Operationalizing Multi-tenancy Support with Kubernetes

(It's Not Just About Security)

October 11, 2018
Your Presenters

Paul Sitowitz
Manager, Software Engineering
Paul is a Software engineer with Capital One who specializes in container technologies and Kubernetes and is a Certified Kubernetes Administrator and a Certified Kubernetes Application Developer. He is currently supporting a Kubernetes-based fraud decisioning platform.

Keith Gasser
Lead Software Engineer
Keith is a Software Engineer specializing in DevOps and Application Security at Capital One currently working on a team which has built a Kubernetes-based streaming and decisioning pipeline for Capital One Bank.
Case Study: Supporting Fast Decisioning Applications With Kubernetes

• Learn more about our Fraud Decisioning Platform at:

https://kubernetes.io/case-studies/capital-one

“a provisioning platform for Capital One applications deployed on AWS that use streaming, big-data decisioning, and machine learning. One of these applications handles millions of transactions a day; some deal with critical functions like fraud detection and credit decisioning. The key considerations: resilience and speed—as well as full rehydration of the cluster from base AMIs”
Agenda
Agenda

1. Introduction
   About what we will be presenting

2. Assumptions
   About our participants

3. Some Definitions
   Workload
   Containerized Workload
   Controllers
   Multi-tenancy

4. Pathway to Multi-tenancy in K8S
   Key Building Blocks
   Self-Healing
   Namespace Isolation

5. Cloud Provider Hosted K8s
   EKS, GKE, AKS

6. Kubernetes Feature Roadmap
   Upcoming features that will help with multi-tenancy

7. Summary
   Recap and take a ways

8. More FinTech Talks Regarding Our Platform
   By our colleagues
1. Introduction
   About what we will be presenting

2. Assumptions
   About our participants

3. Some Definitions
   Workload
   Containerized Workload
   Controllers
   Multi-tenancy

4. Pathway to Multi-tenancy in K8S
   Key Building Blocks
   Self-Healing
   Namespace Isolation

Resource Limitation
Node Isolation
Security Limitation
Network Policy Isolation

5. Cloud Provider Hosted K8s
   EKS, GKE, AKS

6. Kubernetes Feature Roadmap
   Upcoming features that will help with multi-tenancy

7. Summary
   Recap and take a ways

8. More FinTech Talks Regarding Our Platform
   By our colleagues
1. Introduction
   About what we will be presenting

2. Assumptions
   About our participants

3. Some Definitions
   Workload
   Containerized Workload
   Controllers
   Multi-tenancy

4. Pathway to Multi-tenancy in K8S
   Key Building Blocks
   Self-Healing
   Namespace Isolation

   Resource Limitation
   Node Isolation
   Security Limitation
   Network Policy Isolation

5. Cloud Provider Hosted K8s
   EKS, GKE, AKS

6. Kubernetes Feature Roadmap
   Upcoming features that will help with multi-tenancy

7. Summary
   Recap and take a ways

8. More FinTech Talks Regarding Our Platform
   By our colleagues
1. Introduction
   About what we will be presenting

2. Assumptions
   About our participants

3. Some Definitions
   Workload
   Containerized Workload
   Controllers
   Multi-tenancy

4. Pathway to Multi-tenancy in K8S
   Key Building Blocks
   Self-Healing
   Namespace Isolation

   Resource Limitation
   Node Isolation
   Security Limitation
   Network Policy Isolation

5. Cloud Provider Hosted K8s
   EKS, GKE, AKS

6. Kubernetes Feature Roadmap
   Upcoming features that will help with multi-tenancy

7. Summary
   Recap and take aways

8. More FinTech Talks Regarding Our Platform
   By our colleagues
1. Introduction
   About what we will be presenting

2. Assumptions
   About our participants

3. Some Definitions
   Workload
   Containerized Workload
   Controllers
   Multi-tenancy

4. Pathway to Multi-tenancy in K8S
   Key Building Blocks
   Self-Healing
   Namespace Isolation

5. Cloud Provider Hosted K8s
   EKS, GKE, AKS

6. Kubernetes Feature Roadmap
   Upcoming features that will help with multi-tenancy

7. Summary
   Recap and take a ways

8. More FinTech Talks Regarding Our Platform
   By our colleagues
1. Introduction
   About what we will be presenting

2. Assumptions
   About our participants

3. Some Definitions
   Workload
   Containerized Workload
   Controllers
   Multi-tenancy

4. Pathway to Multi-tenancy in K8S
   Key Building Blocks
   Self-Healing
   Namespace Isolation

5. Cloud Provider Hosted K8s
   EKS, GKE, AKS

6. Kubernetes Feature Roadmap
   Upcoming features that will help with multi-tenancy

7. Summary
   Recap and take a ways

8. More FinTech Talks Regarding Our Platform
   By our colleagues
1. Introduction
   About what we will be presenting

2. Assumptions
   About our participants

3. Some Definitions
   Workload
   Containerized Workload
   Controllers
   Multi-tenancy

4. Pathway to Multi-tenancy in K8S
   Key Building Blocks
   Self-Healing
   Namespace Isolation

5. Cloud Provider Hosted K8s
   EKS, GKE, AKS

6. Kubernetes Feature Roadmap
   Upcoming features that will help with multi-tenancy

7. Summary
   Recap and takeaways

8. More FinTech Talks Regarding Our Platform
   By our colleagues
1. Introduction
   About what we will be presenting

2. Assumptions
   About our participants

3. Some Definitions
   Workload
   Containerized Workload
   Controllers
   Multi-tenancy

4. Pathway to Multi-tenancy in K8S
   Key Building Blocks
   Self-Healing
   Namespace Isolation

   Resource Limitation
   Node Isolation
   Security Limitation
   Network Policy Isolation

5. Cloud Provider Hosted K8s
   EKS, GKE, AKS

6. Kubernetes Feature Roadmap
   Upcoming features that will help with multi-tenancy

7. Summary
   Recap and take a ways

8. More FinTech Talks Regarding Our Platform
   By our colleagues
Introduction
Heartache and Pain Points of Multi-tenant Kubernetes

• Coordinated deployments
  • Cluster version baseline (up through the addon stack)
  • Resource starvation & contention
  • “Thundering Herd”
  • Cascading failures
  • Node lockout
  • APIServer Status: Node Unknown (kubelet death)
  • Administrative blindness due to log forwarder saturation

• How do we avert “Tragedy of the Commons”? 
Heartache and Pain Points of Multi-tenant Kubernetes

- Coordinated deployments
- Cluster version baseline (up through the addon stack)
- Resource starvation & contention
- “Thundering Herd”
- Cascading failures
- Node lockout
- APIServer Status: Node Unknown (kubelet death)
- Administrative blindness due to log forwarder saturation

- How do we avert “Tragedy of the Commons”?
Heartache and Pain Points of Multi-tenant Kubernetes

- Coordinated deployments
- Cluster version baseline (up through the addon stack)
- Resource starvation & contention
  - “Thundering Herd”
- Cascading failures
- Node lockout
- APIServer Status: Node Unknown (kubelet death)
- Administrative blindness due to log forwarder saturation

- How do we avert “Tragedy of the Commons”?
Heartache and Pain Points of Multi-tenant Kubernetes

- Coordinated deployments
- Cluster version baseline (up through the addon stack)
- Resource starvation & contention
- “Thundering Herd”
- Cascading failures
- Node lockout
- API Server Status: Node Unknown (kubelet death)
- Administrative blindness due to log forwarder saturation

• How do we avert “Tragedy of the Commons”? 
Heartache and Pain Points of Multi-tenant Kubernetes

• Coordinated deployments
• Cluster version baseline (up through the addon stack)
• Resource starvation & contention
• “Thundering Herd”
• Cascading failures
• Node lockout
• API Server Status: Node Unknown (kubelet death)
• Administrative blindness due to log forwarder saturation

• How do we avert “Tragedy of the Commons”?
Heartache and Pain Points of Multi-tenant Kubernetes

- Coordinated deployments
- Cluster version baseline (up through the addon stack)
- Resource starvation & contention
- “Thundering Herd”
- Cascading failures
- Node lockout
- APIServer Status: Node Unknown (kubelet death)
- Administrative blindness due to log forwarder saturation

- How do we avert “Tragedy of the Commons”? 
Heartache and Pain Points of Multi-tenant Kubernetes

- Coordinated deployments
- Cluster version baseline (up through the addon stack)
- Resource starvation & contention
- “Thundering Herd”
- Cascading failures
- Node lockout
- API Server Status: Node Unknown (kubelet death)
- Administrative blindness due to log forwarder saturation

- How do we avert “Tragedy of the Commons”?
Heartache and Pain Points of Multi-tenant Kubernetes

• Coordinated deployments
• Cluster version baseline (up through the addon stack)
• Resource starvation & contention
• “Thundering Herd”
• Cascading failures
• Node lockout
• API Server Status: Node Unknown (kubelet death)
• Administrative blindness due to log forwarder saturation

• How do we avert “Tragedy of the Commons”?
Heartache and Pain Points of Multi-tenant Kubernetes

- Coordinated deployments
- Cluster version baseline (up through the addon stack)
- Resource starvation & contention
- “Thundering Herd”
- Cascading failures
- Node lockout
- API Server Status: Node Unknown (kubelet death)
- Administrative blindness due to log forwarder saturation

• How do we avert “Tragedy of the Commons”? 
Introduction

• Building large distributed software is not easy especially when:
  • you must support multiple tenants and each have their own workloads to run and SLAs to meet
  • compute and storage resources are limited and need to be shared
  • Careful thought must be given to ensure that resource isolation is obtained to help to address resource contention and avoid starvation
  • Ensuring that you properly employ the right features to keep your workloads well managed is critical!
Introduction

• Building large distributed software is not easy especially when:

  • you must support multiple tenants and each have their own workloads to run and SLAs to meet

  • compute and storage resources are limited and need to be shared

• Careful thought must be given to ensure that resource isolation is obtained to help to address resource contention and avoid starvation

• Ensuring that you properly employ the right features to keep your workloads well managed is critical!
Introduction

• Building large distributed software is not easy especially when:

  • you must support multiple tenants and each have their own workloads to run and SLAs to meet

  • compute and storage resources are limited and need to be shared

• Careful thought must be given to ensure that resource isolation is obtained to help to address resource contention and avoid starvation

• Ensuring that you properly employ the right features to keep your workloads well managed is critical!
• Building large distributed software is not easy especially when:
  • you must support multiple tenants and each have their own workloads to run and SLAs to meet
  • compute and storage resources are limited and need to be shared
• Careful thought must be given to ensure that resource isolation is obtained to help to address resource contention and avoid starvation
• Ensuring that you properly employ the right features to keep your workloads well managed is critical!
• Unfortunately, there is no such thing as a self tuning / self-administering K8S cluster 😞

• These K8S features will be the key ingredients in our recipe for operating a well managed, multi-tenant, Kubernetes cluster

  • Namespaces
  • Taints / Tolerations
  • Affinity / Anti-affinity
  • Liveness / Readiness probes
  • Role Based Access Control
  • Security contexts
  • Pod Resource Requests/Limits

  • Node Selectors
  • Pod Security Policies
  • Secrets
  • Autoscaling
  • Network Policies
  • Limit Ranges
  • Resource Quotas
Introduction (continued)

• Unfortunately, there is no such thing as a self tuning / self-administering K8S cluster 😞

• These K8S features will be the key ingredients in our recipe for operating a well managed, multi-tenant, Kubernetes cluster

  • Namespaces
  • Taints / Tolerations
  • Affinity / Anti-affinity
  • Liveness / Readiness probes
  • Role Based Access Control
  • Security contexts
  • Pod Resource Requests/Limits

  • Node Selectors
  • Pod Security Policies
  • Secrets
  • Autoscaling
  • Network Policies
  • Limit Ranges
  • Resource Quotas
## Pros and Cons of Multi-tenancy in K8S

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Usually more cost effective</td>
<td>• Resource contention / starvation</td>
</tr>
<tr>
<td>• Easier to share common components</td>
<td>• Component sharing can degrade performance</td>
</tr>
<tr>
<td>• Easier to manage</td>
<td>• Hard to get right</td>
</tr>
<tr>
<td>• when a single cluster is shared</td>
<td></td>
</tr>
</tbody>
</table>
## Pros and Cons of Multi-tenancy in K8S

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Usually more cost effective</td>
<td>• Resource contention / starvation</td>
</tr>
<tr>
<td>• Easier to share common components</td>
<td>• Component sharing can degrade performance</td>
</tr>
<tr>
<td>• Easier to manage</td>
<td>• Hard to get right</td>
</tr>
<tr>
<td>• when a single cluster is shared</td>
<td></td>
</tr>
</tbody>
</table>

Pros

- Usually more cost effective
- Easier to share common components
- Easier to manage
  - when a single cluster is shared
### Pros and Cons of Multi-tenancy in K8S

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Usually more cost effective</td>
<td>• Resource contention / starvation</td>
</tr>
<tr>
<td>• Easier to share common components</td>
<td>• Component sharing can degrade performance</td>
</tr>
<tr>
<td>• Easier to manage</td>
<td>• Hard to get right</td>
</tr>
<tr>
<td>• when a single cluster is shared</td>
<td></td>
</tr>
</tbody>
</table>
## Pros and Cons of Multi-tenancy in K8S

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Usually more cost effective</td>
<td>• Resource contention / starvation</td>
</tr>
<tr>
<td>• Easier to share common components</td>
<td>• Component sharing can degrade performance</td>
</tr>
<tr>
<td>• Easier to manage</td>
<td>• Hard to get right</td>
</tr>
<tr>
<td>• when a single cluster is shared</td>
<td></td>
</tr>
</tbody>
</table>
Pros and Cons of Multi-tenancy in K8S

**Pros**
- Usually more cost effective
- Easier to share common components
- Easier to manage
  - when a single cluster is shared

**Cons**
- Resource contention / starvation
- Component sharing can degrade performance
- Hard to get right
## Pros and Cons of Multi-tenancy in K8S

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Usually more cost effective</td>
<td>• Resource contention / starvation</td>
</tr>
<tr>
<td>• Easier to share common components</td>
<td>• Component sharing can degrade performance</td>
</tr>
<tr>
<td>• Easier to manage</td>
<td>• Hard to get right</td>
</tr>
<tr>
<td>• when a single cluster is shared</td>
<td></td>
</tr>
</tbody>
</table>
Our Assumptions about You (the Participant)
Our Assumptions About you (the Participant)

• You are familiar with Docker or other container runtimes
Our Assumptions About you (the Participant)

You are familiar with **Kubernetes (K8S)**
Our Assumptions About you (the Participant)

• You will silence your phones/pagers during our presentation
Our Assumptions About you (the Participant)

• You will hold off on questions until the end
Thank You in advance!! 😊
Some Definitions
Some Definitions

- **Workload**
  - An application that performs some work or processing and requires CPU & Memory resources
    - Server / daemon
    - Batch / scheduled jobs

- **Containerized Workload**
  - A workload that is packaged as an image and deployed inside of a container
    - Docker / Containerd / RKT
  - **A Pod is the smallest unit for deploying a workload in K8S**
    - Hosts 1 or more containers

- **Controllers/Workload Managers**
  - Higher level components used to manage pods
    - Maintains the desired count of available replicas
    - Stateless
      - Deployment
      - Jobs
      - DaemonSets
    - Stateful
      - StatefulSets

- **Multi-tenancy**
  - Disparate workloads hosted on the same cluster
  - Issues exacerbated by varied workload owners
    - Shared resources
    - Contention concerns
Some Definitions

**Workload**
- An application that performs some work or processing and requires CPU & Memory resources
  - Server / daemon
  - Batch / scheduled jobs

**Containerized Workload**
- A workload that is packaged as an image and deployed inside of a container
  - Docker / Containerd / RKT
- A Pod is the smallest unit for deploying a workload in K8S
  - Hosts 1 or more containers

**Controllers/Workload Managers**
- Higher level components used to manage pods
  - Maintains the desired count of available replicas
  - Stateless
    - Deployment
    - Jobs
    - DaemonSets
  - Stateful
    - StatefulSets

**Multi-tenancy**
- Disparate workloads hosted on the same cluster
- Issues exacerbated by varied workload owners
  - Shared resources
  - Contention concerns
Some Definitions

Workload

- An application that performs some work or processing and requires CPU & Memory resources
  - Server / daemon
  - Batch / scheduled jobs

Containerized Workload

- A workload that is packaged as an image and deployed inside of a container
  - Docker / Containerd / RKT

- A Pod is the smallest unit for deploying a workload in K8S
  - Hosts 1 or more containers

Controllers/Workload Managers

- Higher level components used to manage pods
  - Maintains the desired count of available replicas
  - Stateless
    - Deployment
    - Jobs
    - DaemonSets
  - Stateful
    - StatefulSets

Multi-tenancy

- Disparate workloads hosted on the same cluster
- Issues exacerbated by varied workload owners
  - Shared resources
  - Contention concerns
Some Definitions

- **Workload**
  - An application that performs some work or processing and requires CPU & Memory resources
  - Server / daemon
  - Batch / scheduled jobs

- **Containerized Workload**
  - A workload that is packaged as an image and deployed inside of a container
    - Docker / Containerd / RKT
  - **A Pod is the smallest unit for deploying a workload in K8S**
    - Hosts 1 or more containers

- **Controllers/Workload Managers**
  - Higher level components used to manage pods
    - Maintains the desired count of available replicas
    - Stateless
      - Deployment
      - Jobs
      - DaemonSets
    - Stateful
      - StatefulSets

- **Multi-tenancy**
  - Disparate workloads hosted on the same cluster
  - Issues exacerbated by varied workload owners
    - Shared resources
    - Contention concerns
Pathway to Multi-tenancy in Kubernetes
Pathway To Multi-tenancy

Key Building Blocks
- Robust & tested container images
- Controller s / Workload Managers

Self-healing
- Liveness Probes
- Readiness Probes
- Autoscaling

Namespace Isolation & Resource Request & Limitation
- Separate namespace per tenant
- Pod resource requests/limits
- Limit Ranges
- Resource Quotas

Worker Node Isolation
- Taints
- Node Selectors
- Node Affinity
- Pod Affinity/Anti-Affinity

Security Limitation
- Security Contexts
- Pod Security Policies
- Secrets
- Role Based Access Control

Network Policy Isolation
- Control network ingress/egress communication between pods

Well-managed Multi-tenant Kubernetes cluster
Pathway To Multi-tenancy

Key Building Blocks:
- Robust & tested container images
- Controller / Workload Managers

Self-healing:
- Liveness Probes
- Readiness Probes
- Autoscaling

Namespace Isolation & Resource Request & Limitation:
- Separate namespace per tenant
- Pod resource requests/limits
- Limit Ranges
- Resource Quotas

Worker Node Isolation:
- Taints
- Node Selectors
- Node Affinity
- Pod Affinity/Anti-Affinity

Security Limitation:
- Security Contexts
- Pod Security Policies
- Secrets
- Role Based Access Control

Network Policy Isolation:
- Control network ingress/egress communication between pods

Well-managed Multi-tenant Kubernetes cluster
Pathway To Multi-tenancy

Key Building Blocks
- Robust & tested container images
- Controller / Workload Managers

Self-healing
- Liveness Probes
- Readiness Probes
- Autoscaling

Namespace Isolation & Resource Request & Limitation
- Separate namespace per tenant
- Pod resource requests/limits
- Limit Ranges
- Resource Quotas

Worker Node Isolation
- Taints
- Node Selectors
- Node Affinity
- Pod Affinity/Anti-Affinity

Security Limitation
- Security Contexts
- Pod Security Policies
- Secrets
- Role Based Access Control

Network Policy Isolation
- Control networking ingress/egress communication between pods

Well-managed Multi-tenant Kubernetes cluster
Pathway To Multi-tenancy

Key Building Blocks
- Robust & tested container images
- Controller / Workload Managers

Self-healing
- Liveness Probes
- Readiness Probes
- Autoscaling

Namespace Isolation & Resource Request & Limitation
- Separate namespace per tenant
- Pod resource requests/limits
- Limit Ranges
- Resource Quotas

Worker Node Isolation
- Taints
- Node Selectors
- Node Affinity
- Pod Affinity/Anti-Affinity

Security Limitation
- Security Contexts
- Pod Security Policies
- Secrets
- Role Based Access Control

Network Policy Isolation
- Control network ingress/egress communication between pods

Well-managed Multi-tenant Kubernetes cluster
Pathway To Multi-tenancy

Key Building Blocks
- Robust & tested container images
- Controller s / Workload Managers

Self-healing
- Liveness Probes
- Readiness Probes
- Autoscaling

Namespace Isolation & Resource Request & Limitation
- Separate namespace per tenant
- Pod resource requests/limits
- Limit Ranges
- Resource Quotas

Worker Node Isolation
- Taints
- Node Selectors
- Node Affinity
- Pod Affinity/Anti-Affinity

Security Limitation
- Security Contexts
- Pod Security Policies
- Secrets
- Role Based Access Control

Network Policy Isolation
- Control network ingress/egress communication between pods

Well-managed Multi-tenant Kubernetes cluster
Key Building Blocks
- Robust & tested container images
- Controller s / Workload Managers

Self-healing
- Liveness Probes
- Readiness Probes
- Autoscaling

Namespace Isolation & Resource Request & Limitation
- Separate namespace per tenant
- Pod resource requests/limits
- Limit Ranges
- Resource Quotas

Worker Node Isolation
- Taints
- Node Selectors
- Node Affinity
- Pod Affinity/Anti-Affinity

Security Limitation
- Security Contexts
- Pod Security Policies
- Secrets
- Role Based Access Control

Network Policy Isolation
- Control network ingress/egress communication between pods

Well-managed Multi-tenant Kubernetes cluster
Pathway To Multi-tenancy

- It's mostly about well-managed and isolated workloads

Key Building Blocks
- Robust & tested container images
- Controller / Workload Managers

Self-healing
- Liveness Probes
- Readiness Probes
- Autoscaling

Namespace Isolation & Resource Request & Limitation
- Separate namespace per tenant
- Pod resource requests/limits
- Limit Ranges
- Resource Quotas

Worker Node Isolation
- Taints
- Node Selectors
- Node Affinity
- Pod Affinity/Anti-Affinity

Security Limitation
- Security Contexts
- Pod Security Policies
- Secrets
- Role Based Access Control

Network Policy Isolation
- Control network ingress/egress communication between pods

Well-managed Multi-tenant Kubernetes cluster

- Let's take a closer look at each of these in more detail
Key Building Blocks
Robust & Tested Container Images

- The pathway to multi-tenancy starts with the container image!
- Your images should be tested for performance and quality
  - Identify ideal workload resource requests & limits
  - Use automation for repeatable and consistent ongoing testing
- Artifacts needed to build images should be version controlled
- NEVER deploy an image with tag latest
- Always use a secure image registry
- Limit your image size when possible
Use Controllers / Workload Managers

You should NEVER, EVER deploy a single K8S Pod to Production!!!!!

• Un-managed Pods are NOT resilient

• You should instead use controllers/managers like:
  • Deployments, DaemonSets (stateless)
  • Jobs (batch)
  • StatefulSets (stateful)
Use Controllers / Workload Managers

You should NEVER, EVER deploy a single K8S Pod to Production!!!!!

• Un-managed Pods are NOT resilient

• You should instead use controllers/managers like:
  • Deployments, DaemonSets (stateless)
  • Jobs (batch)
  • StatefulSets (stateful)
Use Controllers / Workload Managers

You should **NEVER, EVER** deploy a single K8S Pod to Production!!!!!

• Un-managed Pods are **NOT** resilient

• You should instead use controllers/managers like:
  
  • Deployments, DaemonSets  (stateless)
  • Jobs  (batch)
  • StatefulSets  (stateful)
Self-Healing
Self-Healing

• Sooner or later, software applications will fail! 😞

• **Self-healing** software can identify that it is not operating correctly and, without human intervention, can take action to restore itself to normal operation 😊

• K8S Pods need help to enable self-healing through the use of **Liveness** and **Readiness** probes

• **Autoscaling** can help to keep:
  o the system responsive and appear healthy under heavy loads
  o operational costs down when system load decreases
Self-Healing

• Sooner or later, software applications will fail! 😞

• **Self-healing** software can identify that it is not operating correctly and, without human intervention, can take action to restore itself to normal operation 😊

• K8S Pods need help to enable self-healing through the use of **Liveness** and **Readiness** probes

• **Autoscaling** can help to keep:
  o the system responsive and appear healthy under heavy loads
  o operational costs down when system load decreases
Self-Healing

• Sooner or later, software applications will fail! 😞

• **Self-healing** software can identify that it is not operating correctly and, without human intervention, can take action to restore itself to normal operation 😊

• K8S Pods need help to enable self-healing through the use of **Liveness** and **Readiness** probes

• **Autoscaling** can help to keep:
  - the system responsive and appear healthy under heavy loads
  - operational costs down when system load decreases
Self-Healing

• Sooner or later, software applications will fail! 😞

• **Self-healing** software can identify that it is not operating correctly and, without human intervention, can take action to restore itself to normal operation 😊

• K8S Pods need help to enable self-healing through the use of **Liveness** and **Readiness** probes

• **Autoscaling** can help to keep:
  o the system responsive and appear healthy under heavy loads
  o operational costs down when system load decreases
Types of Probes Available For Liveness & Readiness

**Command**

- Performs a specified command inside the Container
- The diagnostic is considered **successful if the command exits with a status code of 0.**

**TCP**

- Performs a TCP check against the Container’s IP address on a specified port.
- The diagnostic is considered **successful if the port is open.**

**HTTP**

- Performs an HTTP Get request against the Container’s IP address on a specified port and path
- The diagnostic is considered **successful if $200 \leq \text{httpCode} \leq 400.$**
Types of Probes Available For Liveness & Readiness

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Successful Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TCP</strong></td>
<td>Performs a TCP check against the Container’s IP address on a specified port.</td>
<td>successful if the port is open.</td>
</tr>
<tr>
<td><strong>HTTP</strong></td>
<td>Performs an HTTP Get request against the Container’s IP address on a specified port and path</td>
<td>successful if $200 \leq \text{httpCode} \leq 400$.</td>
</tr>
</tbody>
</table>

- Executes a specified command inside the Container
- The diagnostic is considered **successful if the command exits with a status code of 0**.
Types of Probes Available For Liveness & Readiness

**TCP**
- Performs a TCP check against the Container’s IP address on a specified port.
- The diagnostic is considered **successful if the port is open**.

**HTTP**
- Performs an HTTP Get request against the Container’s IP address on a specified port and path.
- The diagnostic is considered **successful if 200 ≤ httpCode ≤ 400**.

**Command**
- Executes a specified command inside the Container.
- The diagnostic is considered **successful if the command exits with a status code of 0**.
Liveness Probes

- Enable Pod containers to recover from a broken state by being restarted

- Define periodic checks to determine if a Pod container is “alive” and if not, then it is killed and re-started

Without Liveness probes, K8S is truly blind and unaware that our workloads may have silently failed or stopped working
Liveness Probes

• Enable Pod containers to recover from a broken state by being restarted

• Define periodic checks to determine if a Pod container is “alive” and if not, then it is killed and re-started

Without Liveness probes, K8S is truly blind and unaware that our workloads may have silently failed or stopped working
Liveness Probes

• Enable Pod containers to recover from a broken state by being restarted

• Define periodic checks to determine if a Pod container is “alive” and if not, then it is killed and re-started

Without Liveness probes, K8S is truly blind and unaware that our workloads may have silently failed or stopped working
apiVersion: v1
class: 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

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes
Readiness Probes

• Allow containers to indicate that they are “not ready” and therefore should temporarily not receive traffic

• Define periodic checks to determine if a Pod container is “ready” and if not, then it will stop receiving traffic until it safely can

• Enforced by removing endpoint IPs for Pods automatically so that they will not receive traffic for services that they support
Readiness Probes

• Allow containers to indicate that they are “not ready” and therefore should temporarily not receive traffic

• Define periodic checks to determine if a Pod container is “ready” and if not, then it will stop receiving traffic until it safely can

• Enforced by removing endpoint IPs for Pods automatically so that they will not receive traffic for services that they support
Readiness Probes

• Allow containers to indicate that they are “not ready” and therefore should temporarily not receive traffic

• Define periodic checks to determine if a Pod container is “ready” and if not, then it will stop receiving traffic until it safely can

• Enforced by removing endpoint IPs for Pods automatically so that they will not receive traffic for services that they support
Readiness Probes (continued)

- When a Pod transitions back to "readiness", it will automatically resume receiving traffic without any intervention.

Without Readiness probes, K8S will send traffic to unready Pods and this can cause failures and unexpected results!
Readiness Probes (continued)

- When a Pod transitions back to “readiness”, it will automatically resume receiving traffic without any intervention.

**Without Readiness probes, K8S will send traffic to unready Pods and this can cause failures and unexpected results!**
```yaml
apiVersion: v1
text:
kind: Pod
metadata:
  labels:
    app: goproxy
    name: 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: 25
          periodSeconds: 20

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes
```
When using Probes

- If your workload requires time to properly startup / initialize

Then include an appropriate value for `initialDelaySeconds`

or else it may never be ready and may always restart
Autoscaling

- **Horizontal**
  - Scale In (light system load)
  - Scale Out (heavy system load)
  - Increase/decrease in the number of replicas
    - Pods - Horizontal Pod Autoscaler (HPA)
    - Nodes - Cluster Autoscaler (CA)

- **Vertical**
  - Increase/decrease in resource usage for a Pod
    - Pod resources – Vertical Pod Autoscaler (VPA)
      - Requires a Pod restart for changes to resources to take effect
  - Increase/decrease in persistent storage resources used by a Pod
Autoscaling

Scale In (light system load)

- Increase/decrease in the number of replicas
  - Pods - Horizontal Pod Autoscaler (HPA)
  - Nodes - Cluster Autoscaler (CA)

Scale Out (heavy system load)

- Increase/decrease in resource usage for a Pod
  - Pod resources – Vertical Pod Autoscaler (VPA)
    - Requires a Pod restart for changes to resources to take effect
- Increase/decrease in persistent storage resources used by a Pod

Horizontal

Vertical
Namespace Isolation & Resource Limitation
Namespace Isolation

- Namespaces scope resource names and can specify constraints for resource consumption to prevent Pods from running with unbounded CPU and memory requests/limits (which they will do by default)

- By default, all resources in Kubernetes are created in a default namespace

- Resources created in one namespace are hidden from other namespaces

Put tenant resources in corresponding & separate namespaces
Namespace Isolation

- Namespaces scope resource names and can specify constraints for resource consumption to prevent Pods from running with unbounded CPU and memory requests/limits (which they will do by default).

- By default, all resources in Kubernetes are created in a default namespace.

- Resources created in one namespace are hidden from other namespaces.

Put tenant resources in corresponding & separate namespaces.
Namespace Isolation

- Namespaces scope resource names and can specify constraints for resource consumption to prevent Pods from running with unbounded CPU and memory requests/limits (which they will do by default)

- By default, all resources in Kubernetes are created in a default namespace

- Resources created in one namespace are hidden from other namespaces

Put tenant resources in corresponding & separate namespaces
Namespace Isolation

- Namespaces scope resource names and can specify constraints for resource consumption to prevent Pods from running with unbounded CPU and memory requests/limits (which they will do by default)

- By default, all resources in Kubernetes are created in a default namespace

- Resources created in one namespace are hidden from other namespaces

Put tenant resources in corresponding & separate namespaces
Always Label Namespaces
Resource Limitation & Request

**Pod Resource Requests & Limits**

- Specified for each container in a Pod (inside a Pod specification)

- NOTE: Pod resource requests and limits are the sum of all resource requests/limits for each container

**Limit Ranges**

- Supports configuring default memory and/or CPU requests and limits for all K8S resources created in a namespace

- The combined resource usage for K8S resources in a namespace can not exceed the defined limit

**Resource Quotas**

- Supports configuring limits for the number of types of K8S resources that can be created within a namespace

- Can even be used to disallow the usage of a given resource within a namespace by setting the number of allowed resources for a type to 0
Resource Limitation & Request (continued)

**Limit Ranges**

- Supports configuring limits for the number of types of K8S resources that can be created within a namespace.
- The combined resource usage for K8S resources in a namespace cannot exceed the defined limit.

**Resource Quotas**

- Supports configuring limits for the number of types of K8S resources that can be created within a namespace.
- Can even be used to disallow the usage of a given resource within a namespace by setting the number of allowed resources for a type to 0.

**Pod Resource Requests & Limits**

- Specified for each container in a Pod (inside a Pod specification).
- NOTE: Pod resource requests and limits are the sum of all resource requests/limits for each container.

**Pod Resource Requests**

- Specified for each container in a Pod (inside a Pod specification).
- NOTE: Pod resource requests and limits are the sum of all resource requests/limits for each container.

**Pod Resource Limits**

- Supports configuring default memory and/or CPU requests and limits for all K8S resources created in a namespace.
- The combined resource usage for K8S resources in a namespace cannot exceed the defined limit.

**Pod Resource Requests & Limits**

- Specified for each container in a Pod (inside a Pod specification).
- NOTE: Pod resource requests and limits are the sum of all resource requests/limits for each container.
Pod Resource Requests & Limits

- Specified for each container in a Pod (inside a Pod specification)
- NOTE: Pod resource requests and limits are the sum of all resource requests/limits for each container

Limit Ranges

- Supports configuring default memory and/or CPU requests and limits for all K8S resources created in a namespace
- The combined resource usage for K8S resources in a namespace can not exceed the defined limit

Resource Quotas

- Supports configuring limits for the number of types of K8S resources that can be created within a namespace
- Can even be used to disallow the usage of a given resource within a namespace by setting the number of allowed resources for a type to 0
### Memory

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
  namespace: mem-example
spec:
  containers:
    - name: memory-demo-ctr
      image: polinux/stress
      resources:
        limits:
          memory: "200Mi"
        requests:
          memory: "100Mi"
      command: ["stress"]
      args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```


### CPU

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo
  namespace: cpu-example
spec:
  containers:
    - name: cpu-demo-ctr
      image: vish/stress
      resources:
        limits:
          cpu: "1"
        requests:
          cpu: "0.5"
      args:
        - -cpus
        - "2"
```

Why are Resource Requests Important?

• They ensure that the minimum required resources are available

• The scheduler bases its decisions only on allocable resource amounts
Why are Resource Limits Important?

• They define the maximum allowed value for a resource

• **Without limits, a Pod can consume as much resources as it likes and can potentially starve other workloads!**

• Exceeding memory limits may cause a Pod to be OOM killed

• Exceeding CPU limits may cause a Pod to be throttled
Quality Of Service Classes (QoS)

• Used to determine the priority order for which workloads will be killed first when the system needs to reclaim memory for higher priority workloads
  o Guaranteed (highest)
  o Burstable (lower)
  o BestEffort (lowest)

• Guaranteed:
  \[ \text{resource limits} = \text{resource requests} \]
Quality Of Service Classes (QoS)

- Used to determine the priority order for which workloads will be killed first when the system needs to reclaim memory for higher priority workloads
  - Guaranteed (highest)
  - Burstable (lower)
  - BestEffort (lowest)

- **Guaranteed:**
  
  \[ \text{resource limits} = \text{resource requests} \]
Quality Of Service Classes (QoS)

- Used to determine the priority order for which workloads will be killed first when the system needs to reclaim memory for higher priority workloads
  
  - Guaranteed (highest)
  - Burstable (lower)
  - BestEffort (lowest)

- **Guaranteed**: resource limits = resource requests
Quality Of Service Classes (QoS)

• Used to determine the priority order for which workloads will be killed first when the system needs to reclaim memory for higher priority workloads
  
  o Guaranteed (highest)
  o Burstable (lower)
  o BestEffort (lowest)

• **Guaranteed:**
  
  \[ \text{resource limits} = \text{resource requests} \]
Quality Of Service Classes (QoS)

• Used to determine the priority order for which workloads will be killed first when the system needs to reclaim memory for higher priority workloads
  
  o Guaranteed (highest)
  o Burstable (lower)
  o BestEffort (lowest)

• **Guaranteed:**

  Set resource limits = resource requests  
  (for all containers in a pod)
LimitRange Example(s)

```yaml
apiVersion: v1
class: LimitRange
metadata:
  name: mem-min-max-demo
spec:
  limits:
    - type: Container:
      max:
        memory: 1Gi
      min:
        memory: 500Mi
    default:
      memory: 1Gi
    defaultRequest:
      memory: 1Gi
    - type: Pod:
      max:
        memory: 1Gi
      min:
        memory: 500Mi

https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-constraint-namespace/
```

```yaml
apiVersion: v1
class: LimitRange
metadata:
  name: cpu-min-max-demo
spec:
  limits:
    - type: Container:
      max:
        cpu: "800m"
      min:
        cpu: "200m"
    default:
      cpu: "800m"
    defaultRequest:
      cpu: "800m"

https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/cpu-constraint-namespace/
```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-quota-demo
spec:
  hard:
    persistentvolumeclaims: "1"
    services.nodeports: "0"
    services.loadbalancers: "0"
    services: "5"
    pods: "5"
    secrets: "2"
    configmaps: "2"
    requests.cpu: 400m
    requests.memory: 200Mi
    limits.cpu: 600m
    limits.memory: 500Mi

https://kubernetes.io/docs/tasks/administer-cluster/quota-api-object/
When using Resource Quotas

• If resource **requests** and **limits** are specified, then each pod that the quota applies to **MUST** also define resource **requests and limits**

• You can always define default requests and limits via a **LimitRange**!
When using Resource Quotas

• If resource requests and limits are specified, then each pod that the quota applies to MUST also define resource requests and limits

• You can always define default requests and limits via a LimitRange!
Worker Node Isolation
Worker Node Isolation

• Workloads are scheduled to run on worker nodes (also referred to as “minions”) in a K8S cluster

By default, multi-tenant workloads can and will be scheduled to run on the same worker nodes and forced to share resources

• unless you explicitly prevent them from doing so

• Let’s take a closer look at this in more detail
Worker Node Isolation

• Workloads are scheduled to run on worker nodes (also referred to as "minions") in a K8S cluster

By default, multi-tenant workloads can and will be scheduled to run on the same worker nodes and forced to share resources

• unless you explicitly prevent them from doing so

• Let’s take a closer look at this in more detail
Worker Node Isolation

• Workloads are scheduled to run on worker nodes (also referred to as “minions”) in a K8S cluster.

By default, multi-tenant workloads can and will be scheduled to run on the same worker nodes and forced to share resources.

• unless you explicitly prevent them from doing so

• Let’s take a closer look at this in more detail.
Worker Node Isolation

• Workloads are scheduled to run on worker nodes (also referred to as "minions") in a K8S cluster

By default, multi-tenant workloads can and will be scheduled to run on the same worker nodes and forced to share resources

• unless you explicitly prevent them from doing so

• Let’s take a closer look at this in more detail
Taints & Tolerations

Taints

- Have a **key**, **value**, and an **effect** to prevent scheduling or execution of a Pod on a Node unless it **Tolerates the Taint**
- A flexible way to keep pods away from nodes or evict those that shouldn’t be running

```
kubectl taint nodes node1 key1=value1:NoSchedule
kubectl taint nodes node1 key1=value1:NoExecute
kubectl taint nodes node1 key2=value2:NoSchedule
```

Tolerations

- Also has a **key**, **value**, and an **effect**
- “matches” a taint if the keys are the same and the effects are the same
- Allows a Pod to be scheduled and/or executed on a Tainted node
Taints & Tolerations

**Taints**

- Have a key, value, and an effect to prevent scheduling or execution of a Pod on a Node unless it **Tolerates the Taint**
- A flexible way to keep pods away from nodes or evict those that shouldn’t be running

**Tolerations**

- Also has a **key, value, and an effect**
- “matches” a taint if the keys are the same and the effects are the same
- Allows a Pod to be scheduled and/or executed on a Tainted node

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
```
Taints & Tolerations (continued)

- The Node controller will automatically taint nodes when certain conditions are true:

  - **node.kubernetes.io/not-ready**: Node is not ready. This corresponds to the NodeCondition Ready being “False”.
  - **node.kubernetes.io/unreachable**: Node is unreachable from the node controller. This corresponds to the NodeCondition Ready being “Unknown”.
  - **node.kubernetes.io/out-of-disk**: Node becomes out of disk.
  - **node.kubernetes.io/memory-pressure**: Node has memory pressure.
  - **node.kubernetes.io/disk-pressure**: Node has disk pressure.
  - **node.kubernetes.io/network-unavailable**: Node’s network is unavailable.
  - **node.kubernetes.io/unschedulable**: Node is unschedulable.
  - **node.cloudprovider.kubernetes.io/uninitialized**: When the kubelet is started with “external” cloud provider, this taint is set on a node to mark it as unusable. After a controller from the cloud-controller-manager initializes this node, the kubelet removes this taint.

Taints & Tolerations (continued)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node.kubernetes.io/not-ready:</td>
<td>Node is not ready. This corresponds to the NodeCondition Ready being “False”.</td>
</tr>
<tr>
<td>node.kubernetes.io/unreachable:</td>
<td>Node is unreachable from the node controller. This corresponds to the NodeCondition Ready being “Unknown”.</td>
</tr>
<tr>
<td>node.kubernetes.io/out-of-disk:</td>
<td>Node becomes out of disk.</td>
</tr>
<tr>
<td>node.kubernetes.io/memory-pressure:</td>
<td>Node has memory pressure.</td>
</tr>
<tr>
<td>node.kubernetes.io/disk-pressure:</td>
<td>Node has disk pressure.</td>
</tr>
<tr>
<td>node.kubernetes.io/network-unavailable:</td>
<td>Node’s network is unavailable.</td>
</tr>
<tr>
<td>node.kubernetes.io/unschedulable:</td>
<td>Node is unschedulable.</td>
</tr>
<tr>
<td>node.cloudprovider.kubernetes.io/uninitialized:</td>
<td>When the kubelet is started with “external” cloud provider, this taint is set on a node to mark it as unusable. After a controller from the cloud-controller-manager initializes this node, the kubelet removes this taint.</td>
</tr>
</tbody>
</table>


Did you know that...

**For kubeadm installs, master nodes are tainted so that only internal K8S resources can run on them**
- These internal components define a matching toleration in their Pod specifications
Node Labels & Selectors

Node Labels

- Key-Value pairs added to a Node for labelling purposes

```bash
kubectl label nodes node1 disktype=ssd
```

Node Selectors

- Defined in a Pod specification to force it to be scheduled only to a Node with a matching label(s)
- Supports equality operators and set-based operators
- Less flexible than Node Affinity
Node Labels & Selectors

**Node Labels**
- Key-Value pairs added to a Node for labelling purposes

**Node Selectors**
- Defined in a Pod specification to force it to be scheduled only to a Node with a matching label(s)
- Supports equality operators and set-based operators
- Less flexible than Node Affinity

```
apiVersion: v1
class: Pod
metadata:
  name: nginx
labels:
  layer: web
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: Always
    nodeSelector:
      diskType: ssd
```
Node Affinity

• Similar to node selectors though much more flexible
• Allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node. There are two types:

requiredDuringSchedulingIgnoredDuringExecution

• “hard” rule that must be met for a Pod to be scheduled to a Node and ran there
• A guarantee that the scheduler will enforce

preferredDuringSchedulingIgnoredDuringExecution

• “soft” rule that may be met for a Pod to be scheduled to a Node
• Supports a weight field (1-100)
  o A greater value means “more preferred”
• Not a guarantee that the scheduler will enforce
Node Affinity

• Similar to node selectors though much more flexible
• Allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node. There are two types:

  requiredDuringSchedulingIgnoredDuringExecution
  • “hard” rule that must be met for a Pod to be scheduled to a Node and ran there
  • A guarantee that the scheduler will enforce

  preferredDuringSchedulingIgnoredDuringExecution
  • “soft” rule that may be met for a Pod to be scheduled to a Node
  • Supports a weight field (1-100)
    o A greater value means “more preferred”
  • Not a guarantee that the scheduler will enforce
Node Affinity (continued)

• Built In Node Labels (Cloud Provider specific)

- kubernetes.io/hostname
- failure-domain.beta.kubernetes.io/zone
- failure-domain.beta.kubernetes.io/region
- beta.kubernetes.io/instance-type
- beta.kubernetes.io/os
- beta.kubernetes.io/arch

https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
              - e2e-az1
              - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
            - key: another-node-label-key
              operator: In
              values:
              - another-node-label-value
  containers:
  - name: with-node-affinity
    image: k8s.gcr.io/pause:2.0

https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity
Pod Affinity / Anti-Affinity

- allow you to constrain which nodes your pod is eligible to be scheduled based on labels on pods that are already running on the node rather than based on labels on the node
- Supports a **topologyKey** which can also match a node label

**Affinity**
- Allows a Pod to run on a node if the node is already running one or more specified Pods

**Anti-Affinity**
- Prevents a Pod from running on a node if the node is already running one or more specified Pods
Pod Affinity / Anti-Affinity

- allow you to constrain which nodes your pod is eligible to be scheduled based on labels on pods that are already running on the node rather than based on labels on the node
- Supports a **topologyKey** which can also match a node label

**Affinity**
- Allows a Pod to run on a node if the node is already running one or more specified Pods

**Anti-Affinity**
- Prevents a Pod from running on a node if the node is already running one or more specified Pods
Pod Affinity / Anti-Affinity Example

apiVersion: v1
type: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
          matchExpressions:
            - key: security
              operator: In
              values:
                - S1
          topologyKey: failure-domain.beta.kubernetes.io/zone
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 100
              podAffinityTerm:
                labelSelector:
                  matchExpressions:
                    - key: security
                      operator: In
                      values:
                        - S2
                topologyKey: kubernetes.io/hostname
  containers:
    - name: with-pod-affinity
      image: k8s.gcr.io/pause:2.0

https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity
Security Limitation
Security Limitation

• While multi-tenant isolation is not only about security, **security certainly plays a big role!**

• Container security can be used to secure the container file system and enable/disable privileged actions and access to host machine Kernel features
  - **Security Contexts** and **Pod Security Policies** are the K8s features available for this

• Network security can be used to control ingress/egress connectivity between Pods
  - **Network Policies** are the K8s features available for this
Security Limitation

• While multi-tenant isolation is not only about security, security certainly plays a big role!

• Container security can be used to secure the container file system and enable/disable privileged actions and access to host machine Kernel features
  o **Security Contexts** and **Pod Security Policies** are the K8s features available for this

• Network security can be used to control ingress/egress connectivity between Pods
  o **Network Policies** are the K8s features available for this
Security Limitation

• While multi-tenant isolation is not only about security, **security** certainly plays a big role!

• Container security can be used to secure the container file system and enable/disable privileged actions and access to host machine Kernel features
  - **Security Contexts** and **Pod Security Policies** are the K8s features available for this

• Network security can be used to control ingress/egress connectivity between Pods
  - **Network Policies** are the K8s features available for this
Security Limitation (continued)

- Role Based Access Control is an approach to restricting system access to authorized users
  - **RBAC** is the K8s feature available for this

- Let’s take a closer look at each of these in more detail
Security Contexts

• Defines privilege and access control settings for a Pod or Container

• Can be defined within a Pod specification and/or within each container running inside of a Pod

⚠️ Defining at the container layer will override one defined at the Pod layer

⚠️ This can be overwritten at a higher level by security context rules inside of a Pod Security Policy
Security Contexts

• Defines privilege and access control settings for a Pod or Container

• Can be defined within a Pod specification and/or within each container running inside of a Pod

⚠️ Defining at the container layer will override one defined at the Pod layer

⚠️ This can be overwritten at a higher level by security context rules inside of a Pod Security Policy
Security Contexts

- Defines privilege and access control settings for a Pod or Container

- Can be defined within a Pod specification and/or within each container running inside of a Pod

**Defining at the container layer will override one defined at the Pod layer**

**This can be overwritten at a higher level by security context rules inside of a Pod Security Policy**
Security Contexts

- Defines privilege and access control settings for a Pod or Container
- Can be defined within a Pod specification and/or within each container running inside of a Pod

⚠️ Defining at the container layer will override one defined at the Pod layer

⚠️ This can be overwritten at a higher level by security context rules inside of a Pod Security Policy
apiVersion: v1
class: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000
  containers:
    - name: sec-ctx-demo-2
      image: gcr.io/google-samples/node-hello:1.0
      securityContext:
        runAsUser: 2000
        allowPrivilegeEscalation: false

https://kubernetes.io/docs/tasks/configure-pod-container/security-context/
Pod Security Policy

- A cluster-level resource that controls security sensitive aspects of the pod specification

- Defines a set of conditions that a pod must run with in order to be accepted into the system

Can override security settings configured by a Pod’s SecurityContext
Pod Security Policy

- A cluster-level resource that controls security sensitive aspects of the pod specification

- Defines a set of conditions that a pod must run with in order to be accepted into the system

Can override security settings configured by a Pod’s SecurityContext
Pod Security Policy

• A cluster-level resource that controls security sensitive aspects of the pod specification

• Defines a set of conditions that a pod must run with in order to be accepted into the system

Can override security settings configured by a Pod’s SecurityContext
Privileged Pod Security Policy Example

https://kubernetes.io/docs/concepts/policy/pod-security-policy
Restricted Pod Security Policy Example

https://kubernetes.io/docs/concepts/policy/pod-security-policy
Secrets

• Objects intended to hold sensitive data
  o Passwords
  o Tokens
  o Keys

• Base 64 encoded (not encrypted)

• Safer and more flexible than putting in an image or Pod definition
  o Reduces the risk of accidental exposure
  o Are mounted/used by Pods to inject sensitive data
Secrets

- Objects intended to hold sensitive data
  - Passwords
  - Tokens
  - Keys

- Base 64 encoded (not encrypted)

- Safer and more flexible than putting in an image or Pod definition
  - Reduces the risk of accidental exposure
  - Are mounted/used by Pods to inject sensitive data
Secrets

• Objects intended to hold sensitive data
  o Passwords
  o Tokens
  o Keys

• Base 64 encoded (not encrypted)

• Safer and more flexible than putting in an image or Pod definition
  o Reduces the risk of accidental exposure
  o Are mounted/used by Pods to inject sensitive data
Secrets

- Objects intended to hold sensitive data
  - Passwords
  - Tokens
  - Keys

- Base 64 encoded (not encrypted)

- Safer and more flexible than putting in an image or Pod definition
  - Reduces the risk of accidental exposure
  - Are mounted/used by Pods to inject sensitive data
Secrets

• Objects intended to hold sensitive data
  o Passwords
  o Tokens
  o Keys

• Base 64 encoded (not encrypted)

• Safer and more flexible than putting in an image or Pod definition
  o Reduces the risk of accidental exposure
  o Are mounted/used by Pods to inject sensitive data
Secrets

- Objects intended to hold sensitive data
  - Passwords
  - Tokens
  - Keys

- Base 64 encoded (not encrypted)

- Safer and more flexible than putting in an image or Pod definition
  - Reduces the risk of accidental exposure
  - Are mounted/used by Pods to inject sensitive data
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm

---

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

https://kubernetes.io/docs/concepts/configuration/secret/
Role Based Access Control (RBAC)

• a method of regulating access to computer or network resources based on the roles of individual users

• Uses Roles and ClusterRoles to represent permissions

• Uses RoleBindings and ClusterRoleBindings to grant role permissions to users
Role Based Access Control (RBAC)

• a method of regulating access to computer or network resources based on the roles of individual users
• Uses Roles and ClusterRoles to represent permissions
• Uses RoleBindings and ClusterRoleBindings to grant role permissions to users
Role Based Access Control (RBAC)

• a method of regulating access to computer or network resources based on the roles of individual users

• Uses Roles and ClusterRoles to represent permissions

• Uses RoleBindings and ClusterRoleBindings to grant role permissions to users
```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
  - apiGroups: [""] # "" indicates the core API group
    resources: ["pods"]
    verbs: ["get", "watch", "list"]

---

# This role binding allows "jane" to read pods in the "default" namespace.

kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
  - kind: User
    name: jane # Name is case sensitive
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

https://kubernetes.io/docs/reference/access-authn-authz/rbac/
```yaml
template: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
  - apiGroups: ['"]
    resources: ['"secrets"]
    verbs: ["get", "watch", "list"]

---

# This cluster role binding allows anyone in the "manager" group to read secrets in any namespace.

kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-secrets-global
subjects:
  - kind: Group
    name: manager # Name is case sensitive
    apiGroup: rbac.authorization.k8s.io

roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

https://kubernetes.io/docs/reference/access-authn-authz/rbac/
Network Policy Isolation
Network Policy Isolation

• Requires that a Network plugin which implements network policies (Calico, Weavenet, etc.) is installed and running on all nodes.

• Enables Pod isolation by explicitly rejecting or allowing connections to/from other Pods and/or other network endpoints.

• Network policies are defined for namespaces.

Without network policies, Pods will accept traffic from any source!
Network Policy Isolation

• Requires that a Network plugin which implements network policies (Calico, Weavenet, etc.) is installed and running on all nodes

• Enables Pod isolation by explicitly rejecting or allowing connections to/from other Pods and/or other network endpoints

• Network policies are defined for namespaces

Without network policies, Pods will accept traffic from any source!
Network Policy Isolation

• Requires that a Network plugin which implements network policies (Calico, Weavenet, etc.) is installed and running on all nodes

• Enables Pod isolation by explicitly rejecting or allowing connections to/from other Pods and/or other network endpoints

• Network policies are defined for namespaces

Without network policies, Pods will accept traffic from any source!
Network Policy Isolation

- Requires that a Network plugin which implements network policies (Calico, Weavenet, etc.) is installed and running on all nodes

- Enables Pod isolation by explicitly rejecting or allowing connections to/from other Pods and/or other network endpoints

- Network policies are defined for namespaces

**Without network policies, Pods will accept traffic from any source!**
Network Policy Isolation (continued)

Diagram created by Mike Knapp, Capital One
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
podSelector:
  matchLabels:
    role: db
policyTypes:
  - Ingress
  - Egress
ingress:
  - from:
    ipBlock:
      cidr: 172.17.0.0/16
    except:
      - 172.17.1.0/24
    namespaceSelector:
      matchLabels:
        tenant: fintech
      podSelector:
        matchLabels:
          role: frontend
    ports:
      - protocol: TCP
        port: 6379
egress:
  - to:
    ipBlock:
      cidr: 10.0.0.0/24
    ports:
      - protocol: TCP
        port: 5978

https://kubernetes.io/docs/concepts/services-networking/network-policies/
Network Policy Isolation
Deny/Allow Ingress Traffic
Examples

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
policyTypes:
  - Ingress

---

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector: {}
ingress:
  - {}

https://kubernetes.io/docs/concepts/services-networking/network-policies/
```
Network Policy Isolation
Deny/Allow Egress Traffic
Examples

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
podSelector: {}
policyTypes:
  - Egress

---

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
podSelector: {}

egress:
- {}
policyTypes:
  - Egress

https://kubernetes.io/docs/concepts/services-networking/network-policies/
```
Cloud Provider Hosted K8S
Cloud Provider Hosted K8S

- EKS (AWS), AKS (Azure), GKE (Google)
- Provision and manage K8S clusters on your behalf
- Can provide additional multi-tenancy related features
Cloud Provider Hosted K8S

- EKS (AWS), AKS (Azure), GKE (Google)

- Provision and manage K8S clusters on your behalf

- Can provide additional multi-tenancy related features
Cloud Provider Hosted K8S

• EKS (AWS), AKS (Azure), GKE (Google)

• Provision and manage K8S clusters on your behalf

• Can provide additional multi-tenancy related features
Kubernetes Roadmap - Multi-tenancy Supporting Features
K8S Roadmap - Multi-tenancy Supporting Features

• Add support for HPA and VPA to work on the same pods

• Add support for VPA to adjust resource limits without requiring a Pod restart

• Affinity
  o requiredDuringSchedulingRequiredDuringExecution
  o preferredDuringSchedulingRequiredDuringExecution
K8S Roadmap - Multi-tenancy Supporting Features

• Add support for HPA and VPA to work on the same pods

• Add support for VPA to adjust resource limits without requiring a Pod restart

• Affinity
  o requiredDuringSchedulingRequiredDuringExecution
  o preferredDuringSchedulingRequiredDuringExecution
K8S Roadmap - Multi-tenancy Supporting Features

• Add support for HPA and VPA to work on the same pods

• Add support for VPA to adjust resource limits without requiring a Pod restart

• Affinity
  o requiredDuringSchedulingRequiredDuringExecution
  o preferredDuringSchedulingRequiredDuringExecution
Summary / Take-Aways
Summary / Take-aways

- **NEVER** deploy an image with tag latest
- **Always** use a secure image registry
- You should **NEVER** deploy a single K8S Pod in Production
- **Always** define Liveness & Readiness Probes for your workloads
- Use autoscaling wherever you can
- **Always** use a tenant specific namespace for your workloads
Summary / Take-aways

• **NEVER** deploy an image with tag latest

• **Always** use a secure image registry

• You should **NEVER** deploy a single K8S Pod in Production

• Always define Liveness & Readiness Probes for your workloads

• Use autoscaling wherever you can

• Always use a tenant specific namespace for your workloads
Summary / Take-aways

• **NEVER** deploy an image with tag latest

• **Always** use a secure image registry

• You should **NEVER** deploy a single K8S Pod in Production

• Always define Liveness & Readiness Probes for your workloads

• Use autoscaling wherever you can

• Always use a tenant specific namespace for your workloads
Summary / Take-aways

• **NEVER** deploy an image with tag latest

• **Always** use a secure image registry

• You should **NEVER** deploy a single K8S Pod in Production

• **Always** define Liveness & Readiness Probes for your workloads

• Use autoscaling wherever you can

• **Always** use a tenant specific namespace for your workloads
Summary / Take-aways

• **NEVER** deploy an image with tag latest

• **Always** use a secure image registry

• You should **NEVER** deploy a single K8S Pod in Production

• Always define Liveness & Readiness Probes for your workloads

• Use autoscaling wherever you can

• Always use a tenant specific namespace for your workloads
Summary / Take-aways

• **NEVER** deploy an image with tag latest

• **Always** use a secure image registry

• You should **NEVER** deploy a single K8S Pod in Production

• Always define Liveness & Readiness Probes for your workloads

• Use autoscaling wherever you can

• Always use a tenant specific namespace for your workloads
Summary / Take-aways (continued)

• Always define resource (cpu/memory) requests and limits for your workloads
  • Set them equal to set the workload’s QoS class to Guaranteed

• Use LimitRange and ResourceQuota resources to further control resource limits across all workloads in a namespace

• Use Taints and Affinity to keep workloads away/near nodes and other workloads when isolation is necessary

• Use SecurityContexts and PodSecurityPolicies to allow/restrict workloads from using host kernel features
Summary / Take-aways (continued)

- Always define resource (cpu/memory) requests and limits for your workloads
  - Set them equal to set the workload’s QoS class to Guaranteed

- Use LimitRange and ResourceQuota resources to further control resource limits across all workloads in a namespace

- Use Taints and Affinity to keep workloads away/near nodes and other workloads when isolation is necessary

- Use SecurityContexts and PodSecurityPolicys to allow/restrict workloads from using host kernel features
Summary / Take-aways (continued)

- Always define resource (cpu/memory) requests and limits for your workloads
  - Set them equal to set the workload’s QoS class to Guaranteed

- Use LimitRange and ResourceQuota resources to further control resource limits across all workloads in a namespace

- Use Taints and Affinity to keep workloads away/near nodes and other workloads when isolation is necessary

- Use SecurityContexts and PodSecurityPolicys to allow/restrict workloads from using host kernel features
Summary / Take-aways (continued)

• Always define resource (cpu/memory) requests and limits for your workloads
  • Set them equal to set the workload’s QoS class to Guaranteed

• Use LimitRange and ResourceQuota resources to further control resource limits across all workloads in a namespace

• Use Taints and Affinity to keep workloads away/near nodes and other workloads when isolation is necessary

• Use SecurityContexts and PodSecurityPolicys to allow/restrict workloads from using host kernel features
Summary / Take-aways (continued)

- Use Secrets to help limit exposure to sensitive data
- Use RBAC for fine grained access control based on user/system roles
- Use NetworkPolicies to allow/restrict network access to/from workloads
- If the features discussed are not enough to isolate your workloads from other tenants, consider using a separate K8S cluster per tenant
• Use Secrets to help limit exposure to sensitive data

• Use RBAC for fine grained access control based on user/system roles

• Use NetworkPolicies to allow/restrict network access to/from workloads

• If the features discussed are not enough to isolate your workloads from other tenants, consider using a separate K8S cluster per tenant
Summary / Take-aways (continued)

- Use Secrets to help limit exposure to sensitive data
- Use RBAC for fine grained access control based on user/system roles
- Use NetworkPolicies to allow/restrict network access to/from workloads
- If the features discussed are not enough to isolate your workloads from other tenants, consider using a separate K8S cluster per tenant
Summary / Take-aways (continued)

- Use Secrets to help limit exposure to sensitive data
- Use RBAC for fine grained access control based on user/system roles
- Use NetworkPolicies to allow/restrict network access to/from workloads
- **If the features discussed are not enough to isolate your workloads from other tenants, consider using a separate K8S cluster per tenant**
More Talks About Our Platform
“Implementing SAAS on Kubernetes”

- When:
  - Thursday, Oct. 11th @ 1:40pm (1st session after lunch)

- Presenters:
  - Mike Knapp & Andrew Gao
More Talks Regarding Our Fraud Decisioning Platform (continued)

- “Will HAL Open the Pod Bay Doors? An (Enterprise FI) Decisioning Platform Leveraging Machine Learning”

  - When:
    - Thursday, Oct. 11th @ 2:50pm (3rd session after lunch)

  - Presenters:
    - Sumit Daryani & Niraj Tank
• “Panel Discussion: Real-World Kubernetes Use Cases in Financial Services: Lessons Learned from Capital One, BlackRock and Bloomberg”

  o When:
    ➢ Thursday, Oct. 11th @ 4:25pm

  o Capital One Panel Member:
    ➢ Jeffrey Odom
THE END