Model Cache Proposal

Shared with KServe Community

Owner: Jin Dong (https://github.com/greenmoon55/) **Working Group:**	**Status**: **WIP** | In Review | Approved | Obsolete **Created**: YYYY-MM-DD **Approvers**: WG-lead-X [], WG-lead-Y [], ...

Motivation / Abstract

[**This should be a short summary of the proposal - 1 paragraph max**. Include the following:

1. Which persona(s) the feature is for
2. What problem is being solved (Problem Statement)
3. What new capability is provided/improved (Feature/Capability)

This should give dux, test, and WG leads an ability to assess how the feature fits into the project.]

LLM models can be over 100 GB, which takes a long time to download, resulting in long service startup time. We would like to introduce a “model cache” which allows operators to specify a list of models to be cached on local disks or other PVs like EFS. For example, we cut down service startup time from 15-20 mins to ~1 minute by caching the Llama3 70B model locally. It also benefits autoscaling as the service can scale up faster.

Background

[Broader background regarding the problem being solved by the proposed feature. Goals / Non-Goals. This is the "what" - The "How" is described in Proposal Design / Approach.]

Before talking about reducing inference service start up time by caching models, let’s review a few common ways to download a model.

1. Build the model into a custom image (not in this doc)
2. Have a model ready in a PV and mount it to kserve-container
3. Downloads the model in storage-container (an init container) before kserve-container starts

We are going to make #2 and #3 better by pre-downloading models to your PV or local disk on nodes before an inference service is created, so we can skip downloading models in the storage initializer. For #1, you might want to use kube-fledged or similar tools to cache models on local nodes.

Overall we would like to support two use cases.

1. **Pre-download models to local disks, and models can be shared across namespaces.**
2. **Leverage custom StorageClass to create PVs and pre-download models to PVs. These models cannot be shared across namespaces.**

Goals

1. For local PVs, cached models can be **shared** across namespaces. Otherwise cached models can only be used in the **same** namespace.
2. Cluster admin manages which models can be cached by a cluster-scoped model cache custom resource. Users can create a namespace-scoped model cache CR with a custom storageclass (e.g. network volume).
3. Cluster admin can choose which nodes to cache a model, like a specific GPU type. The admin can set a limit on the maximum size of the cache per node.
4. Model cache controller downloads models to the PV and deletes the model when the CR is deleted. It ensures a model is not deleted when they are in use.
    a. For local PV, the controller deletes the model in local disks
    b. For custom storage class, controller deletes the PVC and then the PV can be deleted automatically (Reclaim policy)
5. Users can specify original storage uri in their isvc yaml. KServe converts it to PVC uri based on model cache custom resources.
6. Support custom storage protocol
7. Throttle download bandwidth

Non-Goals

1. Sharing across namespaces is not supported for custom storageclass.
2. KServe does not automatically evict models from the cache. Technically this is not a cache.
3. Models under a storage uri (like a S3 path) are considered immutable. Model Cache Controller would not redownload models.
4. Authn and Authz for models

Proposal Design / Approach

["How" the feature is going to work, is designed, implemented, etc. This should be written for an average contributor in the WG area.]

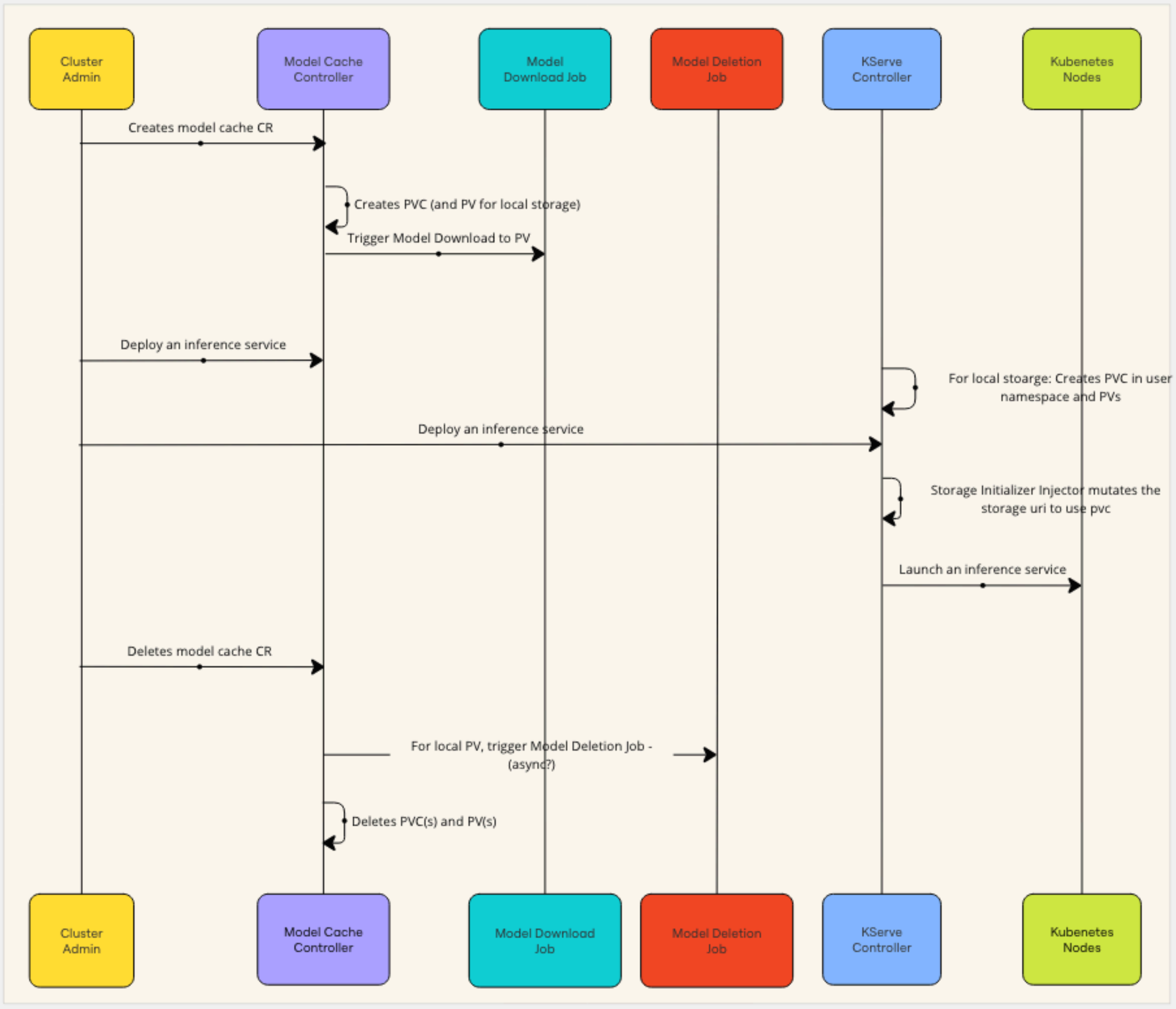
Design

[Design details for the feature at the resource model level. Details such as; How the feature should work, sequence diagrams, schema-level changes, failure modes. For user facing features, this section shouldn't contain code.]

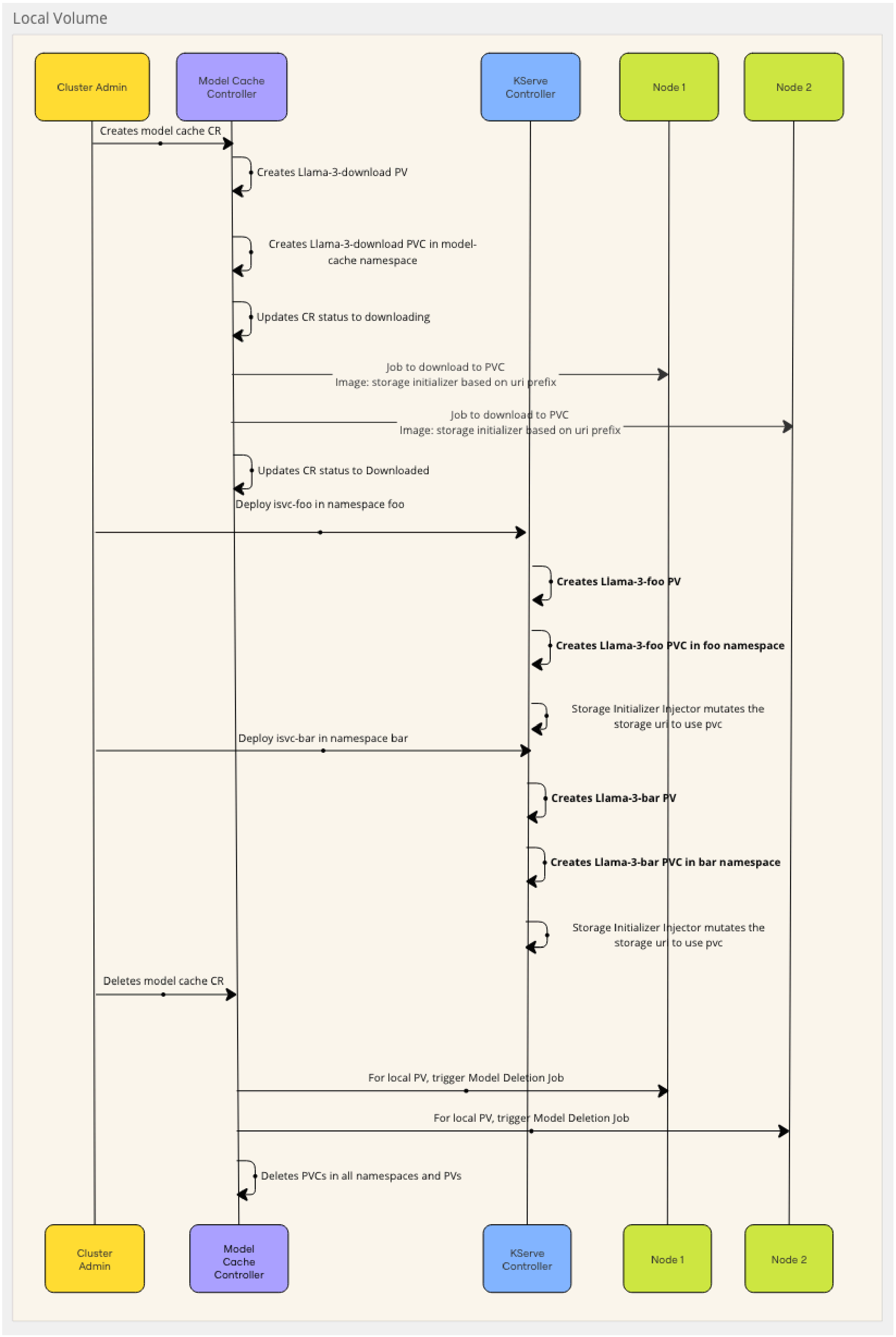
1. Workflow

Here's the overall workflow for caching a model, deploying an isvc and then deleting the model. For details, please check out the next section PV and PVC Management.

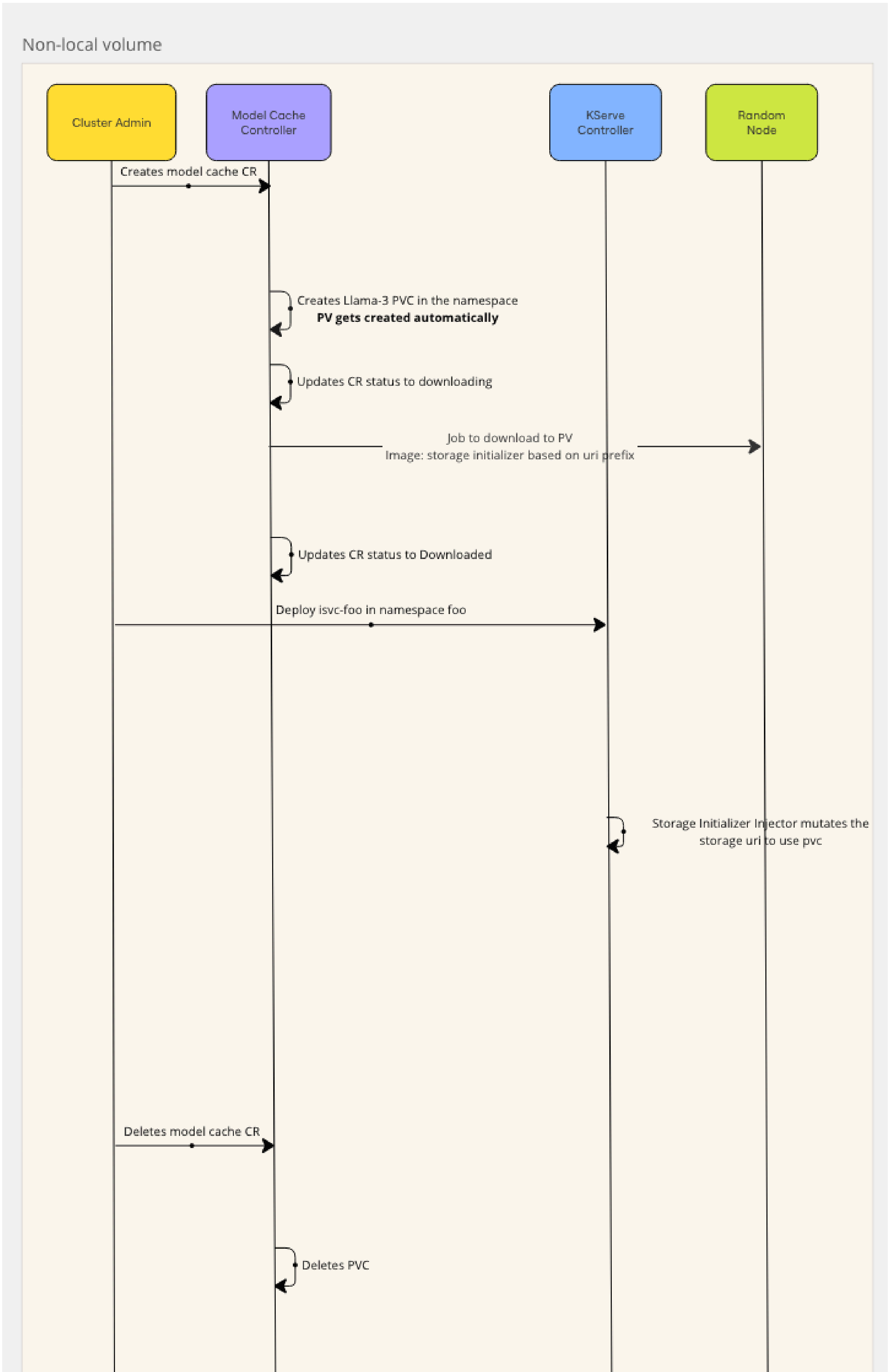
Overall workflow


Cluster Admin
Model Cache Controller
Model Download Job
Model Deletion Job
KServe Controller
Kubenetes Nodes
Creates model cache CR
Creates PVC (and PV for local storage)
Trigger Model Download to PV
Deploy an inference service
For local stoarge: Creates PVC in user namespace and PVs
Deploy an inference service
Storage Initializer Injector mutates the storage uri to use pvc
Launch an inference service
Deletes model cache CR
For local PV, trigger Model Deletion Job - (async?)
Deletes PVC(s) and PV(s)
Cluster Admin
Model Cache Controller
Model Download Job
Model Deletion Job
KServe Controller
Kubenetes Nodes

Workflow for caching on local disks


Local Volume
Cluster Admin
Model Cache Controller
KServe Controller
Node 1
Node 2
Creates model cache CR
Creates Llama-3-download PV
Creates Llama-3-download PVC in model-cache namespace
Updates CR status to downloading
Job to download to PVC Image: storage initializer based on uri prefix
Job to download to PVC Image: storage initializer based on uri prefix
Updates CR status to Downloaded
Deploy isvc-foo in namespace foo
Creates Llama-3-foo PV
Creates Llama-3-foo PVC in foo namespace
Storage Initializer Injector mutates the storage uri to use pvc
Deploy isvc-bar in namespace bar
Creates Llama-3-bar PV
Creates Llama-3-bar PVC in bar namespace
Storage Initializer Injector mutates the storage uri to use pvc
Deletes model cache CR
For local PV, trigger Model Deletion Job
For local PV, trigger Model Deletion Job
Deletes PVCs in all namespaces and PVs
Cluster Admin
Model Cache Controller
KServe Controller
Node 1
Node 2

Workflow for caching on a network volume with custom storage class


Non-local volume
Cluster Admin
Model Cache Controller
KServe Controller
Random Node
Creates model cache CR
Creates Llama-3 PVC in the namespace
PV gets created automatically
Updates CR status to downloading
Job to download to PV
Image: storage initializer based on uri prefix
Updates CR status to Downloaded
Deploy isvc-foo in namespace foo
Storage Initializer Injector mutates the storage uri to use pvc
Deletes model cache CR
Deletes PVC

2. PV and PVC Management

KServe provides two levels of support for PV/PVC creation.

1. KServe manages both PV and PVC. This works for local models. PV spec and PVC spec are customizable.
    a. This should work for S3 Mountpoint CSI driver where they have a PV per S3 bucket.
2. KServe only manages PVC. This is mostly for non-local models where PV is created by the provisioner in a storageClass.

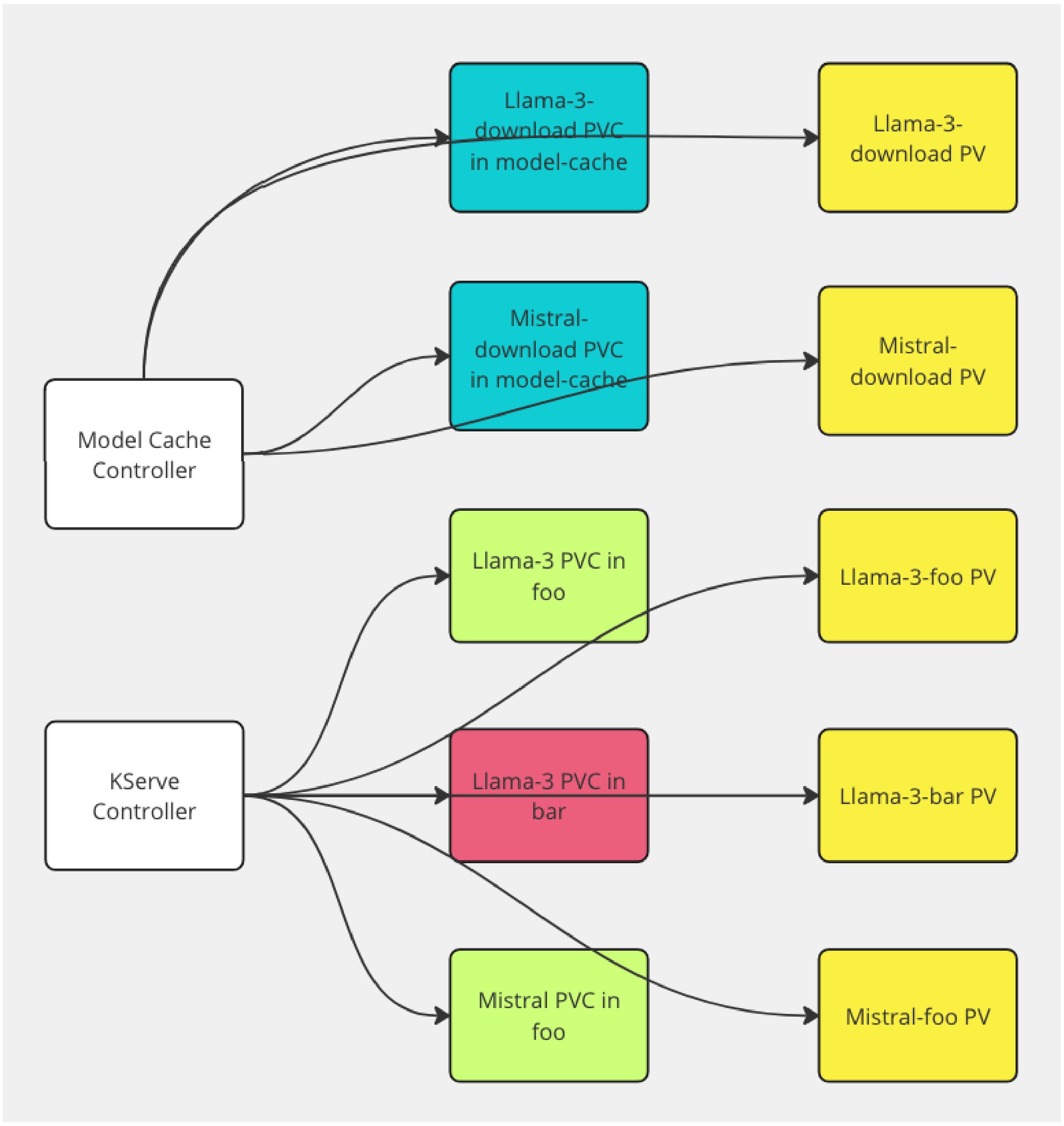
Model cache controller creates and deletes PVCs for model cache resources. For local models, the model cache controller also creates PVs. If the same model is shared in multiple namespaces, there would be two PVs with the same local path because one PV can only be bound to one PVC.

For example, suppose we want to cache the Llama-3 model in the namespace foo and bar.

1. Local models

As you can see in this diagram above, the model cache controller creates `Llama-3-download` PV and `Llama-3-download` PVC in the controller namespace to download the model before service is created.
Since we have two inference services in Namespace foo and bar. We would need two PVs: `Llama-3-foo`, `Llama-3-bar` and two PVCs in each namespace. These PVs and PVCs can be created when the inference service is deployed.

(Preferred) Approach #1: One PV per model



Model cache controller creates the PV and PVC for a namespace when the inference service is deployed. The PV and PVC are deleted when no inference service uses cached models. Model cache controller deletes all PVCs and PVs when the model cache CR is deleted.

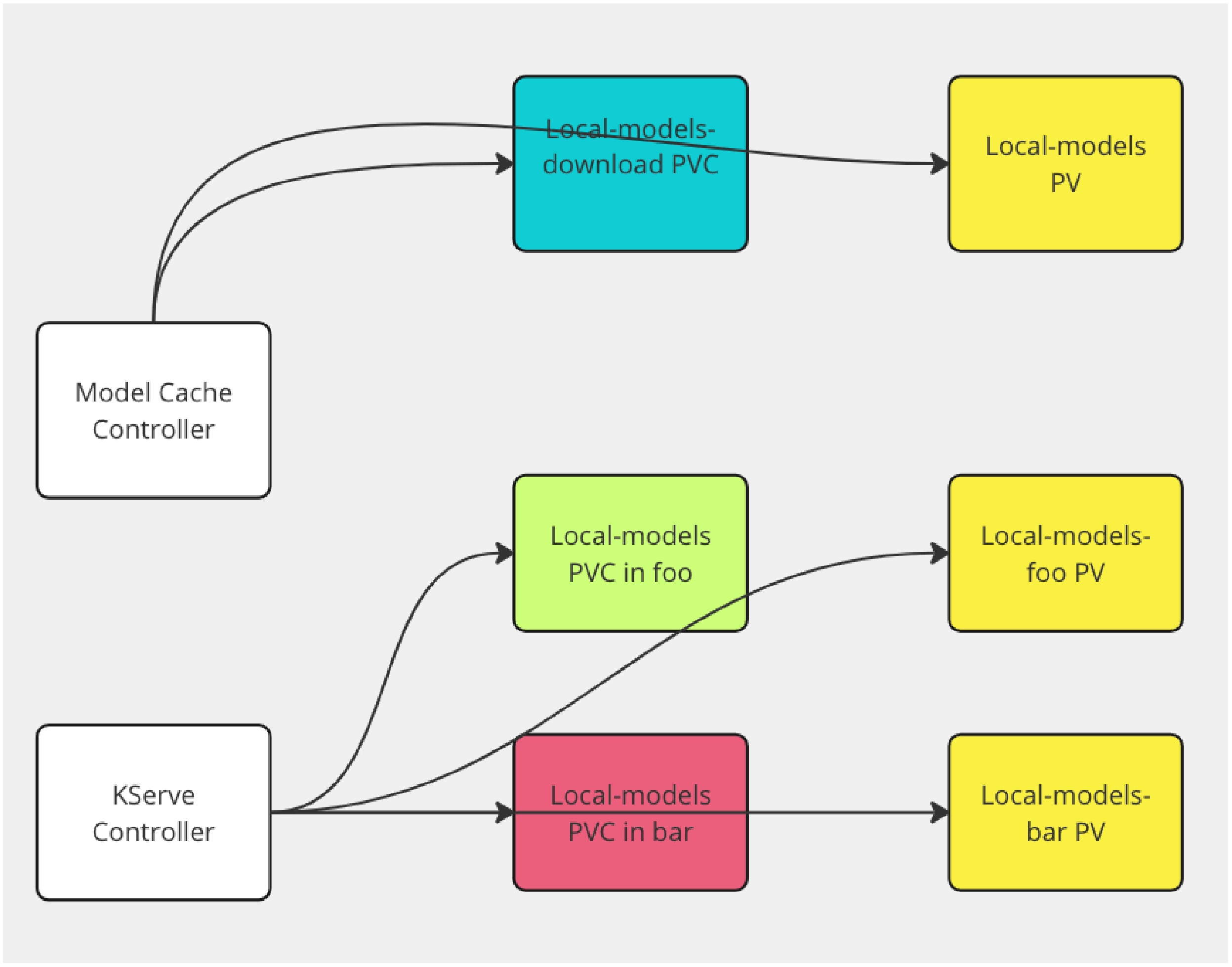
Pros:

1. Similar to non-local model behavior
2. `kubectl get pvc` shows if local models are used in a namespace

Cons:

1. For local PVs, we need (# models) * (# of namespaces) PVs and PVCs because one PV can only be attached to one PVC.

Approach #2: One PV for all models



Pros:

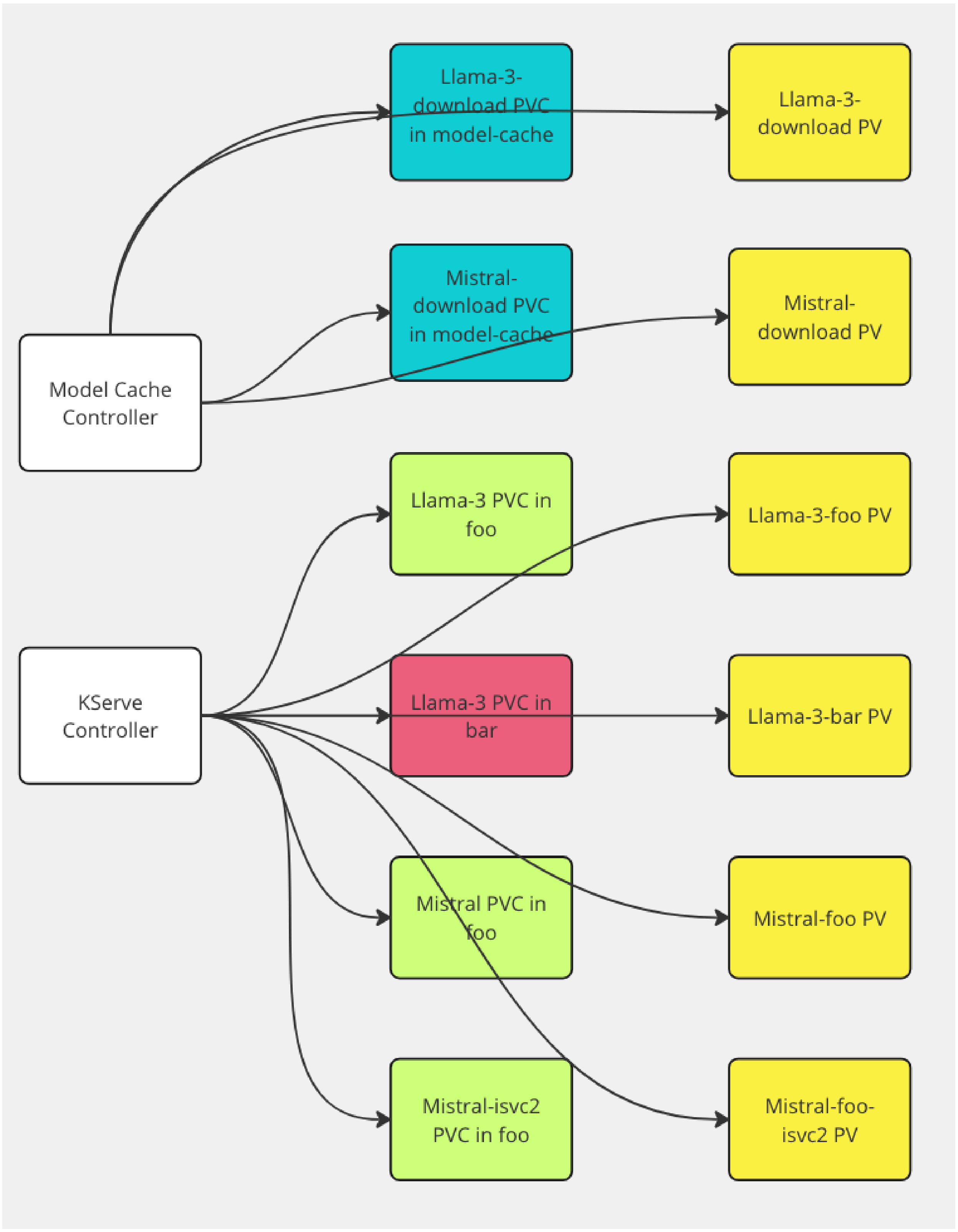
1. For local PVs, we need (# of namespaces) PVs and PVCs instead of (# of namespaces) * (# of models).
2. More like S3 Mountpoint CSI driver where each PV corresponds to a bucket.

Cons:

1. It can get tricky to manage if two models use the same PV.
2. Security and observability

Approach #3: One PV per inference service

Now with mistral-isvc2, we need another PVC and PV pair.



Pros:

1. PV & PVC deletion is simple because they are not shared among different inference services.

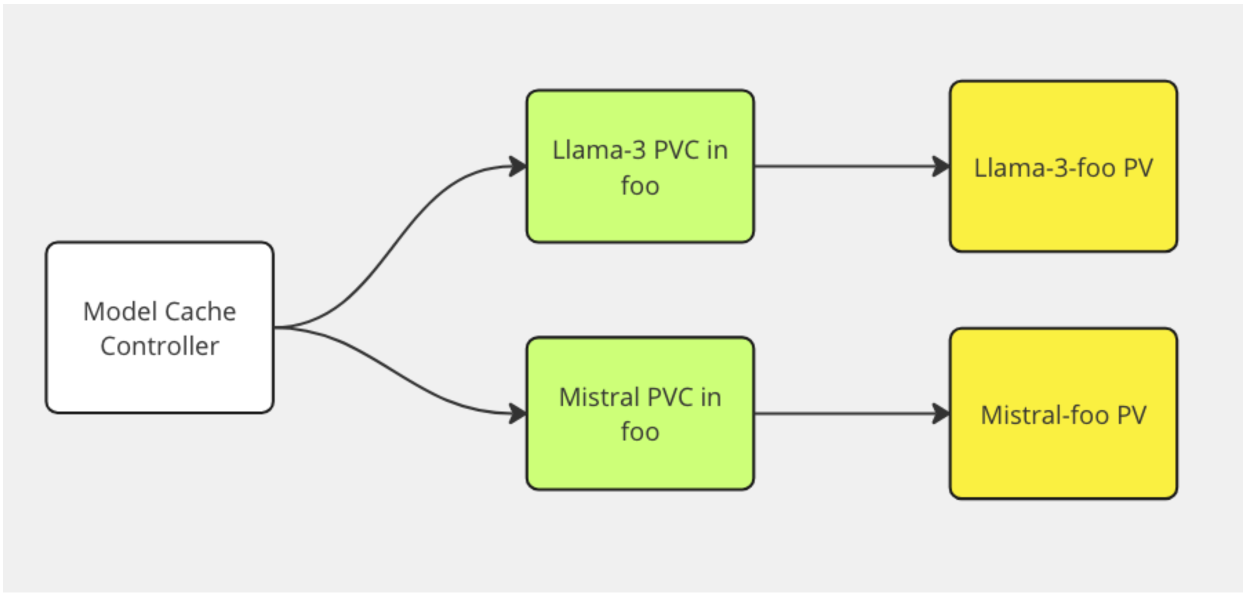
Cons:

1. Too my PVs and PVCs in the namespace.
2. Different from non-local models where one model should map to one PV only.

2. Non-local models

For non-local models (this workflow), the PV is created by the PVC which is created by the model cache controller. So there is only one PVC `Llama-3` in the namespace. Inference services using the same model in the same namespace can share the same PVC. However, sharing the same model across namespaces is not supported.

In this case, PVCs are created by Model cache controller instead of KServe controller.



3. Custom Resource Definition

Cluster Model Cache Custom Resource

```
apiVersion: serving.kserve.io/v1alpha1
kind: ClusterModelCache
metadata:
  name: llama-3-70b
spec:
  storageUri: 's3://llm/llama-3-70b/'  // Immutable
  modelSize: 140Gi
  nodeGroup: GPU-1
  pvSpec:
  pvcSpec:
status:
  storageStatus: pvcCreated/Downloading/Ready(Downloaded on all
nodes)/Deleting?
  nodeStatus:
     node1: ready
     node2: downloading
```

Node group custom resource

This is for local models only. Node group CR is used to decide where we should cache the model and the storage limit per node.

```
apiVersion: serving.kserve.io/v1alpha1
kind: ModelCacheNodeGroup
metadata:
  name: GPU-1
spec:
  storageLimit: 500Gi
  nodeSelector:
    nvidia.com/gpu.product: NVIDIA-xxx
```

Namespace-scoped Model Cache Resource

Without nodeGroup and nodeStatus

```
apiVersion: serving.kserve.io/v1alpha1
kind: ModelCache
metadata:
  name: llama-3-70b
spec:
  storageUri: 's3://llm/llama-3-70b'  // Immutable
  modelSize: 140Gi
  pvSpec:
  pvcSpec:
status:
  storageStatus: pvcCreated/Downloading/Ready(Downloaded)/Deleting?
```

Inference service

```
annotations:
  serving.kserve.io/enable-model-cache: true
spec:
  predictor:
    storageUri: s3://models/llama-3
```

4. Downloading models to the PV

(Preferred) Approach #1: Kubernetes Job

Kubernetes Job that runs storage initializer containers to download models. We can select different images to run based on ClusterStorageContainer CRs.

Pros:
1. Easy to implement - Reuse ClusterStorageContainer CRD to support custom images.
2. Low overhead compared to daemonset

Cons:
1. Cannot detect files deleted on local disk
2. Reconcile on new node joining

Approach #2: DaemonSet

Pros:
1. New node joining - pick up download job immediately

Cons:
1. Additional resources
2. A bit tricky to support custom storage initializer images.
3. Difficult to limit throughput, i.e. number of nodes downloading at the same time.

5. Cleaning up models

When the model cache CR is deleted, the model should be deleted from the PV. For local models, the model cache controller would schedule a deletion job to delete all modes on all nodes. For network volume, the controller deletes the PVC and then the PV can be deleted automatically by setting reclaim policy to delete.

Model cache controller would reject deletion requests if the model is used by any inference service. This can be done by checking annotations on all inference services.

6. Pod scheduling

This section only applies to local models. Pods may be scheduled on nodes where the model is not cached.
Case #1: Pod getting scheduled on a node not in the NodeGroup because the user attached a different node label or nodeSelector in the pod spec in the isvc yaml.
Case #2: Model is not on the node because download is not finished, network issues or disk issues.

For #1, the pod cannot be scheduled to a node because the local PV does not exist.
For #2, the pod will be in CLBO if the container crashes when it cannot get the model locally. It would not be rescheduled.

We have three options here

1. Make sure pod do not get scheduled on nodes without models
2. Crash the pod if model is not on local disks
3. Fallback to default download behavior

Approach #0: KServe does not affect scheduling

Pros:

1. Keep it simple - do not mess with scheduling

Cons:

1. Some serving runtimes may not terminate if the model does not exist, so the inference service may report ready status.

Approach #1: Add node affinity to pods to schedule on ready nodes

```
affinity:
  nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
      - key: kubernetes.io/hostname
           operator: In
           values:
```

```
          - node1
          - node2
```
Pros:

1. In most cases, the controller injects a list of nodes that are not ready, the pod would not be scheduled to these nodes unless the node status is not up to date.

Cons:

1. Complexity - we influence scheduling decision
2. Pod get stuck on the list of nodes after getting killed. The list is not up to date.

Approach #2: Add model name label to nodes and use nodeSelector in Pods

Model cache controller add labels to nodes

```
kubectl label nodes <your-node-name>
model-cache-ready-llama-3-70b=true
Pod:
nodeSelector:
  model-cache-ready-llama-3-70b: true
```
Pros:

1. Compared to Approach #1, if the pod gets killed, it can be scheduled on other nodes using the latest model ready status

Cons:

1. We would add many node labels.

Approach #3: Use lifecycle poststart handler to crash the pod if model is not on local disks

Example: `test -f /mnt/models/model1/.completed`

Similarly

Pros:

1. Easy to implement

Cons:

1. Assuming the user container has commands like “test”, otherwise we can build a small image and use it as an init container.
2. May be stuck in CLBO - one node doesn’t have a model - same node.

Approach #4: Fallback to download in the storage initializer

Pros:

1. May reduce the chance of failure. However this is debatable since the storage initializer are likely to fail if the download job has failed.

Cons:

1. Long startup time - we might prefer fail early
2. Need to handle race condition - too much complexity

7. Changes to Inference Service Spec

Cluster admin can enable/disable model cache in the config map. Users can [set an annotation]() to enable/disable model cache per service.

For storage uri, we can either keep the original storage uri or introduce a `cache://` prefix.

(Preferred) Approach #1: Keeping the original storage uri

Pros:

1. Users can deploy the same isvc yaml when the model cache CR is disabled or deleted
2. Users can see the original storage uri easily

Cons:

1. Users may expect the model to be cached when it is not. There would be no errors when creating the inference service. To mitigate this, we can add an annotation on the pod or a status on the isvc.

Approach #2: Use `cache://model-name`

Pros and cons are the exact opposite of the first approach.

Implementation

[Where is the code going to live? What directories are impacted / changed]

Major changes

1. New CRDs
    a. ClusterModelCache CRD, NodeGroup CRD, Namespace-scoped ModelCache CRD.
2. Model Cache Controller to create model download/deletion jobs, create/delete PVs, and manage Model Cache resources.
3. KServe controller
    a. For local model cache, KServe controller creates PV and PVC when an inference service is deployed.
    b. [TBD] KServe controller modifies storage uri to use pvc
4. [TBD] Storage initializer image writes a COMPLETION file so we know whether the model is on the PV. The lifestyle poststart handler can crash the pod.
5. [TBD] KServe controller may add nodeLabels to the pod to make sure the node has the model.

Milestones

Milestone 1: Basic support for local model cache, pods can be scheduled on nodes without models

1. [P0] ClusterModelCache CRD and NodeGroup CRD

2. [P0] Model Cache controller that triggers jobs for downloading and deleting files, and updating status.
3. [P0] KServe controller mutates the storage uri, and creates PV and PVC when an inference service is deployed.

Milestone 2: Handle cases where some node may not have the model

1. Update Storage Initializer Image to write completion file, or poststart hook/custom init container, scheduling (node affinity)

Milestone 3: Support custom storage class in a single namespace

1. Namespace-scoped ModelCache CRD to support PVC with any Storageclass.

Prerequisites / Dependencies

[Are there any issues / tech that need to be in place for this to work?]

Integration Checklist

Operations

[How is this feature implemented or turned on by the user / operator?]

Observability

[Will this feature need instrumentation or measures that are exposed to specific personas? If so, which personas and optics are needed?]

Test Plan

[How is the feature tested for use? i.e unit testing, E2E, isolated or in conjunction with other components? that conformance tests need to be in place?]

Documentation

[What personas will use this feature and which documented use-cases does this affect? Are there new use-cases that need to be written or existing ones edited?]

Exit Criteria

[What are the requirements to exit each stage]

Alpha

[exit criteria]

Beta

[exit criteria]

GA

[exit criteria]

Alternatives Considered

[What other approaches to solving this problem were considered? What rationale was used to select the specific design over other methods?]

Mapping between models and PVs

Suppose we have M models to cache and they are used in N namespaces.

(Preferred) Approach #1: One PV per model

Pros:

3. For non-local PVs, we can just delete the PV once the model cache is deleted
4. Make sure one model would not affect another.

Cons:

2. For local PVs, we need N * M PVs and PVCs because one PV can only be attached to one PVC.

Approach #2: One PV for all models

Pros:

3. For local PVs, we need N PVs and PVCs instead of N*M.

Cons:

3. It can get tricky to manage if two models use the same PV.

Appendix

How does KServe download a model?

KServe injects an init container to provision model data for the serving container, it also creates a shared volume ([emptyDir]) between the storage-initializer and kserve-container.

```
        // Create a volume that is shared between the storage-initializer and kserve-container
   sharedVolume := v1.Volume{
        Name: StorageInitializerVolumeName,
        VolumeSource: v1.VolumeSource{
                EmptyDir: &v1.EmptyDirVolumeSource{},
        },
   }
   podVolumes = append(podVolumes, sharedVolume)

   // Create a write mount into the shared volume
   sharedVolumeWriteMount := v1.VolumeMount{
        Name:       StorageInitializerVolumeName,
        MountPath: constants.DefaultModelLocalMountPath,
        ReadOnly:  false,
   }
```

```
KServe container:
        - mountPath: /mnt/models
        name: kserve-provision-location
        readOnly: true
Storage initalizer:
        volumeMounts:
        - mountPath: /mnt/models
        name: kserve-provision-location
 volumes:
 - emptyDir: {}
        name: kserve-provision-location
```

How does Kubernetes local Persistent Volumes work?

Kubernetes Persistent Volume is a piece of storage provisioned by a StorageClass. Kubernetes supports local Persistent Volumes with this storageClass. With this, Kubernetes waits to bind a PVC until a Pod using it is scheduled.

Example

We can provision a local PV and a PVC in namespace foo. Then we can specify the pvc in the storage uri of the inference service yaml. KServe controller would skip the storage initializer and attach pvc directly to the KServe container.

Persistent Volume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-models-pv
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
          storage: 1700G
  claimRef:
          apiVersion: v1
          kind: PersistentVolumeClaim
          name: local-models
          namespace: s-foo
  local:
          path: /opt/bb/models/
  nodeAffinity:
          required:
          nodeSelectorTerms:
          - matchExpressions:
          - key: nvidia.com/gpu.product
          operator: In
          values:
          - NVIDIA-XYZ
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  volumeMode: Filesystem
status:
  phase: Bound
```

Persistent Volume Claim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
          pv.kubernetes.io/bind-completed: "yes"
  name: local-models
  namespace: foo
spec:
  accessModes:
  - ReadWriteOnce
  resources:
        requests:
        storage: 1700G
  storageClassName: local-storage
  volumeMode: Filesystem
  volumeName: local-models-pv
status:
  accessModes:
  - ReadWriteOnce
  capacity:
        storage: 1700G
  phase: Bound
```

Inference Service

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: llama-3-70b-instruct
  namespace: foo
spec:
  predictor:
    model:
      storageUri: pvc://local-models/llama-3-70b-instruct/huggingface/
```

Pod

KServe Container

```
  volumes:
  - name: kserve-pvc-source
        persistentVolumeClaim:
        claimName: local-models
volumeMounts:
        - mountPath: /mnt/models
        name: kserve-pvc-source
        readOnly: true
        subPath: llama-3-70b-instruct/huggingface/
```

1. One PV can only be used by one PVC even for local PVs.
2. PVC are namespace scoped but not PVs.
3. To enable sharing between namespaces, we can point two PVs to the same folder

Discussions

Jun 5, 2024

1. PV and PVC management
    a. Support Custom StorageClass
    b. For local PVs, we need n pvs and n pvcs where n is the number of namespaces.
        i. An annotation on the namespace? Feature flag
    c. Support a custom PVC/PV?
    d. When a new node joins. No action is required.
2. Can we avoid downloading models in the storage initializer by tainting nodes?
    a. Approach #1: download models in both places because some nodes may not have the model, e.g. when a new node joins
        i. Cons: Race conditions
            1. Two pods on the same host download at the same time.
            2. Storage initializer starts while the model is being cached by Job/DaemonSet
        ii. Write down a time and temp files. However it's still very messy.
    b. Approach #2: For each model, the controller updates cache status for each node. KServe controller will add taints to nodes or tolerations.
        i. Cons: Per model taints/tolerations? Would it be too many?
        ii. Make sure it doesn't evict existing pods
3. Support custom model registry. Should be easy to manage custom model download code and authorization.
    a. Would be nice if we can reuse "ClusterStorageContainer" CR and use custom images for downloading
4. Daemonset or Job?
    a. Common requirement: manage throughput
    b. DaemonSet
        i. Pros: Report node status
        ii. Cons: Resource usage?
    c. Job:
        i. Pros: does not require resources?
        ii. Cons:
            1. Delay in updating resources or reporting status
            2. Same job for updating all models vs 1 model
    d. As Zoram mentioned, when a node joins, daemonset makes more sense, however a job can work too if a controller starts a job for a new node or we have fallback on the storage initializer.
5. Auth - same credential for all cached models?

        a. Configure envvar for the job or the image used to download.
            i. Is this enough?
6. Manage storage limit and node affinity
    a. Some models are cached on GPU Type X, Some on GPU Type y
        i. KServe does not influence scheduling?
        ii. Manage model limits based on a label.
            1. New model cache CR: do we have enough disk space for GPU Type X.
    b. Model size can be misleading because some models can be compressed?
        i. Ask people to provide accurate size?
7. Garbage collection
    a. When do we delete models from the pvc?
        i. When the CR is deleted?
            1. Do not allow deletion when the uri is in use?
                a. What if an old revision still uses it? Ignore?
        ii. When the inference service is deleted?
8. Handle race conditions - we can put temp files on cache?
    a. Not a issue if we taint nodes
    b. Two pods on the same host download at the same time.
    c. Storage initializer starts while the model is being cached by Job/DaemonSet
9. Provide options
    a. From Zoram - make storage class an optional part of the spec, and also PV access mode (RWO/RWX). Node selector should be optional in case one goes for an RWX PV, such as NFS or CephFS.

Daemon Set/Job

1. Pre-download model when model does not exist
2. Delete models on the pv when the ModelCache CR is deleted
3. KServe controller reconciles creates PVCs in all namespaces

Config Map
Auth - some env var for the daemonset, which would reuse storage initializer code
Secret_name

```
Pvc name:
 nodeSelectorTerms:
      - matchExpressions:
      - key: nvidia.com/gpu.product
      operator: In
      values:
      - NVIDIA-xxx
Pvc size
```

semaphore

Status: which nodes have this model