Model & Dataset CRD Design

Motivation

This document serves as a place for brainstorming ideas for Model & Dataset CRD design. The general goal is to design reusable CRDs that can be shared by various higher level machine learning tasks and frameworks.

Goals

- What do the CRD controllers do? Define the exact responsibilities of model & dataset CRDs and controllers.
- How will the higher level tasks, i.e. federated learning, model serving etc, utilize the services provided by model & dataset CRDs.
- Cloud edge communication mechanism for the CRD controllers: do they share the existing port 10000, or use a new port exclusively for AI purpose? Related: how do cloud workers and edge workers communicate? Cloud workers can be scheduled in cloud worker nodes, which means they can be deployed as a standard K8s service and have an publicly routable endpoint. Can KubeEdge operate in hybrid mode, i.e. having both cloud worker nodes and edge nodes?

Use Cases

Model serving

This is the simplest case. Upon creating a model CRD object, the edge part of the model controller should get notified and prepare the local workspace for the new model. By specifying the target model version to pull, the controller will download the corresponding model files (the whole directory can be compressed into a tar ball) from the provided URL endpoint.

Federated learning

Upon creating a model CRD object, edge needs to download the training scripts (runs on edge) and initial model weights files from provided URL endpoint. Once the gradients have been calculated, they should be reported back to cloud worker.

It looks like the common behavior of the model CRD object is just to download and upload data and executable files (of different types) to a cloud location, i.e. some storage service like AWS S3.

Model preprocess and postprocess

For example:

Preprocess: images need to be resized to be fit for CNN network input.

Postprocess: NMS module in the object detection framework, is used to delete highly redundant boxes.

The model inference service usually requires preprocess and postprocess scripts.

Model Metrics

For a trained model, model metrics (accuracy, precision, recall and etc..) usually indicate the performance of the model. We need to provide this feature for two purposes: i. For users who query model information he can get the model performance; ii. During model selection, the metrics of the model are used as a reference. Further, model metadata may also be used for model selection (for example, it is a object detection model or a facial recognition model? La bels are dog and cat, or car and plane?). Therefore, a metadata field may also be provided.

Support for different dataset format

Different data sets have different formats. Structured data is usually in CSV format, image classification TXT, and object detection XML format. Our crd needs to be aware of the format of the data set. For example, a basic feature of dataset management is to collect statistics on the number of samples, which requires awareness of different dataset formats.

For joint inference, is it necessary to consider as a unified model.

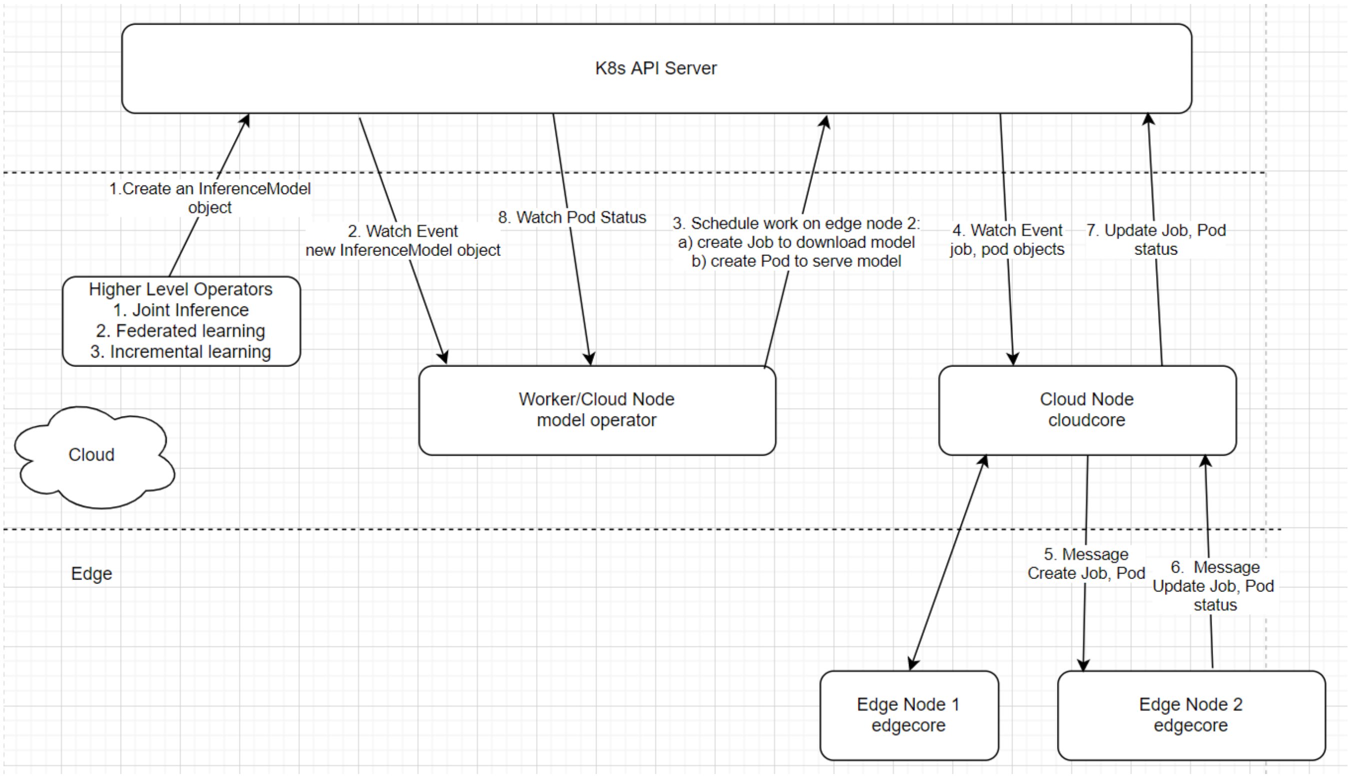
Joint inference aims to be a model on the cloud and a model on the edge. In the current solution, model serving is simple. User only need to train two models separately and deploy services separately. However, if a model is considered as a unified, can the interfaces for users to use the model be more simplified? Users are unaware that the backend are two models, just like loading a single side model for inference.

Cross-edge synchronization of models, datasets, and dataset metadata.

For some algorithms, cross-edge synchronization is required.   More details will be provided later.

Design Details

The model operator is supposed to run as a separate binary, fully decoupled from the KubeEdge platform code. It leverages the KubeEdge platform to schedule work on edge nodes. Here is the high level design diagram for the work flow:


K8s API Server
1.Create an InferenceModel object
2. Watch Event new InferenceModel object
8. Watch Pod Status
3. Schedule work on edge node 2: a) create Job to download model b) create Pod to serve model
4. Watch Event job, pod objects
7. Update Job, Pod status
Higher Level Operators 1. Joint Inference 2. Federated learning 3. Incremental learning
Worker/Cloud Node model operator
Cloud Node cloudcore
Cloud
Edge
5. Message Create Job, Pod
6. Message Update Job, Pod status
Edge Node 1 edgecore
Edge Node 2 edgecore

There are generally two phases for machine learning model development, i.e. training and inference. Model behaviors are quite different depending on whether it is used for training or for inference. So we might as well define two different types of model CRDs:

- InferenceModel
- TrainingModel

Also the whole system will run in a hybrid mode. By hybrid, we mean the KubeEdge cluster will manage worker nodes in both the cloud and the edge. For the cloud running components, we can leverage the full power of standard K8s, i.e. pods, deployments, services, ingress/egress, loadbalancer etc. This means cloud components will be deployed in a way exactly the same as done in a standard Kubernetes cluster. On the other hand, edge components leverage the KubeEdge framework.

InferenceModel CRD

InferenceModel CRD

```
// InferenceModelSpec defines the desired state of an InferenceModel.
// Two ways of deployment are provided. The first is through a docker image.
// The second is through a data store, with a manifest of model files.
// If image is provided, manifest and targetVersion will be ignored.
type InferenceModelSpec struct {
        ModelName     string `json:"modelName"`
        DeployToLayer string `json:"deployToLayer"`
        FrameworkType string `json:"frameworkType"`

        // +optional
        NodeSelector map[string]string `json:"nodeSelector,omitempty"`

        // +optional
        NodeName string `json:"nodeName,omitempty"`

        // +optional
        Image string `json:"image,omitempty"`

        // +optional
        Manifest []InferenceModelFile `json:"manifest,omitempty"`

        // +optional
        TargetVersion string `json:"targetVersion,omitempty"`

        // +optional
        // +kubebuilder:validation:Minimum=0
        ServingPort int32 `json:"servingPort"`

        // +optional
        // +kubebuilder:validation:Minimum=0
        Replicas *int32 `json:"replicas,omitempty"`
}

// InferenceModelFile defines an archive file for a single version of the model
type InferenceModelFile struct {
        Version     string `json:"version,omitempty"`
        DownloadURL string `json:"downloadURL,omitempty"`
        Sha256sum   string `json:"sha256sum,omitempty"`
}

// InferenceModelStatus defines the observed state of InferenceModel
type InferenceModelStatus struct {
        URL            string `json:"url,omitempty"`
        ServingVersion string `json:"servingVersion,omitempty"`
}
```

Sample inferencemodel instance

```
apiVersion: ai.kubeedge.io/v1alpha1
kind: InferenceModel
metadata:
  name: facialexpression
spec:
  modelName: facialexpression
  deployToLayer: edge
  frameworkType: tensorflow
  image:
  nodeSelector:
    kubernetes.io/hostname: precision-5820
  nodeName: precision-5820
  manifest:
    - version: '3'
      downloadURL: http://192.168.1.13/model_emotion_3.tar.gz
      sha256sum: dec87e2f3c06e60e554acac0b2b80e394c616b0ecdf878fab7f04fd414a66eff
    - version: '4'
      downloadURL: http://192.168.1.13/model_emotion_4.tar.gz
      sha256sum: 108a433a941411217e5d4bf9f43a262d0247a14c35ccbf677f63ba3b46ae6285
  targetVersion: '4'
  servingPort: 8080
  replicas: 1
```

An instance of the InferenceModel specifies a single serving service for the provided model.

Two scenarios:

- DeployToLayer == cloud, the controller will create a deployment with specified replicas, and a service to expose the deployment to outside world.
- DeployToLayer == edge, the controller will create a single pod running on the specified edge node (through NodeSelector)

Two ways of deployment

The InferenceModel CRD supports both of the following ways of deployment. If image is provided, manifest and targetVersion will be ignored.

Deployment method	Pros	Cons
Docker image	• Docker images encapsulate all the runtime dependencies • Very flexible as users can build whatever image they want • Docker will manage all the life cycles of the "model offloading"	• Users have to build the images themselves, write the Dockerfile, build the image and upload to a docker registry • Users have to provide a private docker registry if they don't want to use the public dockerhub.
Machine learning model files manifest	• Data scientists directly work with model files. It would be nice if they can just drop their model files somewhere • By using a data store, it opens the door for serverless computing	• Our framework has to manage the whole life cycles of model files deployment, update, delete, etc.

How can the InferenceModel CRD be used?

Simple machine learning offloading to edge

Just create an instance of InferenceModel with "DeployToLayer == edge"

Joint Inference by edge and cloud

Create three resources:

- An instance of InferenceModel to the cloud
- An instance of InferenceModel to the edge
- A pod running on the edge for serving customer traffic. It contains the logic for deciding whether or not to call cloud model serving API.

Joint inference by device, edge and cloud

We can have three models, with different size and accuracies, running on device, edge, and cloud respectively.