Advanced Kubernetes Scheduling and Orchestration with Custom Webhooks

Project Report

Under the guidance of

A. Cyrus Sabzevari

ECE612 –Real-Time Embedded Systems

Spring 2024

GEORGE MASON UNIVERSITY

COLLEGE OF ENGINEERING AND COMPUTING

Team Members:

Anurag Josyula: G01452375

Saiprashanth Vana: G01445432

Contents

List of Tables

List of Figures

1 Introduction

Kubernetes (often abbreviated as K8s) is a powerful open-source platform designed to automate the deployment, scaling, and management of containerized applications. It has become the de facto standard for container orchestration across cloud and on-premises environments due to its flexibility, scalability, and robust feature set.

1.1 Key Concepts

Kubernetes operates on a cluster architecture, composed of the following core concepts:

- **Nodes:** Physical or virtual machines that run the workloads. Nodes can be worker nodes, responsible for running application containers, or control plane nodes, managing the state of the cluster.
- **Pods:** The smallest deployable unit in Kubernetes. A pod encapsulates one or more closely related containers, providing shared networking and storage.
- **Deployments:** A declarative way to define desired state for a set of pods (replicas, scaling behavior). Kubernetes controllers ensure the actual state matches the desired state.
- **Services:** Mechanisms for abstracting network access to pods. Services provide stable internal endpoints for applications, even as individual pods may be created and destroyed.

1.2 Benefits of Kubernetes

- **Resource Optimization:** Kubernetes optimizes resource utilization by intelligently packing containers onto nodes based on resource requirements and constraints.
- **High Availability:** Self-healing mechanisms enable automatic restarting of failed containers, rescheduling of pods on healthy nodes, and rolling updates with zero downtime.
- **Scalability:** Kubernetes easily scales applications horizontally by adjusting the number of pods/replicas, and can be configured to auto scale based on load.
- **Portability:** Kubernetes declarative model enables applications to be deployed consistently across diverse environments (on-premises, cloud, hybrid) without significant modification.
- **Extensibility:** Kubernetes provides a rich ecosystem for customization via custom resource definitions (CRDs), operators, and the ability to integrate with external services and systems.

2 Context

2.1 Real time Scheduling

- **Goal:** Deterministic scheduling with guarantees on very low latency and meeting strict timing deadlines.
- **Scheduling Model:**
  - Strict priority-based scheduling, often using specialized real-time operating system kernels.
  - Pre-emption is very aggressive – higher-priority tasks immediately interrupt lower-priority ones.

2.2 Kubernetes Scheduling

- **Goal:** Best-effort scheduling focused on resource utilization and workload distribution across a cluster.
- **Scheduling Model:**
  - **Predicates:** Filters out nodes that fundamentally cannot run a pod (e.g., not enough resources, incompatible taints).
  - **Priorities:** Ranks nodes based on desirability, influencing pod placement (e.g., spreading pods across nodes, resource availability).
- **Preemption:** Kubernetes can preempt lower-priority pods to make room for higher-priority ones if necessary.

2.3 Kubernetes Scheduling vs Real time Scheduling

Below are the differences between Kubernetes Scheduling and Real-Time Scheduling

Feature	Kubernetes Scheduling	Real-Time Scheduling
Focus	Resource optimization, fairness	Strict deadlines, low latency
Scheduling mechanism	Predicates and priorities	Task priorities, pre-emption
Pre-emption	Lower-priority pods to accommodate higher-priority ones	Aggressive, immediate pre-emption of lower-priority tasks
Latency guarantees	None	Deterministic timing bounds
Workload suitability	General-purpose, some delay tolerance	Time-critical, failure has significant consequences

Table 1 Differences between Kubernetes Scheduling and Real-Time Scheduling

2.4 Integrating with Kubernetes

While Kubernetes is not designed for hard real-time workloads out of the box, there are ways to integrate aspects of real-time scheduling:

- **Node Affinity and Taints:** Ensure time-sensitive pods are scheduled on nodes with dedicated resources and specific hardware.
- **Pod Priority and Pre-emption:** Prioritize critical workloads to reduce the chance of pre-emption.
- **Specialized Kernels:** Run nodes with real-time Linux kernels for fine-grained control.
- **Custom Schedulers:** Develop custom Kubernetes schedulers with real-time scheduling algorithms, though this is a complex undertaking.

2.5 Real-Time Scheduling Works

1. **Task Definition:** Every task is defined with:
    - **Periodicity:** How often it runs.
    - **Deadline:** The maximum time allowed for task execution.
    - **Priority:** A fixed priority level.
2. **Priority-Based Scheduling:** The real-time scheduler always runs the highest priority task that is ready to execute.
3. **Pre-emption:** If a higher-priority task becomes ready, the currently running lower-priority task is immediately suspended, and the higher-priority task takes over.
4. **Timing Guarantees:** The real-time operating system (RTOS) kernel and specific hardware optimizations ensure tasks consistently meet their deadlines.

2.6 Kubernetes Scheduling Works

1. **Submission:** A user submits a pod specification (via YAML, kubectl, etc.) to the Kubernetes API server.
2. **Filtering (Predicates):** The scheduler identifies nodes that could potentially run the pod based on hard requirements:
    - **Node Selector:** Ensures the pod has labels matching the node's labels.
    - **Taints/Tolerations:** Prevents pods from running on nodes with taints, unless they have matching tolerations.
    - **Resource Requests:** Checks if the node has enough CPU, memory, etc. to accommodate the pod.
    - **Pod Affinity/Anti-affinity:** Rules for placing pods together or keeping them separated based on labels.
3. **Scoring (Priorities):** The scheduler ranks suitable nodes to determine the "best" fit:
    - **Resource Fit:** Prefers nodes with enough available resources to match pod requests.
    - **Node Affinity:** Favors nodes where the pod has a preference to run.
    - **Spreading:** Tries to distribute pods across nodes for availability.
    - **Taint Toleration:** Less negative score for nodes the pod tolerates.
    - **Numerous other built-in and customizable priority functions**
4. **Binding:** The scheduler assigns the pod to the highest-scoring node and updates the Kubernetes API server.
5. **Execution:** The kubelet on the assigned node is then responsible for pulling the container images and starting the pod's containers.

3 Scenario

In a real-time traffic light system, safety is paramount. How would real-time scheduling ensure the system prioritizes pedestrian safety when crossing requests are received, while still attempting to optimize traffic flow based on sensor data?

Tasks and Priorities

1. **Vehicle Traffic Monitoring (High Priority):**
    - **Periodicity:** Constant monitoring of road sensors
    - **Deadline:** Extremely short (milliseconds) to react quickly to changes in traffic flow
    - **Task:** Collect data on vehicle presence and density on each road approaching the intersection
2. **Pedestrian Crossing Request (High Priority):**
    - **Periodicity:** Event-driven (occurs when a pedestrian presses a button)
    - **Deadline:** Short (seconds) to ensure pedestrian safety and responsiveness
    - **Task:** Register the request and initiate crossing light changes when safe
3. **Traffic Light Control (Medium Priority):**
    - **Periodicity:** Based on pre-configured cycle times, but adaptable
    - **Deadline:** Flexible within a range (seconds to minutes) to maintain traffic flow
    - **Task:** Change traffic light states (green, yellow, red) based on a decision algorithm

Real-Time Scheduling in Action

- **Baseline:** The scheduler is primarily running the Traffic Light Control task, switching lights according to a standard timing pattern.
- **Vehicle Density Change:** The Vehicle Traffic Monitoring task detects a significant increase in traffic on a particular road. It interrupts the scheduler due to its high priority.
- **Adaptive Scheduling:** A real-time decision algorithm kicks in:
    - It analyzes the sensor data and the pedestrian request queue.
    - It might shorten the current cycle or temporarily skip a direction to accommodate the heavier traffic flow.
- **Pedestrian Request:** A pedestrian button is pressed. This high-priority task interrupts the scheduler.
- **Safety Override:** The algorithm analyzes the traffic situation and determines when it is safe to:
    - Initiate a pedestrian walk signal
    - Modify the traffic light cycle to accommodate the crossing phase

Notes:

The real-time scheduler ensures the system reacts promptly to critical events (pedestrian requests, sudden traffic surges).

Decision-making algorithms still play a crucial role in determining how the lights adapt.

4 Admission Webwooks

Admission webhooks in Kubernetes are HTTP callbacks that intercept requests to the Kubernetes API server during the creation, modification, or deletion of resources. They empower you to implement custom validation, policy enforcement, or even modify incoming objects before they are persisted in the cluster. Admission webhooks offer a flexible way to extend the core Kubernetes functionality, enabling you to tailor cluster behavior according to your organization's specific security, governance, or operational requirements.

4.1 Types of Admission Webhooks

- **Validating Webhooks:** Enforce custom rules and policies. They can only approve or deny requests, not modify them. Use cases:
    - Enforce pod security standards (e.g., disallow privileged containers).
    - Ensure resources adhere to organizational naming conventions.
    - Validate fields against external data sources.
- **Mutating Webhooks:** Modify incoming objects. They can approve/deny requests and introduce changes to the object. Use cases:
    - Inject sidecar containers (for logging, monitoring, etc.)
    - Set default resource limits or labels.
    - Mutate fields based on specific conditions.

4.2 What are Admission Webhooks?

- **HTTP Callouts:** Admission webhooks are essentially HTTP callbacks triggered by events within your Kubernetes cluster, specifically during the process of creating or updating resources.
- **Intercept API Requests:** When a user tries to create, delete, or modify a Kubernetes object (like a pod, deployment, service, etc.), the Kubernetes API server intercepts the request.
- **External Decision Makers:** The API server sends this request to configured admission webhooks *before* persisting the changes to the cluster.
- **Power to Modify or Reject:** Webhooks can inspect the request, apply logic, and then return a response allowing, rejecting, or even modifying the incoming object.

4.3 Why Admission Webhooks are Important

- **Enforce Custom Policies:** They extend Kubernetes beyond its built-in capabilities, allowing you to implement governance, security restrictions, and configuration best practices unique to your organization.
- **Centralized Control:** You can define cluster-wide rules without modifying every application's deployment manifests.
- **Dynamic Behaviour:** Admission webhooks can make decisions based on external systems or real-time conditions, providing flexibility that static configuration cannot.

4.4 How Admission Webhooks Work

1. **Configuration:** You define Validating Webhook Configuration or Mutating Webhook Configuration resources to tell the Kubernetes API server which webhooks to call and for which types of operations (create, update, delete) on specific resources.
2. **API Request Interception:** A user submits a request to the API server.
3. **Calling the Webhook:** The API server sends an Admission Review object (containing the request details) as JSON to the webhook's configured URL.
4. **Webhook Logic:** The webhook service applies its custom logic and responds with an Admission Response object, indicating:
    - Allowed (true/false)
    - Optional patch to modify the object (for mutating webhooks)
5. **API Server Action:** The API server acts on the webhook's response, either persisting the change, rejecting it, or applying the modification.

4.5 Admission Webhook in Real Time Scheduling (Sequences)

Admission webhooks can play a role in real-time scheduling within a Kubernetes environment, especially in scenarios where standard Kubernetes scheduling isn't perfectly tailored for strict real-time requirements:

Use Cases

- **Enforcing Real-Time Constraints:**
    - **Check Resource Requests:** A validating webhook could ensure that pods requesting real-time guarantees specify the appropriate CPU and memory requirements, or even custom labels that the cluster recognizes.
    - **Node Validation:** Ensure pods marked for real-time execution are only scheduled on nodes with compatible hardware (real-time kernels) or those nodes are not oversubscribed.
- **Integration with External Schedulers:**
    - **Hybrid Scheduling:** A mutating webhook could intercept pod requests and add annotations or labels that are understood by a specialized external real-time scheduler operating in parallel with the Kubernetes scheduler.
    - **Passthrough:** In a more complex setup, a validating webhook might act as a gatekeeper, only allowing pods with specific characteristics to proceed to the standard Kubernetes scheduler. Real-time pods could be rerouted to a dedicated scheduler.
- **Dynamic Configuration:**
    - **Time-Based Policies:** A validating webhook could integrate with an external time source and deny the creation of general-purpose pods during designated windows reserved for critical real-time workloads.
    - **QoS Enforcement:** Implement Quality of Service tiers by validating webhooks that enforce resource limits or scheduling priorities based on external data sources or real-time load on the cluster.

4.6 Example

A validating admission webhook could:

1. Check if a pod requests a custom label like "real-time-critical".

2. Verify the pod has sufficient resource requests for its real-time priority.
3. Query an external database to determine if any nodes in the cluster have both the capacity and a real-time compatible kernel version to run the workload.
4. Approve the request only if all conditions are met, potentially providing helpful error messages if not.

Caveats

True hard real-time guarantees usually necessitate a specialized real-time operating system. Admission webhooks in Kubernetes primarily augment standard scheduling to get closer to real-time behaviour or enable interactions with external real-time systems.

5 Mutating Webhooks

Mutating webhooks in Kubernetes are a special type of admission webhook that goes beyond simple approval or denial. They dynamically modify Kubernetes resources (pods, deployments, etc.) during creation or updates. Imagine them as editors that intercept incoming object requests, adding, removing, or changing fields based on your defined logic. This allows you to inject sidecars, enforce resource defaults, or integrate with external systems for configuration, all before the resource is persisted in the cluster.

5.1 How Mutating Webhooks Work

1. **Configuration:** You set up a `MutatingWebhookConfiguration` resource, informing the Kubernetes API server about your webhook and when to call it (e.g., on 'create' operations for Pods).
2. **API Request Interception:** A request to create or update a relevant object is sent to the API server.
3. **Calling the Webhook:** The API server sends the object data to your webhook's configured endpoint.
4. **Webhook Logic:** Your webhook service applies any necessary modifications to the object.
5. **Response:** The webhook sends back a response containing a JSON patch describing the changes.
6. **API Server Action:** The Kubernetes API server applies the patch to the original object and then persists the modified version.

5.2 Understanding Mutating Webhooks in Kubernetes

- **Dynamic Modification:** Mutating webhooks are a type of admission webhook in Kubernetes that have the power to modify incoming object requests before the Kubernetes API server persists them in the cluster.
- **Object Transformation:** They can add, remove, or change fields within pods, deployments, services, and other Kubernetes resources.
- **Beyond Validation:** Unlike validating webhooks (which can only approve or reject), mutating webhooks introduce changes, providing more active control over objects.

5.3 Common Use Cases

- **Injecting Sidecars:** Automatically add containers into pods, such as for logging, monitoring, security, or networking proxies.
- **Enforcing Defaults:** Set default values for labels, annotations, resource limits, or other fields if not explicitly specified by the user.
- **External Integrations:** Modify objects to interact with external systems for configuration, authorization, or data enrichment.
- **Policy Enforcement**: Implement custom rules that transform objects to adhere to security standards or organizational best practices.

5.4 Mutating Webhook Parameters

webhooks: A list of webhook definitions, where you specify the following for each webhook:

- **Name:** A unique name for the webhook.
- **clientConfig:**
  - **service:** Details of how to reach the webhook service:
    - **namespace:** Namespace where the webhook service resides.
    - **name:** Name of the Kubernetes service.
    - **path:** The URL path to call on the service.
  - **caBundle:** Base64-encoded CA certificate to validate the webhook service's TLS certificate.
- **Rules:** Defines which API operations trigger the webhook:
  - **operations:** List of operations (CREATE, UPDATE, DELETE, etc.).
  - **apiGroups:** API groups to apply the rule to (e.g., "*" for all).
  - **apiVersions:** API versions to apply the rule to (e.g., "v1").
  - **resources:** Kubernetes resources to trigger the webhook for (e.g., "pods", "deployments").
- **Failure Policy:** Behavior if the webhook cannot be reached (Ignore or Fail).
- **Namespace Selector:** Restricts the webhook to operate only on specific namespaces.
- **Object Selector:** Fine-grained selection of objects based on labels and fields.
- **Side Effects:** Must be set to "Some" or "None". Indicates if the webhook has potential side-effects outside of the Kubernetes cluster.
- **Admission Review Versions:** List of supported Admission Review versions (usually "v1" or "v1beta1").
- **Timeout Seconds:** Timeout for contacting the webhook.

6 Design Architecture And Flowchart

6.1 Design Architecture

1. **Issuing a command to bring the particular application up**
   - **User Interface:** The user employs a tool to interact with the Kubernetes cluster.
     - **kubectl:** The standard command-line interface for Kubernetes.
     - **Dashboard:** A web-based graphical user interface for cluster management.
     - **Custom/3rd Party Tools:** Tools built on top of the Kubernetes API for specific workflows.
   - **Command Structure:** The command typically defines a Kubernetes resource.
     - **Deployment:** A declarative way to specify how an application should run (replicas, image, etc.).
     - **Pod:** A direct request to create a pod (group of containers), less common for typical usage.
     - **Other Resource Types :** Services, ConfigMaps, etc.
2. **The command goes to the Kube API server**
   - **API Gateway:** The Kubernetes API server is the heart of the control plane. It provides a RESTful API for all cluster interactions.
   - **Authentication/Authorization (AuthN/AuthZ):** Before processing the request, the API server verifies:
     - **Identity:** Who is the user/client? (User accounts, service accounts)
     - **Permissions:** Does the user have the right to create the requested resource? (Role-Based Access Control - RBAC)
3. **The Kube API server sends the command to the Kube scheduler and the scheduler returns a job ID**
   - **Asynchronous Operation:** Kubernetes operations are mostly asynchronous for scalability and resilience. The API server doesn't wait for full deployment but acknowledges the request.
   - **Persistence (etcd):** The desired state of the resource is recorded in etcd, Kubernetes' distributed key-value store.
   - **Job ID:** Serves as a unique reference for tracking the request's progress.
4. **Scheduler will now read the command and it also reads its required specifications**
   - **Scheduling Logic:** The kube-scheduler is a complex component with pluggable algorithms. It considers:
     - **Resource Requests:** CPU, memory, etc. specified in the command.
     - **Node Constraints:** Does the node have sufficient resources, hardware, etc.
     - **Policies:** Node affinity/anti-affinity, taints/tolerations, pod priority.
5. **Scheduler sends its validated specifications to the admission webhook which checks for security enhancements and reports the results back to kube scheduler**
   - **Admission Webhooks:** Powerful mechanism for policy enforcement *before* objects are persisted in etcd.
     - **Security Policies:** Can enforce pod security contexts, image provenance checks, restrict privileged containers, etc.

    - **Custom Logic:** Organizations can implement specific validation or governance rules.

6. **If it does not pass error is served, else the specification is passed to mutating webhook to change or add default configurations**
    - **Error Handling:** User receives an error message with the policy violation reason.
    - **Mutating Webhooks:** These webhooks can modify the object *before* it's stored. Common use cases:
        - **Injecting Sidecars:** Adding monitoring or logging containers.
        - **Setting Defaults:** For labels, annotations, resource limits
7. **Now this fixed configuration is taken and passed to node scheduler**
    - **Binding Decision:** The scheduler has decided which specific node will run the workload. It updates the object in etcd with this binding.
8. **And node scheduler instructs container constructor to run the application, if there is any error it is reported back, else the job is marked as success**
    - **Kubelet:** Agent running on each node, communicates with the API server.
    - **Container Runtime:** Responsible for creating containers.
    - **Monitoring:** Kubelet reports pod status back to the API server, enabling controllers to take corrective actions if needed.

6.2 Design Flowchart

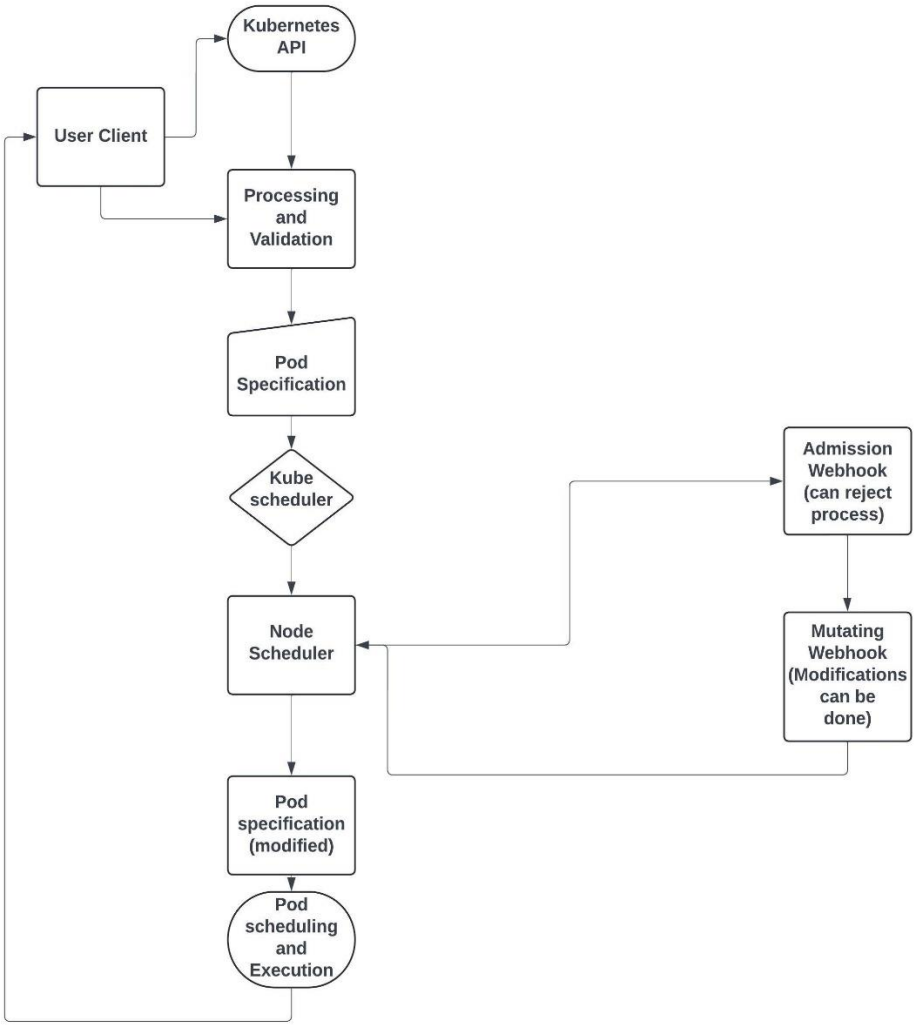


Figure 6-1 Design Architecture

6.3 Project Workflow with Validation and Logging

1. **Project Start:** The project begins.
2. **Log Collection:** The project starts collecting logs, which likely track ongoing activities and events related to the project.
3. **Mutation Request Check:** The process checks if a mutation request has been received. A mutation request suggests a proposed change to some aspect of the project.
4. **Path A - No Mutation Request:**
    a. **Set Lifespan Tolerances:** Acceptable durations for different parts of the project are established.
    b. **Inject Environment:** The necessary tools, technologies, and operating environment for the project are set up or introduced.
    c. **Inject Annotations:** Metadata, comments, or additional information may be added to the project records.
5. **Path B - Mutation Request Received**

a. **Validation Request:** The feasibility and impact of the proposed change are assessed.
b. **Validation Response:** A response is sent indicating whether the mutation request is accepted or rejected.

6. **Mutation Response:** A final response is sent, communicating the decision regarding the mutation request.
7. **Project End:** The project reaches its conclusion.

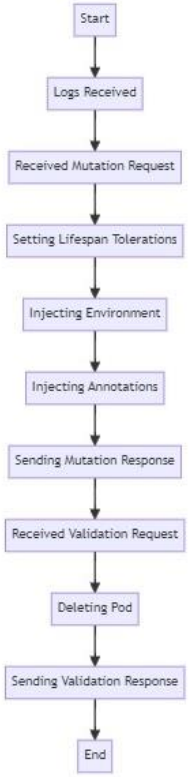


Figure 6-2 Project Workflow

7 RESULTS AND DISCUSSION

7.1 Annotations Used

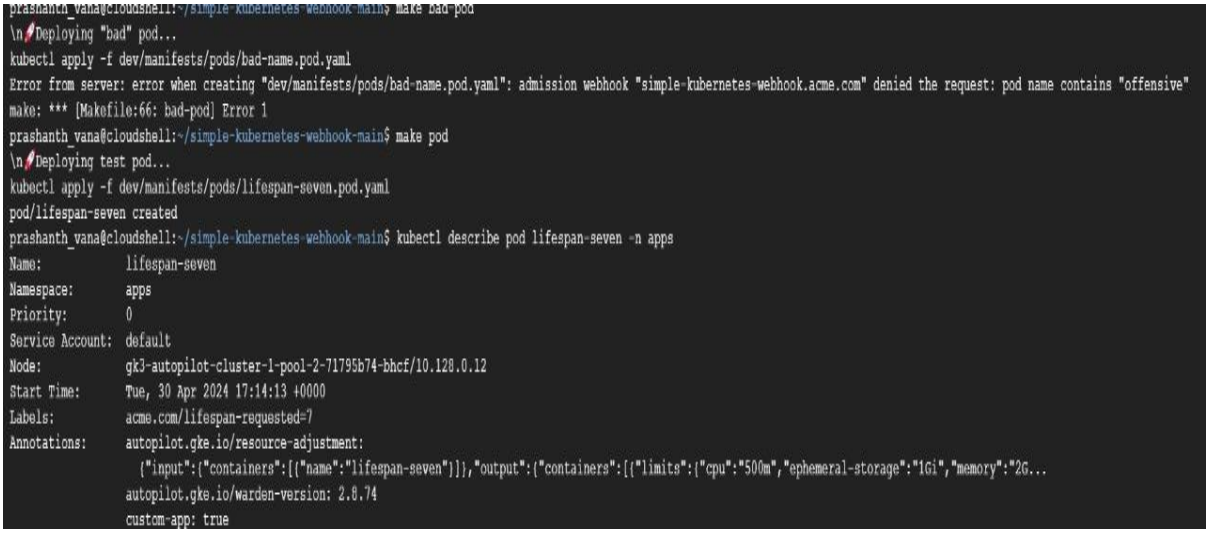


Figure 7-1 Deployment of Pod

- The above picture represents a successful deployment for a pod named "lifespan-seven".
- This pod has labels and annotations associated with it. Labels are key-value pairs used for grouping and selecting pods. In this case, the label "acmo.com/lifespan-requested=7 [invalid URL removed]" suggests a desired lifespan of 7 for this
- Annotations provide non-identifying metadata for informational purposes, we have used gk3-autopilot by google which will be useful in auto-scaler.
- Kubernetes Scheduler assigns pods to nodes based on resource requirements and availability.
- Auto-scaler automatically scales the cluster by provisioning additional nodes if resources are insufficient.
- Cluster Auto-scaler manages node scaling based on cluster load and resource demands.
- Node transitions from "pending" to "running" once scheduled by kube-scheduler; kubelet manages pod lifecycle on the node.

7.2 Determining Bad Pods

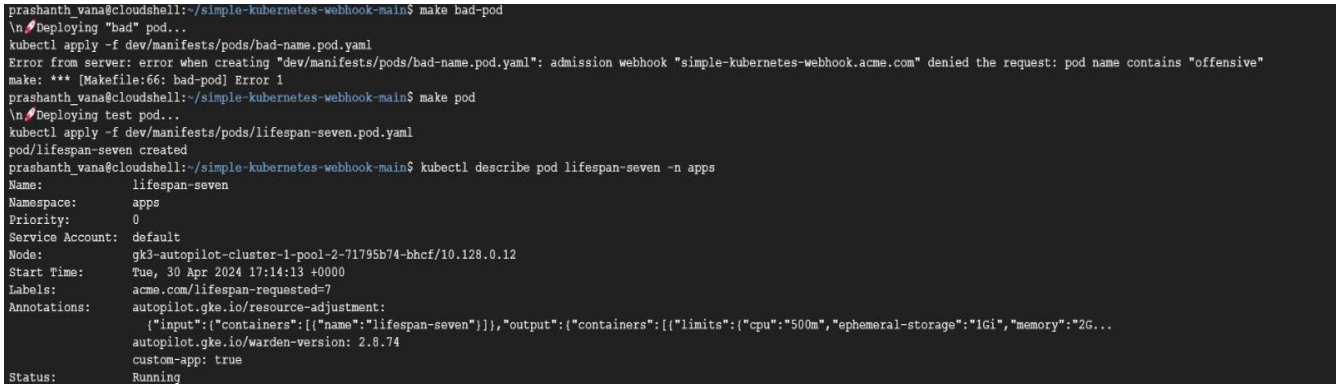


Figure 7-2 determining good and bad pods

Pod Selection

- A wide range of vocabulary is written into the code to determine good and bad pods if the pod does not have the right specification, it shows offensive as show in the picture above, given command make bad pod.
- If make pod command validation is successful, then it moves to the lifespan-seven container for further execution.
- Security Risks: Provides a minimalist set of utilities, it may lack security features found in full-fledged versions of these utilities. This could potentially create security vulnerabilities if not properly managed.
- Limited Functionality: While it provides essential utilities, it may lack some advanced features found in full versions of these tools. This limitation could hinder certain operations within pods.
- Compatibility Issues: Utilities might behave differently from their full-fledged counterparts, leading to compatibility issues with certain applications or scripts designed for standard Unix environments.

7.3 Environment Variables

```
Status:           Running
SeccompProfile:   RuntimeDefault
IP:               10.16.128.70
IPs:
  IP:  10.16.128.70
Containers:
  lifespan-seven:
    Container ID:  containerd://fedf01dcde763c2a53346d930756989fa30519f2ad68917f050ffbba7f53c13c
    Image:         busybox
    Image ID:      docker.io/library/busybox@sha256:6776a33c72b3af7582a5b301e3a08186f2c21a3409f0d2b52dfddbdbe24a5b04
    Port:          <none>
    Host Port:     <none>
    Args:
      sleep
      3600
    State:          Running
      Started:      Tue, 30 Apr 2024 17:14:14 +0000
    Ready:          True
    Restart Count:  0
    Limits:
      cpu:                500m
      ephemeral-storage:  1Gi
      memory:             2Gi
    Requests:
      cpu:                500m
      ephemeral-storage:  1Gi
      memory:             2Gi
    Environment:
      KUBE:  true
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-mhvrr (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
```

Figure 7-3 Environment Variables and Resource Consumption

- The Kubernetes pod "lifespan-seven" is running a container based on the Busy-Box image.
- It has been assigned an IP address and is using some of the allocated CPU and memory, here we took 500m (0.5 core of CPU) 1GB storage and 2GB RAM of resources.
- The pod itself is considered healthy based on the listed condition above showing Ready, Initialized, Containers Ready as true.
- Variables used are KUBE = true which is injected by us (refer inject_env code file) and custom app = true
- Initialized which indicates whether all init containers have completed successfully.
- Ready which indicates whether the Pod is fully operational and can serve requests.
- Containers-Ready which indicates whether all containers in the Pod are ready and able to accept traffic.
- Pod Scheduled which indicates whether the Pod has been scheduled to run on a node.

7.4 Logs

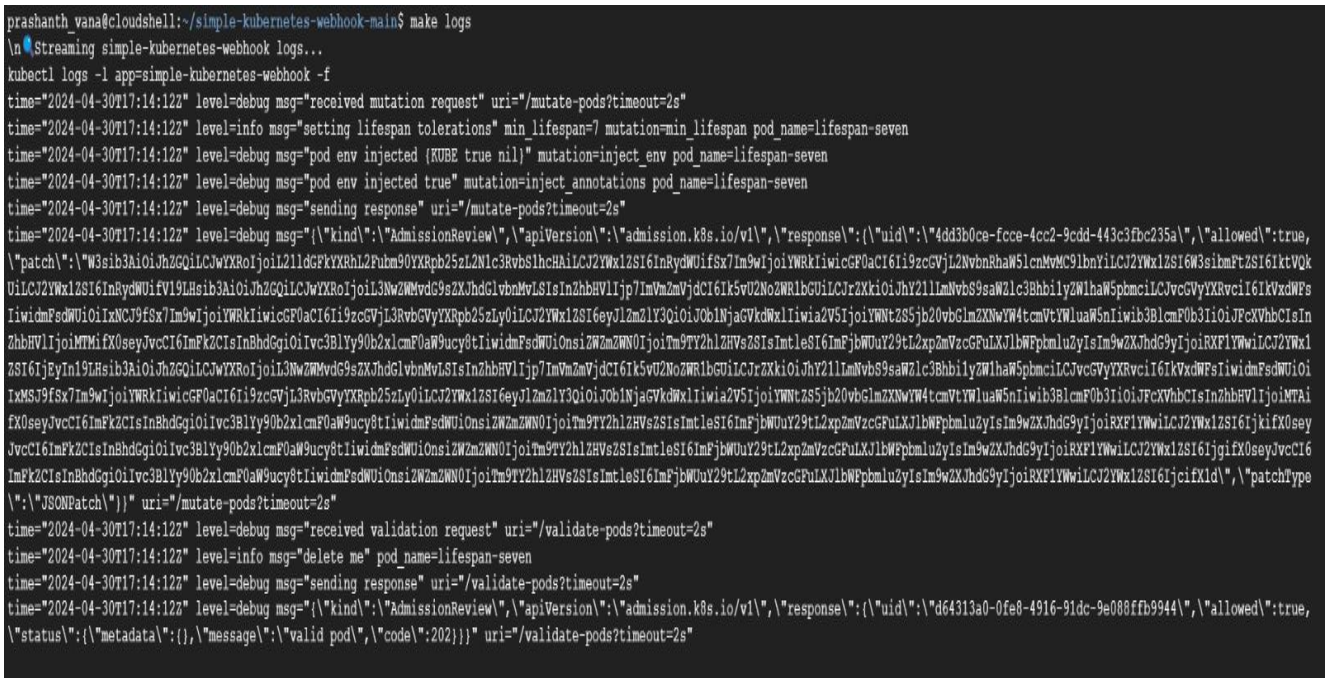
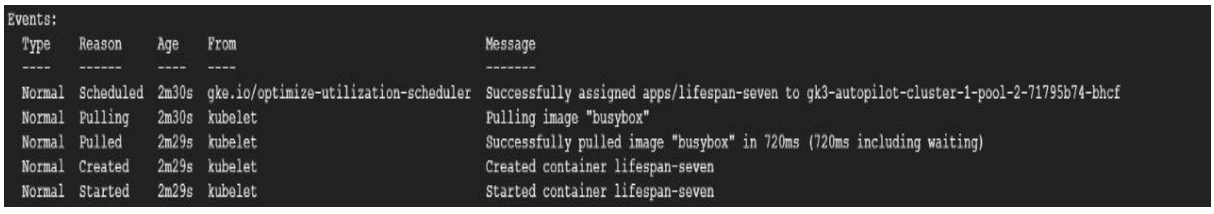


Figure 7-4 Logs

7.5 Event Sections



Type	Reason	Age	From	Message
Normal	Scheduled	2m30s	gke.io/optimize-utilization-scheduler	Successfully assigned apps/lifespan-seven to gk3-autopilot-cluster-1-pool-2-71795b74-bhcf
Normal	Pulling	2m30s	kubelet	Pulling image "busybox"
Normal	Pulled	2m29s	kubelet	Successfully pulled image "busybox" in 720ms (720ms including waiting)
Normal	Created	2m29s	kubelet	Created container lifespan-seven
Normal	Started	2m29s	kubelet	Started container lifespan-seven

Figure 7-5 Event Sections

- Each log usually contains 5 properties: Type: the type of event like Normal, Warning, or Error. Reason.
- A brief code describing the event (e.g. Pulling, Created).
- Age: the time when the event occurred.
- From: the component responsible for logging the event.
- Message: detailed description of the event.
- Lifespan seven is the container created and started and all this information is given by kubelet since it acts as telemetry.

REFERENCES

1) Kubernetes Concepts and tutorial-https://kubernetes.io/docs/

2) Mermaid flow chart tutorial-https://mermaid.live/

3) Reference Documentation-https://erictune.github.io/docs/reference/

4) Intro to DevOps-https://github.com/run-x/awesome-kubernetes

5) Intro and tutorial to GKE- google Kubernetes engine and cloud- https://cloud.google.com/kubernetes-engine/docs

6) Webhook development-https://workos.com/blog/building-webhooks-into-your-application-guidelines-and-best-practices