Kubeflow Serving CRD

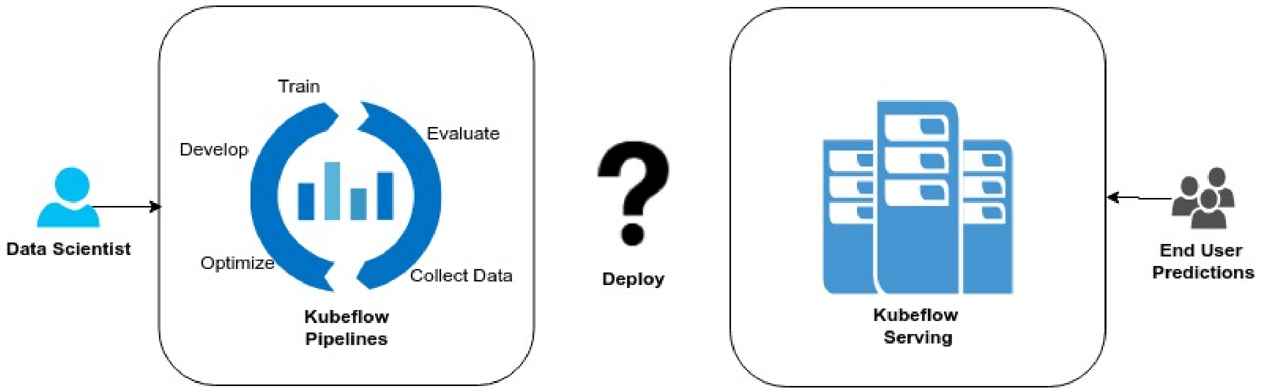
Status: Draft | In Review | **Approved** | Obsolete | Final
Authors: ellisbigelow@google.com
Reviewers: Kubeflow (kubeflow-discuss@googlegroups.com)
Date: 2019-1-17

NOTE: This is not an authoritative architecture for KFServing. The project has evolved significantly, but this is where it all began.

Objective

We propose this Kubernetes CRD interface for serving models on Kubeflow. The scope of this interface is not intended to provide a model lifecycle system, rather it is intended to serve as an integration point for Model Life Cycle and CI/CD systems like Kubeflow Pipelines. This design hopes to encapsulate a best practice architecture from shared experiences within Google and the wider Kubernetes community and demonstrate the best way to serve models from ML frameworks like Tensorflow, PyTorch, XgBoost, and SkLearn.

This aims to solve the problem of API standardization of Model Serving for both Control and Data Plane. The proposal outlines a starting point for Control Plane standardization and assumes standardized Data Planes for Tensorflow, XGBoost, and ScikitLearn. Over time this interface is intended to evolve towards additional frameworks and architectural components related to Model Serving (e.g. Payload Logging, Explainability, etc).



Note: *Other model serving architectures (e.g. Seldon, TFServing) may also be available in Kubeflow, but they will exist as siblings to Kubeflow Serving.*
Credits: *these CRDs were inspired by Knative's CRDs, as well as discussions with rhaertel@google.com, bhupc@google.com, lunkai@google.com, jlewi@google.com.*

Background

Overview

In 2019, ML Serving platforms have diverged interfaces for managing serving endpoints of ML models. Kubeflow, Google Cloud ML Engine, and Amazon Sagemaker have similar, but distinct approaches. These interfaces encapsulate the same information, but through different means. The complexity and divergence of these interfaces results in differing UX, feature potential, raises cognitive load on users. It's not clear which interface provides the best user experience. **Cloud ML Engine's** "Model" is quite similar to Sagemaker's Endpoint. Despite its name, it is effectively a router, sending all traffic to a default version. Compute resources are defined at the Version level, and Versions are available over the network, but are scaled to zero if not in use.

The interface is also imperative, which increases friction with CI/CD tooling compared to declarative APIs. When rolling back a "default version" of a model, the user must maintain their own system to be aware of and restore the previous version, rather than simply reverting a file to a previously declared configuration.

For **Kubeflow**, "TF Serving Components" are a bundle of kubernetes resources generated by user scripts. The API is declarative, which smoothly integrates with CI/CD tooling. However, since the resources are generated client side, users must be aware of the individual components and how they work together. This is great for flexibility, but leaked implementation details increases the complexity of managing the definitions as components evolve over time.

AWS Sagemaker has taken an alternative approach. Their concepts of "Model" and "Endpoint" allow users to create immutable "Models", and deploy them to arbitrary "Endpoints". Additionally, they allow n-way traffic splitting to enable more complex rollout scenarios. However, the UX is more complex, requiring multiple actions for even the most basic use cases.

Product	**Interface**
Google Cloud ML Online Prediction (CMLE)	● Model (Accessible as HTTP Endpoint) a. Name b. Default Version ● Version (Accessible as HTTP Endpoint) a. Assets (GCS) b. Docker Image c. Memory d. CPU/GPU
Kubeflow Serving	● TF Serving Component (Accessible as HTTP Endpoint) ○ Assets (GCS/S3/…)

	○ Docker Image ○ Memory ○ CPU/GPU
Amazon Sagemaker	● Model ○ Assets (S3) ○ Docker Image ● Endpoint (Accessible as HTTP Endpoint) ○ Traffic % -> Model ○ Memory ○ CPU ○ GPU

Goals

The proposed design aims to achieve both the simplicity of a “single-action” semantic, while simultaneously enabling use cases like model history, rollout, rollback, and canarying.

User Experience

1. Meet user expectations for a **Kubernetes Native** experience.
2. Provide a purpose built, ML-specific, **simplified, but flexible** serving interface.
3. Provide a **single action semantic** deployment, rollout, and rollback experience.
4. Provide a **canarying** mechanism to enable production-grade reliability.

Non Goals

1. **Experimentation/AB Testing**. While some serving platforms do provide arbitrary traffic splitting (Sagemaker), we have yet to see a compelling argument transparent (backend) n-way traffic splitting outside of the Canarying or Model Rollout case. We assert that A/B testing use cases are best handled client side or by a generic experiments framework.
2. **Infra Validation** involves preprocessing a model to determine viable/optimal runtime conditions for GPU/CPU/Mem. We do want to provide a CRD for Infra Validation, but this serving component should remain decoupled from that analysis.

User Journeys

The term “User” is to be understood as either manual user actions or automated pipeline steps.

Deploy your first Model
1. User develops and trains a model, or selects off-the-shelf model from TF Hub.
2. User creates an source-controlled YAML CRD file containing the model’s URI.

3. User applies the YAML file using a CLI or SDK.
    a. The CRD is Created.
    b. User may watch the CRD as it rolls out the deployment.
    c. The Service provides a network accessible address when ready.
4. User sends prediction traffic to the address.

Deploy a second revision of a model with no client downtime

1. User iterates on or retrains a model.
2. User edits their source-controlled YAML CRD file with the new model's URI.
3. User applies the YAML file using a CLI or SDK.
    a. The CRD is updated.
    b. User may watch the CRD as it updates the deployment.
4. User continues to send traffic to the address with no client interruption.

Rollback a deployment to a previous deployment

1. User detects issues with a current deployment.
2. User lists previous revisions of their deployment and finds a "golden" one.
3. User edits their "endpoint.yaml"
    a. Either, User reverts to a previous source-controlled revision of the yaml file.
    b. Or, User edits the current YAML file to point to a good revision of the deployment.
4. User applies the YAML file using a CLI or SDK.
    a. The CRD is updated.
    b. User may watch the CRD as it updates the deployment.
5. User continues to send traffic to the address with no client interruption.

Rollout fails and restores the previous deployment

1. User iterates on or retrains a model.
2. User edits their source-controlled YAML file with the new model's GCS URI.
3. User applies the YAML file using a CLI or SDK.
    a. The deployment fails.
    b. User may watch the CRD as it transitions from "Updating" to "Rolled Back".
4. User observes the CRD's error message and remedies the problem or retries.

Canary a revision of a deployment alongside an existing revision

1. User iterates on or retrains a model.
2. User edits their source-controlled YAML file and adds a canary deployment.
3. User applies the YAML file using a CLI or SDK.
    a. The CRD is updated.
    b. User may watch the deployment roll out alongside an existing deployment.
4. User edits their source-controlled YAML file to route % traffic to the canary.
    a. When the deployment is ready, a percentage of traffic is routed to it.
5. User continues to send traffic to the address with no client interruption.

Design

Kubernetes CRDs, by definition, are declarative and come with a common metadata structure and application pattern. CRDs provide a standard that meets requirements for our API without re-inventing the wheel. Therefore, most of these options are designed as Kubernetes CRDs.

Leveraging Kubernetes Name, Labels, and Annotations

Name

A kubernetes resource "name" is a unique identifier for the object as defined here. The format is readable, sanitized, and safe. The name identifier is used as the REST resource id when interfacing with CRUD requests.

Labels

Metadata labels provide a similar but more extensible mechanism for filtering Models and Versions. Users can use CRD labels to define groups of objects which can be efficiently queried from the Kubernetes API. This allows for users to separate groups of models (e.g. per user).

Annotations

Annotations are similar to labels, but are not used for filtering. Resources may be annotated with key-value pairs, which are typically used as references to related objects. We can use annotations to allow users to reference their pipelines or model lifecycle resources.

They may also be used as a flag to specify an alternative implementation of the CRD, enabling optional features like more efficient bin-packing for models, security settings, or other experimental features.

Introducing: Knative

In short, Knative is a recently open sourced Google project that provides a standardized interface and implementation for Serverless Applications on Kubernetes. It's purpose built for clean mechanisms for rollout, rollback, canarying, and observing container based applications.

After years of experience with Google AppEngine, they refined an interface of four components: Service, Configuration, Route, and Revision. These resources are generic, decoupled, and allow flexible mechanisms for managing Service lifecycle activities like Rollback, and Canarying.

For the majority of Knative use cases, “Operators” are recommended to only modify the Service object, which in turn, manages a set of Revisions, Routes, and Configurations. Operators will modify the Service document over time, which creates a snapshotted Revision for each change.

For Rollback cases, Operators may list Revisions and set their service to target the rollback revision. Alternatively, they may simply revert the definition from source control and reapply it.

For Canarying cases, Operators update the Service to pin the current revision and create a new canary revision. Once the canary revision is created, they may set a traffic split % on the new revision, sending some traffic to the new and some to the old. This is a two action semantic.

Option 1: Directly Support Knative’s API

We can directly adopt Knative’s interface. This decision would place a large amount of trust in the team to have architected their interface correctly, but if their decisions are correct, we benefit from their efforts.

Pros

1. Declarative API that cleanly supports Infrastructure as Code, Rollout, Rollback.
2. Built in support for Canarying.
3. Avoids reinventing the wheel and guessing a better serving interface for ML vs generic.
4. Compatible with both Serverless and Serverfull architectures.
5. Knative provides decoupled implementations for each of the underlying components.

Cons

1. We expose a very generic interface which may be over-complicated for ML users.
2. We tie ourselves directly to an interface we don’t control.
    a. Reliant on Knative’s Open Source release process.
    b. Any ML specific features must be implemented through annotations.
3. Knative has made some assumptions that may restrict our freedom
    a. Multiple containers are not supported at the interface, but sidecars are injectable.
    b. Resources requests (e.g., CPU,GPU) are in “pull-request” and not released yet.

Option 2: Extend Knative’s “Service” API (Recommended)

Overview

It doesn’t make sense to fully expose the immense flexibility of the Knative API to ML Users. Instead, we can follow the same interface structure, but abstract much of the configuration away, exposing a small set of ML specific parameters to configure.

We can think of our CRD as a facade of Knative's "Service" CRD. The Kubeflow "Service" CRD takes a small number of parameters like a GCS file, compute resources, ML framework, etc. We then transform those parameters into a complete Knative "Service" CRD with an opinionated configuration or our own implementation if desired (e.g., K8s Vanilla Service + Deployment).

We will provide a "Custom Container" approach, granting users the ability to directly specify an inline Knative Configuration, identical to the Knative's "Service" CRD. This allows users to customize, while maintaining simplicity for the common approach.

It's important to understand that Knative's interface intentionally excludes portions of the PodSpec from its interface. For example Knative excludes "Volumes" and "NodeSelectors". It is this design's intention to follow their leadership in simplifying the PodSpec, but where necessary, we can use MutatingAdmissionControllers to circumvent Knative's restrictions. This will be necessary for GPU-type selection with NodeSelectors in GPU-heterogeneous clusters.

Model Lifecycle Integration

While Model Lifecycle is out of scope of this interface, it's worthwhile to discuss planned integration points. As previously discussed, annotations and labels will be used to provide filtering and referencing mechanisms, but this design takes no opinions on specifically how they will be used.

Multiple Frameworks and Custom Containers

The definition must include one and only one configuration type. These configuration types map to framework, e.g: tensorflow, scikitlearn, xgboost. We will also enable "custom configuration" which is a passthrough Knative ConfigurationSpec. This exposes Knative details, but allows for complete deployment flexibility like custom docker image, env vars, or command.

Blue-Green, Pinning, Rollback, and Canarying

The definition will include Knative's modes: "release" and "runLatest". For more details about how "release" and "runLatest" work, see: "Service". Run latest simply deploys the current configuration as a blue-green deployment. This pattern allows users to follow a deployment pattern of Update CRD, Save CRD to Version Control, Apply CRD.

Release is a flexible mode which enables version pinning, rollback, and canarying. For example,

```
release:
  revisions = ["myapp.01", "myapp.02"]
  rolloutPercent: 50 # 50% of traffic goes to rev2
```

Example

Pros

1. Declarative API that cleanly supports Infrastructure as Code, Rollout, Rollback
2. Built in support for Canarying.
3. Control of the interface is in our hands, should we decide to diverge from Knative.
4. Provides a high-level ML specific interface for common use cases.
5. Provides a flexible path for users interested in tweaking the underlying structure.
6. Compatible with both Serverless and Serverfull architectures.
7. Knative provides decoupled implementations for each of the underlying components.
8. Brief discussions with the Knative team described this approach as “not crazy at all”.

Cons

1. Knative has made some assumptions that may restrict our freedom
    a. Multiple containers are not supported at the interface, but sidecars are injectable.

Option 3: Adopt Cloud ML Engine’s Model/Version API

The CMLE API is logically not far off from enabling our goals. We could maintain a similar structure and naming convention, and with a few minor tweaks, meet our goals.

1. Add a “Canary” parameter to Model, alongside defaultVersion.
2. Move CPU/GPU/Mem definitions from Version to Model’s “defaultVersion” config.
    a. Removing serverless assumption.
3. Make versions no longer directly accessible.
    a. Removing serverless assumption.
4. Create a declarative API (or even CRD) that wraps our imperative API.

This approach aligns us very closely to AWS Sagemaker’s interface where CMLE Model maps to a Sagemaker Endpoint and CMLE Version maps to a Sagemaker Model.

Pros

1. Declarative API that cleanly supports Infrastructure as Code, Rollout, Rollback.
2. Built in support for Canarying.
3. Potentially less disruptive to customers migrating from V1 API.

Cons

1. “Model” and “Version” concepts don’t make sense as currently named.
2. Not a “single-action-semantic”. Users must CreateVersion and then UpdateModel.

3. Wheel has been reinvented, resulting in a potentially worse interface than other options.

The original effort for this design took roughly the approach detailed above, including renaming the components, but this effort was eventually discarded, see [redacted].

FAQ / Alternatives

Why Not Standardize on Existing CMLE API

See option 3.

Why Not continue using Ksonnet Components

The CRD provides a cleaner interface than Ksonnet. Ksonnet is somewhat obtuse, particularly when configuring things like labels, annotations, etc. These serving CRDs are simple documents that generate underlying infrastructure server side, rather than client side. That said, Ksonnet can be useful to generate initial yaml, we can provide a ksonnet component that wraps this serving CRD, but it might not make sense to wire up every single field into the configuration. If a field exists in the Kubeflow Serving CRD, it should be meaningful to an ML Engineer or have a sane default that does not need to be defined.

Why Not Standardize on Existing Sagemaker API

Putting aside issues of whether or not we want to take a dependency on Sagemaker, this is not an effective path for other reasons. However, what we're proposing is not far off from Sagemaker's separation of concerns. Sagemaker's API is complex and allows strange features like n-way traffic splitting. It also is a multi-action semantic, making orchestration and rollout a painful process. Simply put, we think we can do better.

Why Not use Kubernetes.Service Interface?

Kubernetes has native resources for making servable endpoints like Service and Deployment. In fact, Kubeflow generates these resources directly for their TF Serving Component. While it is definitely a workable solution, we think we can do better by generating a higher level abstraction.

This approach is identical to customers doing a DIY solution on top of kubernetes.

Why Not use Istio’s VirtualService Interface?

Istio’s VirtualService is much closer to what we need. It comes with mechanisms for routing and canarying, but still relies on users to specify their own deployments. This does not meet our single-action-semantic goal and exposes too many implementation details to customers.

Why Not have a First Class Model Resource?

We absolutely want to support Model Lifecycle systems and provide features like Infra Validation, but this should be a loosely coupled integration point, rather than a tightly coupled component of the interface. We want to be compatible with Kubeflow Pipelines, homerolled model management systems, or basic off the shelf models without a model management system.

Why Not Build a First Class “Rollout” Semantic?

We experimented with the idea of a Rollout CRD. The main function of this CRD would be to automate some of the actions a User takes as they deploy a Model across different Endpoints, potentially with canarying steps as well. However, in order for this automation to be useful, we would need to implement some sort of “proceed if ok” logic. We found that this level of decision making was difficult to make without other systems and our implementation would simply force the user to implement these hooks.

We’re not opposed to a Rollout CRD in the future, but in the current iteration, it’s unlikely that one we build now would be useful, and it would likely add confusion for the user.

Appendix A: Example Spec for Kubeflow.Service

Full Example

```
apiVersion: serving.kubeflow.org/v1alpha1
kind: Service
metadata:
  name: flowers-endpoint
  namespace: ...
  labels: ...
  annotations: ...
spec:
  release # Pick one of [release, runLatest]
    revisions: ["current", "candidate"]
```

```
rolloutPercent: 50
# Pick one of [ configuration, tensorflow, scikitLearn, xgBoost ]
configuration: # knative.ConfigurationSpec for byo-container use cases
tensorflow: # transform to kubeflow.TensorflowSpec into knative.ConfigurationSpec
  serviceAccount: myaccount.googleserviceaccounts.com
  modelUri: gcs://mybucket/modelDir
  cores: 2
  memory: 10gb
  accelerator:
    Type: nvidia-k80
    count: 1
  version: 1.12
scikitLearn: # transform to kubeflow.ScikitLearnSpec into knative.ConfigurationSpec
  ...
xgBoost: # transform to kubeflow.XGBoostSpec into knative.ConfigurationSpec
  …
```