IBM Watson

Model-mesh: Production model serving at scale

Nick Hill
nickhill@us.ibm.com

IBM

Introduction

- The **model-mesh** framework is a mature, general-purpose model serving management/routing layer
- Designed for production high-scale, high-density and frequently-changing model use cases
- Intended primarily for “model as data” use cases as opposed to “model as code”
- Especially useful where there’s a high degree of usage variability between models
- Underpins many of the IBM Watson AI cloud services/apps including Watson Assistant, Watson Discovery, Watson Natural Language Understanding

Background

- 2015: IBM Natural Language Classifier was the first Watson Cloud service to provide automated end-to-end training and serving of customer-provided data
- Original architecture involved spawning a dedicated container to serve every trained customer model
- Problem:
  - Was growing to tens of thousands of models and didn’t scale well
  - Free plans - many rarely used or effectively abandoned
  - Huge resulting memory footprint => hardware footprint => cost
- Pushing limits of scheduler => seriously impacting performance and stability
- Single point of failure for *all* served models (each only running in one container/VM)
- Excessive time to recover from large-scale outages
- Before Kubernetes and Serverless were ubiquitous

Model-mesh

- **New approach**
  - Serve multiple models per container/process
  - Allow dormant models to be "paged out", loaded just-in-time if/when used again
  - Footprint can be orders of magnitude smaller than that required to serve all models simultaneously
  - Similarities with "serverless" infrastructures (knative , Cloud Functions, …)
- **Philosophy**
  - Decouple and hide all of the (common) concerns of serving production models at scale from the service-specific inferencing logic/technology
  - Do these things once, really well

Key Design Goals

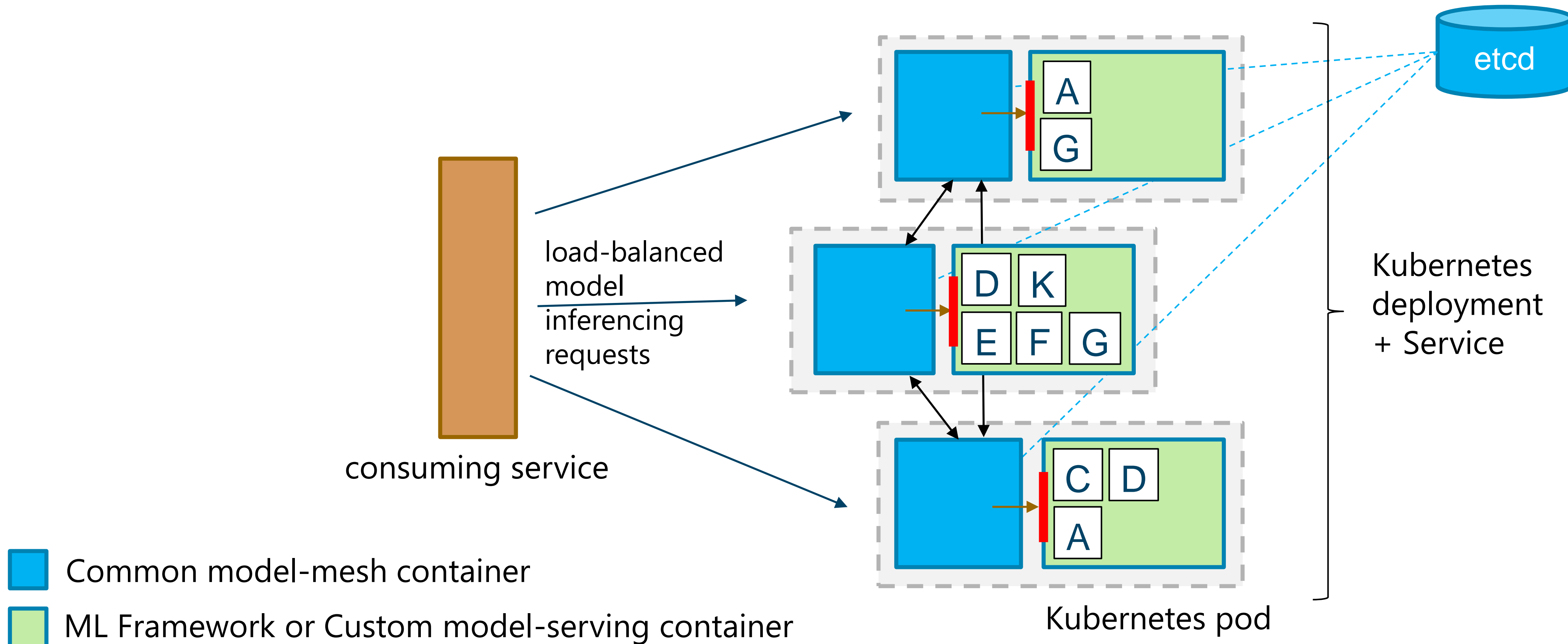
- **Scalability**
  - Both in terms of number of models managed and usage volume of those models
- **Performance**
  - Minimize latency of runtime requests
- **Efficiency**
  - Optimize use of available compute resources
- **Flexibility**
  - Language / ML-framework agnostic, bring-your-own inferencing API
  - Adapt to different usage scenarios, size/performance tradeoffs, etc.

Key Design Goals

- **Simplicity of integration,** short time-to-value for service teams
  - Minimize requirements/concerns of service implementers
  - Should be “plug-and-play”
- **Operational simplicity**
  - Minimize requirements/concerns of provisioning and support teams
  - Autonomy with respect to configuration and tuning
- **Resiliency and minimization of “cache misses” at all costs**
  - In terms of both availability and impacts to latency

Architecture

- Model-mesh is a single Docker image/container which runs alongside a "model runtime" container in a Kubernetes pod, deployed and scaled as a **single** self-contained logical Kubernetes deployment, consumed as a standard Kubernetes Service.



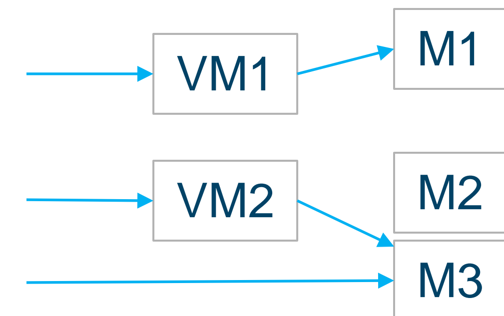
Data Model

"**Models**" have unique ids and are assumed to be immutable

- They generally comprise the static artifacts resulting from a completed training process
- They might be the weights associated with a trained NN, and/or collection of other artifacts used to perform the prediction or inferencing. This could include dictionaries or other data structures used for pre/post-processing
- They reside in some shared backing data store, and can be loaded from the store into memory ready to be used. They do not need to be homogenous and in particular can vary in terms of memory consumption

"**V-Models**" are mutable aliases which point to a single concrete Model

- Can be addressed instead of models for inferencing purposes, and behave equivalently
- Their target model can be updated dynamically
- Intended for versioning of production models



Ready-to-use ML Framework Integrations

- Works with existing model servers that support model multi-tenancy and dynamic reconfiguration of loaded models – custom or third-party

- Built-in adapters to integrate with off-the-shelf model servers for particular ML frameworks:
  - TensorFlow Serving
  - Nvidia Triton (includes Tensorflow 1 & 2, PyTorch, ONNX)
  - Seldon MLServer (includes scikit-learn, XGBoost, LightGBM)
  - Intel OpenVino Model Server

Contributed to KServe Open Source project

- **ModelMesh Serving** surrounds model-mesh with a Kubernetes-native management layer
- `ServingRuntime` custom resource used to specify per model server configuration
- Go-based controller orchestrates heterogeneous modelmesh Deployments behind a single `Service` endpoint
- Models to be served are managed via a `Predictor` custom resource (to be replaced by KServe's existing `InferenceService`) – mapped to mm models and vmodels internally
- Pods for a given `ServingRuntime` are only started if/when there are `Predictors` which require it
- Provides generalized and abstracted runtime-agnostic storage handling (to retrieve models)
- Provides built-in integration with some standard OSS model servers via injected adapter
- Supports KServe V2 REST API via optionally-injected proxy container

https://github.com/kserve/modelmesh-serving

KServe ModelMesh Serving Architecture

How model-mesh works in KServe:

model-mesh

AI Application

inferencing requests

Custom Resource manipulation

Kubernetes API Server

kubectl

Service

Register/ unregister models

watches / updates status

Mesh

Mesh

Mesh

Mesh

Puller

Puller

Puller

Puller

Model Server X

A C B D

Multi-model POD 1

Model Server X

E C B

Multi-model POD 2

Runtime X Deployment

Model Server Y

K L M P

Multi-model POD 3

Model Server Y

Q M

Multi-model POD 4

Runtime Y Deployment

Model Serve Controller

watches (read-only)

etcd

load models (read-only)

Other storage options to follow

COS (S3)

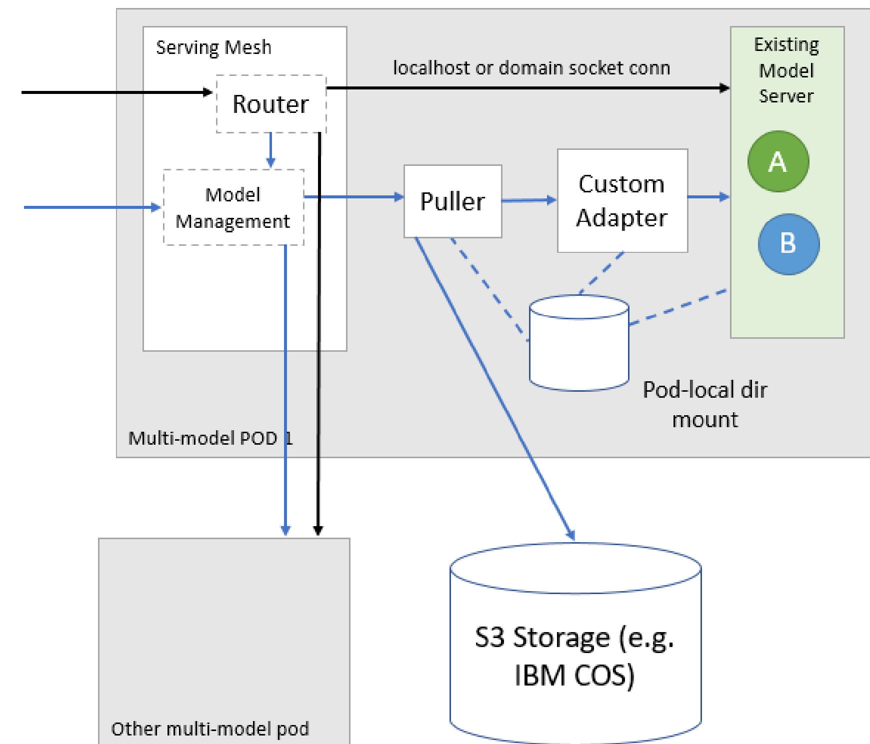
Data Plane

Control Plane

Adapter Pattern for Existing Model Servers

- Puller is injected automatically when used as part of KServe
- Otherwise, adapter or model server is responsible for retrieving model data
- ID Injection allows for direct inferencing path without needing to pass thorough adapter

Note: The remainder of these charts cover the function/design of the core model-mesh container only, not how it is managed/exposed within KServe



APIs

Model-mesh uses gRPC and is based on three logical service APIs and two logical service boundaries (internal and external):

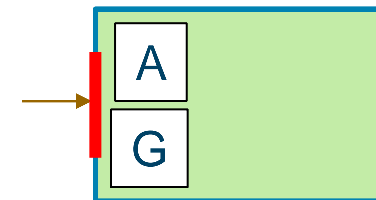
1. **Internal model management SPI** (internal)
   - Comprises two primary methods: `loadModel` and `unloadModel`
   - Implemented by internal "model runtime" containers - load a specified model from backing storage into memory, ready to be served
2. **External model management API** (external)
   - Exposed externally by model-mesh, provides methods to register and unregister new models with the platform, and create/update/delete vmodels
3. **Runtime Inferencing API** (both)
   - One or more *arbitrary* gRPC service definitions*, new or existing
   - Wraps prediction/inferencing logic to invoke already-loaded model of specified id
   - Implemented by internal "model runtime" containers, *automatically/transparently* exposed by the external model-mesh service

*Streaming rpcs not yet supported

Internal SPI

Wrap existing model-inferencing logic in a gRPC server implementing the following API, and package as a Docker container. Called only from within the same pod via localhost.

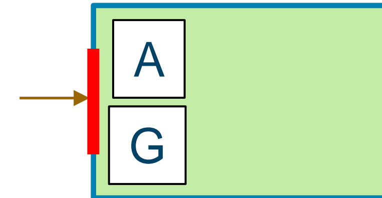
- **`loadModel`** `( modelId, modelInfo )`
  - return successfully when load is complete, or fail otherwise
- **`unloadModel`** `( modelId )`
- **`modelSize`** `( modelId )`
  - return memory consumed by previously-loaded model
- **`runtimeStatus`** `()`
  - return one of `STARTING`, `READY`, `FAILING`; plus total mem capacity for holding loaded models and max loading concurrency
- **Plus**, one or more *arbitrary* grpc services comprising any number of rpc methods, for performing prediction logic using prior-loaded model - id specified in a metadata header



Internal SPI Requirements and Guarantees

Requirements

- Must support concurrent (multithreaded) API access*
- Must be able to measure size of loaded models relatively accurately
- Loaded models are expected to remain available until explicitly unloaded via the `unloadModel` method (no eviction or cache management required)
- Inferencing/invocation API methods must read the target `modelId` from a gRPC metadata param (payload agnostic header), and be idempotent (retryable)



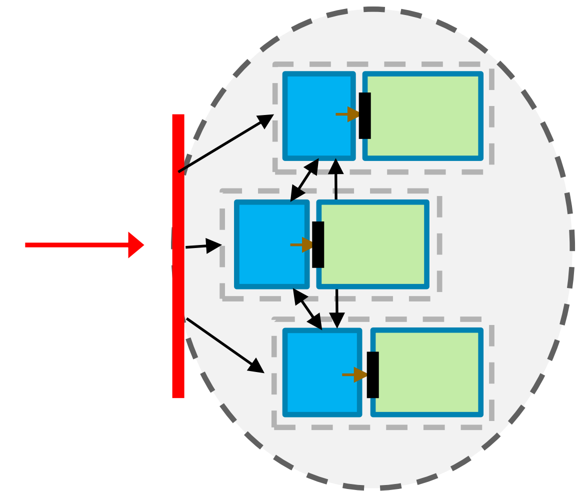
Guarantees

- No more than the indicated maximum number of `loadModel` requests will be in-progress concurrently
- Model invocation requests will only be made for already-loaded models

*No longer a hard requirement – see “latency-based autoscaling”

External API

Exposed by the model-mesh grpc-service. **All** methods are idempotent.



- **`registerModel`** `( modelId, modelInfo, loadNow, sync )`
  - registers a model with the cluster, optionally triggering an immediate load (sync or async); returns current status
- **`unregisterModel`** `( modelId )`
  - removes/unregisters model from the service
- **`getStatus`** `( modelId )`
  - returns one of NOT_LOADED, LOADING, LOADED, LOADING_FAILED, NOT_FOUND
- **`ensureLoaded`** `( modelId, sync, lastUsedTime )`
  - ensure model is loaded, optionally with specified last-used timestamp; returns current status
- **`setVModel(…), deleteVModel(…), checkVModelStatus(…)`**

- **Plus**, all inferencing API methods exposed by the configured model server container

Integration with Model Lifecycle Management

- Upon completion of training, call `registerModel` to add the model to the platform
- The model is immediately available for use, but inferencing requests will be delayed until loading completes
- `LOADED` state can be waited for synchronously or by polling, after which inferencing requests will return immediately
- If this is a new version of an existing model, `setVModel` can be called to update the vmodel to point to this new model

- When a model is no longer needed, `unregisterModel` flags it for removal. Subsequent inferencing requests for it will immediately fail with a not-found error, and it is asynchronously unloaded if necessary
- If the model in question is the current target of any vmodels, those must be deleted first

Integration with Model Lifecycle Management

- `ensureLoaded` can be used to verify that a particular model is loaded and trigger its loading if not. It also "touches" the model so that it moves up in the LRU ordering
- This can be a useful optimization for when there is an outside hint that a model may soon be needed

- For versioned models, the vmodel APIs (`setVModel`, `deleteVModel`) can be used by themselves:
  - `setVModel` can register a new model and update the vmodel's target to it in a combined operation
  - The "auto-delete" option can be used to automatically deregister models when no longer referenced by any vmodels

When an existing vmodel is set to a new target model, the transition is asynchronous and managed to ensure no impact to inferencing traffic (ensuring new model is loaded/scaled first)

Integration with Model-Consuming Services

- Call any of the custom inferencing API methods on the external model-mesh Kubernetes Service
- Provide one or more model ids *or* vmodel ids in `mm-model-id` / `mm-vmodel-id` headers
  - Unless id extraction is configured (see next chart)
- **If more than one id is provided, all specified (v)models will be applied in parallel**, with results concatenated
  - Assuming the API's response message comprises of protobuf maps and/or repeated fields, the result will be a simple aggregation (due to protobuf's wire encoding)
- Requests should be load-balanced between all model-mesh pods in an even manner (e.g. random or round-robbin)
- If evenly-balanced requests are guaranteed, `mm-balanced=true` header should be included to benefit from additional optimizations

Model ID Extraction and Injection

- By default, inferencing requests must include a gRPC metadata header indicating the target model or vmodel id
  - Unfortunately can be cumbersome for clients to set this
- Model servers are expected to read a model id header to determine which of their loaded models to use for a given request (and it may not be the same as the incoming external request, for example in the vmodel case)
  - Means existing model-servers require an adapter to “move” the id to the appropriate place (typically within the protobuf request message)

- ID extraction addresses the first problem and injection addresses the second.
- A string protobuf field can be configured per service method from/in which to extract/inject the model id, specified as a “path” of protobuf field numbers
- Model-mesh extracts/injects surgically based on offsets without copying data or serializing/deserializing

Features – Cache Management and HA

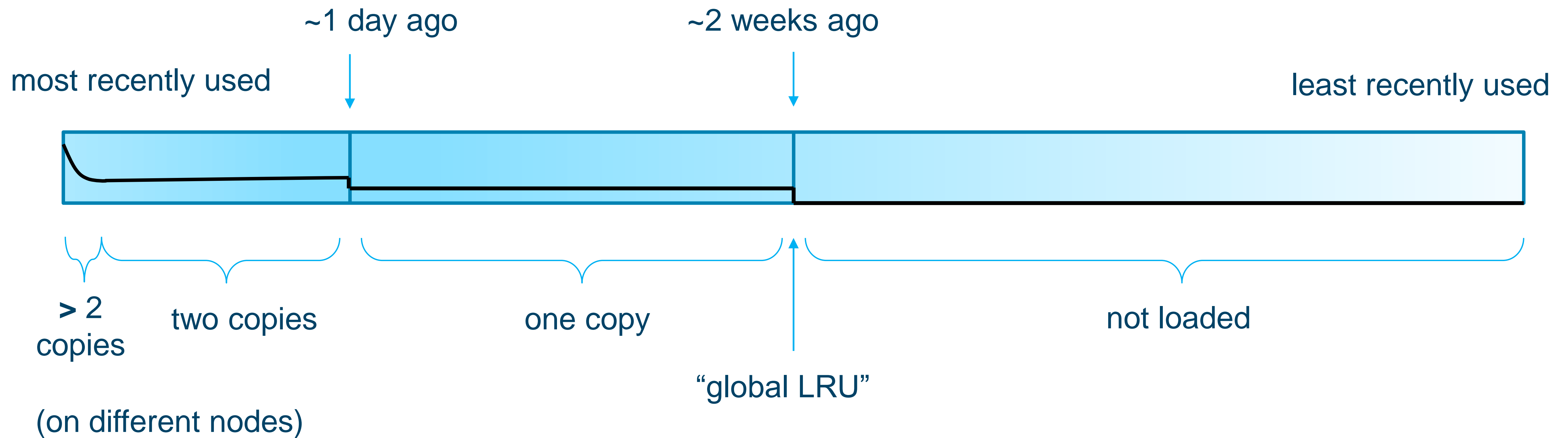
- The cluster of service instances is managed internally as a distributed LRU cache, with the available model server container filled with registered models
- Runtime inferencing requests can hit any instance (pod) and are forwarded (when necessary) to an instance holding the target loaded model
- If the target model is not currently loaded, a load will be immediately triggered (in the instance deemed most optimal at that point in time) and the request (and any others for the same model) will be delayed until the load completes.
  - This is done atomically – if multiple requests hit at the same time (across any number of “entry” instances), only the exact intended number of copy loads will be triggered
  - This cuts down on unnecessary churn/load

Features – Cache Management and HA

- Management of how many copies of a particular model are loaded, where, and when, is handled automatically within the framework. This includes:
  - Ensuring all “recently used” models have at least **two** copies loaded
  - Scaling beyond this if (and only if) certain request load thresholds are exceeded – a model under sufficient load can grow to have copies in every cluster instance
  - Scaling down from >2 to 2 and from 2 to 1 copy is done when appropriate based on load heuristics and last-used time respectively – the most-busy instance copies take precedence
  - Proactive loading of not-currently-loaded models where there is free space or to replace less-recently-used loaded models

Features – Cache Management and HA

- Visual representation of typical cache distribution



Features – Cache Management and HA

- The **global LRU** is the last-used time of the least-recently-used model which is currently loaded anywhere in the cluster
- Framework ensures that all models used since this time are loaded and ready for immediate use
- It only has significance if the cluster is full – i.e. insufficient capacity to hold an appropriate number of copies of all registered models
- This is a key metric to track w.r.t. sizing a model serving cluster, and correlates with the frequency of cache misses (requests which can't be served immediately and must wait for a model load)
- Its value changes over time and can be adjusted by scaling the size of the cluster – would be a suitable metric for auto-scaling the deployment size, but based on experience so far changes slowly enough that this isn't typically necessary
- For some implementations, model loading may take an amount of time which is unacceptable to *ever* have to wait on an inferencing request path – in this case just set the pod count to be large enough to hold all models

Features – Upgrades, Migrations, Cluster Scaling

- These operations are really functionally equivalent from a Kubernetes point of view
- When a pod is stopped, be it during a rolling upgrade, node evacuation/migration, or cluster scale-down, the following steps occur:
  - The instance becomes immediately removed as a target for new placements
  - Locally loaded models are propagated (in parallel) to other instances (i.e. new loadings are triggered), based on balanced/optimal placement and prioritized based on usage recency. Global LRU positioning and aggregate copy counts also propagated/preserved
  - Queued or in-progress model loads are aborted
  - The instance waits for the in-use / most recently used propagated models (not all) to finish loading before proceeding
  - The instance is unregistered from service discovery, while continuing to serve requests for a few more seconds and complete any in-flight
  - The container/pod exits

Features – Upgrades, Migrations, Cluster Scaling

- If instance versions aren't consistent across the cluster (as will be the case during an upgrade), model placement will automatically avoid older instances
  - Results in a more efficient upgrade process, ensuring that models don't get propagated to instances which are also about to be stopped

- This shutdown sequence ensures that the serving service can be treated operationally like any other homogenous microservice; no special or separate orchestration is needed for any of these operations

Features – Heterogeneous Clusters

- Different model server types and/or configurations can be combined as a single logical model-mesh cluster/service
- Placement is controlled via arbitrary labels that can be assigned to individual or subsets of instances (Pods), and a set of constraints mapping model types to required and/or preferred labels
- Uses include:
  - A Kubernetes cluster where only some nodes have GPUs and certain kinds of models may require or be able to exploit GPUs while other kinds of models in the same model-mesh service use CPU only
  - Managing heterogenous model server runtimes behind the same service
    - VModels can transition between different runtime implementations assuming they expose the same inferencing API

Optimization – Controlled, Prioritized loading

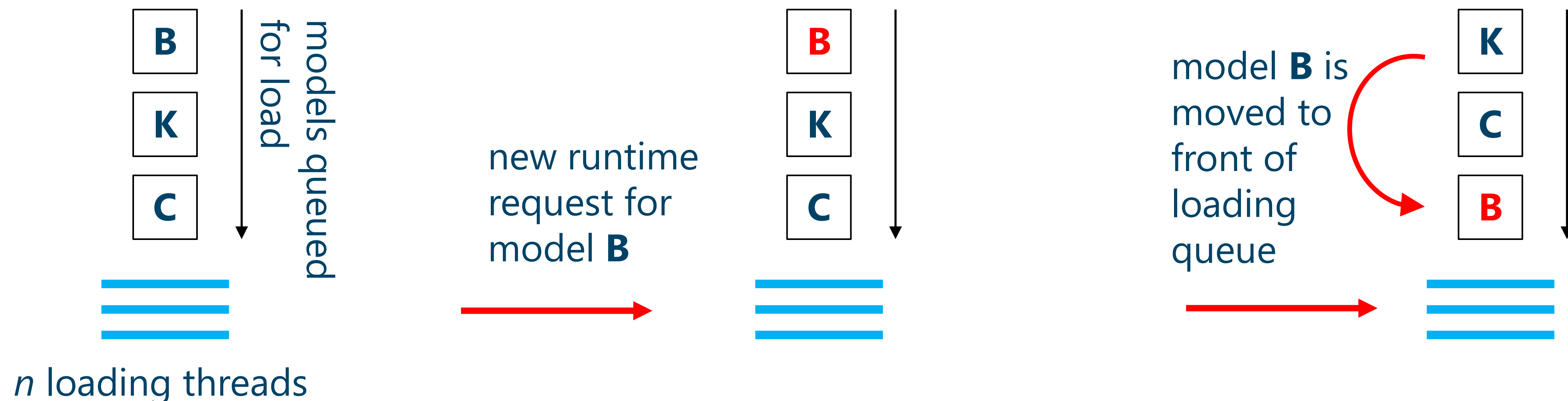
- The framework controls the number of concurrently loading models on each instance. The "cost" of loading may depend on the model server implementation, which can specify a suitable maximum

- This limit is important to ensure that loading activity doesn't noticeably impact processing of latency-sensitive inferencing requests

- Models placed on an instance which is at maximum loading capacity are queued, resulting in a greater total loading time. To minimize this, current loading queue lengths are advertised by each instance and taken into account in placement decisions (though not as a first-order consideration)

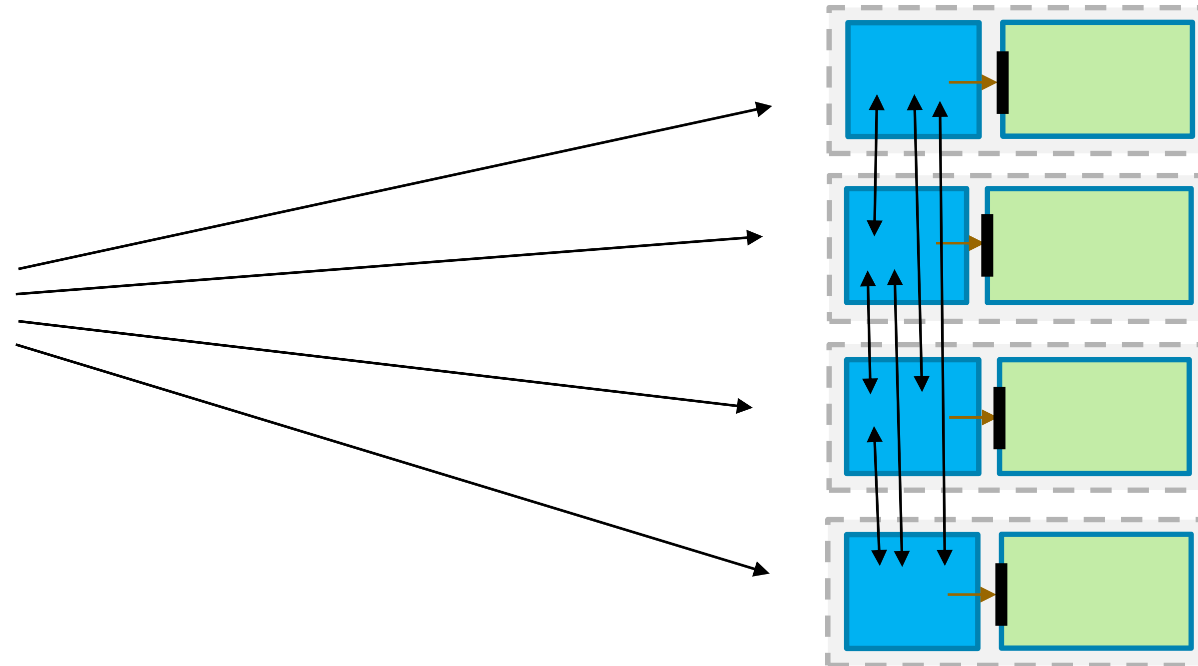
Optimization – Controlled, Prioritized loading

- The queue is prioritized (not FIFO) – models with a more recent LRU time will overtake less recently used ones (for example old models proactively loaded to fill up available space)
- In particular, upmost priority is given to loadings which have an inferencing request waiting for them (cache-miss scenario). If already in the loading queue for another reason, a runtime request for a model will dynamically move the load in-front of all other models which *don't* have a runtime request waiting for them – such as second copies triggered due to recent model usage.



Resiliency – Architecture Benefits

- Autonomy w.r.t. operational procedures minimizes scope for errors
- Structural simplicity (symmetry and few moving parts) results in a manageable number of distributed failure scenarios/permutations to reason about, less room for different components' views of data to become out of sync



Runtime Inferencing Resiliency

- A model-mesh deployment essentially forms a decentralized “service mesh”. From the outside it behaves like any other clustered service; internally requests are routed in a deliberate manner
- Under normal circumstances, the majority of runtime requests will result in a single internal hop, with some taking zero and some taking two

- Per-container cpu resource allocations ensure heavily-loaded or misbehaving model serving container doesn’t impact capacity/performance in request routing layer

- If a routed request fails due to a connection error it will be automatically retried against the next instance which has the model loaded, or else treated as a cache-miss if there are no more
  - This means there’s minimal impact if an instance dies, even to model requests currently being processing

Model Loading Resiliency

- The framework is designed to expect failures during loading, to insulate against temporary unavailability of object storage or whatever data store the models are retrieved from
  - Failed model loads are immediately retried on up to three separate instances, after which the model's state will move to `LOADING_FAILED`
  - Failure records remain on each instance, prevent the same model from loading there until they have "expired" (10 minutes), and no load of the model will be attempted while there's at least 3 failure records for it in total
  - This means that if appropriate given demand/cache management decisions, failed models will continue to be retried on a regular basis
  - Provides self-recovery with no manual intervention required – such as when there's an extended data store outage, or a bug in the service runtime code (in the latter case, after an upgrade the failed models will automatically retry and succeed)
- Any runtime requests waiting for the failed model are automatically re-routed to the retry instances – if a retry succeeds the in-flight requests won't notice (apart from extra latency)

Rate-Tracking and Auto-Scaling

- Real-time inferencing request rates are tracked in each node on both a per-instance and per-model basis
  - Assessed every ~10 secs, only models used in prior time window require processing
  - These are used for ongoing placement and scaling decisions
  - Highest-loaded / most-frequently-used models are placed on least loaded instances *and vice versa* – for example “old” models that are proactively loaded will actually prefer to land on *heavily* loaded instances (which have available memory)
  - This has proved very effective in maintaining a balanced load across large clusters
- Scale-up of a model is done in a decentralized manner
  - Each instance makes scale-up decisions about the *active* models which it’s serving, based on a dynamically-configurable *per-instance* request-rate threshold
  - Since inferencing requests are balanced evenly between the model’s copies, this will result in the instance’s “share” of the requests subsequently dropping below the threshold

Latency-based Auto-Scaling

- New auto-scaling mode, **currently in beta**
- Enabled globally for a particular model-mesh cluster
- Differences from existing request-rate-based auto-scaling:
  - Each loaded model reports a maximum concurrency for inflight requests
  - Model-mesh queues requests as needed
  - Scaling is triggered as needed to minimize queuing time
  - Explicit request load threshold parameter is no longer required or used
  - Particularly suited for use-cases where:
    - Per-model inferencing logic is not multi-threaded
    - Per-request inferencing cost and time is high and/or variable

Implementation Details - Model Loading Stats

- As well as request rates, aggregate model-loading time statistics are continually tracked (average and variance)
  - Used as a heuristic within various other routing and timing decisions

- Self-tuning means the framework will fit use-cases with varying distributions of model-loading times without manual configuration

Implementation Details – etcd

- An **etcd** instance is required for coordinating operations and persisting model/instance state
  - Powerful abstraction layer means that Zookeeper can be used interchangeably if desired
- A robust java etcd client was developed from scratch to use in production (**`etcd-java`**)
  - It has now been open-sourced: https://github.com/IBM/etcd-java
- Care is taken to minimize load on the key/value store, in terms of data volume and request frequency
- Continually-changing state (model usage times) is updated in a lazy fashion, precise cache ordering, per-model usage rates, and model size info are only kept locally and sent via internal RPCs
- Only aggregated instance stats are published, and only when changed by a sufficient amount

Implementation Details - Background Tasks

- **Janitor**
  - Runs in every instance every 6 minutes
  - Cross-references and reconciles local cache with model registry
  - Scales-down model copies as/when appropriate
- **Rate Tracker**
  - Runs in every instance every 10 seconds
  - Processes models used since the last iteration, triggers scale-ups as necessary
- **Reaper**
  - Runs on a single elected leader instance every 7 minutes
  - Keeps track of “missing” instances referenced in the model registry and cleans up orphaned registrations (which might exist if an instance died)
  - Triggers proactive model loads if there is sufficient free space in the cluster, or any more-recently-used not-loaded models than those which are loaded

Implementation Details – etcd Registry

- Three “tables” are maintained within etcd – **models**, **vmodels**, **instances**
- Every instance watches all of these tables and mirrors their contents in local memory
- Reads are from the local cache in almost all cases
- Entries are JSON-based but size is kept to a minimum (we’re considering a change to protobuf)

- The **vmodels** table has an entry for every vmodel with target and active model ids, used for resolving vmodel-addressed inference requests to a concrete model id
- The **models** table has an entry for *every* registered model
  - The local cache of this registry in each instance is separate from the logical weighted LRU cache of the models that are actually loaded/loading on that instance
  - These entries contain:
    - Minimal metadata required by model servers to locate/load the model data
    - List of instance ids where the model is loaded/loading and failed
    - A refcount and some timestamps used in loading/unloading decisions

Implementation Details – etcd Registry

- The **instances** table has an entry for each running instance, created/updated only by that instance and deleted when the instance shuts down
  - These entries are tied to a session lease associated with the client of the corresponding instance, so that they will be deleted automatically after a short time if the instance dies
  - They contain a collection of high-level stats for the instance, updated when those values change but rate-limited to not more than one update per 1-2 seconds
    - LRU model time for the instance, model count, capacity, usage, loads in progress, current request load ("rpm" average over last few minute), some other metadata
  - These values are used to make placement decisions when a new copy of a model is to be loaded somewhere

Implementation Details – etcd Registry

- Lifecycle of etcd **model** records
  - Created upon registration, deleted upon deregistration (instance independent)
  - List of associated instance ids used to route requests
  - *Before* loading a model, instance adds its own id to that model's record via an etcd transaction
    - If two instances attempt this concurrently, one will fail, see the updated record and then re-assess loading decision / route to the other instance
  - There is a periodic task run in each instance which reviews its set of loaded models and decides which to unload based on various factors. When unload decision is made, an etcd transaction is first performed to remove the instance's own id from that model's record
  - There is a *global* periodic task (run in elected leader instance) to clean up instance ids from model records corresponding to Pods which did not shut down gracefully

Implementation Details – etcd Registry

Model record (indexed by model id)

```
{
  # Mostly-arbitrary metadata
  "type": "tensorflow",
  "encKey": “abcde",
  "mPath": "path/to/model",

  # Where model is loaded
  # and time each load was initiated
  "instanceIds": {
    "df5b84-5f9qb": 1654150842935,
    "df5b84-d9d4z": 1654150276533
  },

  # Where model failed to load
  # and time of failure
  "failedIn": {
    "df5b84-wn7sh": 1654150270218
  },

  # Ref count (from vmodels)
  "refs": 1,
  # Whether to auto-delete when refs==0
  "autoDel": true,
  # Last-used time (lazily updated)
  "lu": 1654186975550
}
```

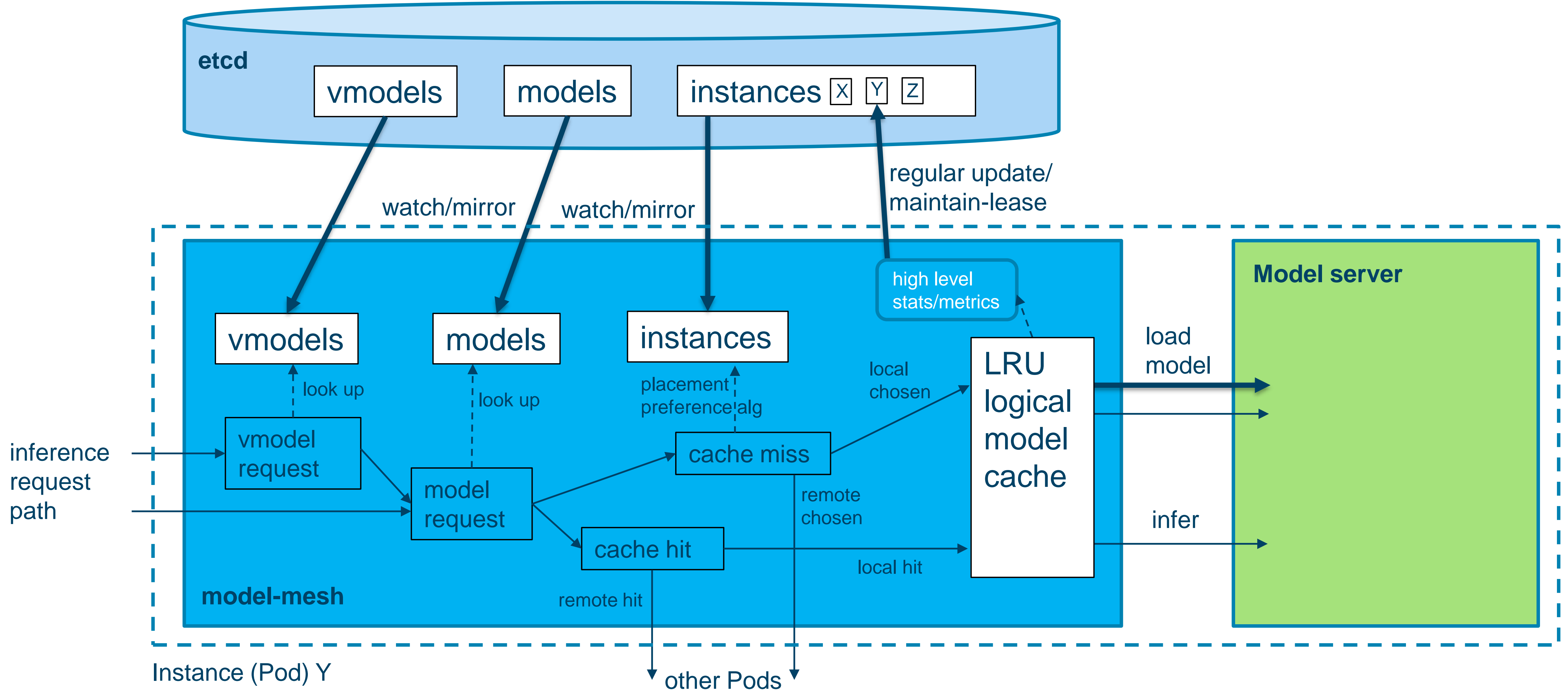
VModel record (indexed by vmodel id)

```
{
  # Active model id
  "amid": "my-model-v1",
  # Target model id
  # (equals active id in steady state)
  "tmid": "my-model-v2",

  # can only be true when amid != tmid
  "failed": false,

  "o": "optional-owner"
}
```

Implementation Details – Data Flows

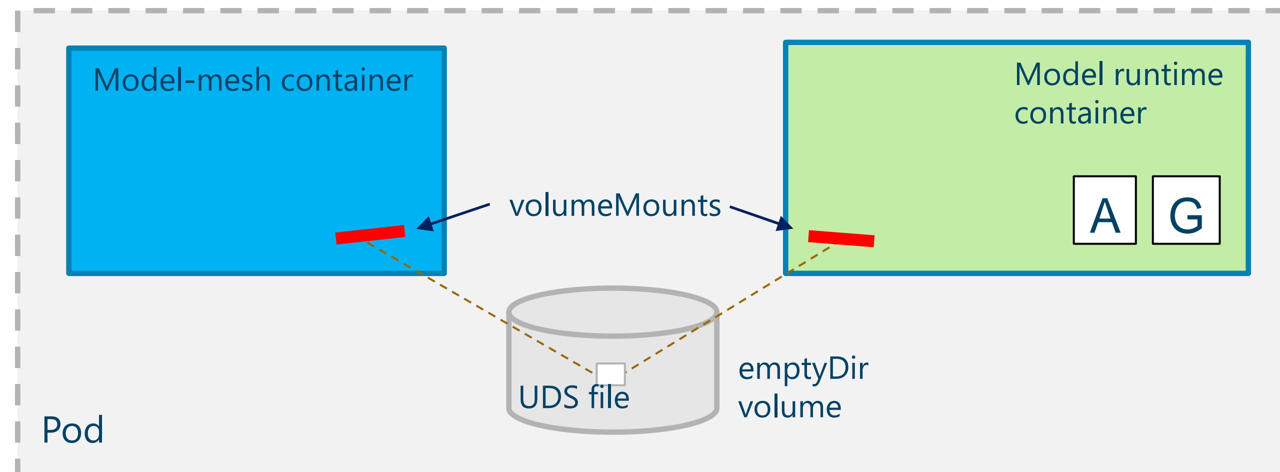


Performance

- Built with strong focus on performance and scalability
  - Runtime path highly optimized (network, CPU, GC, lock contention), close to zero-copy I/O
    - Written by committer of popular netty networking library
  - Real-time prioritization of "critical path" model loads
  - Loading/unloading rate controlled to prevent impact to runtime processing

Unix Domain Socket for intra-Pod communication

- The default mechanism for the model-mesh container to talk to the collocated model runtime container is now via a mounted **unix domain socket**
- Significantly better performance than a regular IP connection via localhost
- Requires very small change to built gRPC server config and pod spec
- localhost TCP port connection still works for backwards compatibility



Dashboard

- Drop-in image, deploy in your Kubernetes namespace
- Uses same etcd secret as model-mesh deployments
- Shows real-time state/stats of all model-mesh deployments

Note: The dashboard has not yet been open sourced, but we intend to do so in future

Cluster Summary

Instances	Global RPM	Global LRU	Usage (MB)	Capacity (MB)	Avg. Model Size (MB)	Loaded Models	Models Loading	Model Copies	Managed Models	Models w/ Load Failures
20	2888	7 days ago	168435.9 [90.4%]	186360.9	180.919	762 [2.2%]	4	927	33892	1

Instances *Showing 20 of 20 entries*

Search:

	ID	Host	Zone	RPM	LRU	Loading	Count	Usage	Capacity	Models	Failures	Endpoint	Version	Registration	L
86.8%	86d549-pfnxr	10.187.35.114		3	6 days ago	0	51	8085.2	9318.0	51	0	172.30.231.149:8080	20181005-2004-144	an hour ago	
84.0%	86d549-92mdc	10.94.17.234		153	6 days ago	0	42	7829.0	9318.0	42	0	172.30.5.203:8080	20181005-2004-144	an hour ago	
87.6%	86d549-kljq5	10.185.57.185		176	7 days ago	0	44	8159.0	9318.0	44	0	172.30.118.182:8080	20181005-2004-144	an hour ago	
91.0%	86d549-wbtjr	10.93.209.184		133	6 days ago	1	47	8481.2	9318.0	46	1	172.30.222.6:8080	20181005-2004-144	an hour ago	
94.6%	86d549-l77zp	10.93.211.202		268	7 days ago	0	49	8819.0	9318.0	48	0	172.30.106.155:8080	20181005-2004-144	an hour ago	
93.1%	86d549-lpssb	10.185.56.5		157	7 days ago	0	46	8678.1	9318.0	45	1	172.30.150.202:8080	20181005-2004-144	an hour ago	
96.3%	86d549-pp8jd	10.187.35.247		187	6 days ago	0	48	8976.0	9318.0	48	0	172.30.103.101:8080	20181005-2004-144	an hour ago	
85.0%	86d549-6g4wz	10.93.211.120		0	7 days ago	1	48	7920.0	9318.0	48	0	172.30.96.215:8080	20181005-2004-144	an hour ago	
98.2%	86d549-q4v6s	10.94.17.184		159	7 days ago	0	50	9149.0	9318.0	50	0	172.30.216.106:8080	20181005-2004-144	an hour ago	
94.6%	86d549-sfpkj	10.187.15.217		174	6 days ago	0	48	8819.0	9318.0	48	0	172.30.105.28:8080	20181005-2004-144	an hour ago	
96.4%	86d549-gzlnr	10.93.211.29		203	7 days ago	0	49	8984.0	9318.0	49	0	172.30.236.214:8080	20181005-2004-144	an hour ago	
79.7%	86d549-vs7tk	10.94.17.216		44	7 days ago	0	45	7425.0	9318.0	45	0	172.30.214.31:8080	20181005-2004-144	an hour ago	
98.2%	86d549-vf9dz	10.93.211.142		163	7 days ago	0	51	9149.1	9318.0	51	0	172.30.173.106:8080	20181005-2004-144	an hour ago	
89.3%	86d549-rfr9m	10.185.56.15		174	7 days ago	0	45	8324.0	9318.0	45	0	172.30.123.26:8080	20181005-2004-144	an hour ago	
80.5%	86d549-k9pjs	10.94.17.101		149	7 days ago	0	41	7499.1	9318.0	41	0	172.30.171.109:8080	20181005-2004-144	an hour ago	
91.1%	86d549-xz8p2	10.187.15.225		160	7 days ago	1	47	8489.1	9318.0	47	0	172.30.200.241:8080	20181005-2004-144	an hour ago	
94.9%	86d549-hbx2c	10.93.209.222		160	6 days ago	0	45	8843.0	9318.0	45	0	172.30.52.247:8080	20181005-2004-144	an hour ago	
89.3%	86d549-prw29	10.93.211.34		122	7 days ago	1	46	8324.1	9318.0	45	1	172.30.242.174:8080	20181005-2004-144	an hour ago	

IBM Watson

IBM

Detailed metrics published (Prometheus or statsd)