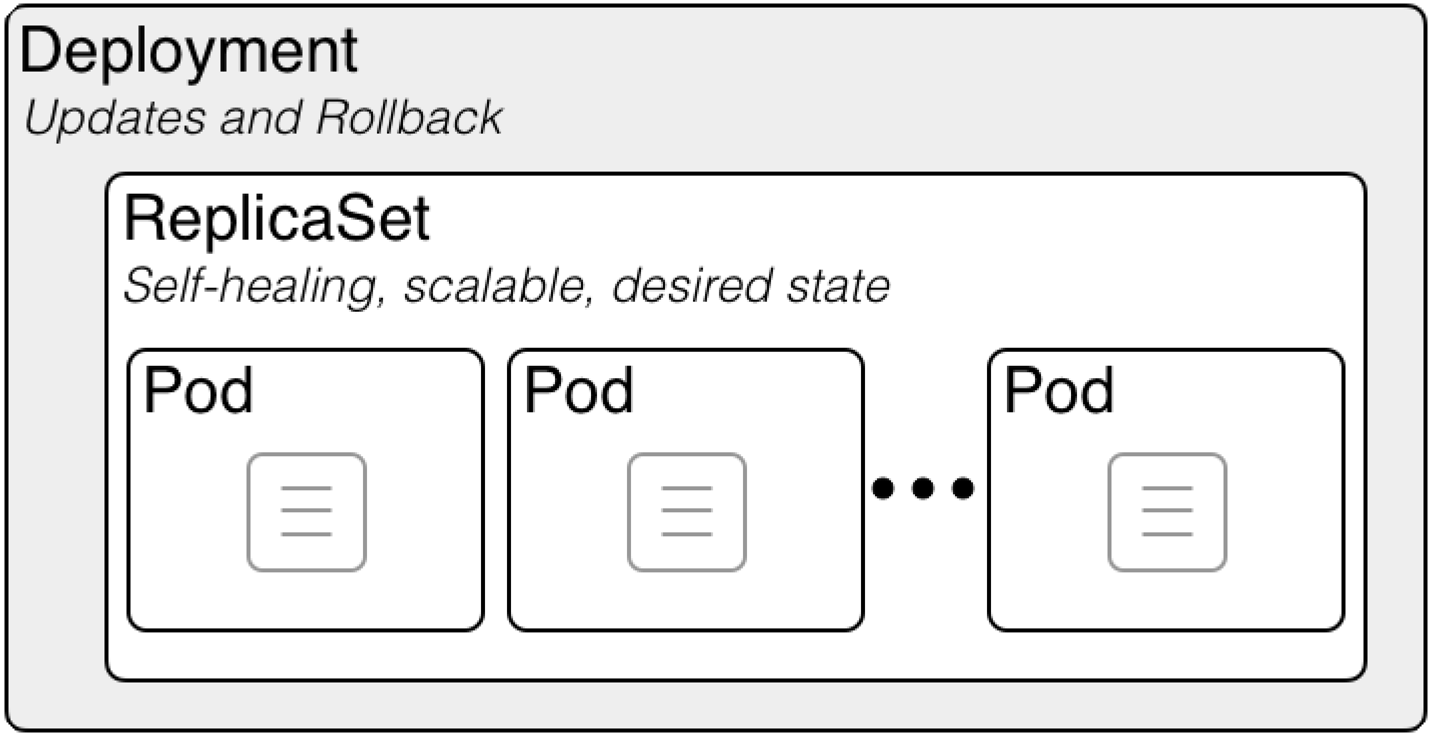
KUBERNETES DEPLOYMENTS

K8S DEPLOYMENTS



OPEN IN A TERMINAL:

```
watch -n 1 kubectl get  pod -l app=hello-world
```

OPEN IN OTHER TERMINAL:

```
$ kubectl scale --replicas=4 deployment/hello-world-deployment
```

K8S DEPLOYMENTS UPDATES

ROLLOUT

The kubectl rollout command is used to manage the rollout of updates to applications running on the platform, as part of the Kubernetes deployment process. It can be used to manage three Kubernetes objects: Deployment, DaemonSet, and StatefulSet. The rollout process is a gradual, step-by-step recreation of pods.

```
aureliano@aureliano-N141CU ~ $ kubectl rollout status deployment/hello-world-deployment
deployment "hello-world-deployment" successfully rolled out

aureliano@aureliano-N141CU ~ $ kubectl rollout history deployment/hello-world-deployment
deployment.apps/hello-world-deployment
REVISION  CHANGE-CAUSE
1         <none>

kubectl rollout restart deployment/hello-world-deployment
```

K8S DEPLOYMENTS UPDATES

Rollout Strategies

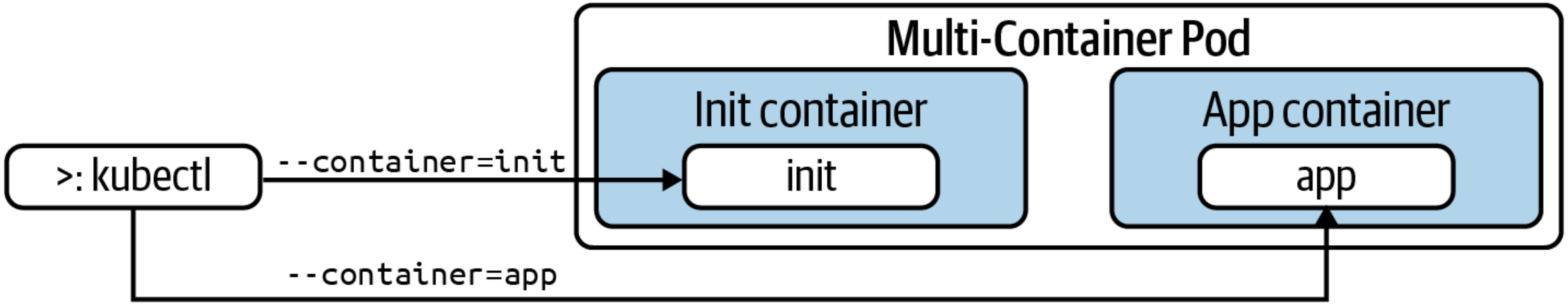
There are mainly two different ways for rolling out an update for Deployment resources, namely **Recreate** and **RollingUpdate**.

The **Recreate** rollout method recreates all the pod instances at the same time. This can be problematic for running services that are serving live traffic as the recreation of all the instances at the same time causes downtime.

The **RollingUpdate** strategy, which is the default rollout method for Deployment resources takes a more gradual approach. Instead of recreating all the pods at the same time, we can perform the update to a subset of the pods, leaving the rest of the pods to continue to serve the traffic.

```
OPEN IN A TERMINAL:

watch -n 1 kubectl get  pod -l app=hello-world

Update in hello world deployment example asinatra/secure:v1.0.3 → asinatra/secure:v1.0.4 then

OPEN IN OTHER TERMINAL:

k apply -f ~/conteneurisation/k8s/local/helloworld/deployment.yaml
```

INIT CONTAINERS



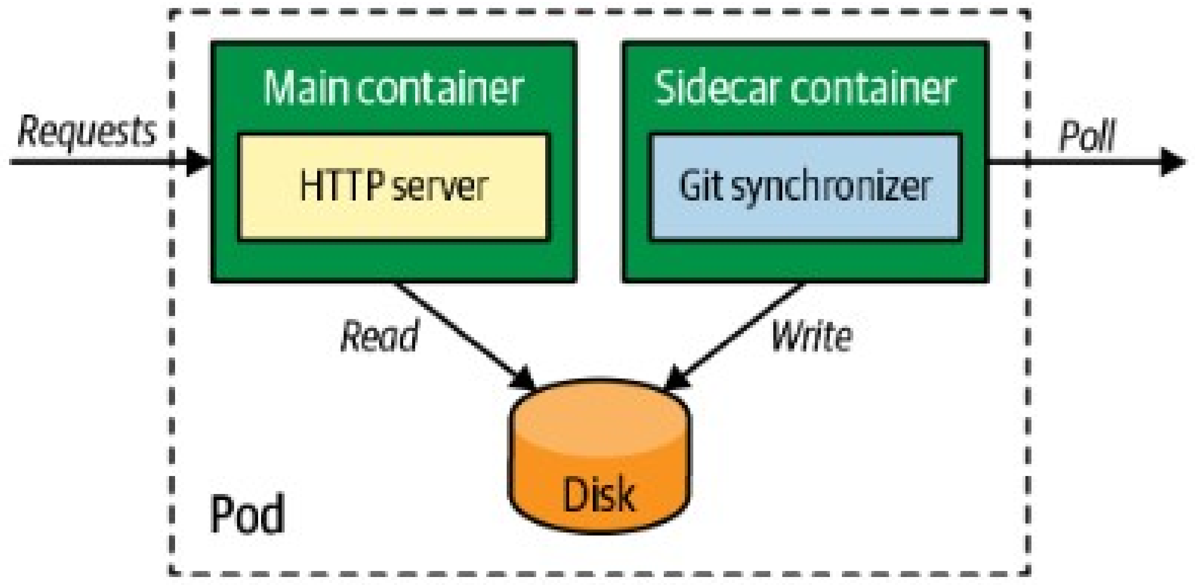
```
aureliano@aureliano-N141CU ~ $ k get pod
NAME                                   READY   STATUS     RESTARTS      AGE
hello-world-deployment-init-549c895b74-274wh   0/1     Init:0/1   0             10s
hello-world-deployment-init-549c895b74-8clcc   0/1     Init:0/1   0             10s
hello-world-deployment-init-549c895b74-hq5nw   0/1     Init:0/1   0             10s
hello-world-deployment-init-549c895b74-nzggz   0/1     Init:0/1   0             10s
```

GIVE ME ALL LOGS BY LABEL AND CONTAINER

```
aureliano@aureliano-N141CU ~ $ k logs -l app=hello-world-init -c init-myapp
The app is running!
The app is running!
The app is running!
The app is running!
```

Init containers in Kubernetes are part of the Pod definition, and they separate all containers in a Pod into two groups: init containers and application containers.
All init containers are executed in a sequence, one by one, and all of them have to terminate successfully before the application containers are started up.

SIDECARS



A sidecar container extends and enhances the functionality of a preexisting container without changing it.
A typical example demonstrating this pattern is of an HTTP server and a Git synchronizer. The HTTP server container is focused only on serving files over HTTP and does not know how or where the files are coming from. Similarly, the Git synchronizer container’s only goal is to sync data from a Git server to the local filesystem. It does not care what happens once synced—its only concern is keeping the local folder in sync with the remote Git server.

HEALTH PROBES

The Health Probe pattern indicates how an application can communicate its health state to Kubernetes.

Liveness Probes
If your application runs into a deadlock, it is still considered healthy from the process health check’s point of view. To detect this kind of issue and any other types of failure according to your application business logic, Kubernetes has liveness probes, regular checks performed by the Kubelet agent that asks your container to confirm it is still healthy. It is important to have the health check performed from the outside rather than in the application itself, as some failures may prevent the application watchdog from reporting its failure. Regarding corrective action, this health check is similar to a process health check, since if a failure is detected, the container is restarted.
However, it offers more flexibility regarding which methods to use for checking the application health, as follows:

HTTP probe: Performs an HTTP GET request to the container IP address and expects a successful HTTP response code between 200 and 399.
TCP Socket probe: Assumes a successful TCP connection.
Exec probe: Executes an arbitrary command in the container’s user and kernel namespace and expects a successful exit code (0).
gRPC probe: Leverages gRPC’s intrinsic support for health checks.

HEALTH PROBES

The Health Probe pattern indicates how an application can communicate its health state to Kubernetes.

Readiness Probes
Rather than restarting the con-
tainer, a failed readiness probe causes the container to be removed from the service endpoint and not receive any new traffic. Readiness probes signal when a container is ready so that it has some time to warm up before getting hit with requests from the service. It is also useful for shielding the container from traffic at later stages, as readiness probes are performed regularly, similarly to liveness checks.

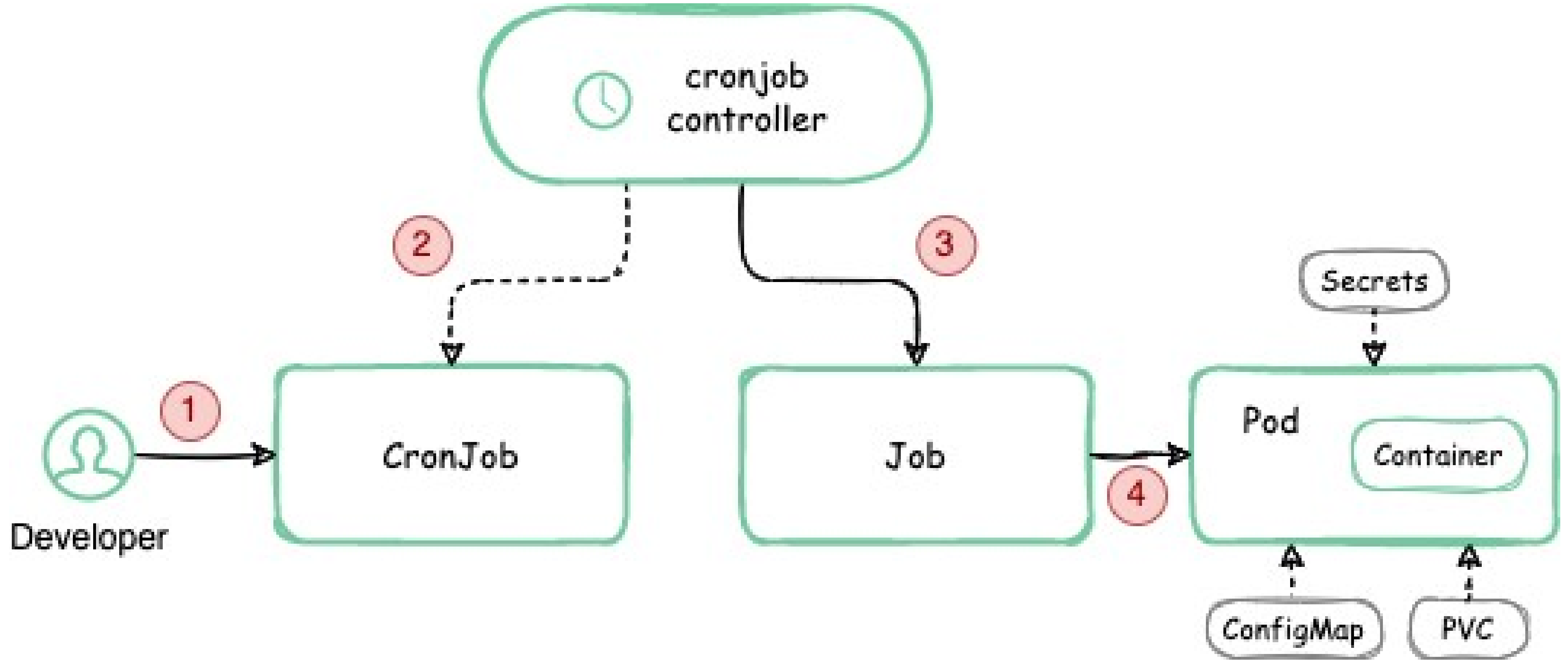
```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-readiness-check
spec:
  containers:
  - image: asinatra/flask-app
    name: hello-world
    readinessProbe:
      httpGet:
          path: /health
          port: 5000
```

HEALTH PROBES

Startup Probes
When applications take minutes to start (for example, Jakarta EE application servers), Kubernetes provides startup probes.
Startup probes are configured with the same format as liveness probes but allow for different values for the probe action and the timing parameters. The periodSeconds and failureThreshold parameters are configured with much larger values compared to the corresponding liveness probes to factor in the longer application startup.
Liveness and readiness probes are called only after the startup probe reports success.
The container is restarted if the startup probe is not successful within the configured failure threshold.

```
apiVersion: v1
kind: Pod
metadata:
name: pod-with-startup-check
spec:
  containers:
  - image: quay.io/wildfly/wildfly
    name: wildfly
    startupProbe:
      exec:
        command: [ "stat",
"/opt/jboss/wildfly/standalone/tmp/startup-marker" ]
      initialDelaySeconds: 60
      periodSeconds: 60
      failureThreshold: 15
    livenessProbe:
      httpGet:
        path: /health
        port: 9990
      periodSeconds: 10
      failureThreshold: 3
```

JOBS / CRON JOBS


cronjob
controller
2
3
1
Developer
CronJob
Job
4
Secrets
Pod
Container
ConfigMap
PVC