Linux Administration: Kubernetes

Services and Security

Run application in production

Resource Management

Runtime constraints on resources

- User memory constraints.
- There is no memory limit for the container. The container can use as much memory as needed
- -m, --memory="" Memory limit (format: <number>[<unit>]). Number is a positive integer. Unit can be one of b, k, m, or g. Minimum is 4M.
- --memory-swap="" Total memory limit (memory + swap, format: <number>[<unit>]). Number is a positive integer. Unit can be one of b, k, m, or g.
- --memory-reservation="" Memory soft limit (format: <number>[<unit>]). Number is a positive integer. Unit can be one of b, k, m, or g.

Runtime constraints on resources

- We set memory limit only, this means the processes in the container can use 300M memory and 300M swap memory, by default, the total virtual memory size (--memory-swap) will be set as double of memory, in this case, memory + swap would be 2*300M, so processes can use 300M swap memory as well.

  docker run -it -m 300M ubuntu:14.04 /bin/bash

- We set both memory and swap memory, so the processes in the container can use 300M memory and 700M swap memory.

  docker run -it -m 300M --memory-swap 1G ubuntu:14.04 /bin/bash

Runtime constraints on resources

- Memory reservation is a kind of memory soft limit that allows for greater sharing of memory. Under normal circumstances, containers can use as much of the memory as needed and are constrained only by the hard limits set with the -m/--memory option.
- When memory reservation is set, Docker detects memory contention or low memory and forces containers to restrict their consumption to a reservation limit.

```
docker run -it -m 500M --memory-reservation 200M \
    ubuntu:14.04 /bin/bash
```

CPU share constraint

- CPU period constraint
- If there is 1 CPU, this means the container can get 50% CPU worth of run-time every 50ms.

```
docker run -it --cpu-period=50000 --cpu-quota=25000 \
    ubuntu:14.04 /bin/bash
```

- cpu-period=50000 (50ms)
- cpu-quota=25000 (50% : 25000 of 50000)

CPU share constraint

- Cpuset constraint
- This means processes in container can be executed on cpu 1 and cpu 3.

```
docker run -it --cpuset-cpus="1,3" ubuntu:14.04 /bin/bash
```

- This means processes in container can be executed on cpu 0, cpu 1 and cpu 2.

```
docker run -it --cpuset-cpus="0-2" ubuntu:14.04 /bin/bash
```

Block IO bandwidth (Blkio) constraint

- By default, all containers get the same proportion of block IO bandwidth (blkio). This proportion is 500. To modify this proportion, change the container's blkio weight relative to the weighting of all other running containers using the --blkio-weight flag.
- The --blkio-weight flag can set the weighting to a value between 10 to 1000. For example, the commands below create two containers with different blkio weight:

```
docker run -it --name c1 --blkio-weight 300 ubuntu:14.04 /bin/bash
docker run -it --name c2 --blkio-weight 600 ubuntu:14.04 /bin/bash
```

Block IO bandwidth (Blkio) constraint

- The --blkio-weight-device="DEVICE_NAME:WEIGHT" flag sets a specific device weight.
- The DEVICE_NAME:WEIGHT is a string containing a colon-separated device name and weight.
- For example, to set /dev/sda device weight to 200:

```
docker run -it  --blkio-weight-device "/dev/sda:200"   Ubuntu
```

- If you specify both the --blkio-weight and --blkio-weight-device, Docker uses the --blkio-weight as the default weight and uses --blkio-weight-device to override this default with a new value on a specific device.

Block IO bandwidth (Blkio) constraint

- The --device-read-bps flag limits the read rate (bytes per second) from a device. For example, this command creates a container and limits the read rate to 1mb per second from /dev/sda:

```
docker run -it --device-read-bps /dev/sda:1mb ubuntu
```

- The --device-write-bps flag limits the write rate (bytes per second) to a device. For example, this command creates a container and limits the write rate to 1mb per second for /dev/sda:

```
docker run -it --device-write-bps /dev/sda:1mb ubuntu
```

Block IO bandwidth (Blkio) constraint

- The --device-read-iops flag limits read rate (IO per second) from a device. For example, this command creates a container and limits the read rate to 1000 IO per second from /dev/sda:

```
docker run -ti --device-read-iops /dev/sda:1000 ubuntu
```

- The --device-write-iops flag limits write rate (IO per second) to a device. For example, this command creates a container and limits the write rate to 1000 IO per second to /dev/sda:

```
docker run -ti --device-write-iops /dev/sda:1000 ubuntu
```

Kubernetes

Resource Quotas

Resource Quotas

- A resource quota, defined by a ResourceQuota object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by resources in that project.
- Compute Resource Quota
- Storage Resource Quota
- Object Count Quota

Compute Resource Quotas

- limits.cpu Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
- limits.memory Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.
- requests.cpu Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value.
- requests.memory Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value.

Storage Resource Quotas

- requests.storage Across all persistent volume claims, the sum of storage requests cannot exceed this value.
- persistentvolumeclaims The total number of persistent volume claims that can exist in the namespace.
- <storage-class-name>.storageclass.storage.k8s.io/requests.storage
  Across all persistent volume claims associated with the storage-class-name, the sum of storage requests cannot exceed this value.
- <storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims
  Across all persistent volume claims associated with the storage-class-name, the total number of persistent volume claims that can exist in the namespace.

Object Count Quotas

- Here is an example set of resources users may want to put under object count quota:

- count/persistentvolumeclaims
- count/services
- count/secrets
- count/configmaps
- count/replicationcontrollers
- count/deployments.apps
- count/replicasets.apps
- count/statefulsets.apps
- count/jobs.batch
- count/cronjobs.batch
- count/deployments.extensions

Object Count Quotas

```
apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-high
  spec:
    hard:
      cpu: "1000"
      memory: 200Gi
      pods: "10"
    scopeSelector:
      matchExpressions:
      - operator : In
        scopeName: PriorityClass
        values: ["high"]
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-medium
```

Object Count Quotas

```
spec:
    hard:
      cpu: "10"
      memory: 20Gi
      pods: "10"
    scopeSelector:
      matchExpressions:
      - operator : In
        scopeName: PriorityClass
        values: ["medium"]
- apiVersion: v1
  kind: ResourceQuota
```

```
metadata:
  name: pods-low
spec:
  hard:
    cpu: "5"
    memory: 10Gi
    pods: "10"
  scopeSelector:
    matchExpressions:
    - operator : In
      scopeName: PriorityClass
      values: ["low"]
```

Run application in production

Swarm

Swarm Feature highlights

- Cluster management integrated with Docker Engine
- Decentralized design
- Declarative service model
- Scaling
- Desired state reconciliation
- Multi-host networking
- Service discovery
- Load balancing
- Secure by default
- Rolling updates

Swarm mode key concepts

- A swarm consists of multiple Docker hosts which run in swarm mode and act as managers (to manage membership and delegation) and workers (which run swarm services).
- A node is an instance of the Docker engine participating in the swarm. You can also think of this as a Docker node.
- A service is the definition of the tasks to execute on the manager or worker nodes. It is the central structure of the swarm system and the primary root of user interaction with the swarm.
- The swarm manager uses ingress load balancing to expose the services you want to make available externally to the swarm.

Create a swarm

- Open a terminal and ssh into the machine where you want to run your manager node.
- This tutorial uses a machine named manager1.

- Run the following command to create a new swarm:

```
docker swarm init --advertise-addr <MANAGER-IP>
```

Create a swarm

- Run the following command to create a new swarm (10.0.0.1 – ip address of the manager):

```
docker swarm init --advertise-addr 10.0.0.1
```

- To add a worker to this swarm, run the following command on the another node:

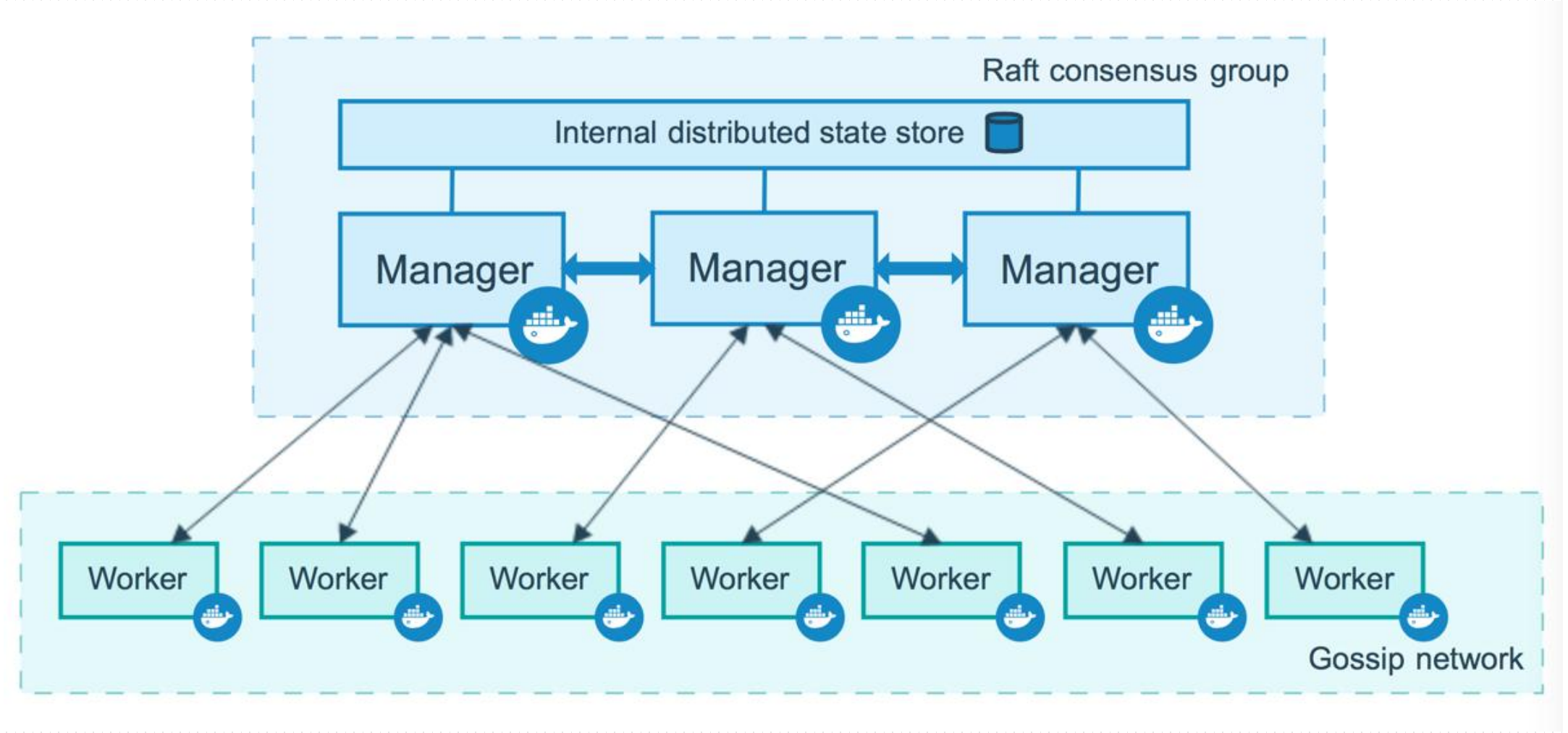
```
docker swarm join
--token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c  10.0.0.1:2377
```

Create a swarm

- Open a terminal and ssh into the machine where the manager node runs and run the docker node ls command to see the worker nodes:

```
docker node ls
ID                            HOSTNAME        STATUS  AVAILABILITY  MANAGER STATUS
03g1y59jwfg7cf99w4lt0f662     worker2   Ready                 Active
9j68exjopxe7wfl6yuxml7a7j     worker1   Ready                 Active
dxn1zf6l61qsb1josjja83ngz *   manager1  Ready                 Active        Leader
```

Create a fault tolerant swarm



Create a swarm: Change roles

- You can promote a worker node to be a manager by running docker node promote. For example, you may want to promote a worker node when you take a manager node offline for maintenance.

  docker node promote <node name>

- You can also demote a manager node to a worker node.

  docker node demote <node name>

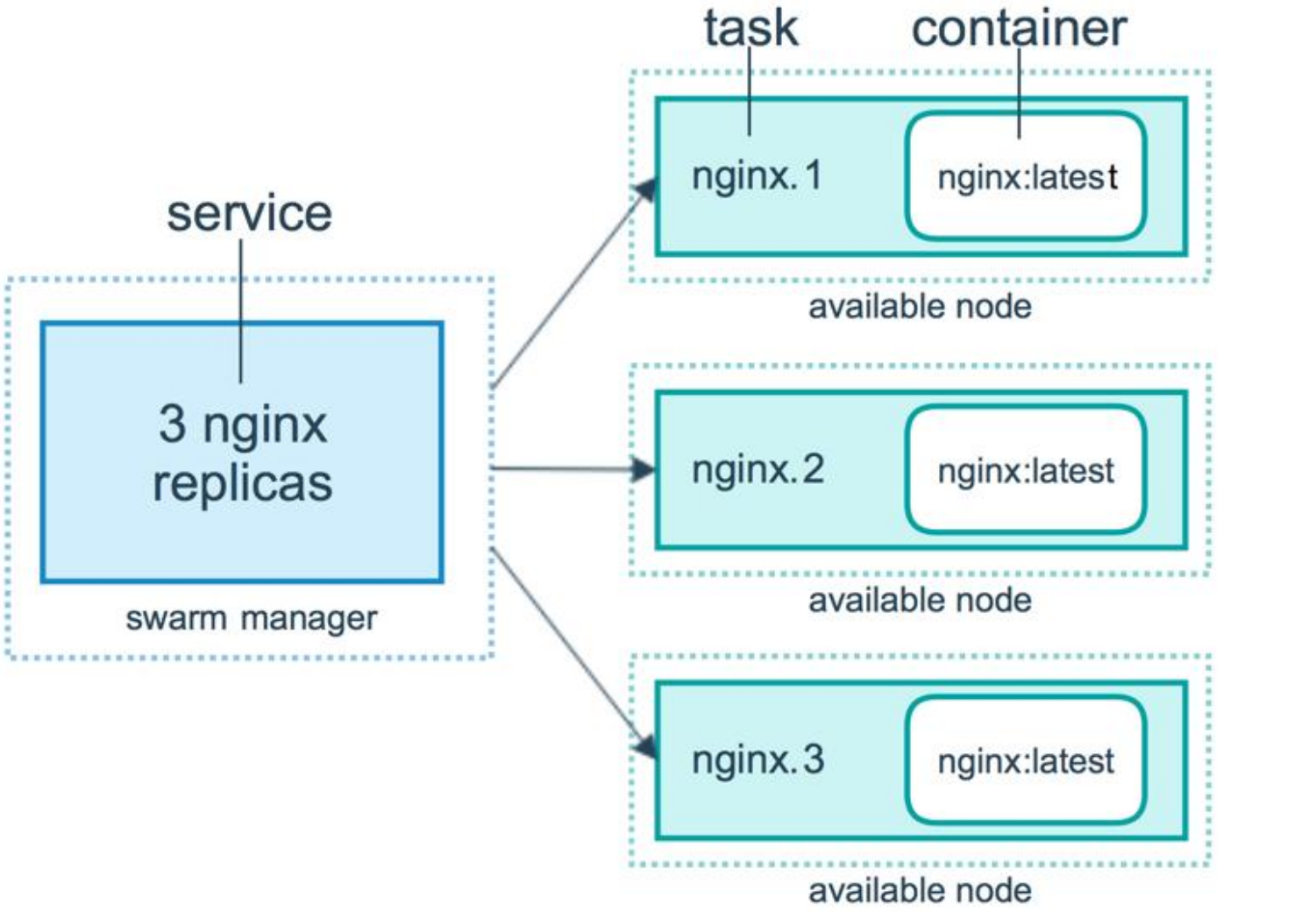
Run application in production

Services

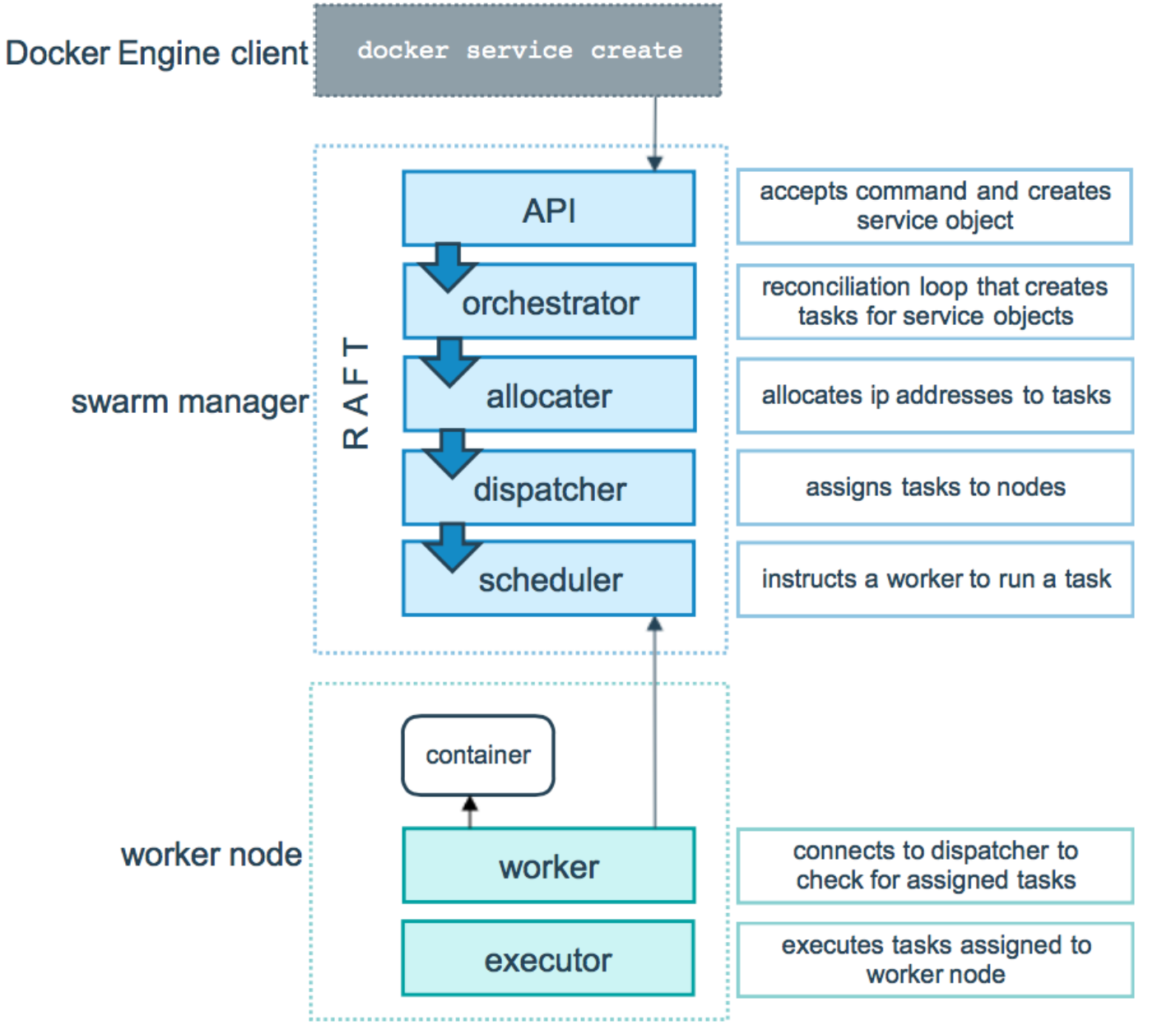
About services

- In a distributed application, different pieces of the app are called “services”.
- Services are really just “containers in production.”
- A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on.
- Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

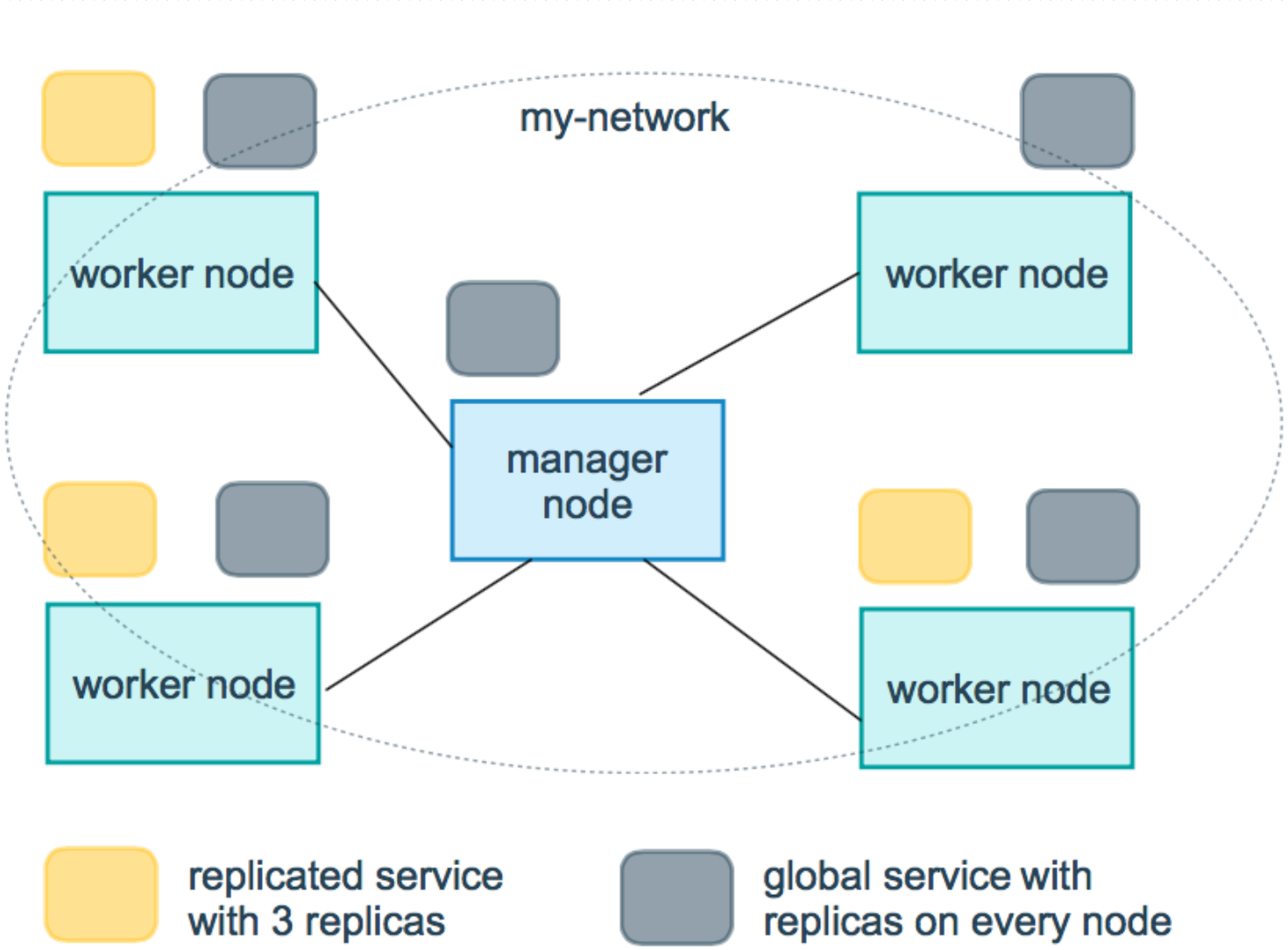
About services



Start service



Replicated and global services



Create a service

- Run the following command:

```
docker service create --replicas 1 \
        --name helloworld alpine ping docker.com
```

- The --name flag names the service helloworld.
- The --replicas flag specifies the desired state of 1 running instance.
- The arguments alpine ping docker.com define the service as an Alpine Linux container that executes the command ping docker.com.

Create a service

- Run docker service ls to see the list of running services:

```
docker service ls

ID                  NAME        SCALE  IMAGE   COMMAND
9uk4639qpg7n  helloworld  1/1    alpine  ping docker.com
```

Inspect and remove a service

- Run docker service inspect --pretty <SERVICE-ID> to display the details about a service in an easily readable format.

```
docker service inspect --pretty helloworld
```

- Run docker service rm helloworld to remove the helloworld service.

```
docker service rm helloworld
helloworld
```

Scale the service in the swarm

- Run the following command to change the desired state of the service running in the swarm:

```
docker service scale helloworld=5
helloworld scaled to 5

docker service ps helloworld
NAME                                    IMAGE   NODE     DESIRED STATE  CURRENT STATE
helloworld.1.8p1vev3fq5zm0mi8g0as41w35  alpine  worker2  Running        Running 7 minutes
helloworld.2.c7a7tcdq5s0uk3qr88mf8xco6  alpine  worker1  Running        Running 24 seconds
helloworld.3.6crl09vdcalvtfehfh69ogfb1  alpine  worker1  Running        Running 24 seconds
helloworld.4.auky6trawmdlcne8ad8phb0f1  alpine  manager1 Running        Running 24 seconds
helloworld.5.ba19kca06l18zujfwxyc5lkyn  alpine  worker2  Running        Running 24 seconds
```

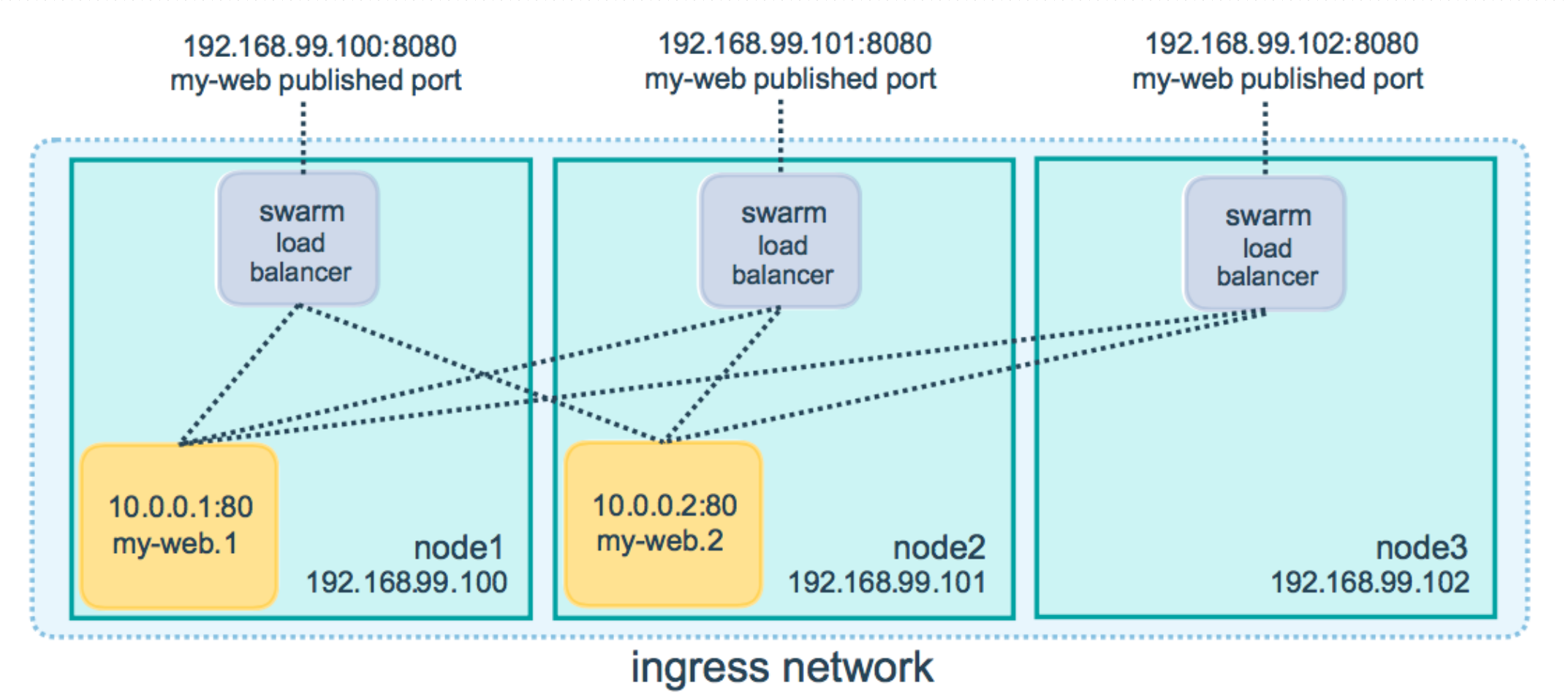
Drain a node on the swarm

- Sometimes, such as planned maintenance times, you need to set a node to DRAIN availability. DRAIN availability prevents a node from receiving new tasks from the swarm manager. It also means the manager stops tasks running on the node and launches replica tasks on a node with ACTIVE availability.

- Change docker node availability availability state:

```
docker node update --availability drain worker1
docker node update --availability active worker1
```

Publish a port for a service



Drain a node on the swarm

- The following command publishes port 80 in the nginx container to port 8080 for any node in the swarm:

```
docker service create --name my-web \
  --publish published=8080,target=80 --replicas 2 nginx
```

- You can publish a port for an existing service using the following command:

```
docker service update \
  --publish-add published=8080,target=80
```

Run application in production

Stacks

Using stacks

- A stack is a group of interrelated services that share dependencies, and can be orchestrated and scaled together.
- A single stack is capable of defining and coordinating the functionality of an entire application (though very complex applications may want to use multiple stacks).

Define service docker-compose.yml

- version: "3"
- services:
- web:
- image: username/repo:tag
- deploy:
- replicas: 5
- resources:
- limits:
- cpus: "0.1"
- memory: 50M
- restart_policy:
- condition: on-failure
- ports:
- - "4000:80"
- networks:
- - webnet
- networks:
- webnet:

Create or update the stacked service

- Re-run the docker stack deploy command on the manager, and whatever services need updating are updated:

```
docker stack deploy -c docker-compose.yml getstartedlab
Updating service getstartedlab_web (id: angi1bf5e4to03qu9f93trnxm)
Creating service getstartedlab_visualizer (id: l9mnwkeq2jiononb5ihz9u7a4)
```

Docker-compose

- Create a new Dockerfile .

- FROM nginx:latest
- EXPOSE 80

Docker-compose

- Create docker-compose.yml .

```
version: "3.3"
services:
  web:
    build: .
    ports:
      - "8088:80"
```

Docker-compose

- Build and run your app with Compose

- From your project directory, start up your application by running docker-compose up.

- docker-compose up

Kubernetes

Access Control

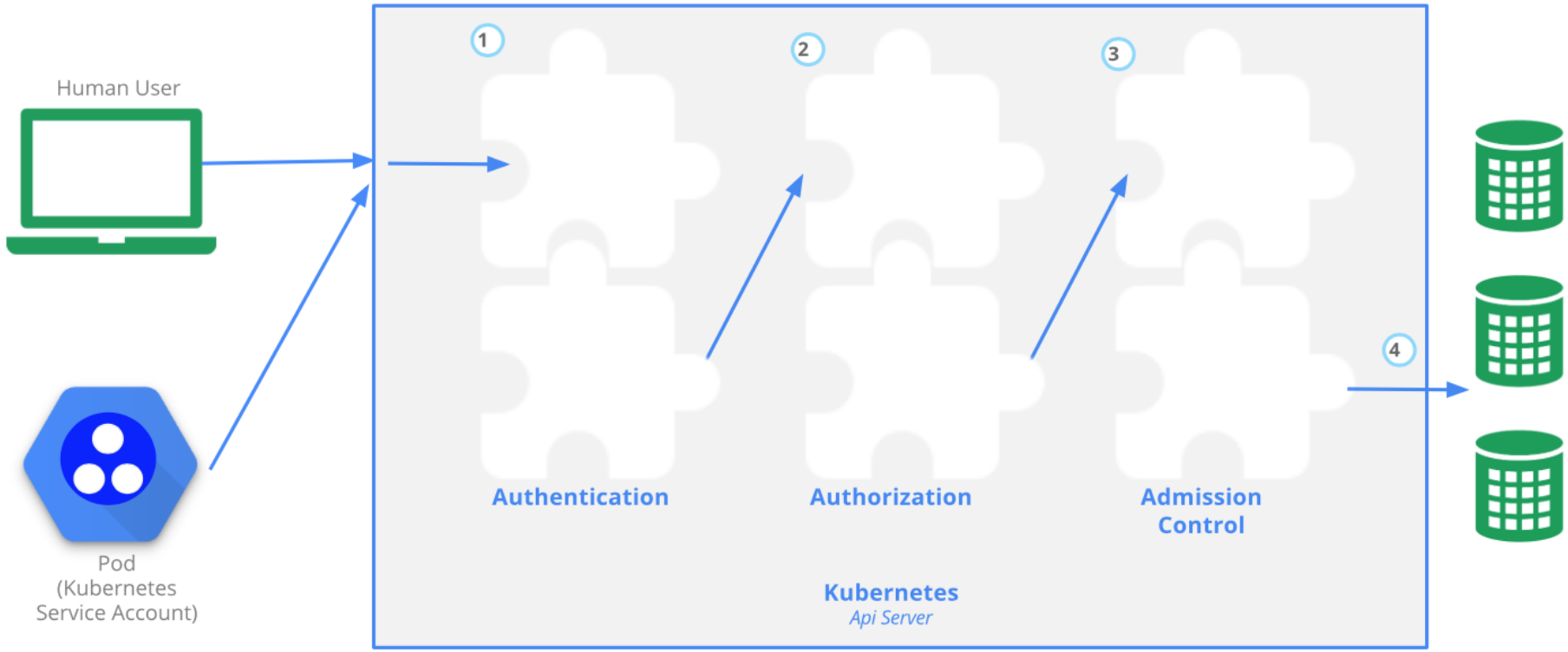
Namespaces

- If multiple users and teams use the same Kubernetes cluster we can partition the cluster into virtual sub-clusters using Namespaces. The names of the resources/objects created inside a Namespace are unique, but not across Namespaces in the cluster.
- To list all the Namespaces, we can run the following command:
- kubectl get namespaces
- Generally, Kubernetes creates four Namespaces out of the box: kube-system, kube-public, kube-node-lease, and default.

Authentication, Authorization, and Admission Control

- To access and manage Kubernetes resources or objects in the cluster, we need to access a specific API endpoint on the API server. Each access request goes through the following access control stages:
- Authentication : Logs in a user.
- Authorization: Authorizes the API requests submitted by the authenticated user.
- Admission Control : Software modules that validate and/or modify user requests based.

Authentication, Authorization, and Admission Control



Authentication

- Kubernetes does not have an object called user, nor does it store usernames or other related details in its object store. However, even without that, Kubernetes can use usernames for the Authentication phase of the API access control, and to request logging as well.
- Kubernetes supports two kinds of users:
  - Normal Users are managed outside of the Kubernetes cluster via independent services like User/Client Certificates, a file listing usernames/passwords, Google accounts, etc.
  - Service Accounts allow in-cluster processes to communicate with the API server to perform various operations. Most of the Service Accounts are created automatically via the API server, but they can also be created manually. The Service Accounts are tied to a particular Namespace and mount the respective credentials to communicate with the API server as Secrets.

Authentication

- For authentication, Kubernetes uses a series of authentication modules:
  - X509 Client Certificates (passing the --client-ca-file=SOMEFILE option to the API server)
  - Static Token File (the --token-auth-file=SOMEFILE option to the API server)
  - Bootstrap Tokens
  - Service Account Tokens
  - OpenID Connect Tokens (to offload the authentication to external services)
  - Webhook Token Authentication (offload to a remote service)
  - Authenticating Proxy

Authorization

- After a successful authentication, users can send the API requests to perform different operations.
- Here, these API requests get authorized by Kubernetes using various authorization modules, that allow or deny the requests.
- Some of the API request attributes that are reviewed by Kubernetes include user, group, extra, Resource, Namespace, or API group, to name a few. Next, these attributes are evaluated against policies. If the evaluation is successful, then the request is allowed, otherwise it is denied. Similar to the Authentication step, Authorization has multiple modules, or authorizers. More than one module can be configured for one Kubernetes cluster, and each module is checked in sequence. If any authorizer approves or denies a request, then that decision is returned immediately

Authorization

- Authorization modes
- Node authorization is a special-purpose authorization mode which specifically authorizes API requests made by kubelets. It authorizes the kubelet's read operations for services, endpoints, or nodes, and writes operations for nodes, pods, and events.
- Attribute-Based Access Control (ABAC)
  With the ABAC authorizer, Kubernetes grants access to API requests, which combine policies with attributes.
- In Webhook mode, Kubernetes can request authorization decisions to be made by third-party services, which would return *true* for successful authorization, and *false* for failure

Authorization : Role

- Authorization modes
- Role-Based Access Control (RBAC) regulates the access to resources based on the Roles of individual users. In Kubernetes, multiple Roles can be attached to subjects like users, service accounts, etc.
- In RBAC, we can create two kinds of Roles:
- A Role grants access to resources within a specific Namespace.
- A ClusterRole grants the same permissions as Role does, but its scope is cluster-wide.

Authorization : Role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: lfs158
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Authorization : RoleBinding

- Once the role is created, we can bind it to users with a RoleBinding object.
- There are two kinds of RoleBindings:
- **RoleBinding**
  It allows us to bind users to the same namespace as a Role. We could also refer a ClusterRole in RoleBinding, which would grant permissions to Namespace resources defined in the ClusterRole within the RoleBinding's Namespace.
- **ClusterRoleBinding**
  It allows us to grant access to resources at a cluster-level and to all Namespaces.

Authorization : RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-read-access
  namespace: lfs158
subjects:
- kind: User
  name: student
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Admission Control

- Admission Controllers are used to specify granular access control policies, which include allowing privileged containers, checking on resource quota, etc.
- We force these policies using different admission controllers, like ResourceQuota, DefaultStorageClass, AlwaysPullImages, etc.
- They come into effect only after API requests are authenticated and authorized.

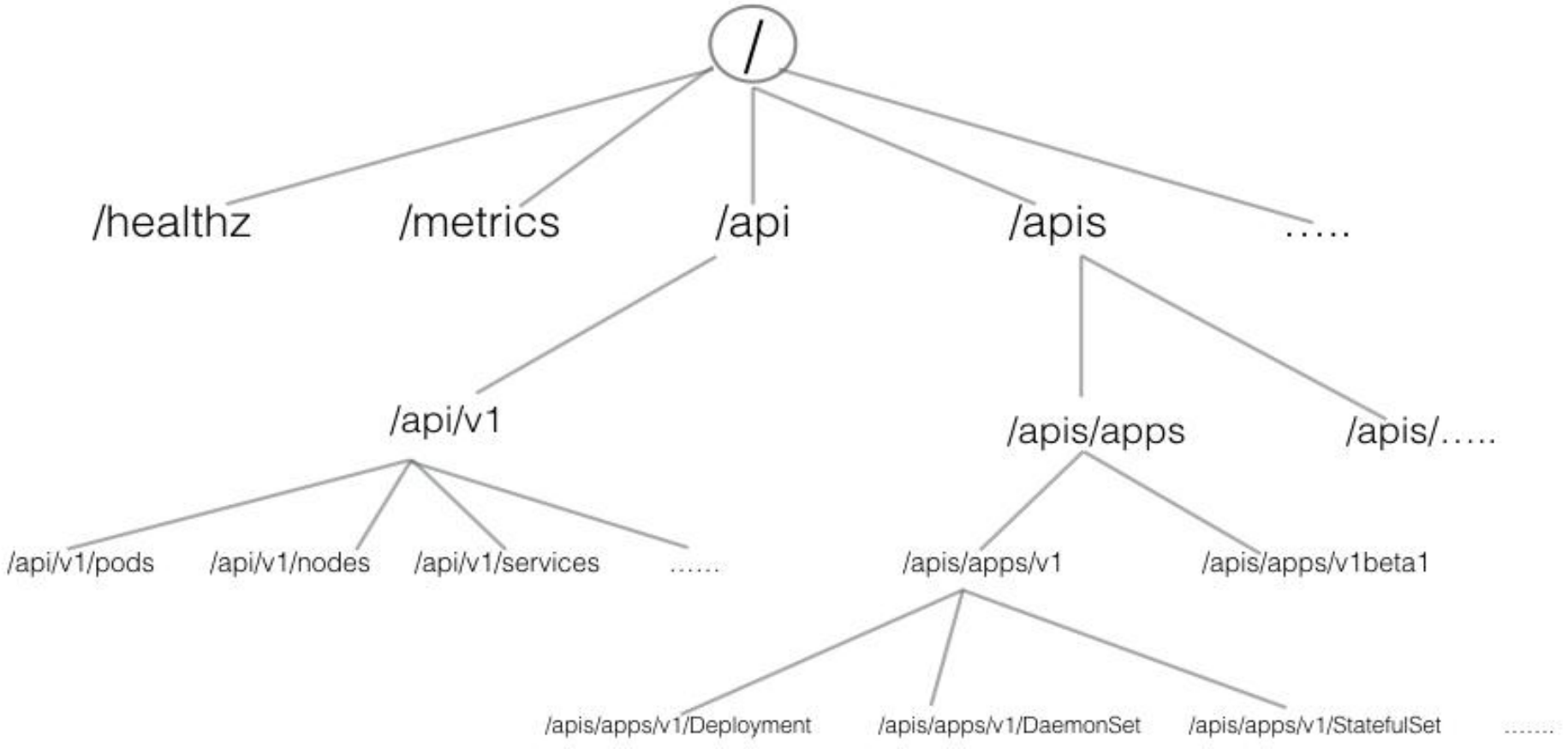
Admission Control

- Admission Controllers are used to specify granular access control policies, which include allowing privileged containers, checking on resource quota, etc.
- We force these policies using different admission controllers, like ResourceQuota, DefaultStorageClass, AlwaysPullImages, etc.
- They come into effect only after API requests are authenticated and authorized.

Kubernetes API

- The main component of the Kubernetes control plane is the **API server**, responsible for exposing the Kubernetes APIs.
- The APIs allow operators and users to directly interact with the cluster. Using both CLI tools and the Dashboard UI, we can access the API server running on the master node to perform various operations to modify the cluster's state.
- The API server is accessible through its endpoints by agents and users possessing the required credentials.

Kubernetes API



APIs with Authentication

- When not using the **kubectl proxy**, we need to authenticate to the API server when sending API requests.
- We can authenticate by providing a **Bearer Token** when issuing a **curl**, or by providing a set of **keys** and **certificates**.
- A **Bearer Token** is an **access token** which is generated by the authentication server (the API server on the master node) and given back to the client. Using that token, the client can connect back to the Kubernetes API server without providing further authentication details, and then, access resources.

APIs with Authentication

- Get the token
- TOKEN=$(kubectl describe secret -n kube-system \
  $(kubectl get secrets -n kube-system | grep default | cut -f1 -d ' ') \|
  grep -E '^token' | cut -f2 -d':' | tr -d '\t' | tr -d " ")

- Get the API server endpoint:
- APISERVER=$(kubectl config view | grep https | cut -f 2- -d ":" | tr -d " ")

APIs with Authentication

- Access the API server using the curl command, as shown below:
- curl $APISERVER --header "Authorization: Bearer $TOKEN" --insecure

- Instead of the **access token,** we can extract the client certificate, client key, and certificate authority data from the **.kube/config** file. Once extracted, they can be encoded and then passed with a **curl** command for authentication
- curl $APISERVER --cert encoded-cert --key encoded-key --cacert encoded-ca

Kubernetes

ConfigMaps and Secret

Kubernetes ConfigMaps

- Use a ConfigMap for setting configuration data separately from application code.
- For example, imagine that you are developing an application that you can run on your own computer (for development) and in the cloud (to handle real traffic). You write the code to look in an environment variable named DATABASE_HOST. Locally, you set that variable to localhost. In the cloud, you set it to refer to a Kubernetes Service that exposes the database component to your cluster.

Kubernetes ConfigMaps

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: 3
  ui_properties_file_name: "user-interface.properties"
#
# file-like keys
game.properties: |
  enemy.types=aliens,monsters
  player.maximum-lives=5
user-interface.properties: |
  color.good=purple
  color.bad=yellow
  allow.textmode=true
```

Using ConfigMaps

- Defining a volume and mounting it inside the demo container as /config creates four files:

- /config/player_initial_lives
- /config/ui_properties_file_name
- /config/game.properties
- /config/user-interface.properties

Using ConfigMaps

- Create a config map or use an existing one. Multiple Pods can reference the same config map.
- Modify your Pod definition to add a volume under .spec.volumes[]. Name the volume anything, and have a .spec.volumes[].configmap.localObjectReference field set to reference your ConfigMap object.
- Add a .spec.containers[].volumeMounts[] to each container that needs the config map. Specify .spec.containers[].volumeMounts[].readOnly = true and .spec.containers[].volumeMounts[].mountPath to an unused directory name where you would like the config map to appear.
- Modify your image or command line so that the program looks for files in that directory. Each key in the config map data map becomes the filename under mountPath.

Kubernetes ConfigMaps

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    configmap:
      name: myconfigmap
```

Information Security

Secrets Management

https://kubernetes-security.info/#securing-your-container-images

Overview of Secrets

- To use a Secret, a Pod needs to reference the Secret. A Secret can be used with a Pod in three ways:

- As files in a volume mounted on one or more of its containers.
- As container environment variable.
- By the kubelet when pulling images for the Pod.

Types of Secret

- When creating a Secret, you can specify its type using the type field of a Secret resource, or certain equivalent kubectl command line flags (if available).
- The type of a Secret is used to facilitate programmatic handling of different kinds of confidential data.
- Kubernetes provides several builtin types for some common usage scenarios. These types vary in terms of the validations performed and the constraints Kubernetes imposes on them

Builtin Type	Usage
`Opaque`	arbitrary user-defined data
`kubernetes.io/service-account-token`	service account token
`kubernetes.io/dockercfg`	serialized `~/.dockercfg` file
`kubernetes.io/dockerconfigjson`	serialized `~/.docker/config.json` file
`kubernetes.io/basic-auth`	credentials for basic authentication
`kubernetes.io/ssh-auth`	credentials for SSH authentication
`kubernetes.io/tls`	data for a TLS client or server
`bootstrap.kubernetes.io/token`	bootstrap token data

Service account token Secrets

- apiVersion: v1
- kind: Secret
- metadata:
- name: secret-sa-sample
- annotations:
- kubernetes.io/service-account.name: "sa-name"
- type: kubernetes.io/service-account-token
- data:
- # You can include additional key value pairs as you do with Opaque Secrets
- extra: YmFyCg==

Docker config Secrets

- You can use one of the following type values to create a Secret to store the credentials for accessing a Docker registry for images.
- kubernetes.io/dockercfg
- kubernetes.io/dockerconfigjson

- The kubernetes.io/dockercfg type is reserved to store a serialized ~/.dockercfg which is the legacy format for configuring Docker command line. When using this Secret type, you have to ensure the Secret data field contains a .dockercfg key whose value is content of a ~/.dockercfg file encoded in the base64 format.
- The kubernetes.io/dockerconfigjson type is designed for storing a serialized JSON that follows the same format rules as the ~/.docker/config.json file which is a new format for ~/.dockercfg. When using this Secret type, the data field of the Secret object must contain a .dockerconfigjson key, in which the content for the ~/.docker/config.json file is provided as a base64 encoded string.

Docker config Secrets

- Below is an example for a kubernetes.io/dockercfg type of Secret:

- apiVersion: v1
- kind: Secret
- metadata:
- name: secret-dockercfg
- type: kubernetes.io/dockercfg
- data:
- .dockercfg: |
- "<base64 encoded ~/.dockercfg file>"

Basic authentication Secret

- The kubernetes.io/basic-auth type is provided for storing credentials needed for basic authentication. When using this Secret type, the data field of the Secret must contain one of the following two keys:

- username: the user name for authentication;
- password: the password or token for authentication.

- Both values for the above two keys are base64 encoded strings. You can, of course, provide the clear text content using the stringData for Secret creation.

SSH authentication secrets

- The builtin type kubernetes.io/ssh-auth is provided for storing data used in SSH authentication. When using this Secret type, you will have to specify a ssh-privatekey key-value pair in the data (or stringData) field as the SSH credential to use.

- apiVersion: v1
- kind: Secret
- metadata:
- name: secret-ssh-auth
- type: kubernetes.io/ssh-auth
- data:
- # the data is abbreviated in this example
- ssh-privatekey: |
- MIIEpQIBAAKCAQEAulqb/Y ...

TLS secrets

- Kubernetes provides a builtin Secret type kubernetes.io/tls for storing a certificate and its associated key that are typically used for TLS .
- This data is primarily used with TLS termination of the Ingress resource, but may be used with other resources or directly by a workload.
- When using this type of Secret, the tls.key and the tls.crt key must be provided in the data (or stringData) field of the Secret configuration, although the API server doesn't actually validate the values for each key.

TLS secrets

- apiVersion: v1
- kind: Secret
- metadata:
- name: secret-tls
- type: kubernetes.io/tls
- data:
- # the data is abbreviated in this example
- tls.crt: |
- MIIC2DCCAcCgAwIBAgIBATANBgkqh ...
- tls.key: |
- MIIEpgIBAAKCAQEA7yn3bRHQ5FHMQ ...

Using Secrets as files from a Pod

- To consume a Secret in a volume in a Pod:

- Create a secret or use an existing one. Multiple Pods can reference the same secret.
- Modify your Pod definition to add a volume under .spec.volumes[]. Name the volume anything, and have a .spec.volumes[].secret.secretName field equal to the name of the Secret object.
- Add a .spec.containers[].volumeMounts[] to each container that needs the secret. Specify .spec.containers[].volumeMounts[].readOnly = true and .spec.containers[].volumeMounts[].mountPath to an unused directory name where you would like the secrets to appear.
- Modify your image or command line so that the program looks for files in that directory. Each key in the secret data map becomes the filename under mountPath.

Projection of Secret keys to specific paths

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
      - key: username
        path: my-group/my-username
```

Projection of Secret keys to specific paths

- What will happen:

- username secret is stored under /etc/foo/my-group/my-username file instead of /etc/foo/username.
- password secret is not projected.

- If .spec.volumes[].secret.items is used, only keys specified in items are projected. To consume all keys from the secret, all of them must be listed in the items field. All listed keys must exist in the corresponding secret. Otherwise, the volume is not created.

Using Secrets as environment variables

- To use a secret in an environment variable in a Pod:

- Create a secret or use an existing one. Multiple Pods can reference the same secret.
- Modify your Pod definition in each container that you wish to consume the value of a secret key to add an environment variable for each secret key you wish to consume. The environment variable that consumes the secret key should populate the secret's name and key in env[].valueFrom.secretKeyRef.
- Modify your image and/or command line so that the program looks for values in the specified environment variables.

Using Secrets as environment variables

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
  restartPolicy: Never
```

Immutable Secrets

- FEATURE STATE: Kubernetes v1.21 [stable] : **immutable: true**

- The Kubernetes feature Immutable Secrets and ConfigMaps provides an option to set individual Secrets and ConfigMaps as immutable. For clusters that extensively use Secrets (at least tens of thousands of unique Secret to Pod mounts), preventing changes to their data has the following advantages:

- protects you from accidental (or unwanted) updates that could cause applications outages
- improves performance of your cluster by significantly reducing load on kube-apiserver, by closing watches for secrets marked as immutable.

Questions?

andry.cheredarchuk@gmail.com