
Deployments

Reliable, scalable distributed systems

- Many services are being deployed as API’s
- If they fail part of the system crashes
- Software must be available even during software maintenance, version and fix rollouts etc.
- When more and more people rely on the services, they must be scalable to avoid downtime or unavailability.

Reasons to start with containerized systems

- Velocity
  Measured not in terms of the raw number of features you can ship per hour or day, but rather in terms of the number of things you can ship while maintaining a highly available service.
- Scaling (of both software and teams)
- Abstracting your infrastructure
- Efficiency

Velocity through Immutability

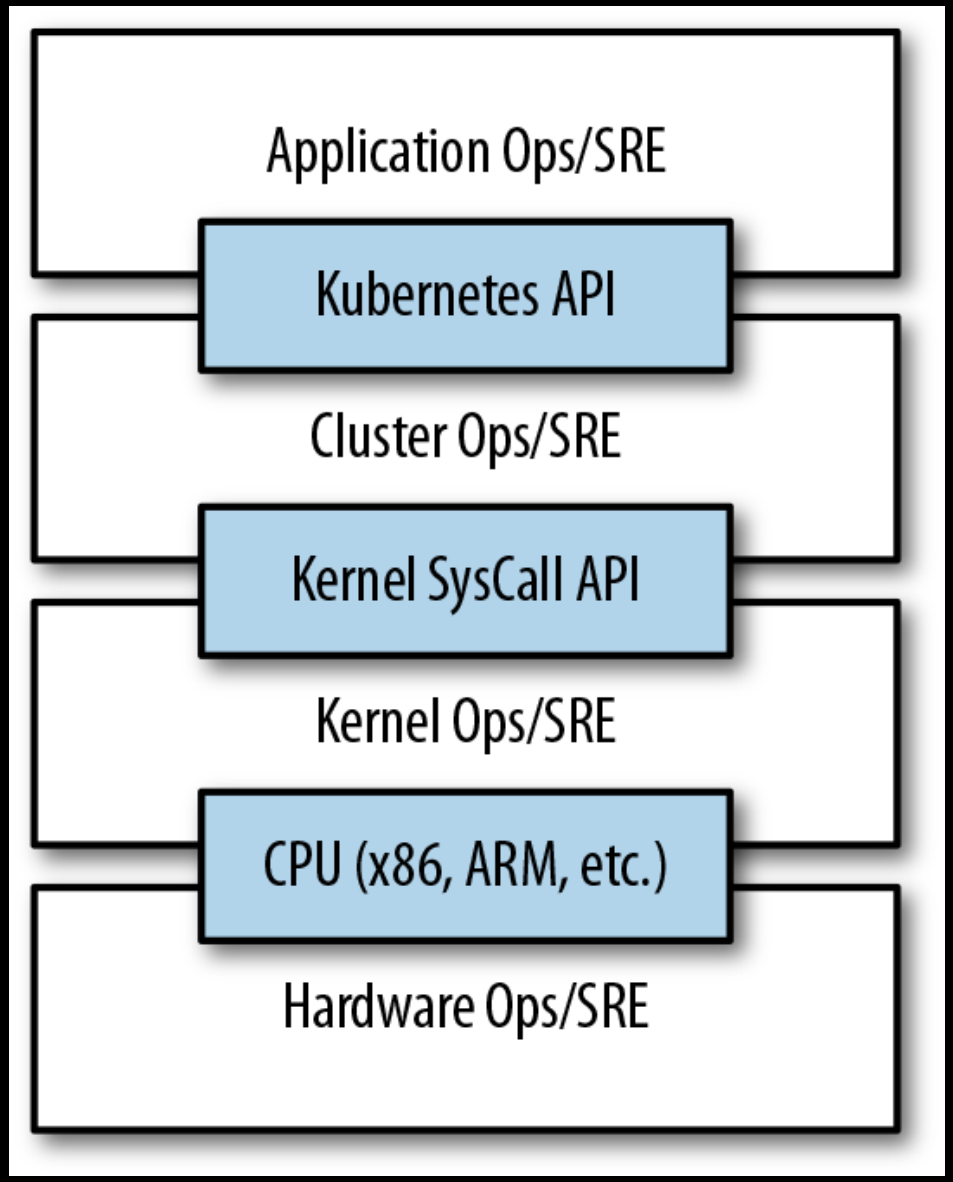
- Immutable infrastructure
- Declarative Configuration
- Self-Healing Systems

Scaling

- Decoupling
  … systems through separating API’s and services, connected by load balancers
- Easy Scaling for Applications and Clusters
  Independently scale parts or even whole applications. Extend computing power by adding more nodes to a cluster
- Scaling Teams
  Teams can be organized around services, applications, cluster-management etc.

Abstracting infrastructure

- Public cloud movement
  easy-to-use, self-service infrastructure for developers to consume
- Focus on services and applications
- Infrastructure components, like VMs, storage and physical networking are hidden with Kubernetes
- Hence easy movement between on-premise clusters and different public cloud offerings


Application Ops/SRE
Kubernetes API
Cluster Ops/SRE
Kernel SysCall API
Kernel Ops/SRE
CPU (x86, ARM, etc.)
Hardware Ops/SRE

Efficiency

- No/less thinking in terms of machines by developers
- Multiple tasks/services can be packed tightly together on the same machine
- Power usage / useful work ratio is better
- Toolset, and knowledge of them, is being reused
- Testing services and applications can be done in an already available cluster, no need for separate machines (isolate by namespaces)

Creating, Updating, and Destroying Kubernetes Objects

- Objects in the Kubernetes API are represented as JSON or YAML file

```
kubectl apply -f obj.yaml

kubectl edit <resource-name> <obj-name>

kubectl describe <resource-name> <obj-name>

kubectl delete -f obj.yaml
```

Labels and annotations

Resources can have labels and annotations (metadata), more on this later.

```
kubectl label pods bar color=red

# remove label color:
kubectl label pods bar color-

kubectl annotate pods bar color
```

Applying labels to pods

```
kubectl run my-pod --image nginx \
-- labels="env=prod, app=web"

kubectl run my-pod2 --image nginx \
-- labels="env=staging, app=web"

kubectl label deployments my-prod "other=label"
```

Label selectors

```
kubectl get pods --show-labels

kubectl get pods --selector="app=web“

# logical AND
kubectl get pods --selector="app=web,env=staging“

# in set (notin)
kubectl get pods --selector="app in (web,api)“

# has label (!)
kubectl get pods --selector="app"
```

Label selectors in API Objects (manifest)

- Referring to other Kubernetes objects by using labels
- Instead of a simple string we can query objects

```
spec:
  selector:
    matchLabels:
      app: myapp
    matchExpressions:
      - key: env
        operator: In
        values:
          - "staging"
```

Annotations

- Labels are primarily used to identify and group objects
- Annotations provide extra information
- Object-scoped key/value store
- When in doubt use an annotation and promote to label if you need a selector using it.
- I.e. Build, release or image information not appropriate for labels
- Deployments can use it to manage rollouts (more later)
  - Track status, rollback information, previous states

Annotations

- Same as label keys. For Annotations it’s more common to include ‘namespace’ information:
  - deployment.kubernetes.io/revision
  - kubernetes.io/change-cause
- Value is free-form text
  - Might contain JSON

Deployments

Deployments

- Simply and reliably roll out new software versions
- No downtime or errors
- Managed by Deployment Controller in K8s
- Can run unattended

Deployments

- So far we've created pods:
  `kubectl run nginx --image=nginx:1.7.12`

- Actually that created also a deployment in the background

```
kubectl get deployments nginx \
  -o jsonpath --template {.spec.selector.matchLabels}
```

Replicasets

- Deployments manage replicasets
- Replicasets manage pods

```
kubectl get replicasets --selector=run=nginx
kubectl scale deployments nginx --replicas=2
```

Replicasets

- Redundancy

  Multiple running instances mean failure can be tolerated.

- Scale

  Multiple running instances mean that more requests can be handled.

- Sharding

  Different replicas can handle different parts of a computation in parallel.

Replicasets

- Describe desired state
  - How many of which containers should run?
- Reconciliation loop in Kubernetes monitors and takes action
- Labels are used to find the corresponding pods

Replicaset example

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
        version: "2"
    spec:
      containers:
        - name: nginx
          image: nginx:latest
```

Replicaset – Pod templates

```
# Pods are recreated based on the Pod template supplied:
template:
  metadata:
    labels:
      app: helloworld
      version: v1
  spec:
    containers:
      - name: helloworld
        image: kelseyhightower/helloworld:v1
        ports:
          - containerPort: 80
```

Replicasets

```
kubectl get pods -l app=nginx,version=2

# imperative scaling
kubectl scale replicasets nginx --replicas=4

kubectl autoscale rs kuard --min=2 --max=5 --cpu-percent=80

kubectl get horizontalpodautoscalers
kubectl get hpa

kubectl delete rs kuard --cascade=false
```

Deployments

- Get the deployment:

```
kubectl get deployments nginx --export -o yaml > nginx-deployment.yaml

kubectl replace -f nginx-deployment.yaml --save-config
```

Deployment example

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  labels:
    run: nginx
  name: nginx
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      run: nginx
```

```
strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
 template:
    metadata:
      labels:
        run: nginx
    spec:
      containers:
      - image: nginx:1.7.12
        imagePullPolicy: Always
      dnsPolicy: ClusterFirst
      restartPolicy: Always
```

Manage deployments

```
kubectl apply –f nginx-deployment.yml

kubectl rollout status deployments nginx

kubectl rollout pause deployments nginx

kubectl rollout resume deployments nginx
```

Manage deployments

```
kubectl rollout history deployment nginx
```

```
kubectl rollout undo deployments nginx --to-revision=3
```

Deployment strategies

- Recreate
  - Simpler version
  - Changes replicaset
  - Terminates pods
  - Re-creates pods with new version

- RollingUpdate
  - Incrementally updates pods to newer version
  - No downtime
  - Slower

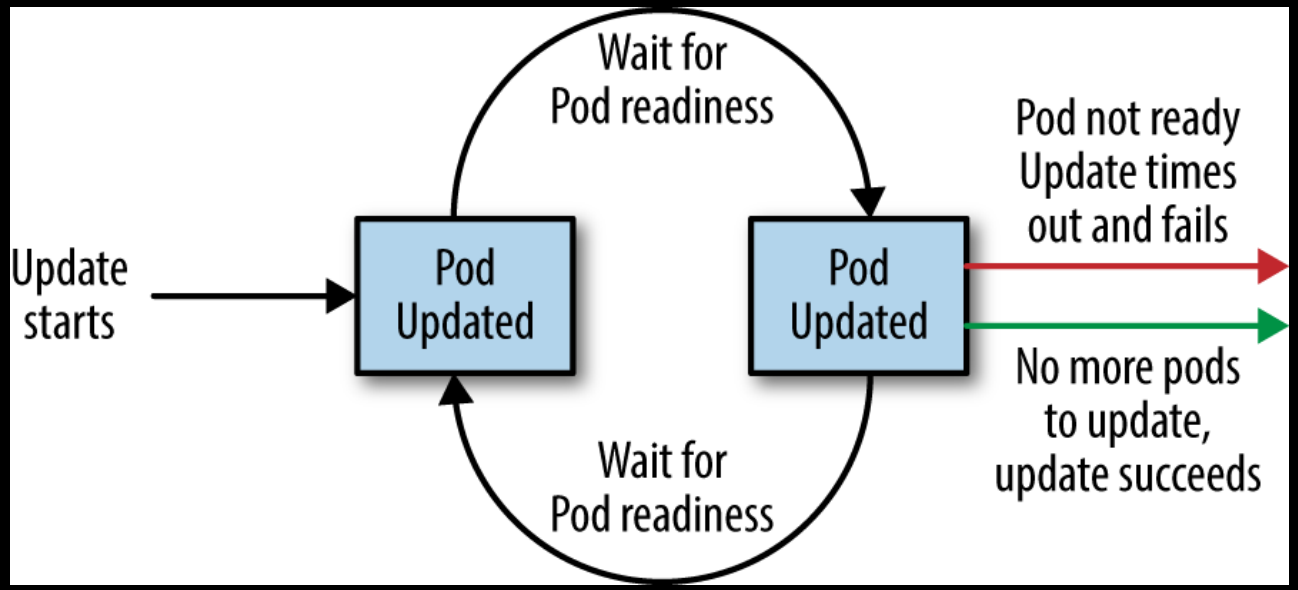
Rolling update

- With rollingupdates we have some control with the following settings:

- maxUnavailable
  - How many pods can the update take down?
  - Set to fixed number or percentage
- maxSurge
  - How many additional resources might be used during update?
  - maxUnavailable = 0%, maxSurge = 25%

Rolling update

- minReadySeconds
  - Time to wait between updating pods
  - Useful to slow down migration process
- progressDeadlineSeconds
  - Maximum time updating one



Deleting deployment

```
kubectl delete deployments nginx
```

```
kubectl delete -f nginx-deployment.yaml
```