Chapter 6: Kubernetes Logging, Monitoring and Health checks

Kubernetes Monitoring

Metrics to Monitor

Cluster Monitoring

- *Node resources* – network bandwidth, disk/CPU/memory utilization
- *Number of nodes* – health of nodes & price justification
- *Running pods* – how many nodes may fail before the remaining nodes can run all desired resources

Pod Monitoring

- *K8s metrics* – pod handling by orchestrator, # of replica, rolling update, in-progress deployments
- *Container metric* – CPU, network, memory of each container
- *Application metric* – depends on business rules

Monitoring

Components required

collect logs -> store data -> aggregation -> view results

Why it is hard in Kubernetes?

- Containers are disposable
- Several layers of abstraction between the application and the underlying hardware to ensure portability and scalability
- Lots of moving parts - multiple services, namespaces, roles, nodes, etc. are spread across the infrastructure
- So many moving parts contributes to a significant blind spot when it comes to conventional monitoring

Default Monitoring Options

Kubernetes Dashboard

- A web-based UI add-on for Kubernetes clusters.
- Allow users to create & manage workloads as well as do discovery, configuration, storage, and monitoring.
- Offers views for CPU & memory usage metrics aggregated across all nodes.
- Used to monitor the health status of workloads (pods, deployments, replica sets, cron jobs, etc.)

Limitation - It is helpful for small clusters

Default Monitoring Options

Kube-state-metrics

- Listens to the Kubernetes API server and generates metrics about the state of numerous Kubernetes objects, including cron jobs, config maps, pods, and nodes
- Uses the Golang Prometheus client to export metrics in the Prometheus metrics exposition format and expose metrics on an HTTP endpoint. Prometheus consume the web endpoint

Limitation - This tool is not oriented toward performance and health but rather toward cluster-wide, state-based metrics such as the number of desired pod replicas for deployment or the total CPU resources available on a node.

Default Monitoring Options

cAdvisor (Container advisor)

- Container resource usage and performance analysis agent; it's integrated into the Kubelet binary
- Auto-discovers all containers in a machine and collects statistics about memory, network usage, filesystem, and CPU
- Does not operate at the pod level, but on each node

Limitation - cAdvisor can't be used to store metrics for long-term use or perform complex monitoring actions.

https://hub.docker.com/r/google/cadvisor

Default Monitoring Options

Probes

- Monitor the **health of a container**
- Kubelet use liveliness probes to know when to restart a container
- Kubelet use readiness probes to know when a container is ready to take traffic

Limitation – Only covers liveliness and readiness of containers

Default Monitoring Options

Heapster

- Enables monitoring and performance analysis in Kubernetes Clusters
- Collects signals from kubelets and the api server, processes them, and exports them via REST APIs or to a configurable timeseries storage backend
- Acts as a normal pod and discovers every cluster node via the Kubernetes API

Limitation – Deprecated and no longer supported

Problems in Default Monitoring Options

- **Persistence**: These tools provide information about the current health of Kubernetes, they lack data storage capabilities. Either InfluxDB or Prometheus (time-series databases) is added to provide persistence
- **Visualization**: open-source tools such as Grafana or Kibana are used
- **Logging**: Additional log collectors are added as well
- **Access control and HA**: Additional third-party integration needs reliability, scalability to store data, additional RBAC

Bottom line: Monitoring in Kubernetes is hard, so default options are rarely used. ** *Next few slides describes what options are available*

Other Monitoring Options

Prometheus and Grafana

- Prometheus stores all its data as a time series
- This data can be queried via the PromQL query language and visualized with a built-in expression browser
- Since Prometheus is not a dashboard, it relies on Grafana for visualizing data

Other Monitoring Options

ELK or EFK stack (Elasticsearch, logstash/fluentd, Kibana)

- Works well with one another and together represent a reliable solution used for Kubernetes monitoring and log aggregation
- Fluentd collects logs from pods running on cluster nodes, then routes them to a centralized Elasticsearch
- Elasticsearch ingests these logs from Fluentd and stores them in a central location. It is also used to efficiently search text files
- Kibana is the UI; the user can visualize the collected logs and metrics and create custom dashboards based on queries

Other Monitoring Options

Kubewatch

- A Kubernetes watcher that publishes event notifications in a **Slack channel**. This tool allows you to specify the resources you want to monitor

Commonly used toolset

- Heapster + InfluxDB + Grafana
- Prometheus + Grafana
- Heapster + ELK

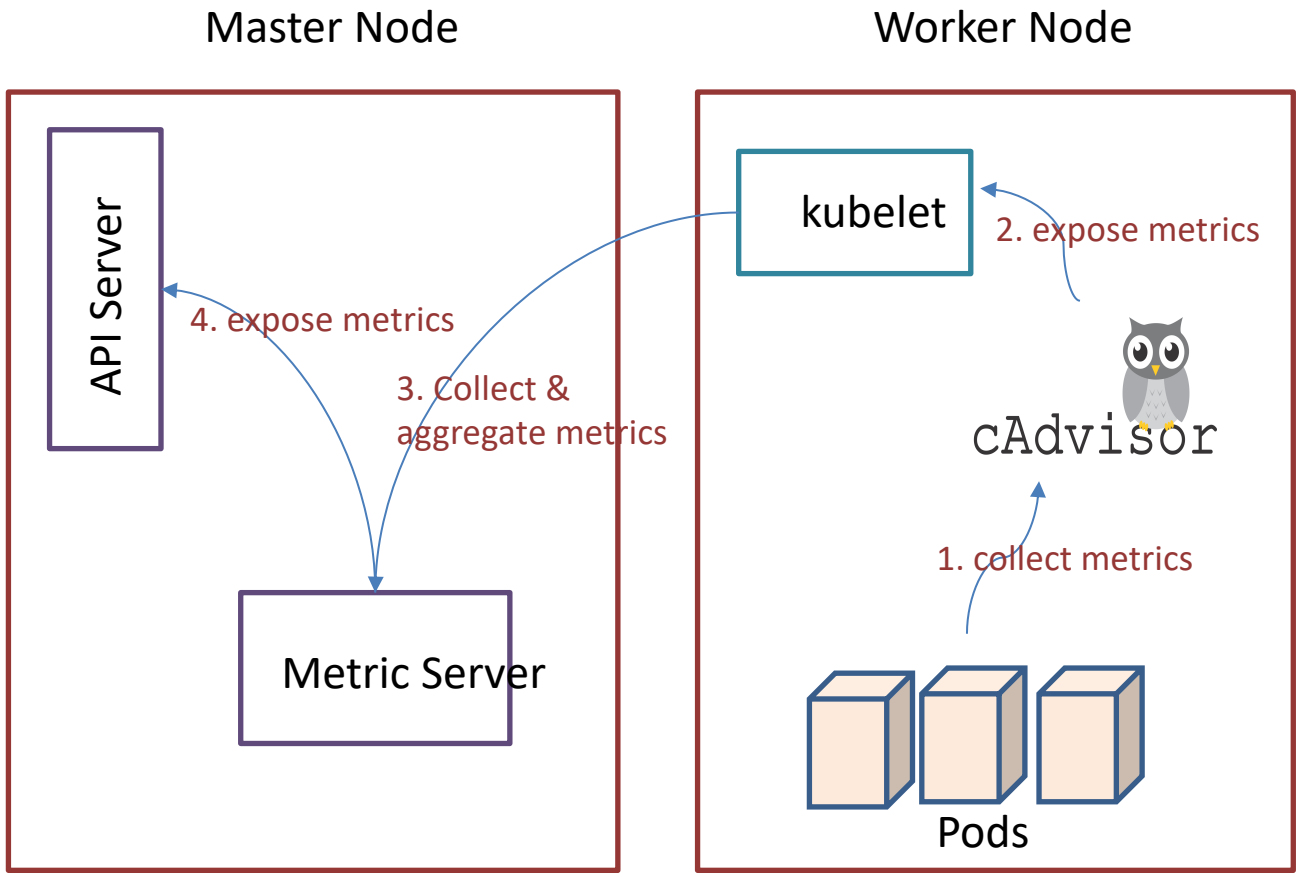
With Heapster deprecated, the most common toolset remains is – {Prometheus + Grafana}

Vendors for Monitoring

- Datadog
- AppDynamics
- Sumo Logic
- Rancher
- Dynatrace
- Sysdig

Core Metric Monitoring

Also known as Resource Metric Pipeline



>> CAdvisor collects metrics about containers
and nodes that on which it is installed.
* CAdvisor is installed by default
on all cluster nodes

>> Kubelet exposes these metrics (default is one-minute resolution) through
Kubelet APIs.

>> Metrics Server discovers all available nodes
and calls Kubelet API to get containers and
nodes resources usage.

>> Metrics Server exposes these metrics
through Kubernetes aggregation API.

Lab: Kubernetes Monitor (cAdvisor+Prometheus)

https://github.com/shekhar2010us/kubernetes_teach_git/blob/master/k8s_monitor1/monitor.md

Lab: Kubernetes Monitor (cAdvisor+Prometheus+Grafana)

https://github.com/shekhar2010us/kubernetes_teach_git/blob/master/k8s_monitor2/monitor.md

Kubernetes Logging

What to Log?

In order to monitor Kubernetes cluster activities we need to be able to log at the cluster level as well as at the node level.

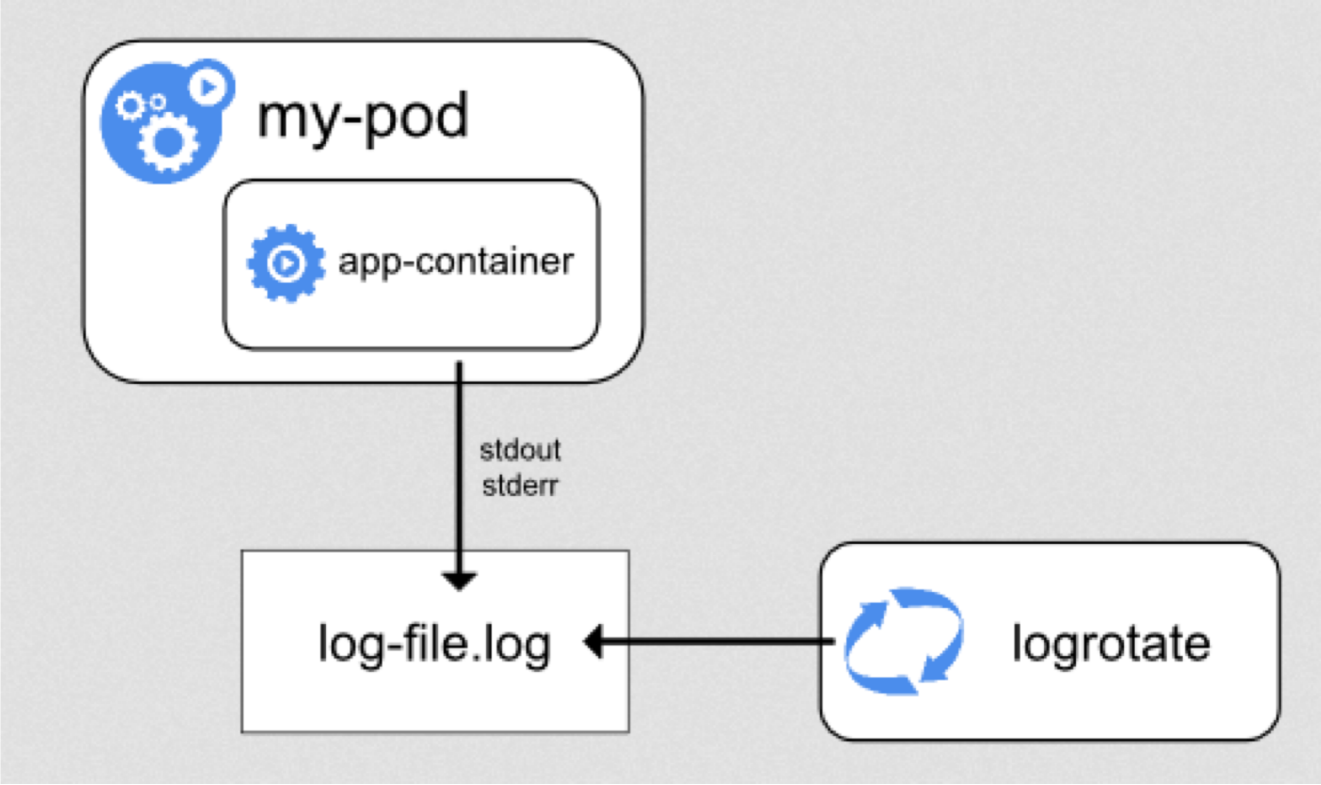
For this, we need to augment the native logging functionality of a container. Example if a container dies, the pod gets evicted or if a node dies and the replication gets rescheduled, we still need to be able to access the application logs.

This necessitates separate storage and lifecycle independent of nodes, pods or containers.

Lab: Basic Logging

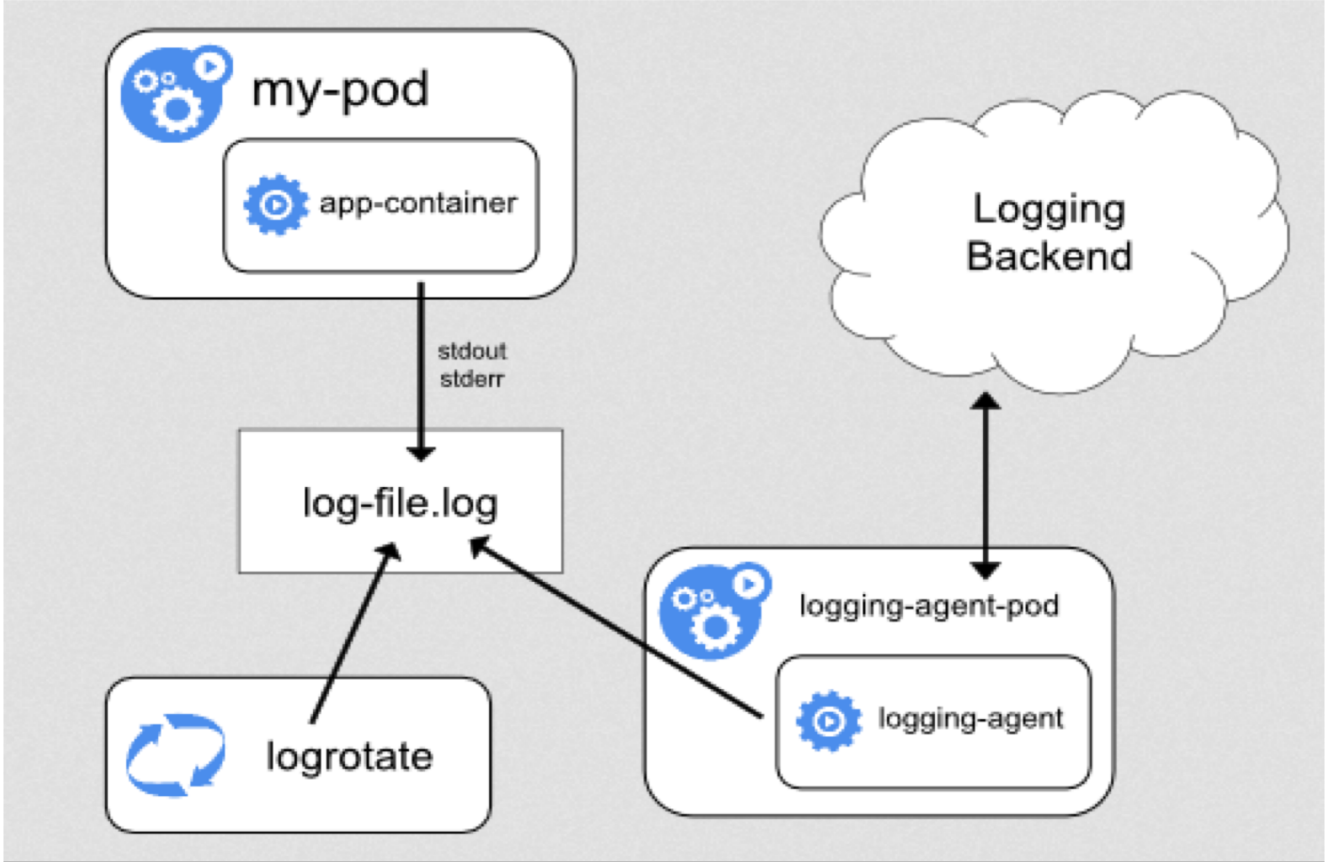
https://github.com/shekhar2010us/kubernetes_teach_git/blob/master/k8s_logging/basic_logging.md

Cluster Level Logging in Kubernetes



Everything the container (application) writes to stdout and stderr are sent to an external log. We must implement log rotation to ensure that the space is not filled up!

Cluster Level Logging in Kubernetes



By adding a node-level logging agent on each node we can push logs to a backend service to take the log lifecycle beyond that of a node.

Before we setup a EFK cluster for logging, let's get introduced with **Helm**

Helm

What is Helm

- Package manager in Kubernetes
- Introduce concept of **charts**
- Charts can be pre-defined or user-defined

Components:

- **Helm Client** – To interact with helm
- **Tiller** - receives the incoming request and installs the appropriate package. Also manage release and versions. Tiller contains the resource definitions to install a Kubernetes application

Helm Charts

What is Helm Charts

- Packaging format – Collection of files (in a directory) having Kubernetes resources
- Directory name = Chart name
- Can be simple (deploy Redis) or complex (full stack web application)

An example directory

```
wordpress/
  Chart.yaml          # A YAML file containing information about the chart
  LICENSE             # OPTIONAL: A plain text file containing the license for th
  README.md           # OPTIONAL: A human-readable README file
  requirements.yaml   # OPTIONAL: A YAML file listing dependencies for the chart
  values.yaml         # The default configuration values for this chart
  charts/             # A directory containing any charts upon which this chart d
  templates/          # A directory of templates that, when combined with values,
                      # will generate valid Kubernetes manifest files.
  templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes
```

Helm Charts

Chart.yaml

apiVersion: <chart api version>

name: <name of the chart>

version: <version>

kubeVersion: <kubernetes version>

description: <about project>

spec: {{.Values.imageRegistry}}

engine: <template engine name, default gotpl>

tillerVersion: <tiller version>

requirements.yaml – dependency on other charts

dependencies:

- name: apache

  version: 1.2.3

  repository: http://example.com/charts

- name: mysql

  version: 3.2.1

  repository: http://another.example.com/charts

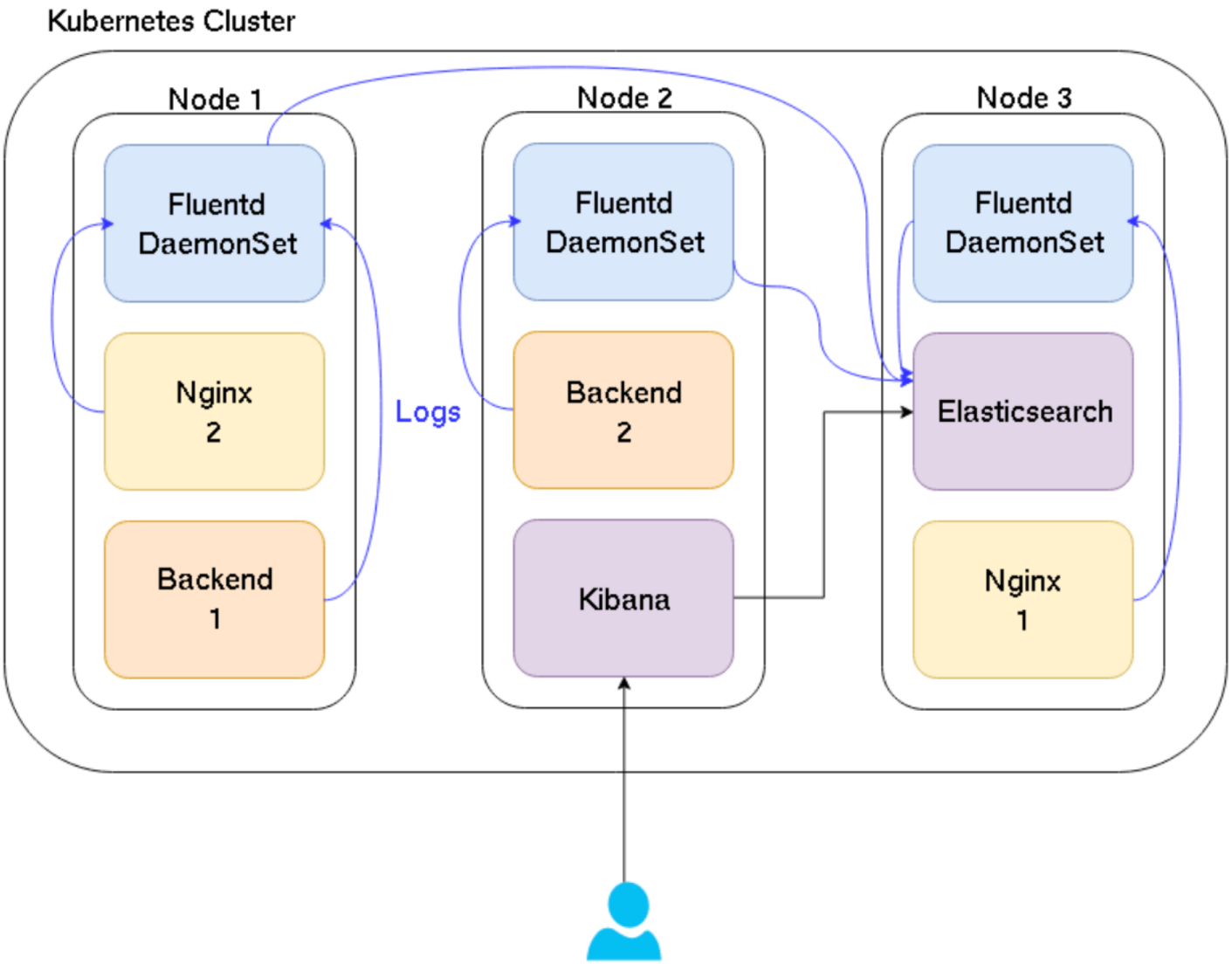
values.yaml – provide values to templates

imageRegistry: “my-registry”

Lab: Installing Helm

https://github.com/shekhar2010us/kubernetes_teach_git/blob/master/k8s_logging/helm.md

EFK Architecture



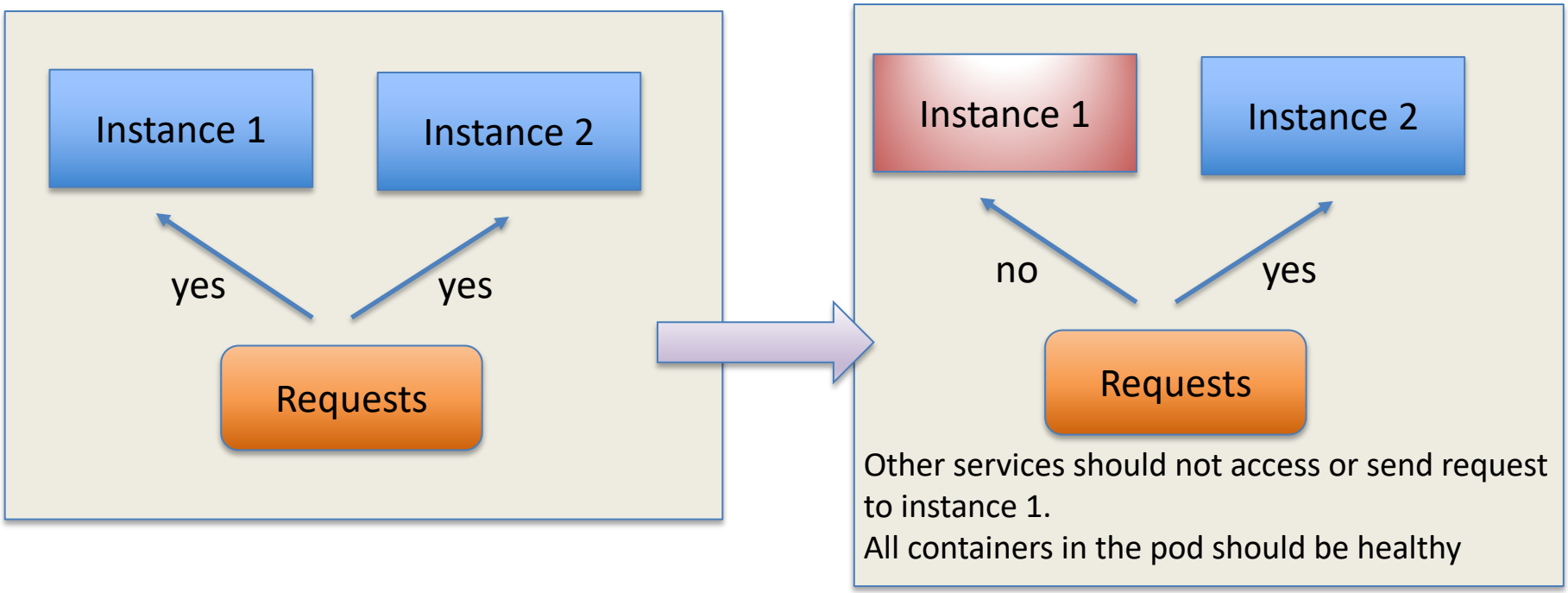
Simplified schema of an EFK

Lab: Logging using EFK

https://github.com/shekhar2010us/kubernetes_teach_git/blob/master/k8s_logging/efk_logging.md

Kubernetes Health Check - Liveness and Readiness

Health Check Objectives – default options in k8s



Custom Health Check Options

Container Probes

Container probes

A Probe is a diagnostic performed periodically by the kubelet on a Container.

Two kinds of probes on running Containers:

- **Readiness probes:**
  - To let know k8s when the app is ready to serve traffic. K8s will make sure that readiness probe passes before passing the traffic to it
  - If it fails, k8s will stop sending traffic until it passes again
- **Liveness probes:**
  - To let know k8s if app is alive or dead
  - If the app is dead, k8s will remove the pod and start a new

Liveness vs. Readiness Probes

When should you use liveness or readiness probes?

- If you'd like your Container to be killed and restarted if a probe fails, then specify a **liveness probe**, and specify a restartPolicy of Always or OnFailure.

- If you'd like to start sending traffic to a Pod only when a probe succeeds, specify a **readiness probe**
- If you want your Container to be able to take itself down for maintenance, you can specify a **readiness probe**

Container Probes Handlers

There are **three types of handlers** (being called by the kubelet running in the node):

- **ExecAction (Command)**: Executes a specified command inside the Container. The diagnostic is considered successful if the command exits with a **status code of 0**.
- **TCPSocketAction**: Performs a TCP check against the Container's IP address on a specified port. The diagnostic is considered successful if the **port is open**.
- **HTTPGetAction**: Performs an HTTP Get request against the Container's IP address on a specified port and path. The diagnostic is considered successful if the response has a **status code greater than or equal to 200 and less than 400**.

A probe has one of these 3 results:
Success: The Container passed the diagnostic.
Failure: The Container failed the diagnostic.
Unknown: The diagnostic failed, so no action should be taken.

Container Probes Handlers

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

HTTP Type

- perform a liveness probe every 3 seconds
- send a HTTP GET request to the server
- If HTTP returns between 200-400, all good
  - Otherwise, it will restart the pod
- First 10 seconds, the command will succeed

Container Probes Handlers

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

Command Type

- perform a liveness probe every 5 seconds
- executes the command cat /tmp/healthy
- If command succeeds, all good
  - Otherwise, restart the pod
- First 30 seconds, the command will succeed

Container Probes Handlers

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

TCP Type

- Similar to HTTP probe
- Use both liveness and readiness
- perform a liveness probe every 20 seconds
- perform a readiness probe every 10 seconds
- Try to connect to goproxy container at 8080

THANK YOU !!