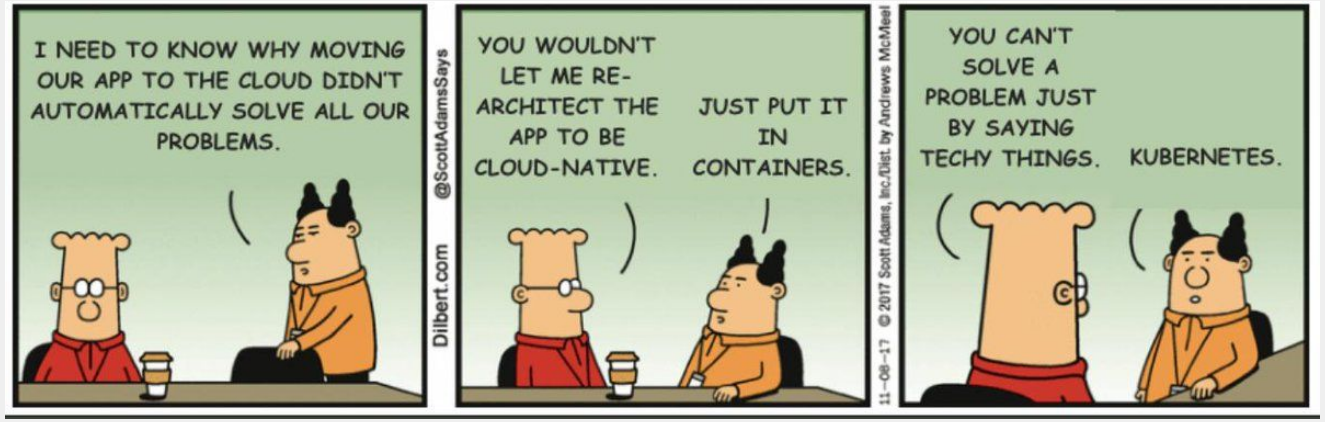


Best Practices for Container Based Design

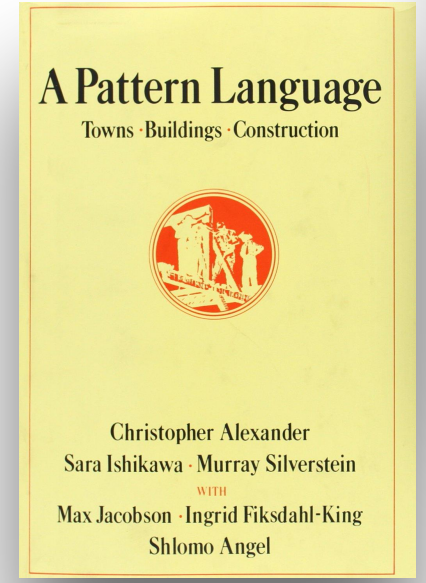
Microservices and Kubernetes-based Principles and Patterns

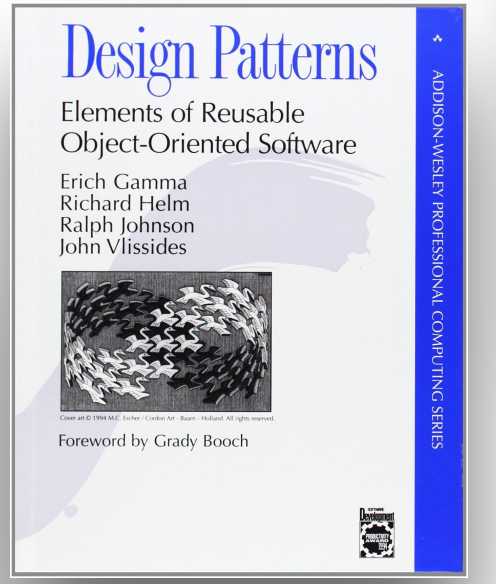
Eric Deandrea
Senior Principal Developer Advocate
edeandrea@redhat.com

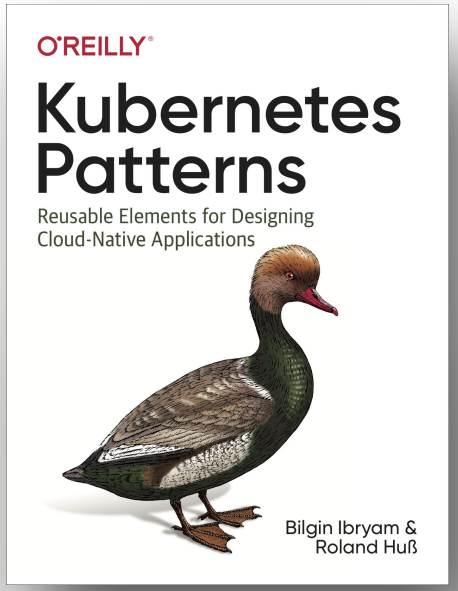
Red Hat

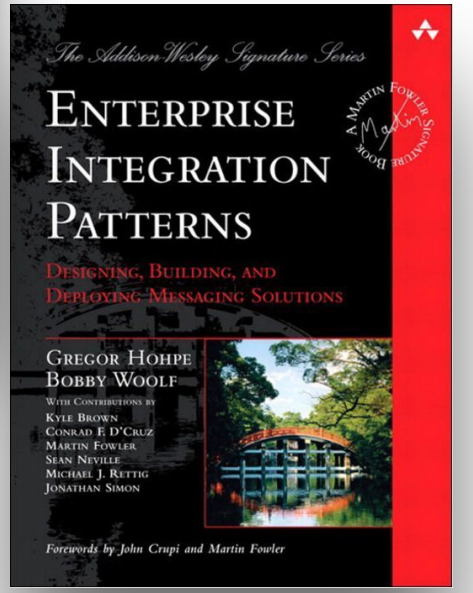
Agenda

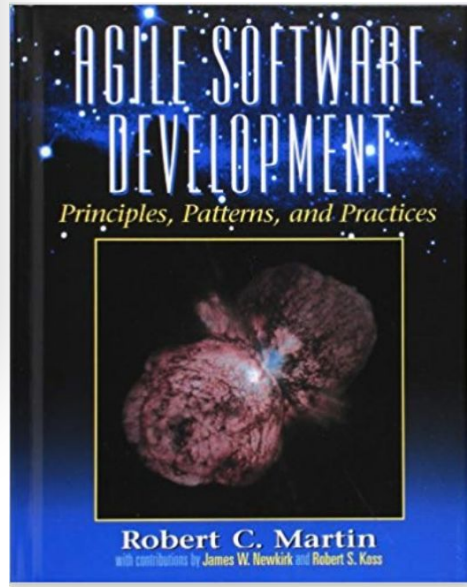
- Container Design Principles
- Kubernetes Patterns
  - Foundational Patterns
  - Structural Patterns
  - Configurational Patterns
  - Behavioral Patterns
  - Advanced Patterns





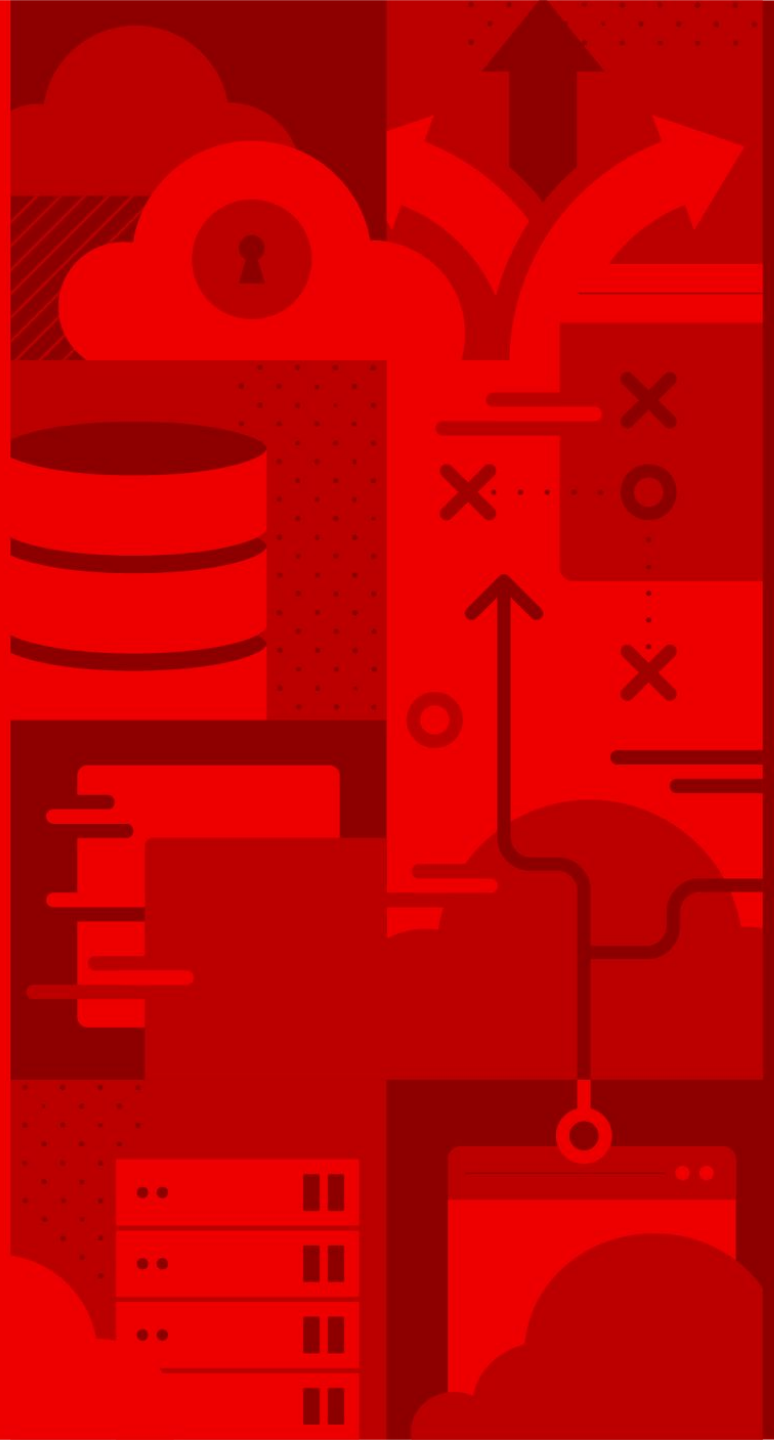






https://k8spatterns.io
https://github.com/k8spatterns/examples
https://www.redhat.com/en/engage/kubernetes-containers-architecture-s-201910240918
https://developers.redhat.com/blog/2020/06/08/commit-to-excellence-java-in-containers

Red Hat

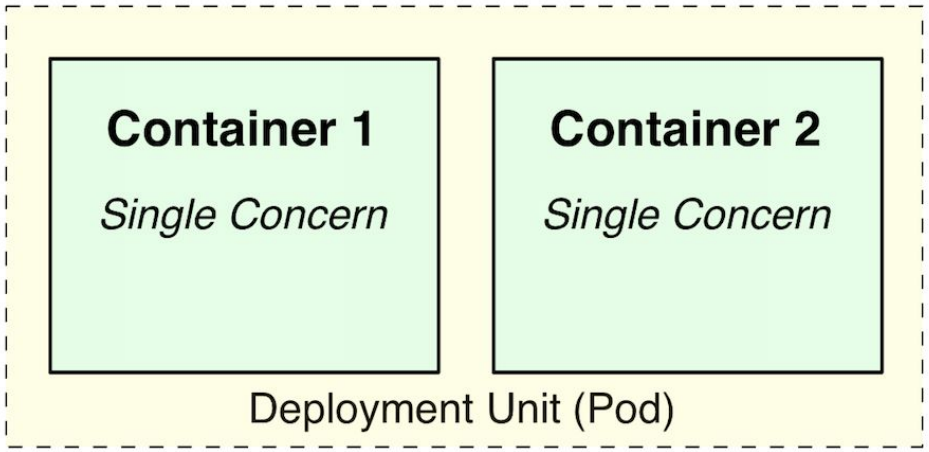


Container Design Principles

Red Hat

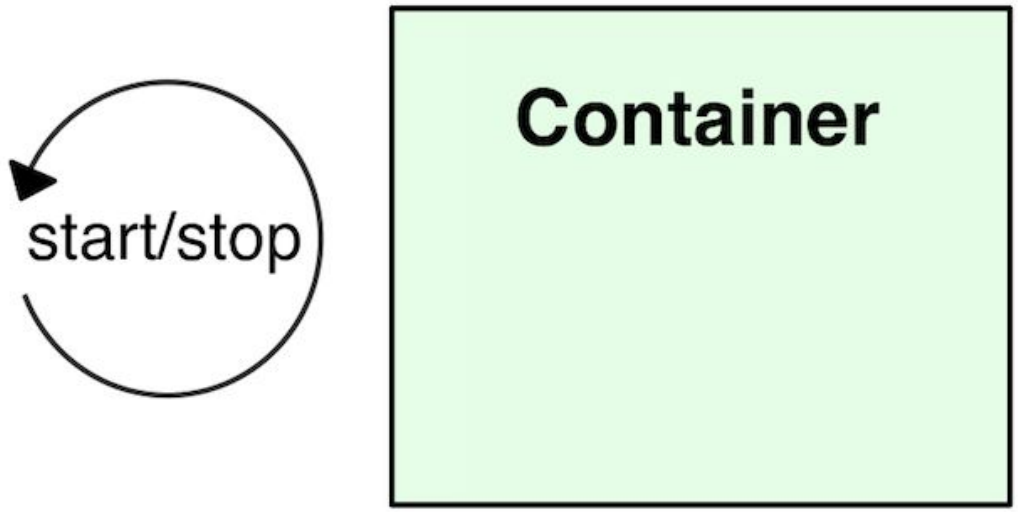
Single Concern Principle (SCP)

- Every container should address a single concern and do it well
- Use sidecars and/or init-containers to combine multiple containers into a single pod
  - Each container still handles a single concern



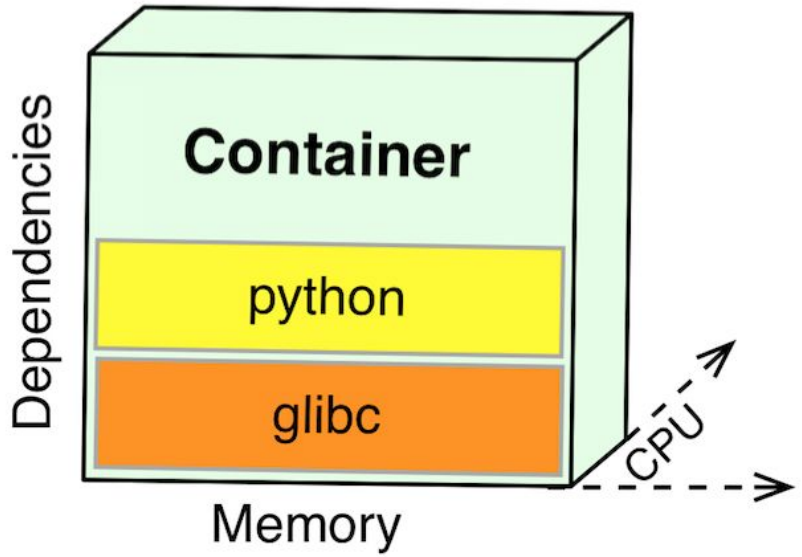
Process Disposability Principle (PDP)

- A running container should be as ephemeral as possible and ready to be replaced by another container instance at any time

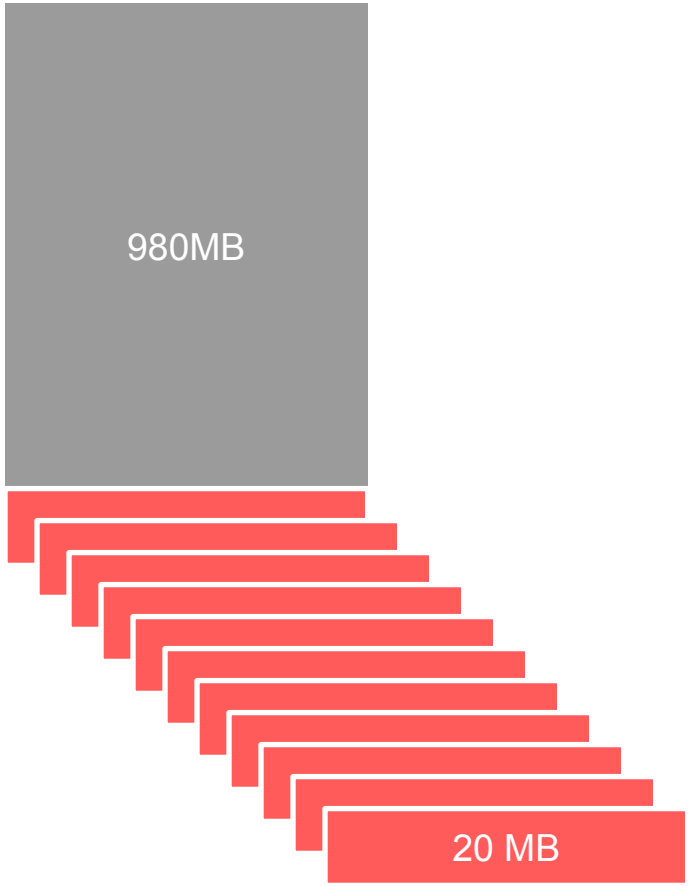


Runtime Confinement Principle (RCP)

- Containers have multiple dimensions at runtime, such as memory and CPU



Does Image Size Really Matter?



1 x 980 MB base image

10 x 20 MB project specific image

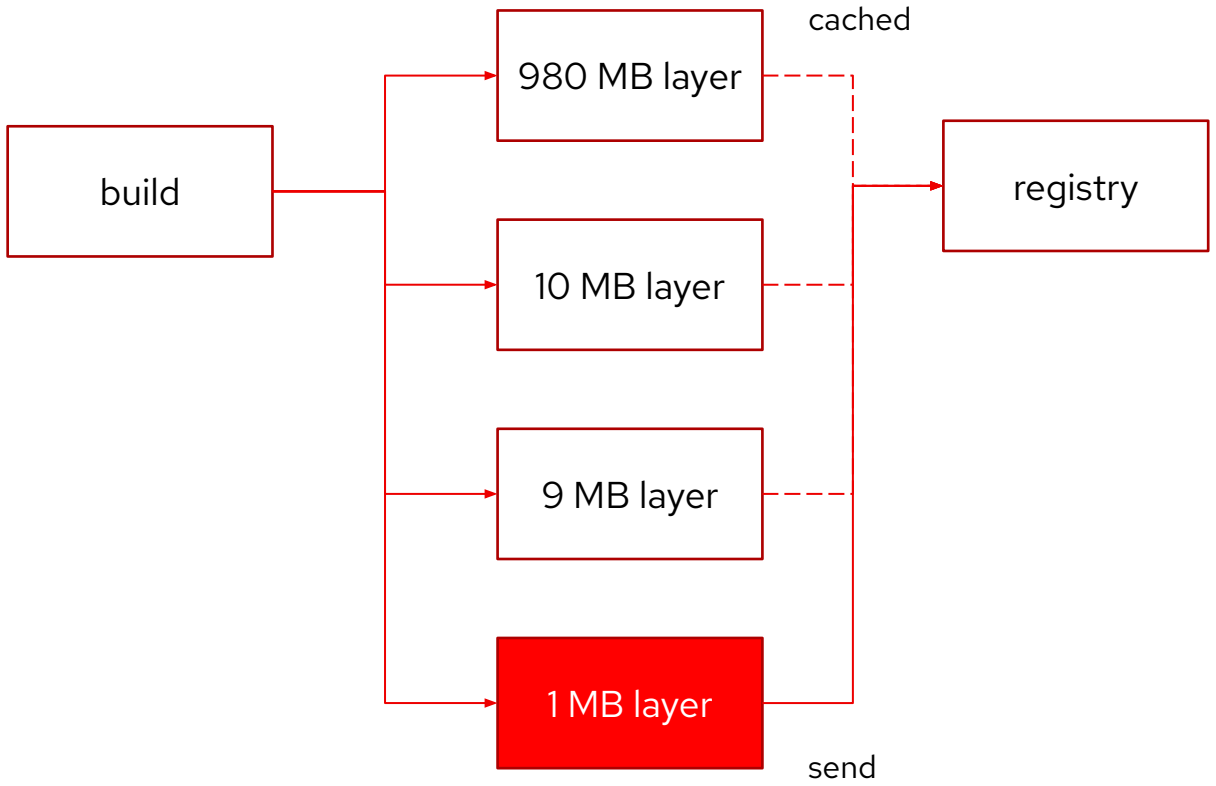
980 + 10*20 = 980 + 200 = **1180 MB**

≈ **200 MB**

- Total image size does not matter!
- Base image size does not matter!
- What really matters is the size of frequently changing layers!

https://developers.redhat.com/blog/2020/06/08/commit-to-excellence-java-in-containers-5

Construct Application Layers the Right Way

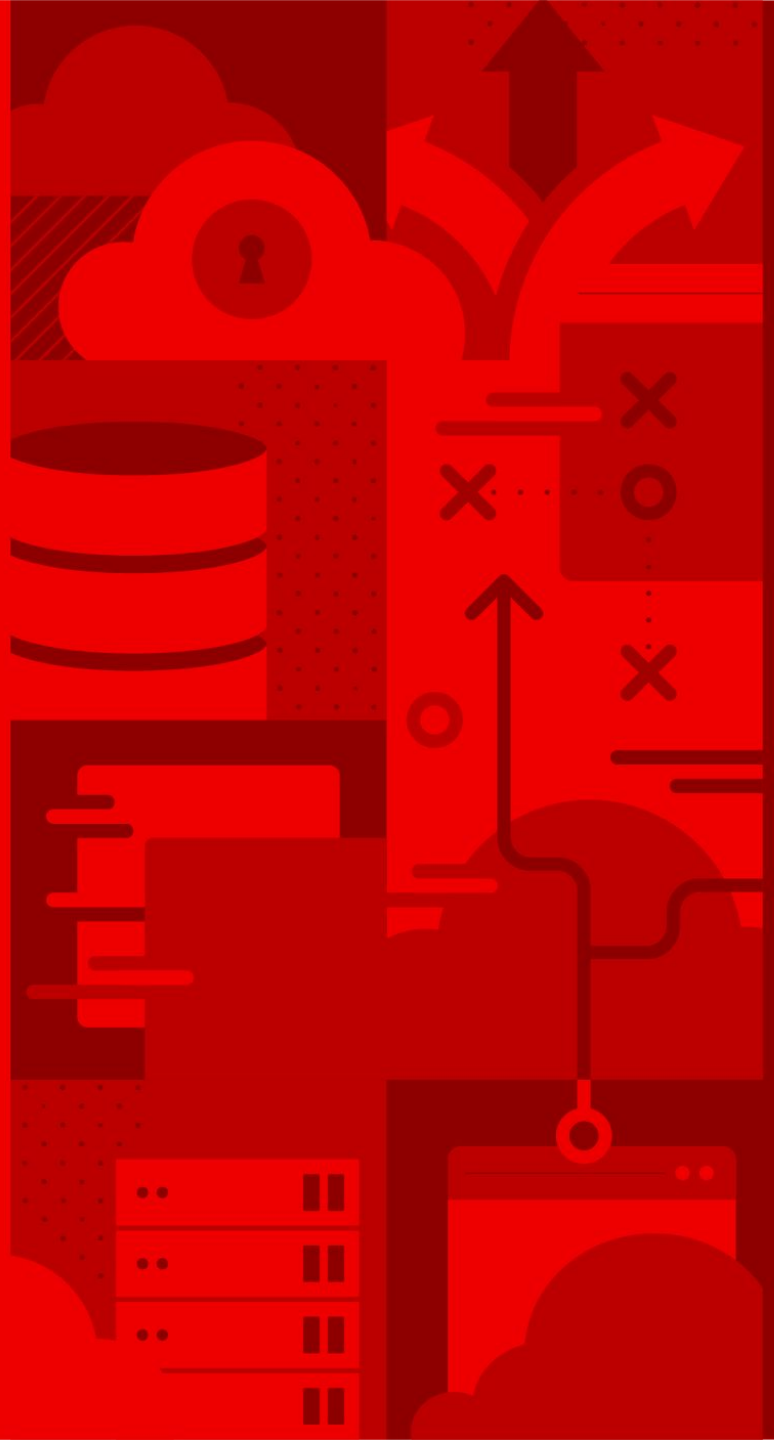


https://developers.redhat.com/blog/2020/06/08/commit-to-excellence-java-in-containers-5

Construct Application Layers the Right Way

980 MB layer	FROM registry.redhat.io/ubi8/openjdk-17:latest
10 MB layer	COPY target/dependencies /app/dependencies
9 MB layer	COPY target/resources /app/resources
1 MB layer	COPY target/classes /app/classes
	ENTRYPOINT java –cp /app/dependencies/*:/app/resources:/app/classes my.app.Main

https://developers.redhat.com/blog/2020/06/08/commit-to-excellence-java-in-containers-5
https://spring.io/guides/topicals/spring-boot-docker/



Foundational Patterns

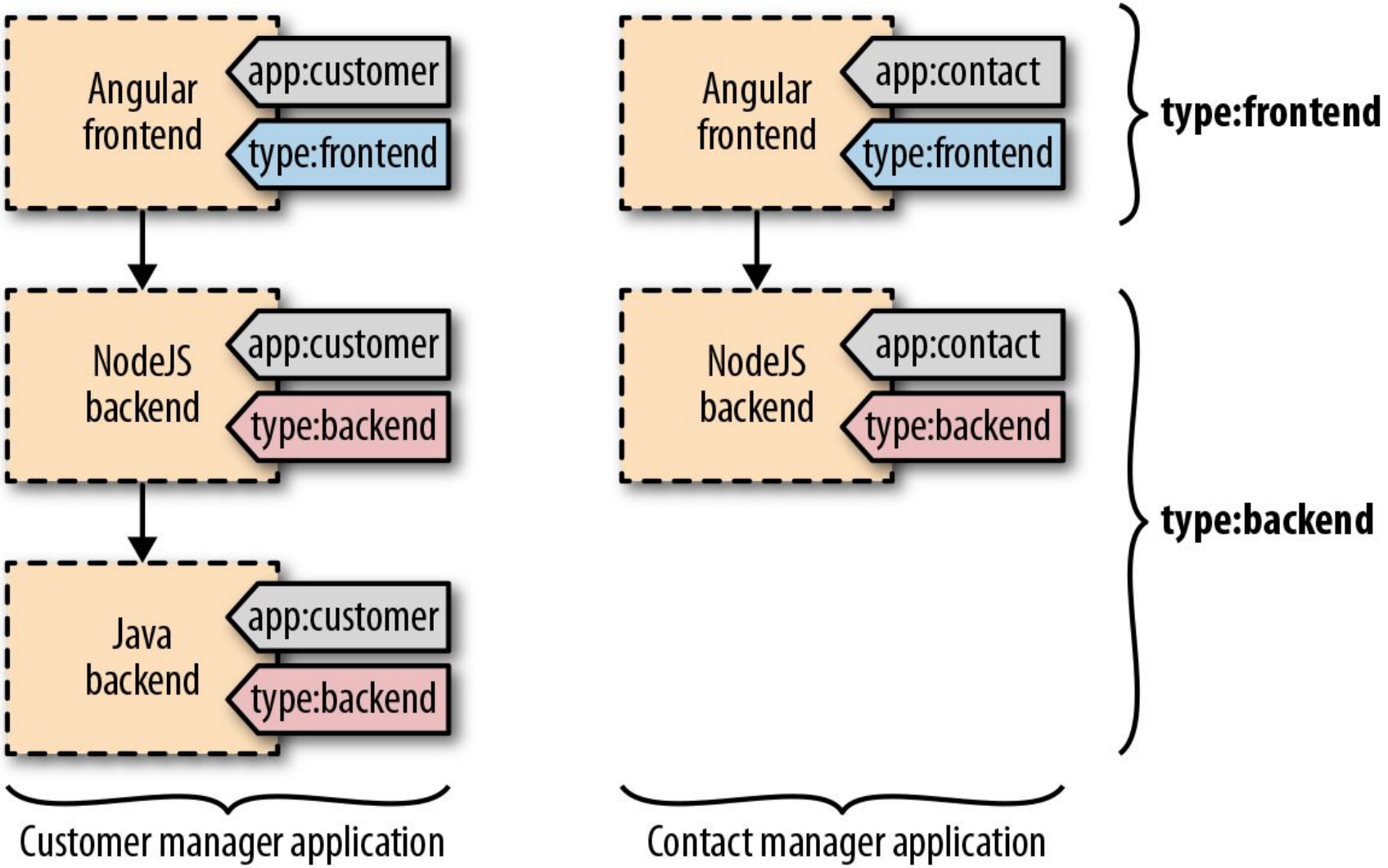
Red Hat

Automatable Unit

How can we create and manage applications with Kubernetes.

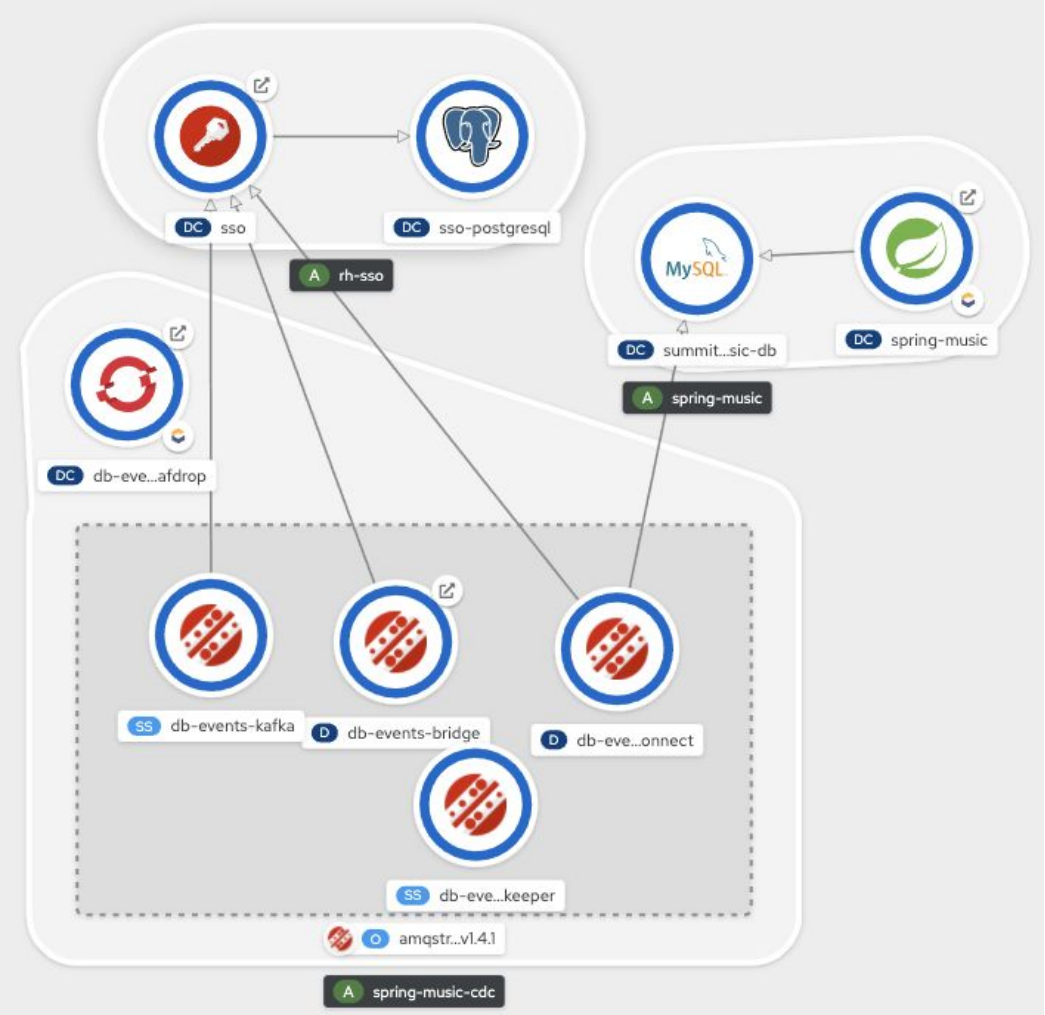
- Pods
  - Atomic unit within Kubernetes
- Services
  - Entry point to pods
- Grouping via Labels, Annotations, and Namespaces

Labels



Special Labels/Annotations

- app.kubernetes.io/part-of
- app.openshift.io/runtime
- app.openshift.io/runtime-version
- app.openshift.io/vcs-url
- app.openshift.io/vcs-ref
- app.openshift.io/connects-to



https://docs.openshift.com/container-platform/4.12/applications/odc-viewing-application-composition-using-topology-view.html#odc-labels-and-annotations-used-for-topology-view_viewing-application-composition-using-topology-view

Predictable Demands

How can we handle resource requirements deterministically?

- Declared requirements
  - Scheduling decisions
  - Capacity planning
  - Matching infrastructure services
- Runtime dependencies
  - Persistent Volumes
  - Host ports
  - Dependencies on **ConfigMaps** and **Secrets**

Resource Profile Requests & Limits

- **Resources**
  - CPU, Network (compressible)
  - Memory (incompressible)
- **App**
  - Declaration of resource requests and limits
- **Platform**
  - Resource quotas and limit ranges

Quality-of-Service Classes

- **Best Effort**
  - No requests or limits
  - Lowest priority
- **Burstable**
  - requests < limits
- **Guaranteed**
  - requests == limits
  - Highest priority

Health Probe

How to communicate an application's health state to Kubernetes?

- **Process health checks**
  - Checks running process
  - Restarts container if container process died
- **Application provided Health Probes**
  - Liveness Probe - Check application health
  - Readiness Probe - Check readiness to process requests

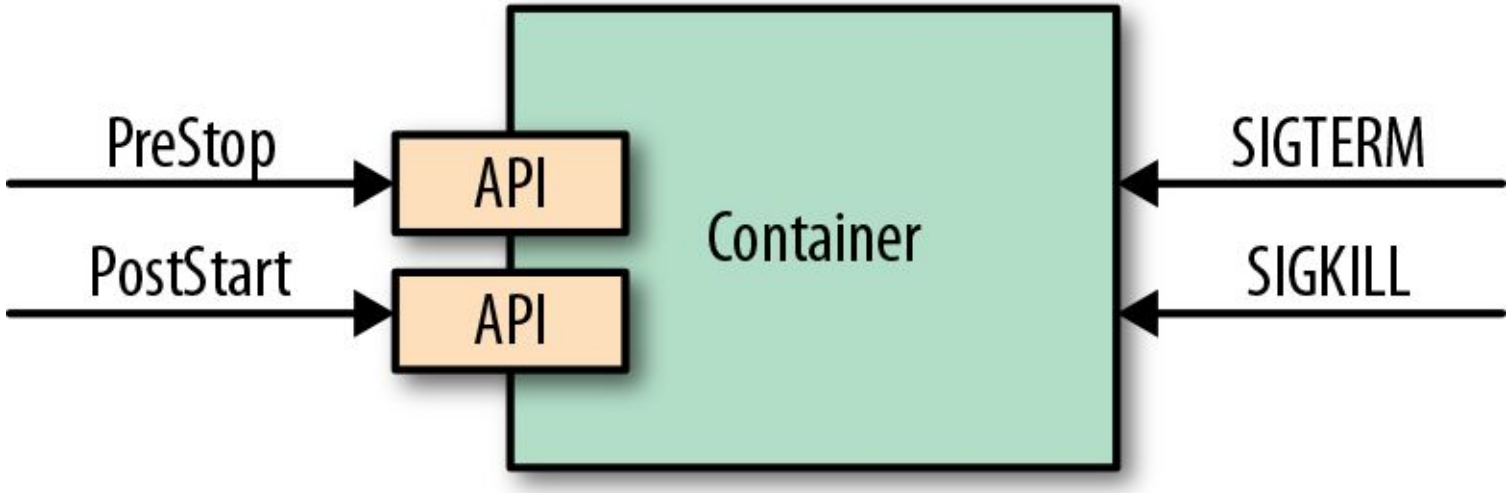
Liveness & Readiness

- Liveness Probe
  - Restarting containers if liveness probes fail
- Readiness Probe
  - Removing from service endpoint if readiness probe fails
- Probe methods
  - HTTP endpoint
  - TCP socket endpoint
  - Unix command’s return value

```
apiVersion: v1
kind: Pod
metadata:
  name: app-with-readiness-check
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    livenessProbe:
      httpGet:
        path: /q/health
        port: 8080
      initialDelaySeconds: 30
    readinessProbe:
      exec:
        command: [ "stat", "/var/run/random-generator-ready" ]
```

Managed Lifecycle

How should applications react to lifecycle events?



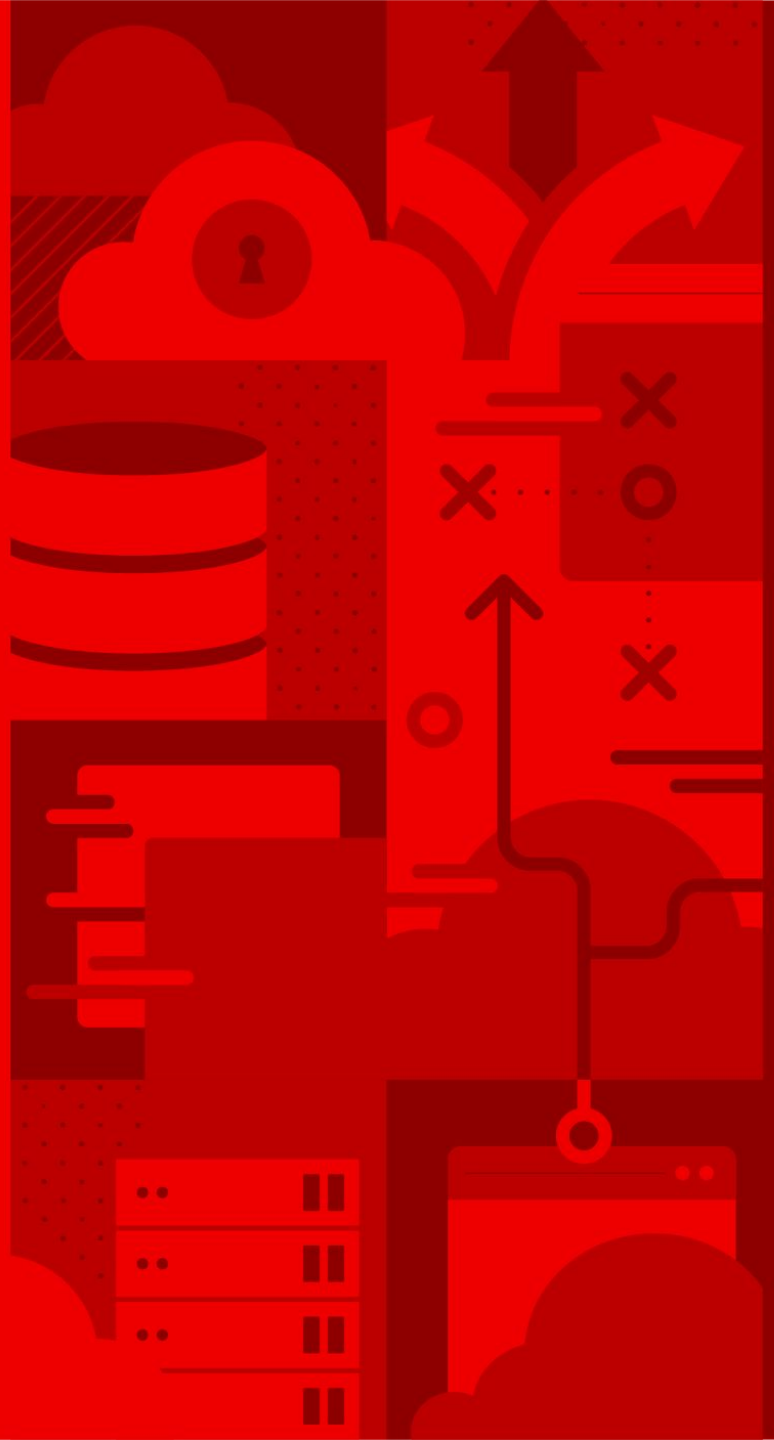
Lifecycle Events

- SIGTERM
  - Initial event issued when a container is going to shutdown
  - Application should listen to this event to cleanup properly and then exit
- SIGKILL
  - Final signal sent after a grace period
    - Can't be caught by application
  - `terminationGracePeriodSeconds`
    - Time to wait after SIGTERM (default: 30s)

Lifecycle Hooks

- postStart
  - Called after container is created
  - Runs in parallel to the main container
  - Keeps Pod in status *Pending* until exited successfully
  - `exec` or `httpGet` handler types (like *Health Probe*)
- preStop
  - Called before container is stopped
  - Same purpose & semantics as for SIGTERM

```
apiVersion: v1
kind: Pod
metadata:
  name: post-start-pre-stop-hook
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    lifecycle:
      postStart:
        exec:
          command:
          - sh
          - -c
          - sleep 30 && echo "Wake up!" > /tmp/postStart_done
      preStop:
        httpGet:
          port: 8080
          path: /shutdown
```



Structural Patterns

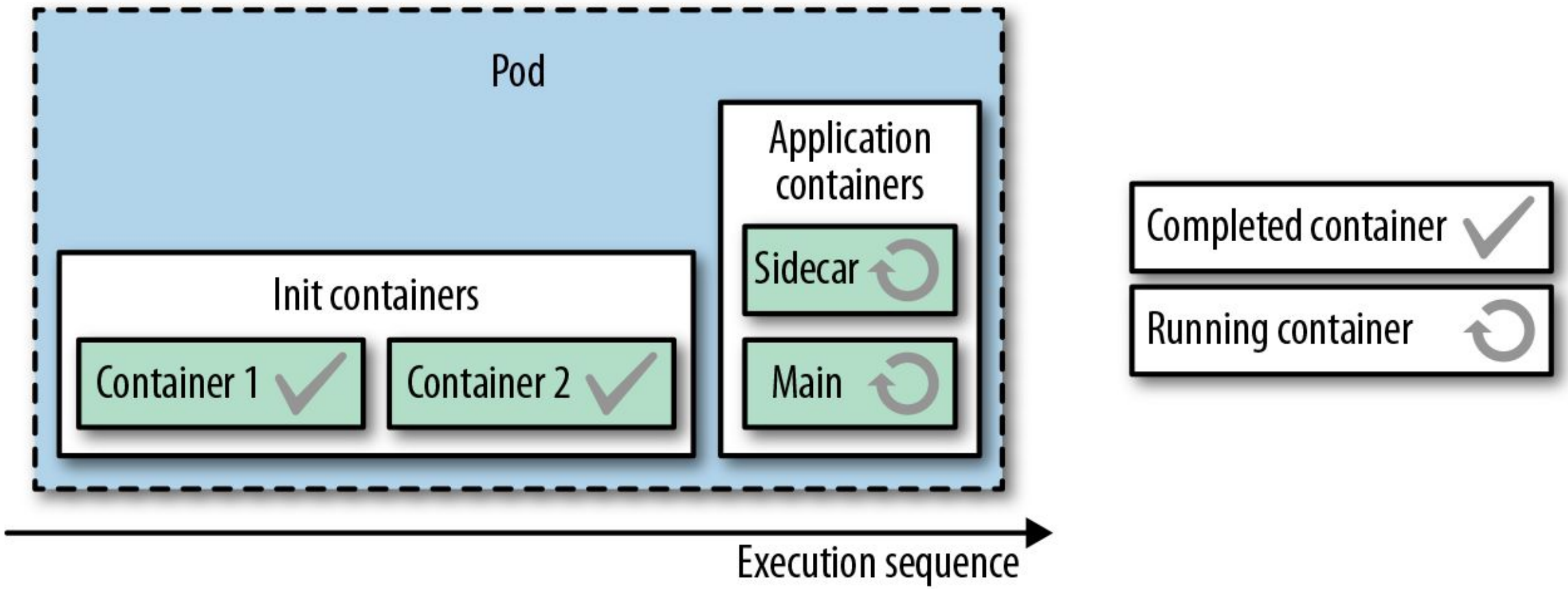
How can I structure & organize my container(s) within a Pod?

Red Hat

Init Container

How can we initialize our containerized applications?

- Init Containers
  - Part of a Pod
  - One shot actions before application starts
  - Needs to be idempotent
  - Has own resource requirements


Pod
Application containers
Sidecar
Main
Init containers
Container 1
Container 2
Execution sequence
Completed container
Running container

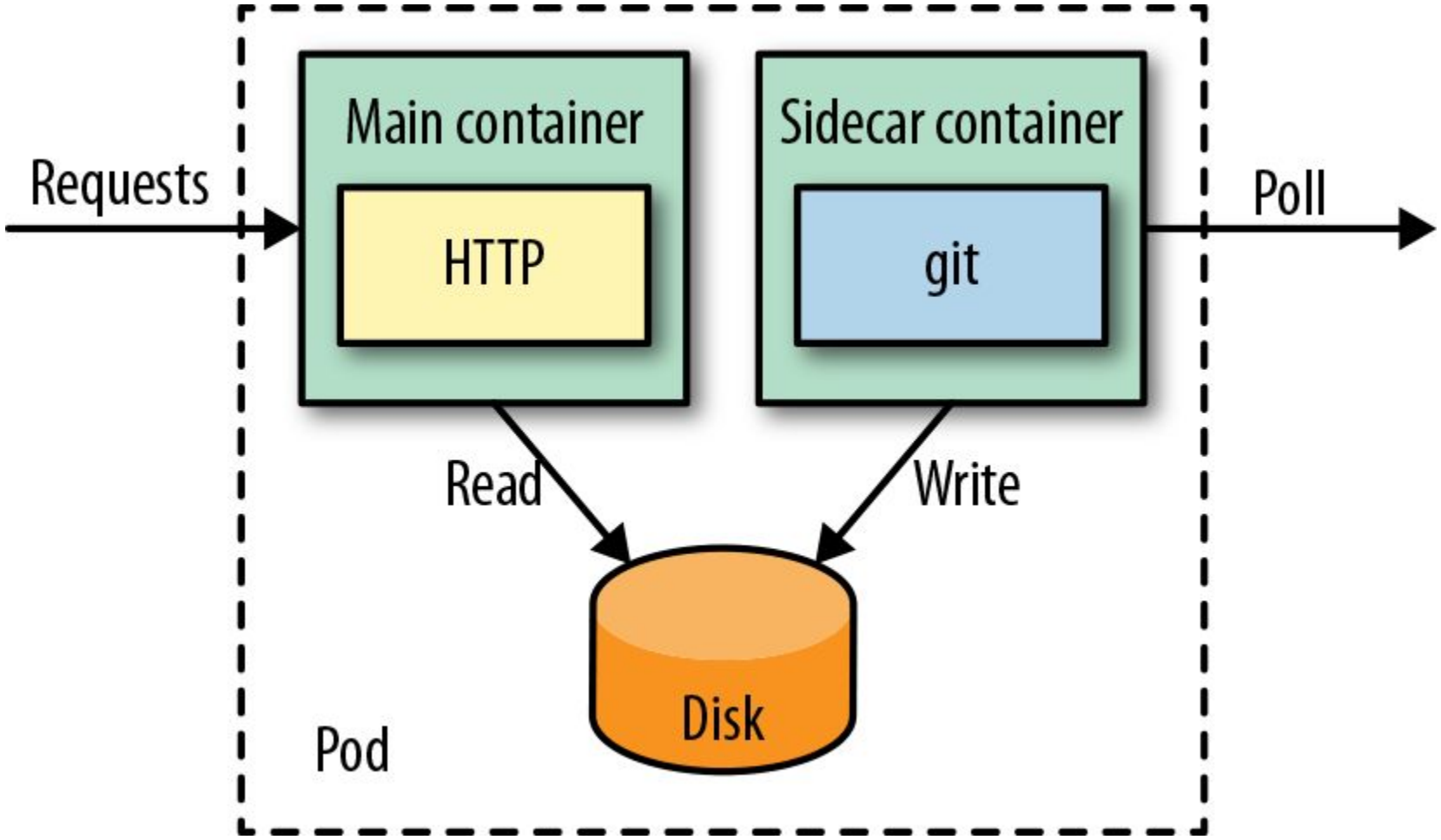
```
apiVersion: v1
kind: Pod
....
spec:
  initContainers:
  - name: download
    image: axeclbr/git
    command: [ "git","clone","https://github.com/myrepo","/data"]
    volumeMounts:
    - mountPath: /var/lib/data
      name: source
  containers:
  - name: run
    image: docker.io/centos/httpd
    volumeMounts:
    - mountPath: /var/www/html
      name: source
  volumes:
  - emptyDir: {}
    name: source
```

Sidecar Pattern

How do we enhance the functionality of an application without changing it?

- Runtime collaboration of containers
- Connected via shared resources
  - Network
  - Volumes
- Similar what AOP is for programming
- Separation of concerns

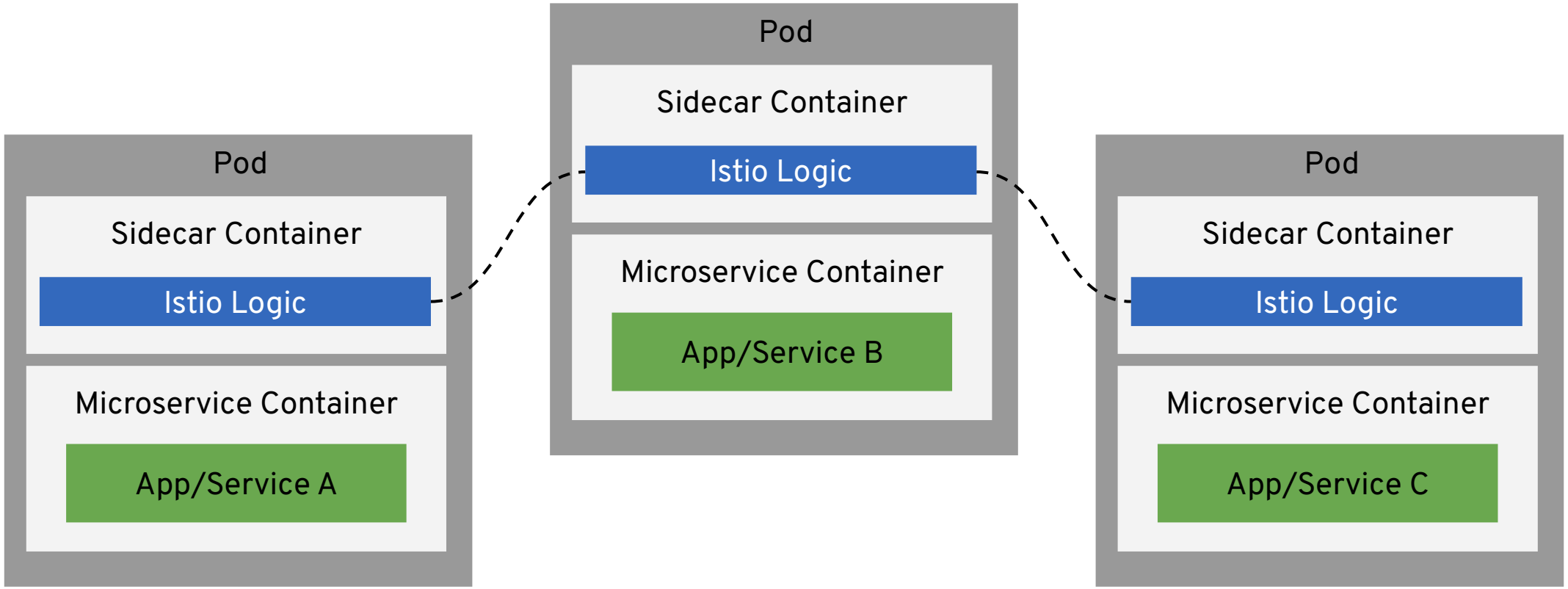
Sidecar



Red Hat

```
apiVersion: v1
kind: Pod
....
spec:
  containers:
  - name: app
    image: docker.io/centos/httpd
    ports:
    - containerPort: 80
    volumeMounts:
    - mountPath: /var/www/html
      name: git
  - name: gitpoll
    image: axeclbr/git
    volumeMounts:
    - mountPath: /var/lib/data
      name: git
    env:
    - name: GIT_REPO
      value: https://github.com/mdn/beginner-html-site-scripted
    command:
    - “sh”
    - “-c”
    - “git clone ${GIT_REPO} . && watch -n 600 git pull”
    workingDir: /var/lib/data
  volumes:
  - emptyDir: {}
    name: git
```

Service Mesh

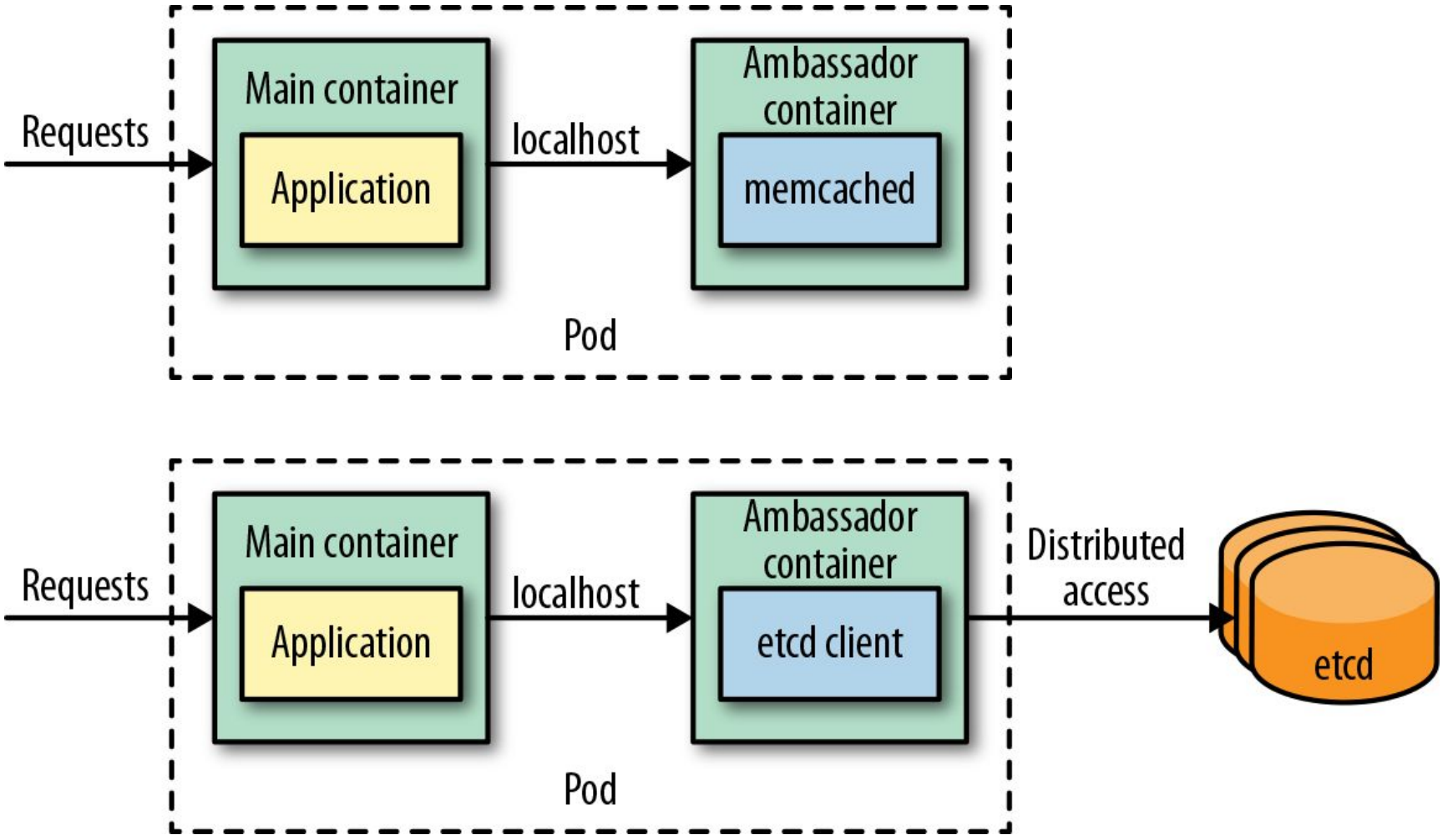


Ambassador Pattern

How to decouple a container's access to the outside world?

- Also known as **Proxy**
- Specialization of a Sidecar
- Examples for infrastructure services
  - Circuit breaker
  - Tracing

Ambassador



Red Hat

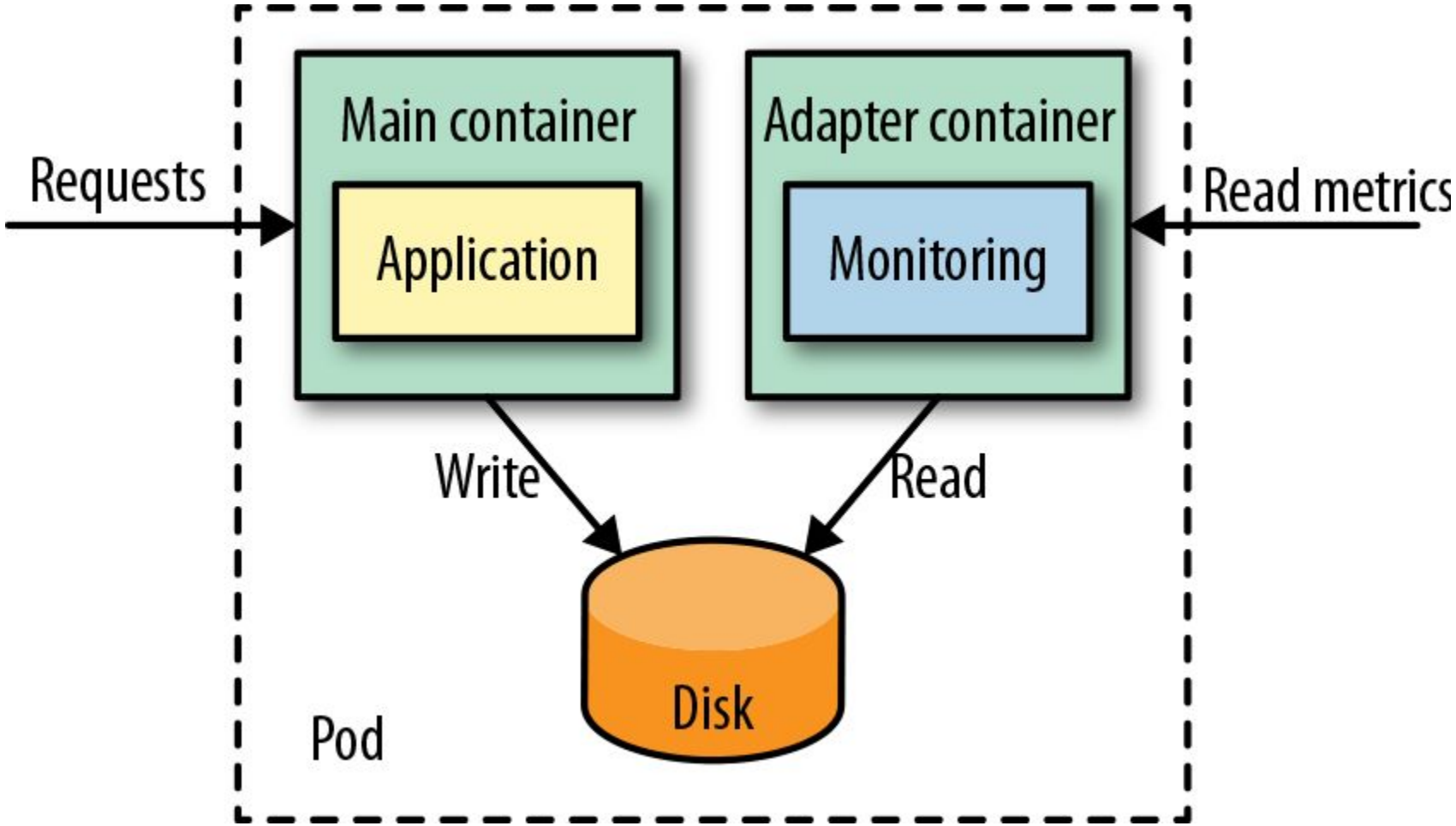
```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
  labels:
    app: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: main
    env:
    - name: LOG_URL
      value: http://localhost:9009
    ports:
    - containerPort: 8080
      protocol: TCP
  - image: k8spatterns/random-generator-log-ambassador
    name: ambassador
```

Adapter Pattern

How to decouple access to a container from the outside world?

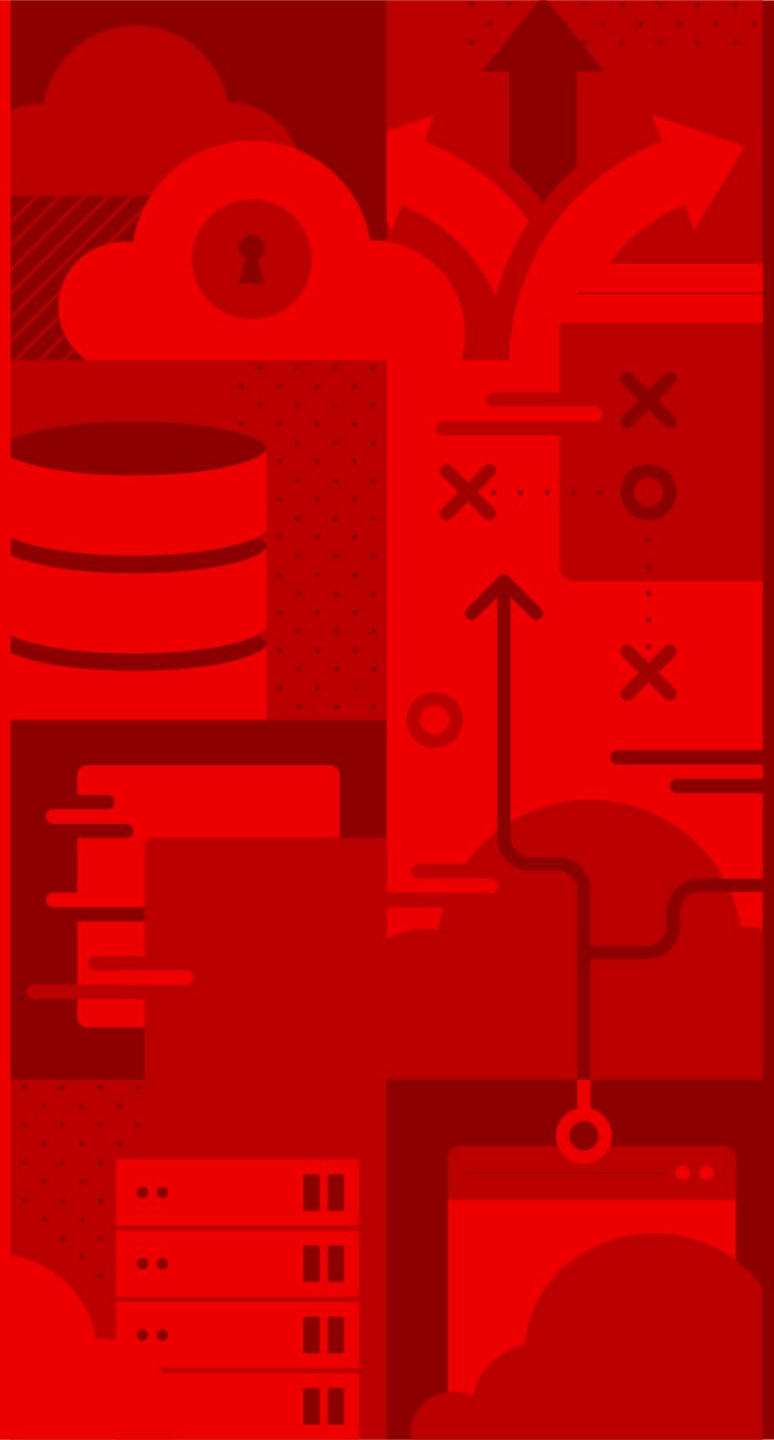
- Opposite of Ambassador
- Uniform access to an application
- Examples
  - Monitoring
  - Logging

Adapter



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: random-generator
spec:
  replicas: 1
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        env:
        - name: LOG_FILE
          value: /logs/random.log
        ports:
        - containerPort: 8080
          protocol: TCP
        volumeMounts:
        - mountPath: /logs
          name: log-volume
```

```
      - image:
k8spatterns/random-generator-exporter
        name: prometheus-adapter
        env:
        - name: LOG_FILE
          value: /logs/random.log
        ports:
          - containerPort: 9889
            protocol: TCP
        volumeMounts:
          - mountPath: /logs
            name: log-volume
      volumes:
      - name: log-volume
        emptyDir: {}
```



Configurational Patterns

How can applications be configured for different environments?

Red Hat

EnvVar Configuration

- Universally applicable
- Recommended by the *Twelve Factor App* manifesto
- **The Good**
  - Universal - every language understands them
- **The Bad**
  - Can only be set during startup of an application
  - Tedious if lots of them
  - Hard to know where they come from

ConfigMap

- Key-Value Map
- Use in Pods as
  - Environment variables
  - Volumes with keys as file names and values as file content

```
kubectl create cm spring-boot-config \
   --from-literal=JAVA_OPTIONS=-Djava.security.egd=file:/dev/urandom \
   --from-file=application.properties
```

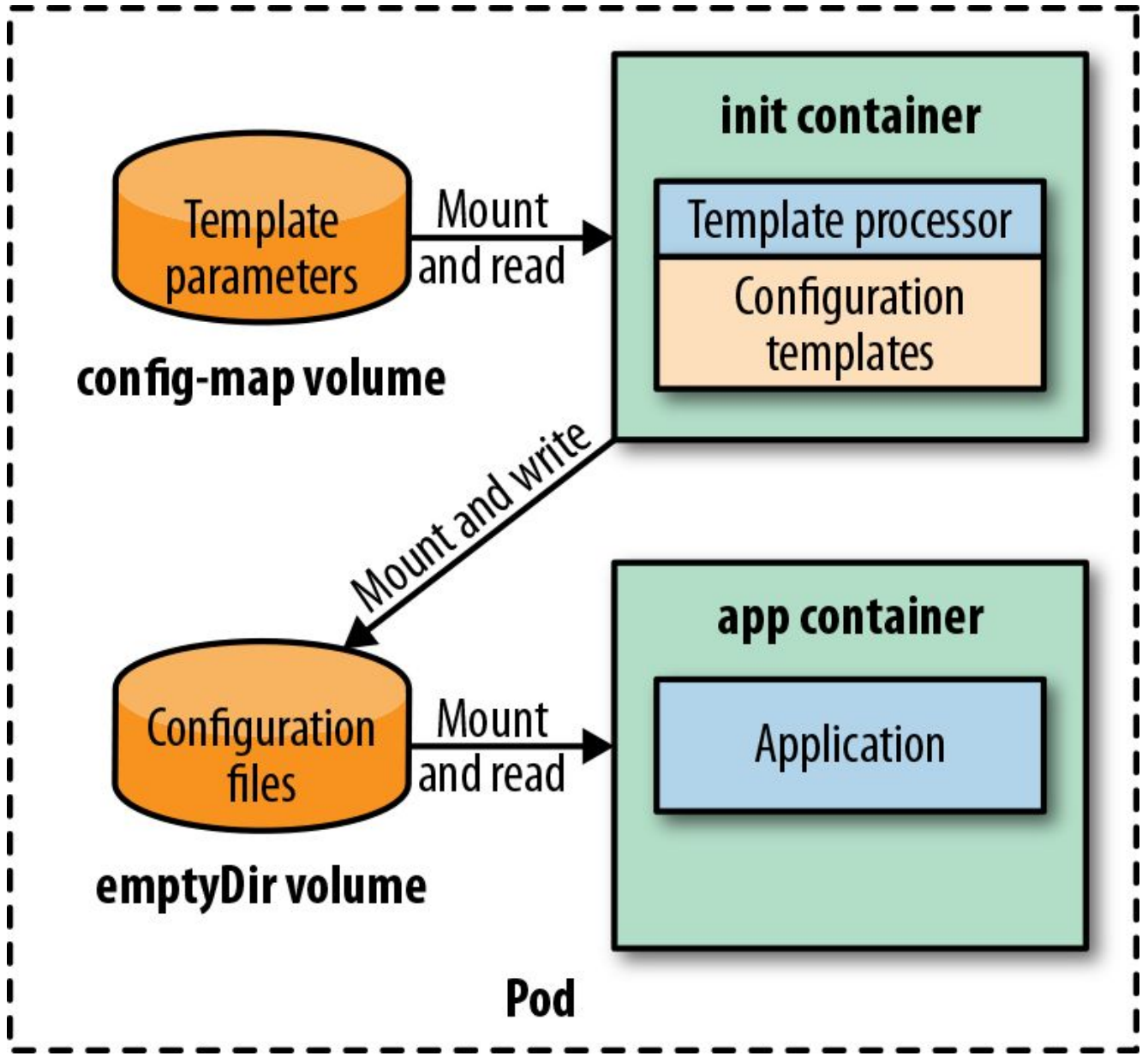
Secret

- Like **ConfigMap** but content Base64 encoded
- Secrets are ...
  - ... only distributed to nodes running Pods that need it
  - ... only stored in memory in a tmpfs and never written to physical storage
  - ... stored encrypted in the backend store (etcd)
- Access can be restricted with RBAC rules
- For high security requirements application based encryption is needed

Configuration Template

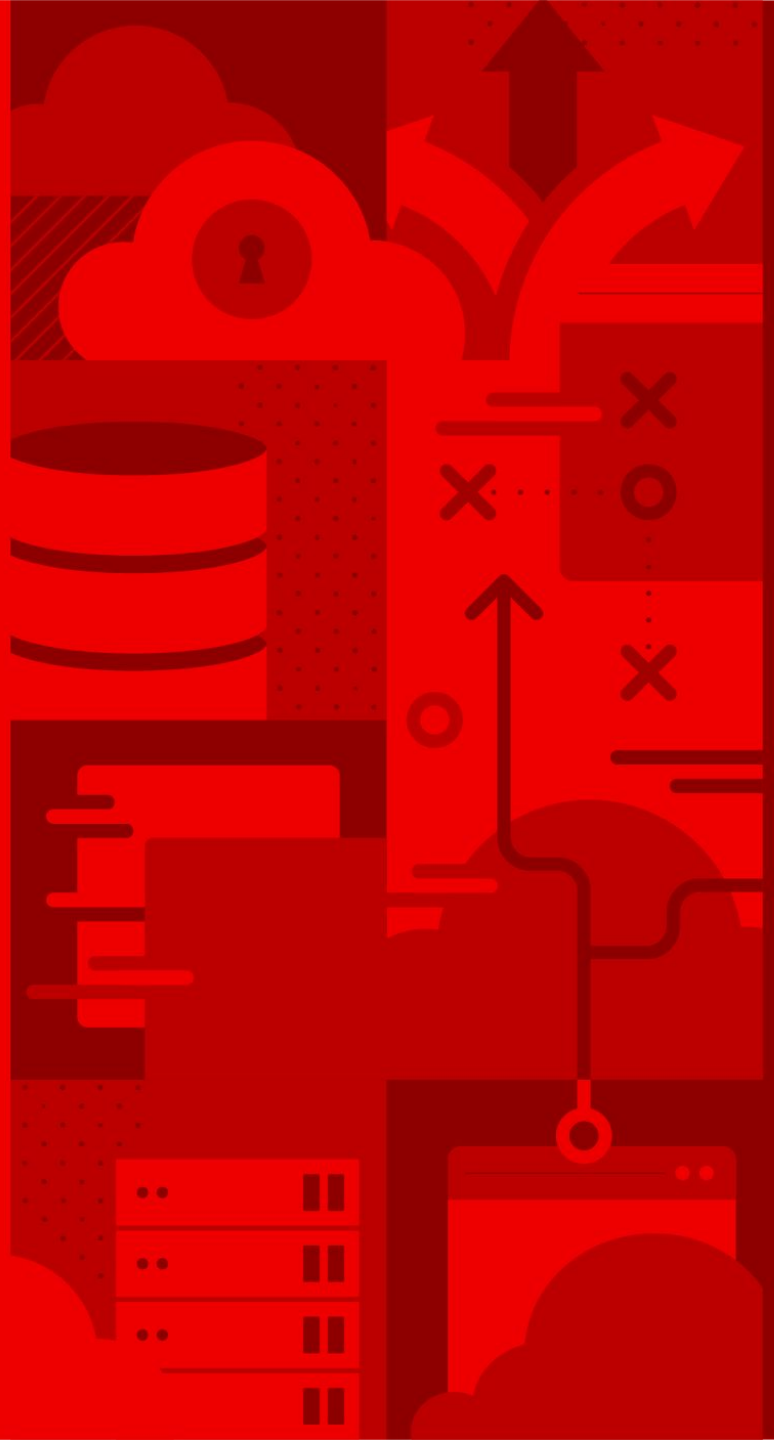
How to manage large and complex similar configuration data?

- **ConfigMap** not suitable for large configuration
- Managing similar configuration
- Ingredients
  - Init-container with template processor and templates
  - Parameters from a **ConfigMap** Volume
- Good for large, similar configuration sets per env
- Parameterization via **ConfigMaps** easy
- More complex


init container
Template processor
Configuration templates
Template parameters
Mount and read
config-map volume
Mount and write
app container
Application
Configuration files
Mount and read
emptyDir volume
Pod

```
apiVersion: v1
kind: Deployment
metadata:
  name: wildfly-cm-template
spec:
  replicas: 1
  template:
    spec:
      initContainers:
      - image: k8spatterns/config-init
        name: init
        volumeMounts:
        - mountPath: "/params"
          name: wildfly-parameters
        - mountPath: "/out"
          name: wildfly-config
```

```
containers:
- image: jboss/wildfly:10.1.0.Final
  name: server
  volumeMounts:
  - mountPath: "/config"
    name: wildfly-config
volumes:
- name: wildfly-parameters
  configMap:
    name: wildfly-params-cm
- name: wildfly-config
  emptyDir: {}
```



Behavioral Patterns

How can my application communicate/interact with the underlying platform?

Red Hat

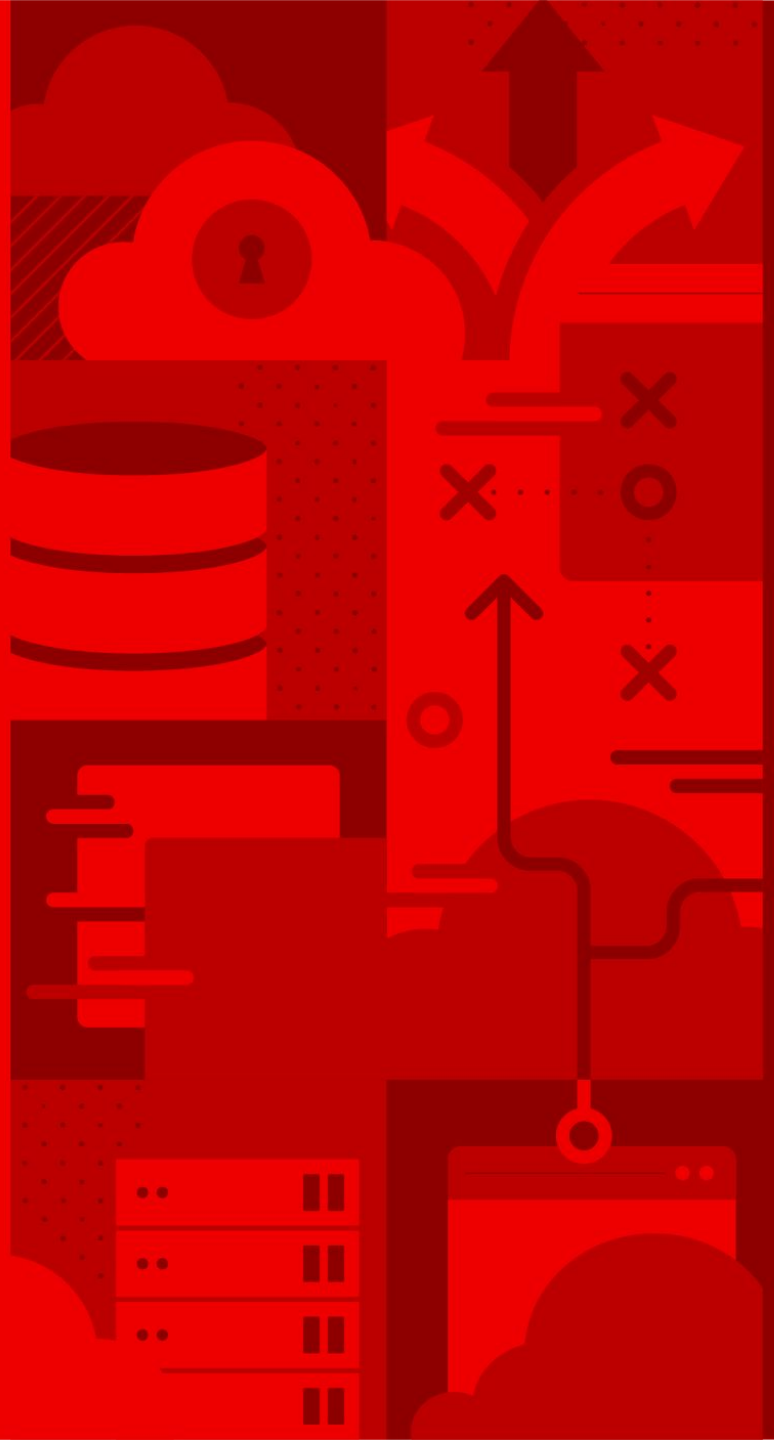
Self Awareness

How to gather information about a running Pod from within the Pod?

- Runtime data via Downward API
  - Pod name, ip address, hostname of underlying host
  - Specified resource requests & limits
  - Annotations & labels
- Application doesn’t need to use a client
- Application can remain Kubernetes agnostic

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: MEMORY_LIMIT
      valueFrom:
        resourceFieldRef:
          container: random-generator
```

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - name: pod-info
      mountPath: /pod-info
    volumes:
    - name: pod-info
      downwardAPI:
        items:
        - path: labels
          fieldRef:
            fieldPath: metadata.labels
        - path: annotations
          fieldRef:
            fieldPath: metadata.annotations
```



Advanced Patterns

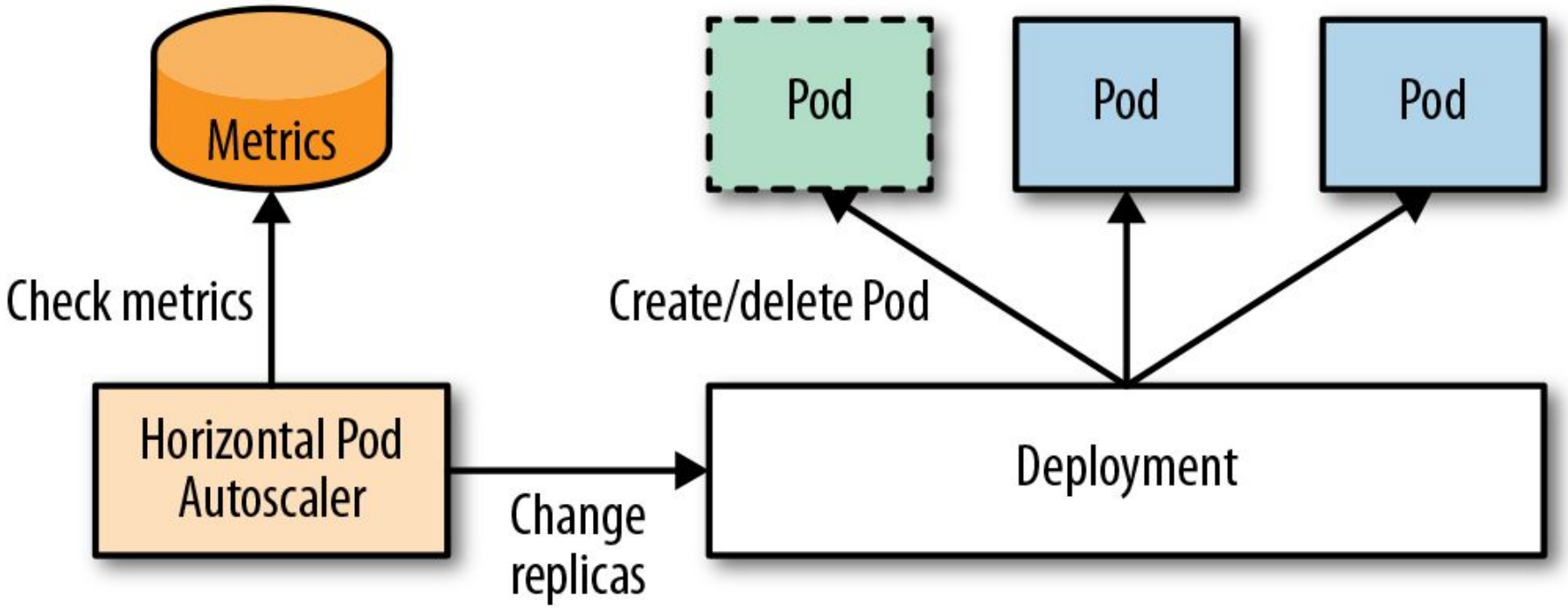
Red Hat

Elastic Scale

How to automatically react to dynamically changing resource requirements?

- **Horizontal:** Changing replicas of a Pod
- **Vertical:** Changing resource constraints of containers in a single Pod
- **Cluster:** Adding new nodes to a cluster
- **Manual:** Changing scale parameters manually, imperatively, or declaratively
- **Automatic:** Change scaling parameters based on observed metrics

Horizontal Pod Autoscaler (HPA)




```
kubectl autoscale deployment random-generator --cpu-percent=50 --min=1 --max=5
```

HorizontalPodAutoscaler

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: random-generator
spec:
  minReplicas: 1
  maxReplicas: 5
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: random-generator
  metrics:
  - resource:
      name: cpu
      target:
        averageUtilization: 50
        type: Utilization
    type: Resource
```

KEDA

Kubernetes Event Driven Autoscaling

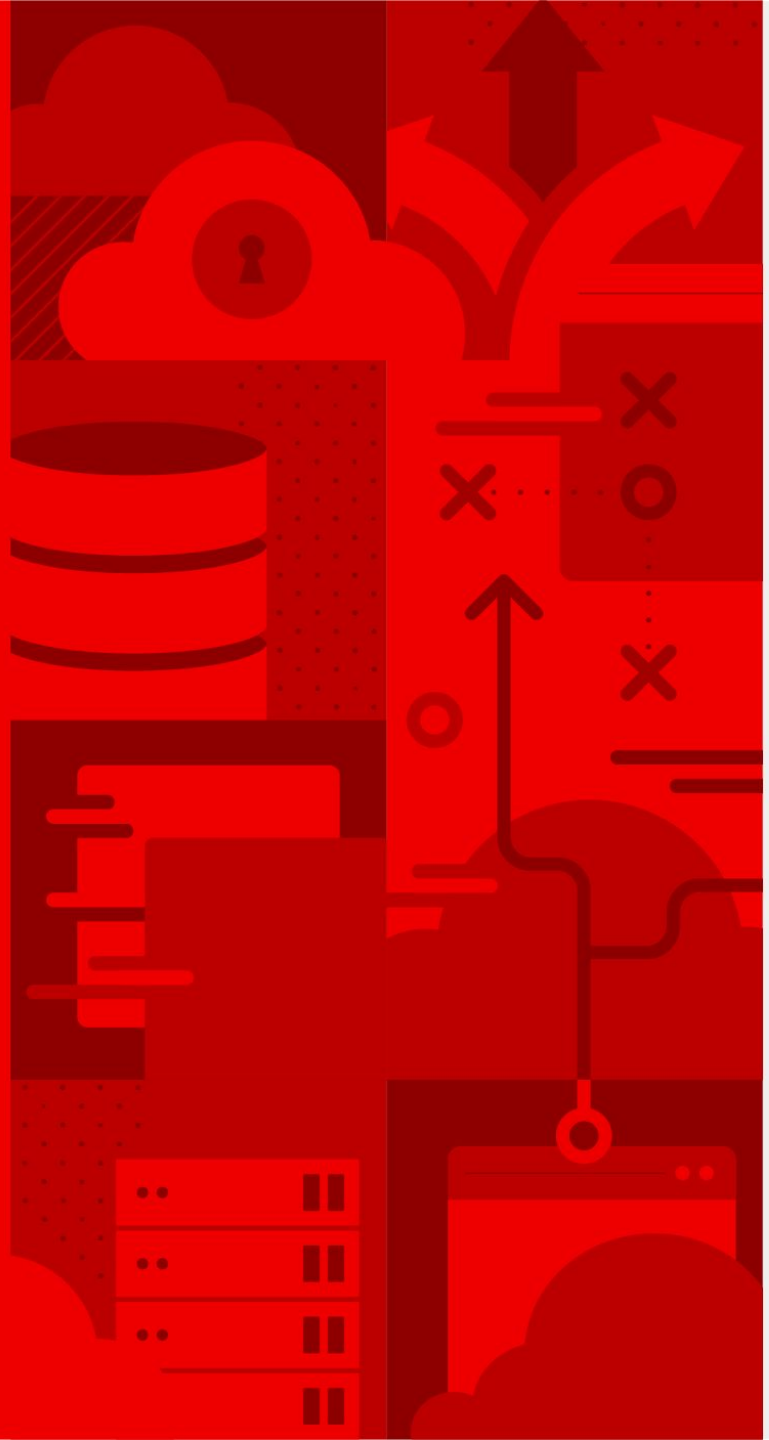
```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: random-generator-so
spec:
  minReplicaCount: 1
  maxReplicaCount: 5
  scaleTargetRef:
    name: random-generator
  triggers:
  - type: kafka
    metadata:
      bootstrapServers: kafka.svc:9092
      consumerGroup: my-group
      topic: my-topic
      lagThreshold: ‘5’
```


https://youtu.be/TwO5b7YCbTU

Red Hat

Metrics & Challenges

- Metrics
  - **Standard Metrics** - CPU & Memory Pod data obtained from Kubernetes metrics server
  - **Custom Metrics** - Metrics delivered via an aggregated API server at the `custom.metrics.k8s.io` API path
  - **External Metrics** - Metrics obtained from outside the cluster
- Challenges
  - **Metric Selection** - Correlation between metric value and replica counts
  - **Preventing Thrashing** - Windowing to avoid scaling on temporary spikes
  - **Delayed Reaction** - Delay between cause and scaling reaction



Thank you!

https://k8spatterns.io/

https://github.com/k8spatterns/examples

twitter.com/ro14nd

facebook.com/redhatinc

youtube.com/user/RedHatVideos

Red Hat