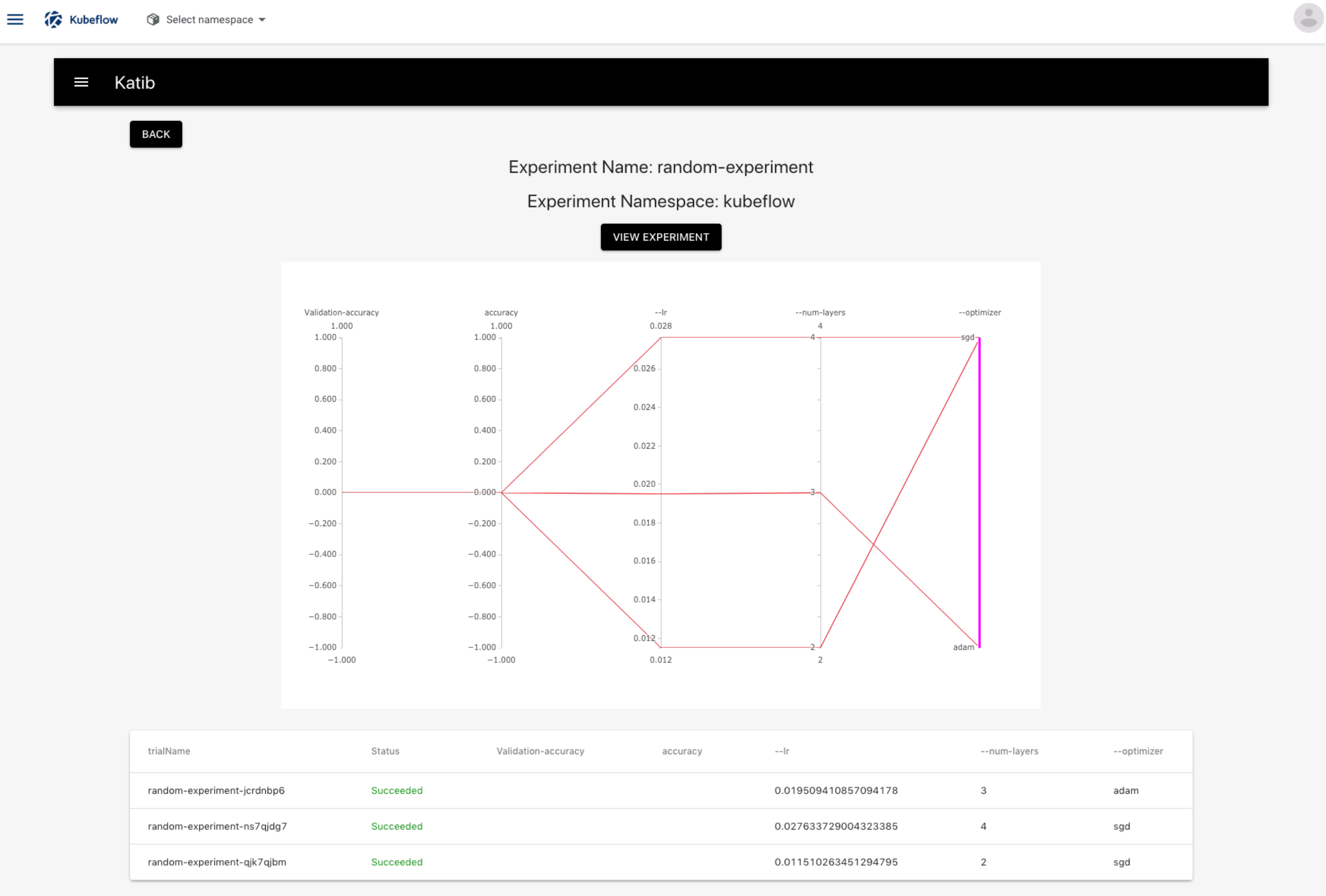
Check experiment status

- Click the Katib tab, then choose Monitor under HP on the left side



View Experiment

- Click the experiment name, it will show the experiment also the status of trial



Check status from command line

- At your K8S cluster command line:

```
(base) Qianyangs-MBP:kevin-kubeflow-demo-0521 qianyangyu$ kubectl get experiment -n kubeflow
NAME                STATUS    AGE
random-experiment   Running   132m
(base) Qianyangs-MBP:kevin-kubeflow-demo-0521 qianyangyu$
```

Check Experiment's CR

- Get the experiment CR from command line

kevin-kubeflow-demo-0521 — -bash — 113×34

```
(base) Qianyangs-MBP:kevin-kubeflow-demo-0521 qianyangyu$ kubectl get experiment random-experiment -n kubeflow -o
 yaml
apiVersion: kubeflow.org/v1alpha3
kind: Experiment
metadata:
  creationTimestamp: "2020-06-25T22:22:10Z"
  finalizers:
  - update-prometheus-metrics
  generation: 1
  name: random-experiment
  namespace: kubeflow
  resourceVersion: "10842626"
  selfLink: /apis/kubeflow.org/v1alpha3/namespaces/kubeflow/experiments/random-experiment
  uid: f3868bd1-1ccb-4de4-beb7-852d6c67d8f8
spec:
  algorithm:
    algorithmName: random
    algorithmSettings: []
  maxFailedTrialCount: 3
  maxTrialCount: 12
  objective:
    additionalMetricNames:
    - accuracy
    goal: 0.99
    objectiveMetricName: Validation-accuracy
    type: maximize
  parallelTrialCount: 3
  parameters:
  - feasibleSpace:
      max: "0.03"
      min: "0.01"
    name: --lr
    parameterType: double
  - feasibleSpace:
```

Conti. Experiment CR

```
spec:
  algorithm:
    algorithmName: random
    algorithmSettings: []
  maxFailedTrialCount: 3
  maxTrialCount: 12
  objective:
    additionalMetricNames:
    - accuracy
    goal: 0.99
    objectiveMetricName: Validation-accuracy
    type: maximize
  parallelTrialCount: 3
  parameters:
  - feasibleSpace:
      max: "0.03"
      min: "0.01"
    name: --lr
    parameterType: double
  - feasibleSpace:
      max: "5"
      min: "2"
    name: --num-layers
    parameterType: int
  - feasibleSpace:
      list:
      - sgd
      - adam
      - ftrl
    name: --optimizer
    parameterType: categorical
  trialTemplate:
    goTemplate:
      templateSpec:
        configMapName: trial-template
        configMapNamespace: kubeflow
        templatePath: defaultTrialTemplate.yaml
```

- Algorithm: Katib supports random, grid, hyperband, bayesian optimization and tpe algorithms.
- MaxFailedTrialCount: specify the max the tuning with failed status
- MaxTrialCount: specify the limit for the hyper-parameters sets can be generated.
- Objective: Set objetiveMetricName and additionalMetricNames.
- ParalleTrialCount: how many set of hyper-parameter to be tested in parallel.

Fields in Experiment's spec

```yaml
spec:
  algorithm:
    algorithmName: random
    algorithmSettings: []
  maxFailedTrialCount: 3
  maxTrialCount: 12
  objective:
    additionalMetricNames:
    - accuracy
    goal: 0.99
    objectiveMetricName: Validation-accuracy
    type: maximize
  parallelTrialCount: 3
  parameters:
  - feasibleSpace:
      max: "0.03"
      min: "0.01"
    name: --lr
    parameterType: double
  - feasibleSpace:
      max: "5"
      min: "2"
    name: --num-layers
    parameterType: int
  - feasibleSpace:
      list:
      - sgd
      - adam
      - ftrl
    name: --optimizer
    parameterType: categorical
  trialTemplate:
    goTemplate:
      templateSpec:
        configMapName: trial-template
        configMapNamespace: kubeflow
        templatePath: defaultTrialTemplate.yaml
```

- TrialTemplate: Your model should be packaged by image, model's hyper-parameter must be configurable by argument or environment variable.
- Parameter: defines the range of the hyper-parameters you want to tune your model.
- MetricsCollectorSpec: The metric collectors for stdout, file or tfevent. Metric collecting will run as a sidecar if enabled.

Trial

- Katib internally generate a Trial CR, it is for internal logic control.

```
(base) Qianyangs-MBP:kevin-kubeflow-demo-0521 qianyangyu$ kubectl get trial -n kubeflow
NAME                         TYPE        STATUS    AGE
random-experiment-jcrdnbp6   Succeeded   False     138m
random-experiment-ns7qjdg7   Succeeded   False     138m
random-experiment-qjk7qjbm   Succeeded   False     138m
(base) Qianyangs-MBP:kevin-kubeflow-demo-0521 qianyangyu$ kubectl get trial -n kubeflow -o yaml
apiVersion: v1
items:
- apiVersion: kubeflow.org/v1alpha3
  kind: Trial
  metadata:
    creationTimestamp: "2020-06-25T22:22:47Z"
    finalizers:
    - clean-metrics-in-db
    generation: 1
    labels:
      experiment: random-experiment
    name: random-experiment-jcrdnbp6
    namespace: kubeflow
    ownerReferences:
    - apiVersion: kubeflow.org/v1alpha3
      blockOwnerDeletion: true
      controller: true
      kind: Experiment
      name: random-experiment
      uid: f3868bd1-1ccb-4de4-beb7-852d6c67d8f8
    resourceVersion: "10842624"
    selfLink: /apis/kubeflow.org/v1alpha3/namespaces/kubeflow/trials/random-experiment-jcrdnbp6
    uid: 7837e2c9-8b57-47d5-b396-1d94256c81f4
  spec:
    metricsCollector: {}
    objective:
      additionalMetricNames:
      - accuracy
      goal: 0.99
      objectiveMetricName: Validation-accuracy
      type: maximize
    parameterAssignments:
```

Suggestion

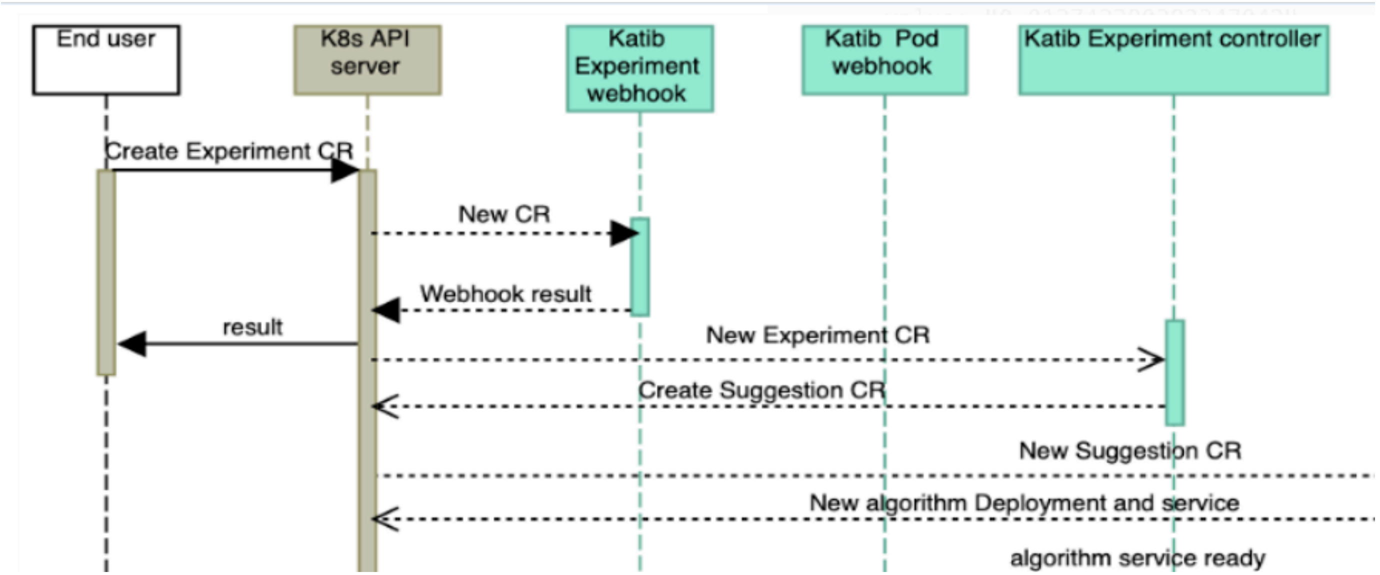
- Katib internally create a suggestion CR for each experiment CR. It includes hyper-parameter algorithm name and how many sets of hyper-parameter katib is asking to be generated by requests field.

```
kubectl get suggestion -n kubeflow
NAME                TYPE      STATUS   REQUESTED   ASSIGNED   AGE
random-experiment   Running   True     3           3          13h
(.venv) (base) Qianyangs-MBP:kevin-kubeflow-iks-2032 qianyangyu$ kubectl get suggestion random-experiment -n kubeflow -o
yaml
apiVersion: kubeflow.org/v1alpha3
kind: Suggestion
metadata:
  creationTimestamp: "2020-04-14T04:30:08Z"
  generation: 1
  name: random-experiment
  namespace: kubeflow
  ownerReferences:
  - apiVersion: kubeflow.org/v1alpha3
    blockOwnerDeletion: true
    controller: true
    kind: Experiment
    name: random-experiment
    uid: b925fd88-45fa-48d8-813b-5ad9e88c98b5
  resourceVersion: "37729974"
  selfLink: /apis/kubeflow.org/v1alpha3/namespaces/kubeflow/suggestions/random-experiment
  uid: 528f2b9e-56ae-4c03-8fc5-fe76a6dacdfb
spec:
  algorithmName: random
  requests: 3
status:
  conditions:
  - lastTransitionTime: "2020-04-14T04:30:08Z"
    lastUpdateTime: "2020-04-14T04:30:08Z"
    message: Suggestion is created
    reason: SuggestionCreated
    status: "True"
    type: Created
  - lastTransitionTime: "2020-04-14T04:30:28Z"
    lastUpdateTime: "2020-04-14T04:30:28Z"
    message: Deployment is ready
    reason: DeploymentReady
    status: "True"
    type: DeploymentReady
  - lastTransitionTime: "2020-04-14T04:30:49Z"
    lastUpdateTime: "2020-04-14T04:30:49Z"
    message: Suggestion is running
    reason: SuggestionRunning
    status: "True"
    type: Running
```

Conti. Suggestion

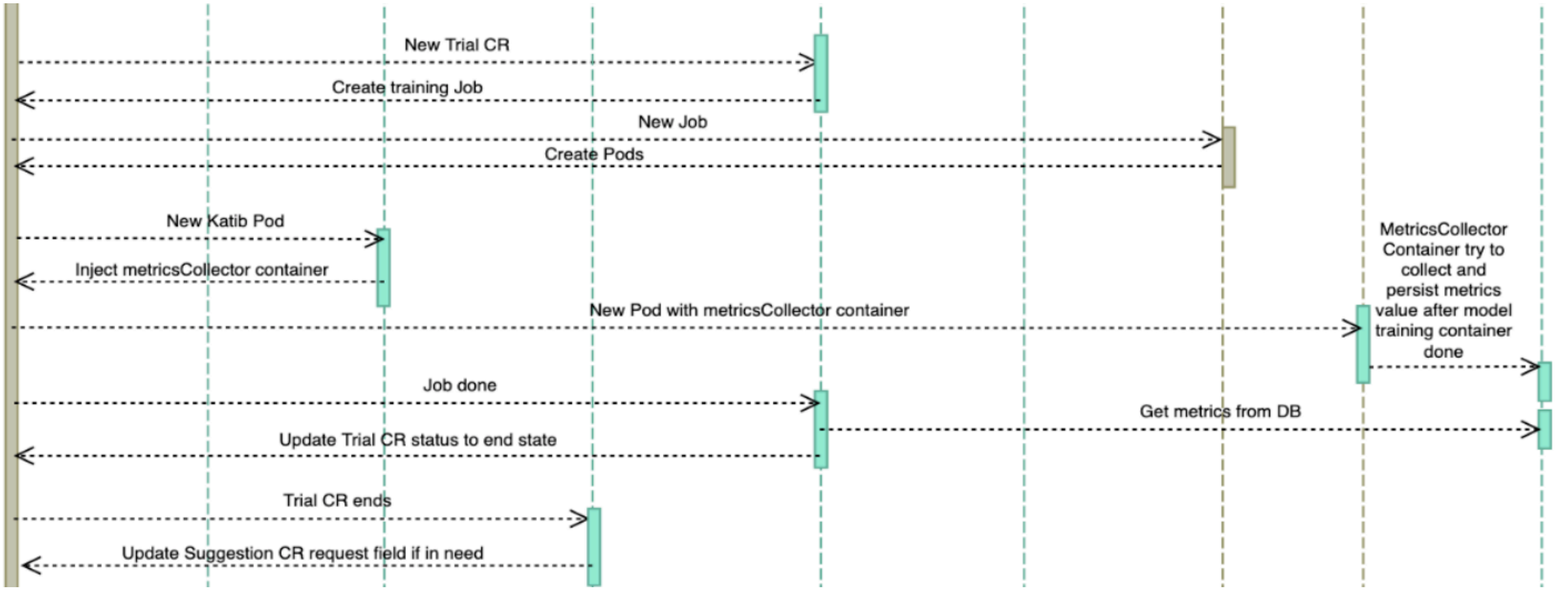

```yaml
startTime: "2020-04-14T04:30:08Z"
suggestionCount: 3
suggestions:
- name: random-experiment-pq8dtx5h
  parameterAssignments:
  - name: --lr
    value: "0.0269665166782524"
  - name: --num-layers
    value: "2"
  - name: --optimizer
    value: sgd
- name: random-experiment-tnfb6ztg
  parameterAssignments:
  - name: --lr
    value: "0.014498585230091017"
  - name: --num-layers
    value: "3"
  - name: --optimizer
    value: ftrl
- name: random-experiment-pprjwngk
  parameterAssignments:
  - name: --lr
    value: "0.011259413563300284"
  - name: --num-layers
    value: "2"
  - name: --optimizer
    value: sgd
```

Katib controller flow(step1 to 3)



1. A experiment CR is submitted to K8S API server; Katib experiment mutating and validating webhook will be called to set default value for the Experiment CR and validate the CR.
2. Experiment controller create a suggestion CR
3. Suggestion controller create the algorithm deployment and service based on the new suggestion CR

Katib controller flow(Step4 to 6)



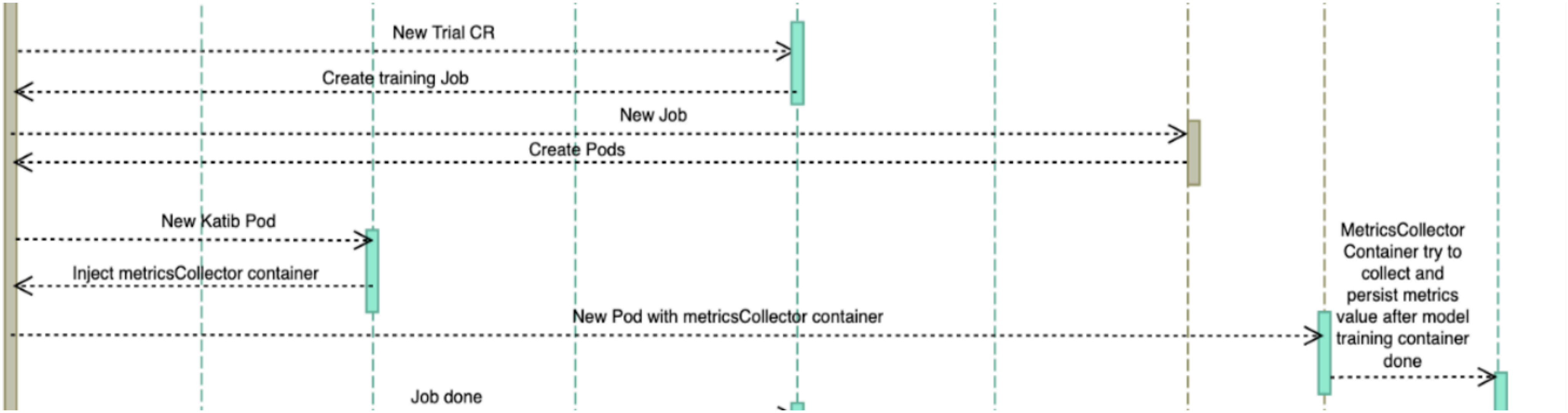
4. Suggestion controller verifies the algorithm service is ready;

generates spec.request - len(status.suggestions) and append them into status.suggestions

5. Experiment controller detects the suggestion CR has been updated, generate each Trial for each new hyper-parameters set

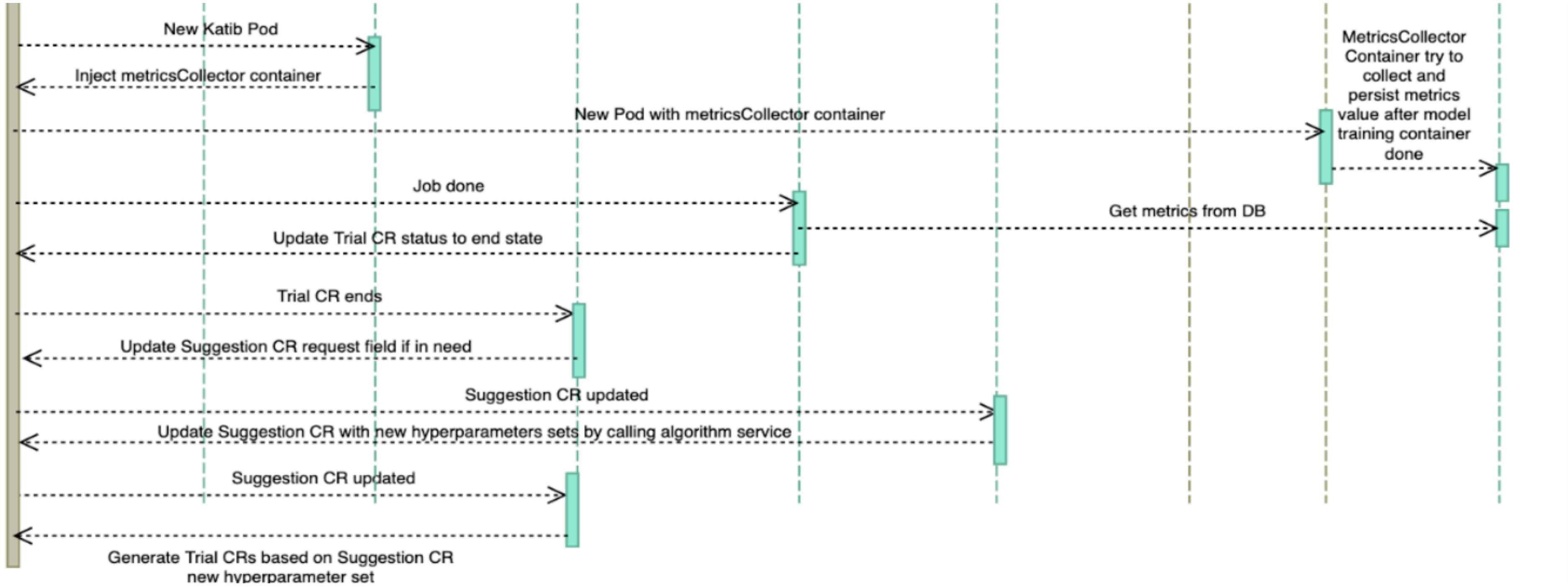
6. Trial controller generates job based on runSpec manifest with the new hyper-parameter set.

Katib controller flow(Step7 to 9)



7. Related job controller (k8s batch job, kubeflow pytorchJob or Kubeflow TFJob) generated Pods.
8. Katib Pod mutating webhook to inject metrics collector sidecar container to the candidate Pod.
9. Metrics collector container tries to collect metrics from it and persists them into Katib DB backend.

Katib controller flow(Step10 to 11)



10. When the ML model job ends, Trial controller will update corresponding Trial CR's status.

11. When a Trial CR goes to end, Experiment controller will increase request field of corresponding suggestion CR, then go to step 4 again. If it ends, it will record the best set of hyper-parameters in .status.currentOptimalTrial field.

Demo

Demo

Thank you!

Further Resources

- Distributed Training:
  - https://github.com/kubeflow/tf-operator
  - https://github.com/kubeflow/pytorch-operator
  - https://github.com/kubeflow/mpi-operator
- Katib
  - https://github.com/kubeflow/katib