Introduction to Kubernetes

Bob Rotsted

What is a Kubernetes!?

Clusters, Nodes, Control Planes

- A Kubernetes **cluster** consists of a set of worker machines called **nodes**, that run containerized applications. Every cluster has at least one worker node.
- The worker nodes host the application workloads.
- The **control plane** manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

Review: Running a Container

Local Management vs. Cluster Orchestration

- **On Your Laptop**: Running Docker containers directly on your laptop involves manual management using Docker CLI commands. You're responsible for starting, stopping, and managing the containers and their configurations individually.
- **In Kubernetes**: Kubernetes orchestrates containers at scale across a cluster of machines. It manages the deployment, scaling, and networking of containers automatically according to defined configurations (e.g., YAML files).

Review: Running a Container

Single Host vs. Multi-Host Deployment

- **On Your Laptop:** Containers run on a single host, your laptop. This setup is suitable for development, testing, or small-scale production environments where high availability and scalability are not critical concerns.
- **In Kubernetes:** Containers are deployed across multiple nodes in a cluster, offering high availability, load balancing, and scalability. Kubernetes ensures that applications can handle increased load or failures in the cluster by adjusting the number of running containers as needed.

Review: Running a Container

Manual Scaling vs. Automated Scaling and Self-healing

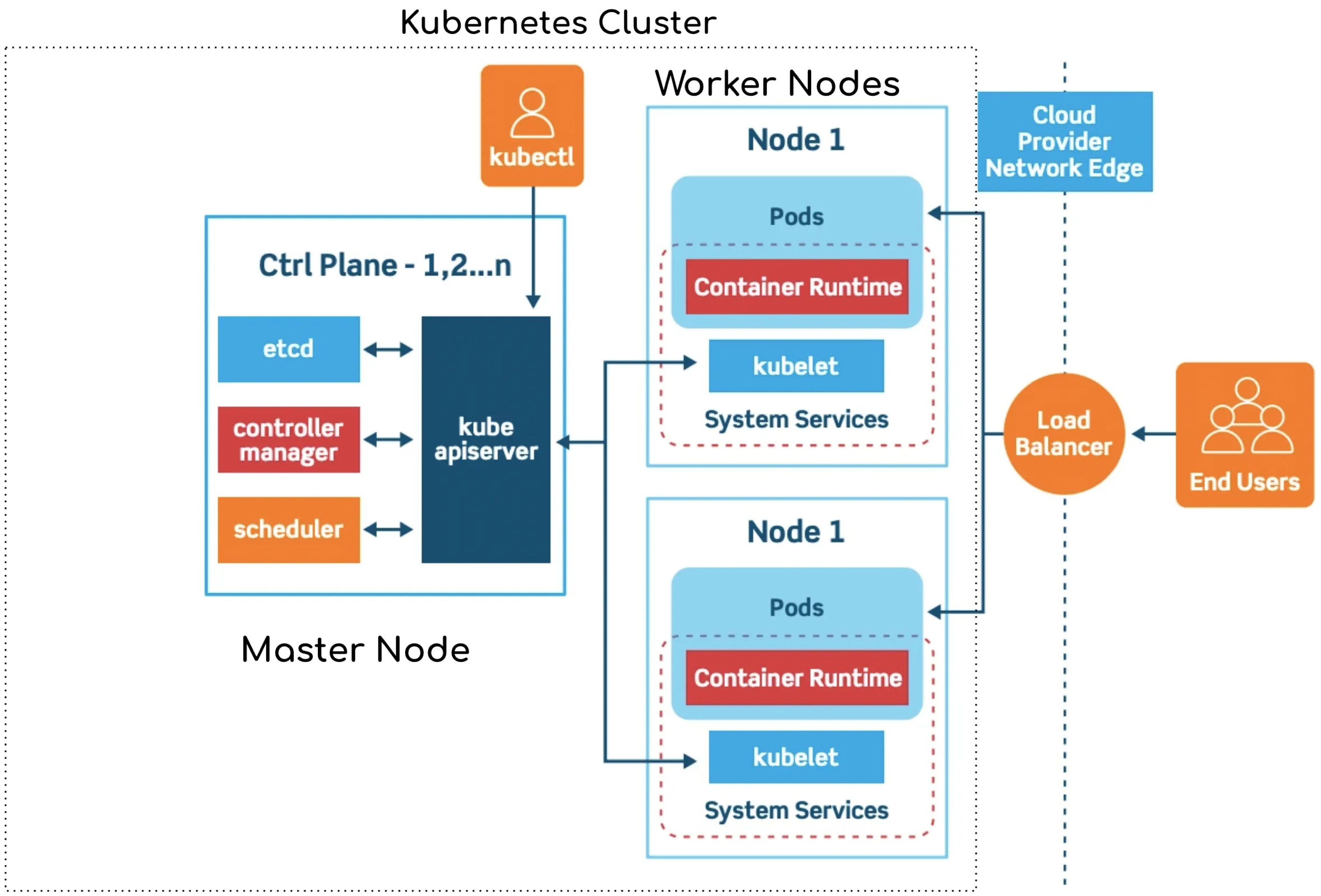
- **On Your Laptop:** Scaling involves manually starting more containers or stopping them as needed. There is no built-in mechanism for automatically adjusting the number of running containers based on demand or for replacing failed containers.
- **In Kubernetes:** Kubernetes supports automated scaling based on metrics like CPU usage or custom metrics. It also provides self-healing by automatically replacing containers that fail, are unresponsive, or don't meet the user-defined health criteria.

From Monoliths to Microservices

Running infrastructure the Kubernetes way

- Kubernetes is designed around the idea of microservices—small, loosely coupled services that communicate over well-defined APIs. This requires developers to think about applications as a collection of independent services rather than a single, monolithic codebase.
- Services must be designed to operate independently, allowing for scaling, updates, and failure recovery without affecting the entire application.

Cluster Infrastructure



https://medium.com/the-programmer/kubernetes-fundamentals-for-absolute-beginners-architecture-components-1f7cda8ea536

Control Plane

The infrastructure that orchestrates containers

- **API Server**: The frontend to the control plane that exposes the Kubernetes API.
- **etcd**: A consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.
- **Scheduler**: Watches for newly created Pods with no assigned node, and selects a node for them to run on.
- …

Nodes

The infrastructure that executes the containers

- **kubelet**: An agent that runs on each node in the cluster, ensuring containers are running in a Pod according to their specifications.
- **Kube-Proxy**: Manages network rules on each node, enabling network communication to Pods from within or outside the cluster.
- **Container Runtime**: The underlying software used to run containers, such as Docker, containerd, or any Kubernetes CRI (Container Runtime Interface)-compliant runtime.
- **Node Controller**: Part of the control plane that monitors the health and status of nodes, handling tasks like lifecycle operations and maintenance.
- **Container Network Interface (CNI)**: A set of standards and libraries for configuring network interfaces for Linux containers. CNI is used in Kubernetes to facilitate Pod networking, allowing Pods to communicate with each other and with the outside world in a standardized way.
- …

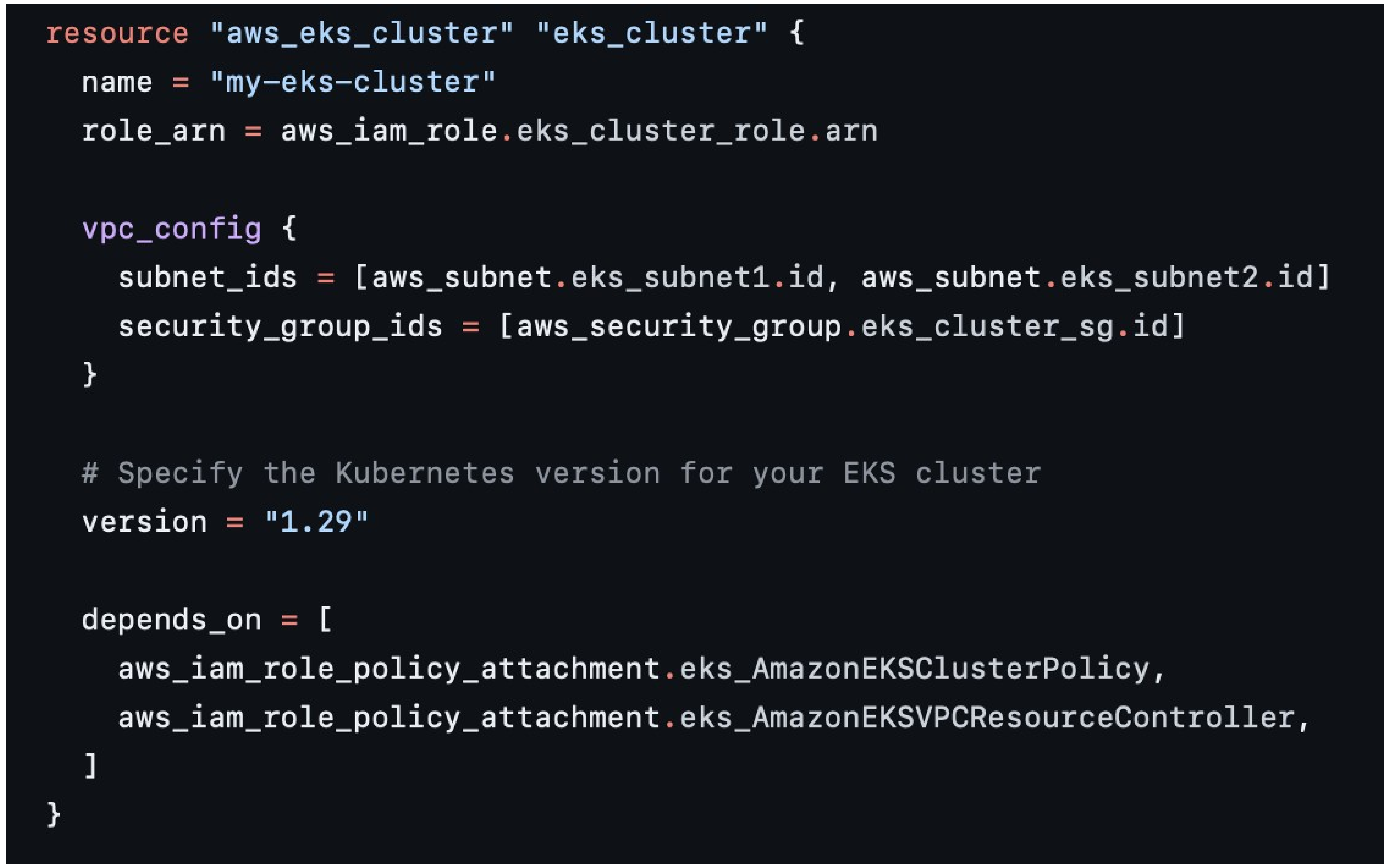
Managed Kubernetes

Elastic Kubernetes Service (EKS)

- **Cluster Management**: Automates the provisioning, setup, and scaling of Kubernetes clusters, reducing the complexity of cluster management.
- **High Availability**: Ensures clusters are highly available and distributed across multiple availability zones to minimize downtime.
- **Security**: Integrates with cloud provider security services to manage access and encryption, enhancing the security posture of your applications.
- **Scalability**: Automatically scales resources based on demand, ensuring efficient resource utilization and performance.

Managed Kubernetes: EKS

How we built today’s lab environment



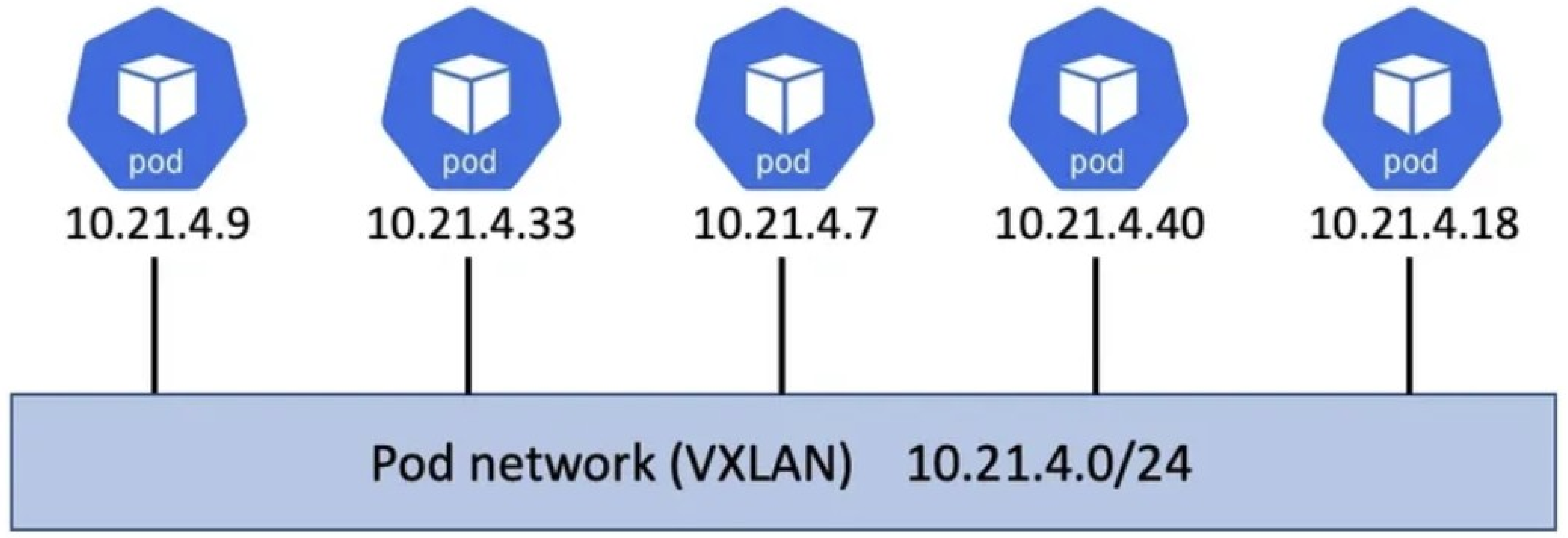
```
resource "aws_eks_cluster" "eks_cluster" {
  name = "my-eks-cluster"
  role_arn = aws_iam_role.eks_cluster_role.arn

  vpc_config {
    subnet_ids = [aws_subnet.eks_subnet1.id, aws_subnet.eks_subnet2.id]
    security_group_ids = [aws_security_group.eks_cluster_sg.id]
  }

  # Specify the Kubernetes version for your EKS cluster
  version = "1.29"

  depends_on = [
    aws_iam_role_policy_attachment.eks_AmazonEKSClusterPolicy,
    aws_iam_role_policy_attachment.eks_AmazonEKSVPCResourceController,
  ]
}
```

- Built and deployed with Terraform
- Code is in Github: https://github.com/nsrc-org/cloud-virt-labs/blob/main/eks-terraform/main.tf
- AWS is managing “node groups” for our cluster and can auto-scale the number of nodes in our cluster based on utilization

Networking

Communication Within the Cluster

- **Flat Network Space**: Kubernetes assigns a unique IP address to each pod across the cluster, enabling direct communication between pods without NAT (Network Address Translation).
- **No Overlapping IPs**: Pods can communicate with each other using their IP addresses, assuming they know those IPs, without any IP overlap issues.



https://nigelpoulton.com/demystifying-kubernetes-service-discovery/

Networking

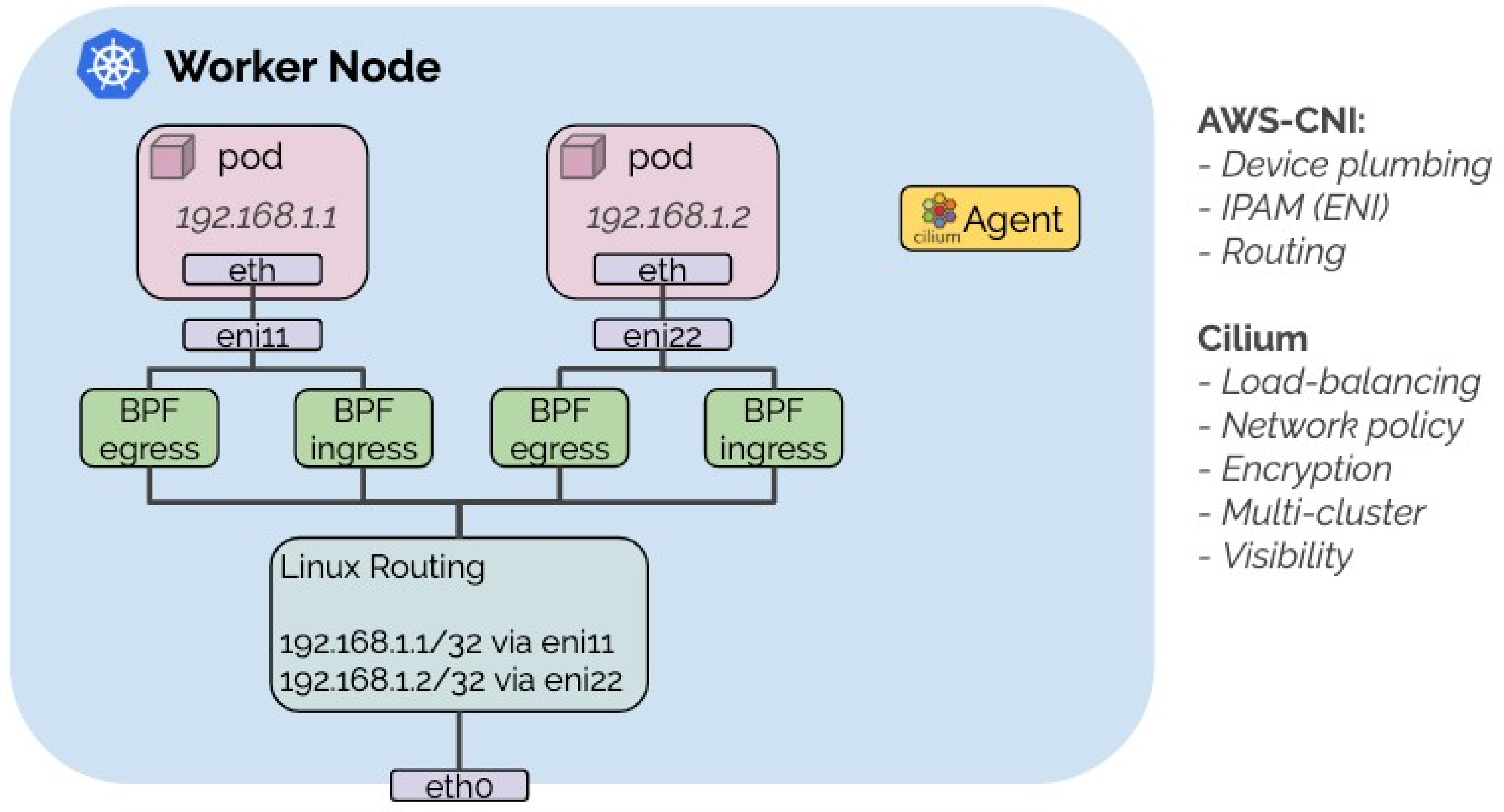
Services - Stable access in a dynamic environment

- **Stable Access Point**: Pods can be created, destroyed, and moved frequently, causing their IP addresses to change. A Service acts as an abstraction layer that provides a single, fixed entry point for accessing a set of pods that offer the same functionality. For reliability purposes, a pod should speak to a service abstraction rather than directly to a pod's IP address.
- **Load Balancing**: A Service automatically distributes incoming network traffic across all pods that match its selector criteria, effectively load balancing requests to maintain application performance and reliability.
- **Service Discovery**: Kubernetes uses DNS for service discovery. Each Service is assigned a DNS name, making it easy for internal communication within the cluster. Pods can resolve the Service name to its stable IP address to access the service.

Networking

What is a CNI?

- **Pod Networking**: When a Pod is created or destroyed, Kubernetes invokes the configured CNI plugin to attach or detach the network from the Pod. This process includes assigning IP addresses, setting up routes, and managing DNS settings.
- **Pluggable Architecture**: Kubernetes does not provide a default network implementation but relies on third-party CNI plugins to implement the networking layer. Examples include Flannel, Calico, Weave Net, and Cilium.
- **Network Isolation**: Beyond basic connectivity, CNI plugins can enforce network policies, providing isolation between Pods across different namespaces or within the same namespace, enhancing security within a Kubernetes cluster.

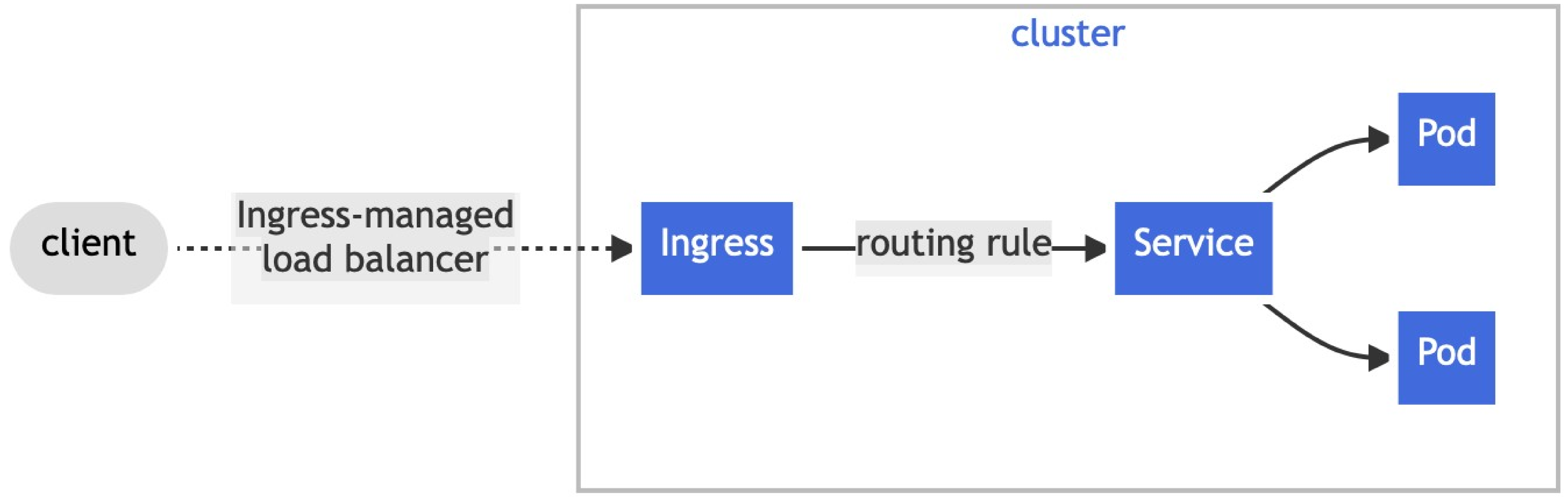


https://docs.cilium.io/en/stable/installation/cni-chaining-aws-cni/

Networking

Ingress - Expose your Service to Clients

- Ingress in Kubernetes helps manage access from the outside world to your services inside the Kubernetes cluster.
- It makes sure that incoming internet traffic gets to the right place safely and efficiently.
- NGINX can serve as an Ingress controller in a Kubernetes environment
- NGINX's ingress controller can handle handle SSL termination, load balancing, and authentication



https://www.nginx.com/blog/announcing-nginx-ingress-controller-for-kubernetes-release-1-7-0/

Resources

Resource Types

Pods, Deployments and StatefulSets

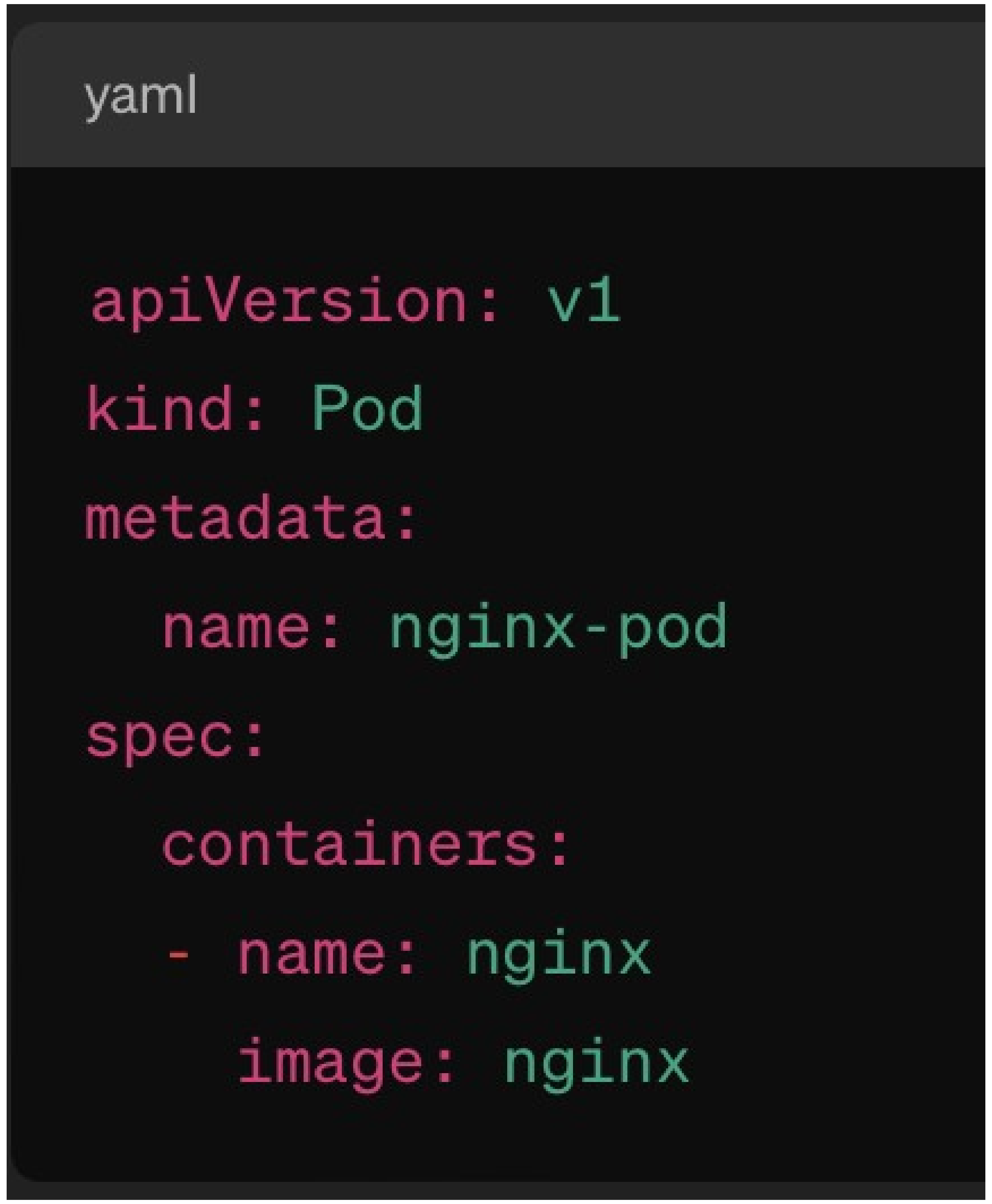
- **Pods**: The smallest, most basic deployable objects in Kubernetes that represent a single instance of a running process in your cluster. Pods contain one or more containers, such as Docker containers.
- **Deployments**: Higher-level management entities that describe the desired state of Pods. Deployments control the scaling and lifecycle of a set of Pods, ensuring that the specified number of Pods are running and updating them as needed.
- **StatefulSets:** Designed for stateful applications that require persistent storage, stable, unique network identifiers, and orderly deployment and scaling. StatefulSets are ideal for applications like databases that need to maintain state across restarts and scaling operations.

Resource Manifest

A YAML file that declares a resources desired state

- It specifies various settings and configurations such as the containers the Pod should run, their images, resource limits, environmental variables, volumes, and more.

- The Kubernetes API server uses this manifest to create, update, and manage the Pod according to the specifications outlined in the file.

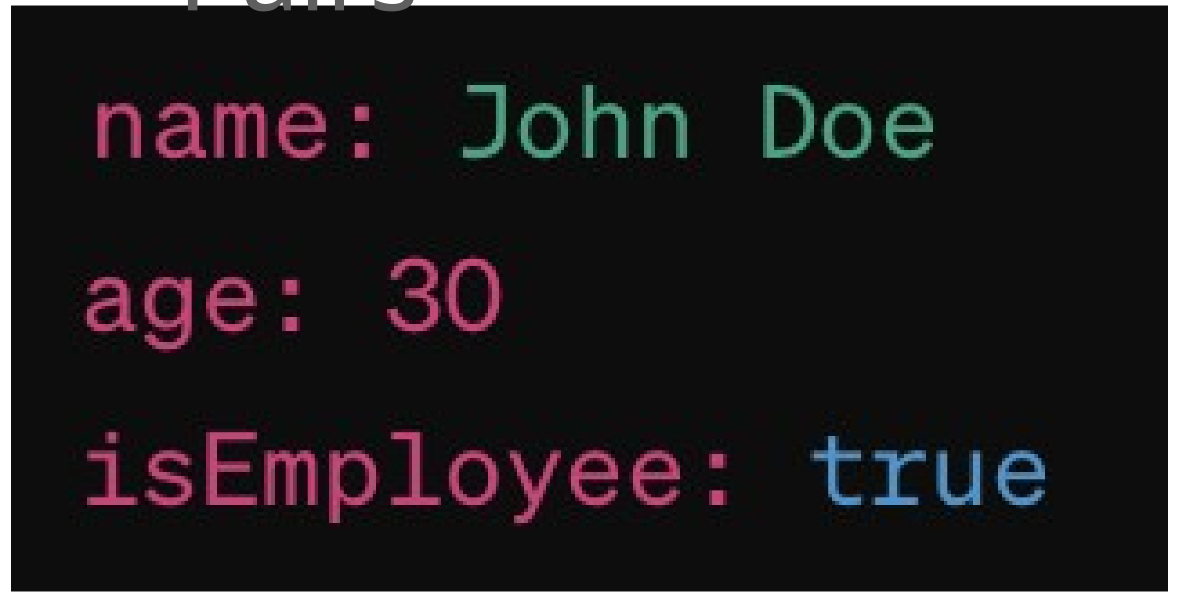


```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx
```

YAML Basics

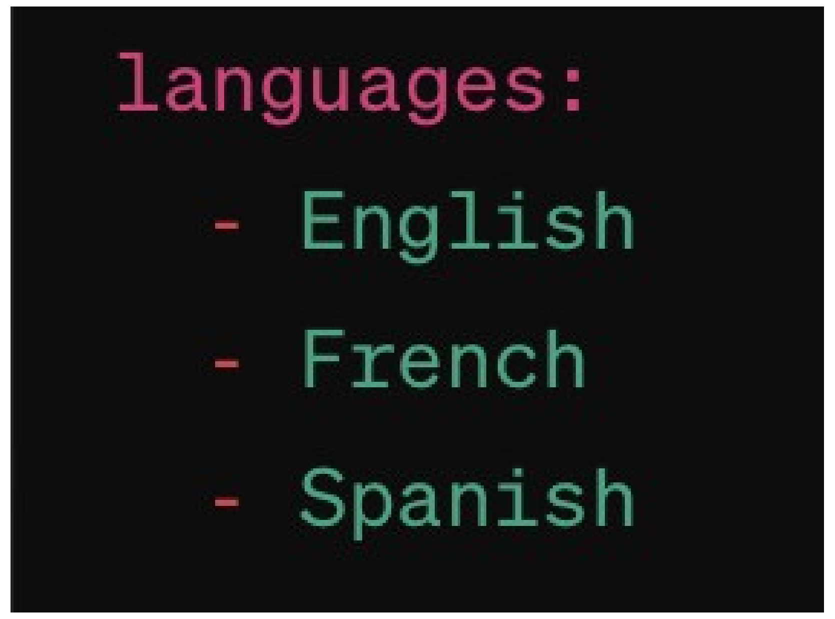
Example Data Structures

Key / Value Pairs



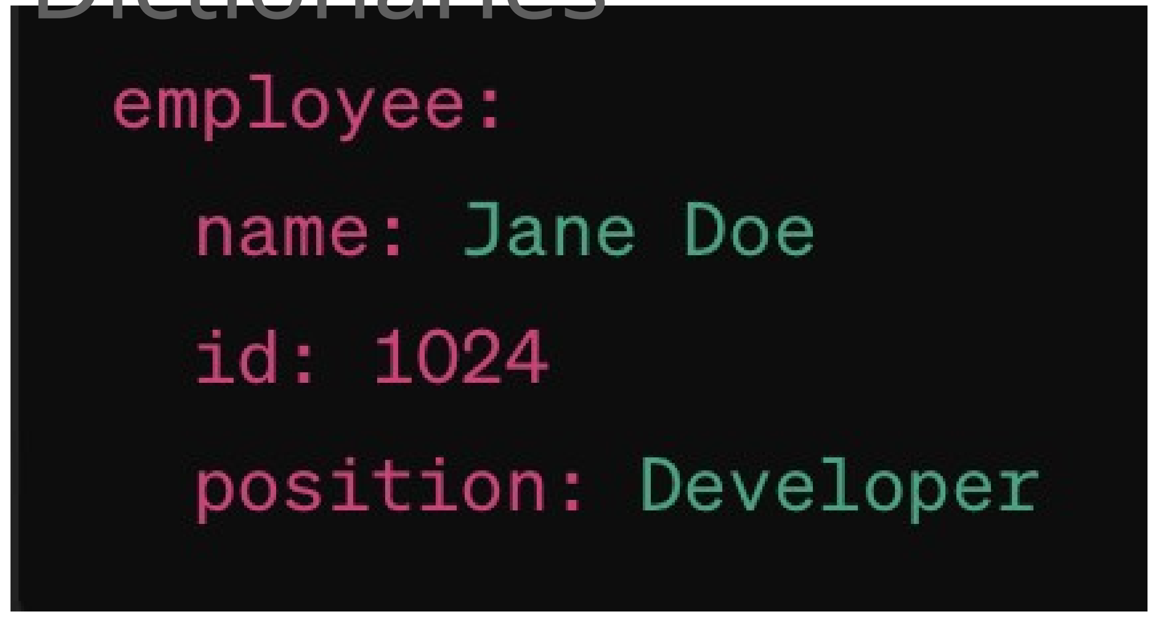
```yaml
name: John Doe
age: 30
isEmployee: true
```

Lists



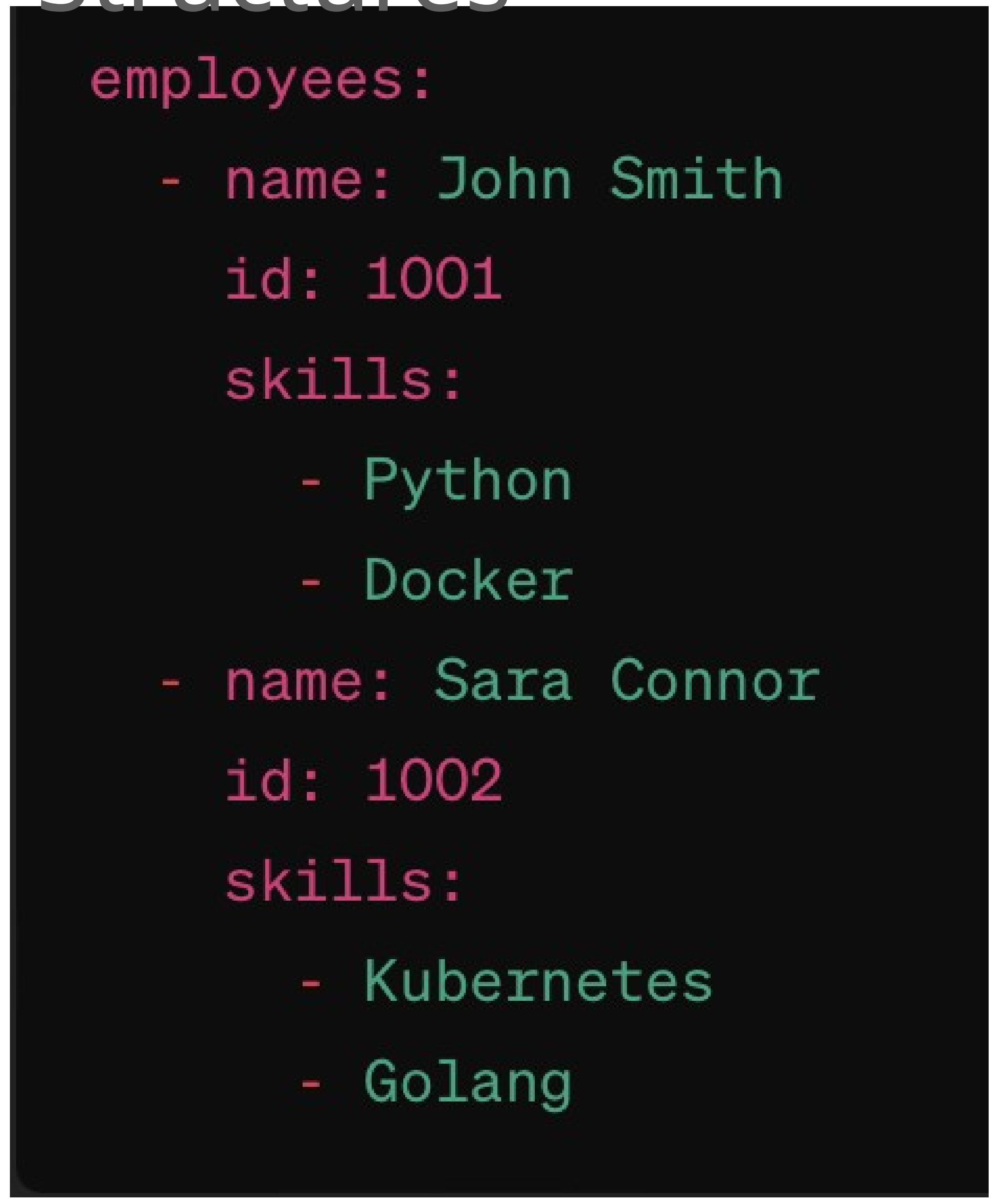
```yaml
languages:
  - English
  - French
  - Spanish
```

Maps / Dictionaries



```yaml
employee:
  name: Jane Doe
  id: 1024
  position: Developer
```

Nested Structures



```yaml
employees:
  - name: John Smith
    id: 1001
    skills:
      - Python
      - Docker
  - name: Sara Connor
    id: 1002
    skills:
      - Kubernetes
      - Golang
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    app: example
spec:
  containers:
  - name: example-container
    image: example/image
    ports:
    - containerPort: 80
    env:
    - name: EXAMPLE_ENV
      value: "example-value"
    resources:
      limits:
        cpu: "1"
        memory: "500Mi"
      requests:
        cpu: "0.5"
        memory: "200Mi"
    volumeMounts:
    - mountPath: /path/in/container
      name: example-volume
  volumes:
  - name: example-volume
    persistentVolumeClaim:
      claimName: example-pvc
```

Pods

A single instance of a running process

- **apiVersion**: Specifies the version of the Kubernetes API you're using to create the object. For Pods, it's typically v1.
- **kind**: The type of Kubernetes object you're defining. For a Pod manifest, this is Pod.
- **metadata**: Contains data that helps uniquely identify the Pod, including its name and labels.
- **spec**: The specification section where you define the containers the Pod will run, their images, ports, environment variables, and more.
- **containers**: A list of containers to run in the Pod. Each container's name, image, and other settings like ports, env (environment variables), resources (CPU and memory limits and requests), and volumeMounts are specified here.
- **volumes**: Defines the storage volumes that the Pod can use. These volumes can then be mounted into containers.

Pods

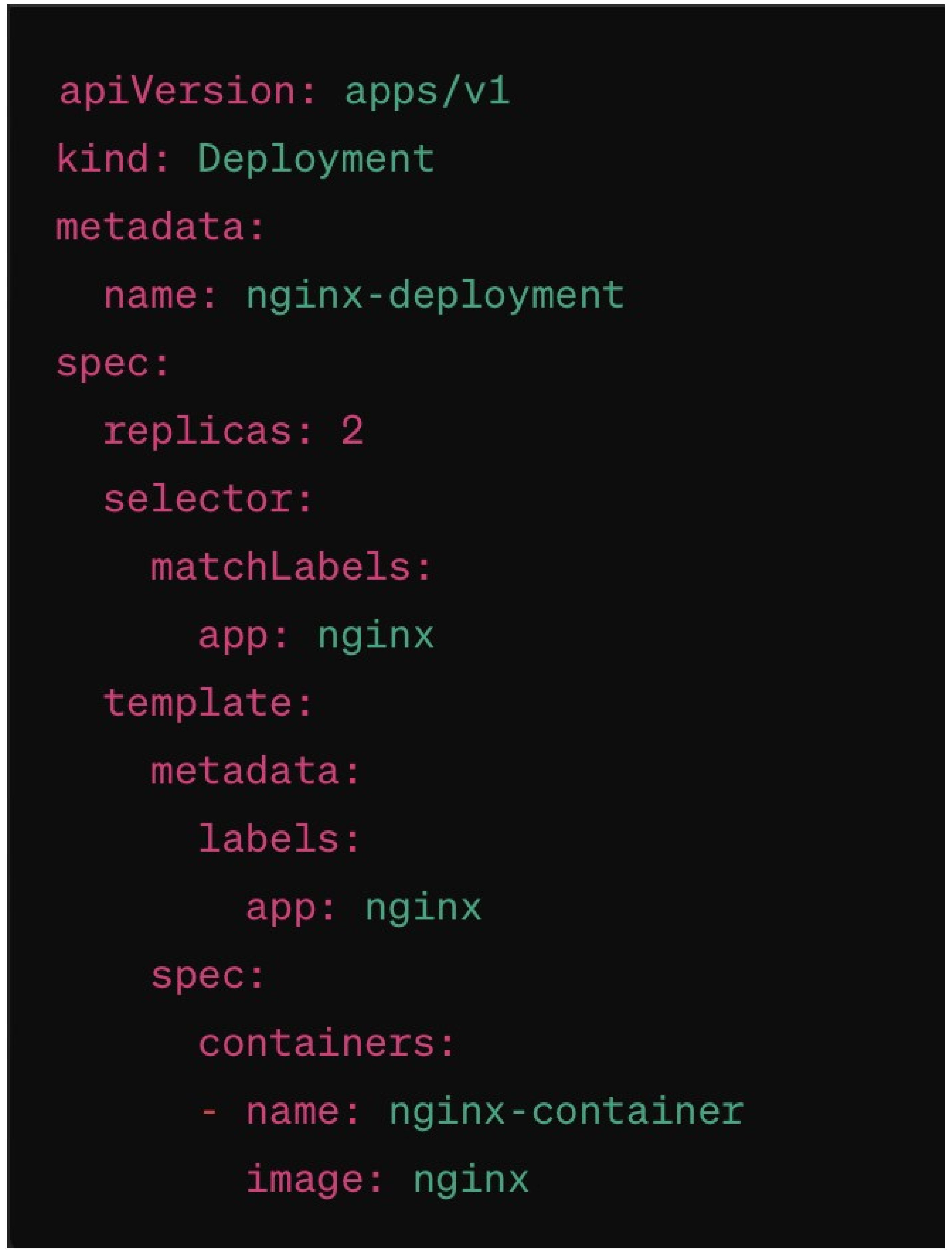
Best Practices

- **One Application per Pod**: As a general rule, it's recommended to run one application per Pod. This simplifies management, scaling, and deployment
- Reserve multi-container Pods for scenarios where the containers are tightly coupled and need to be managed as a single unit
- You may want to deploy multiple applications in a pod in cases where you need a logging agent that shares storage volumes with an application. This is called a “**sidecar**” pattern.

Deployments

Control the scaling and lifecycle of a set of Pods

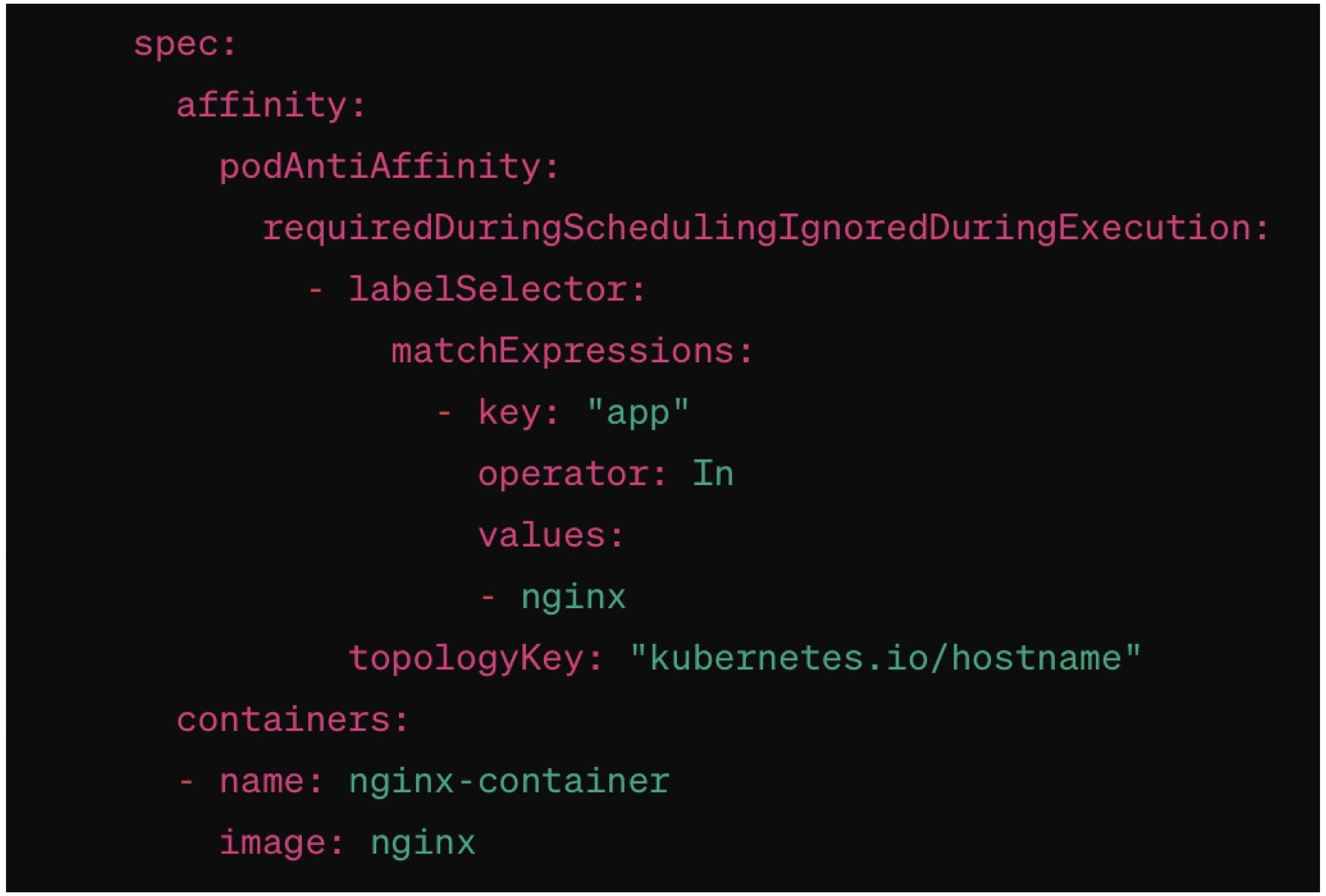
- **Replicas**: Specifies that 2 instances of the nginx container should be running.
- **Selector and Labels**: Uses app: nginx to match the deployment with the pods it manages.
- **Container Image**: The nginx container will use the nginx image from a container registry.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx-container
        image: nginx
```

Deployments

Control the scaling and lifecycle of a set of Pods



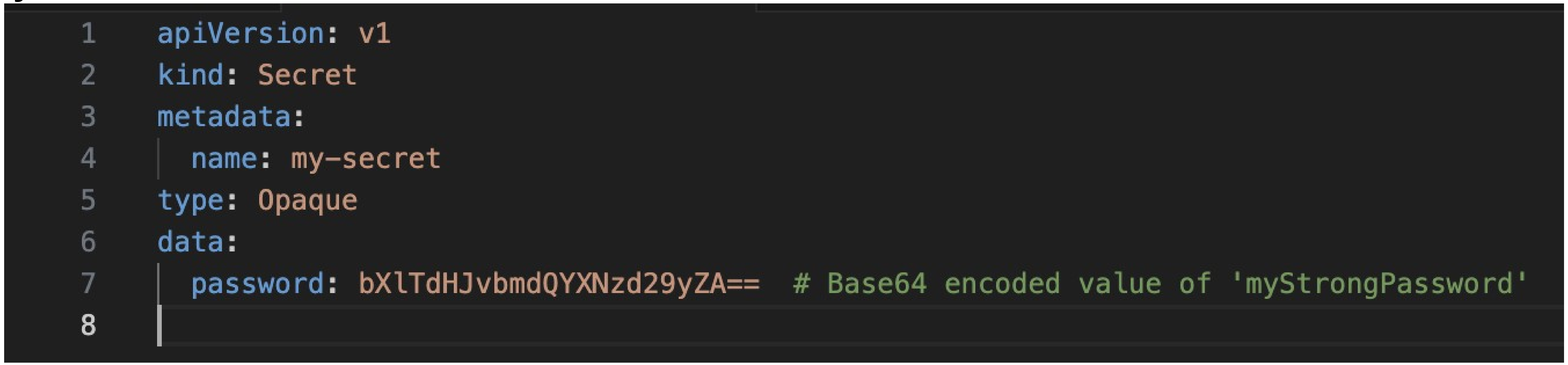
```yaml
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: "app"
                operator: In
                values:
                - nginx
          topologyKey: "kubernetes.io/hostname"
  containers:
  - name: nginx-container
    image: nginx
```

- **Control Pod Placement**: Deployments allow you to manage pod affinity or anti-affinity, which means you can influence how pods are distributed across nodes in your cluster.
- **Enforce Anti-Affinity**: In the example provided, we've used pod anti-affinity to ensure that the nginx pods do not share the same host. This is achieved by setting rules that prevent scheduling them together.

Resource: Secret

Securely Managing Sensitive Data

- A Secret is used to store sensitive information, such as passwords, OAuth tokens, and ssh keys, in key-value pairs.
- Secrets can be mounted as volumes inside pods or their key-value pairs can be used as environment variables
- The data in a secret is required to be base64 encoded ensuring that the secret's data can be safely represented as a string. This is important for binary data or data that may contain special characters that could be misinterpreted by systems or tools processing the YAML or JSON files.

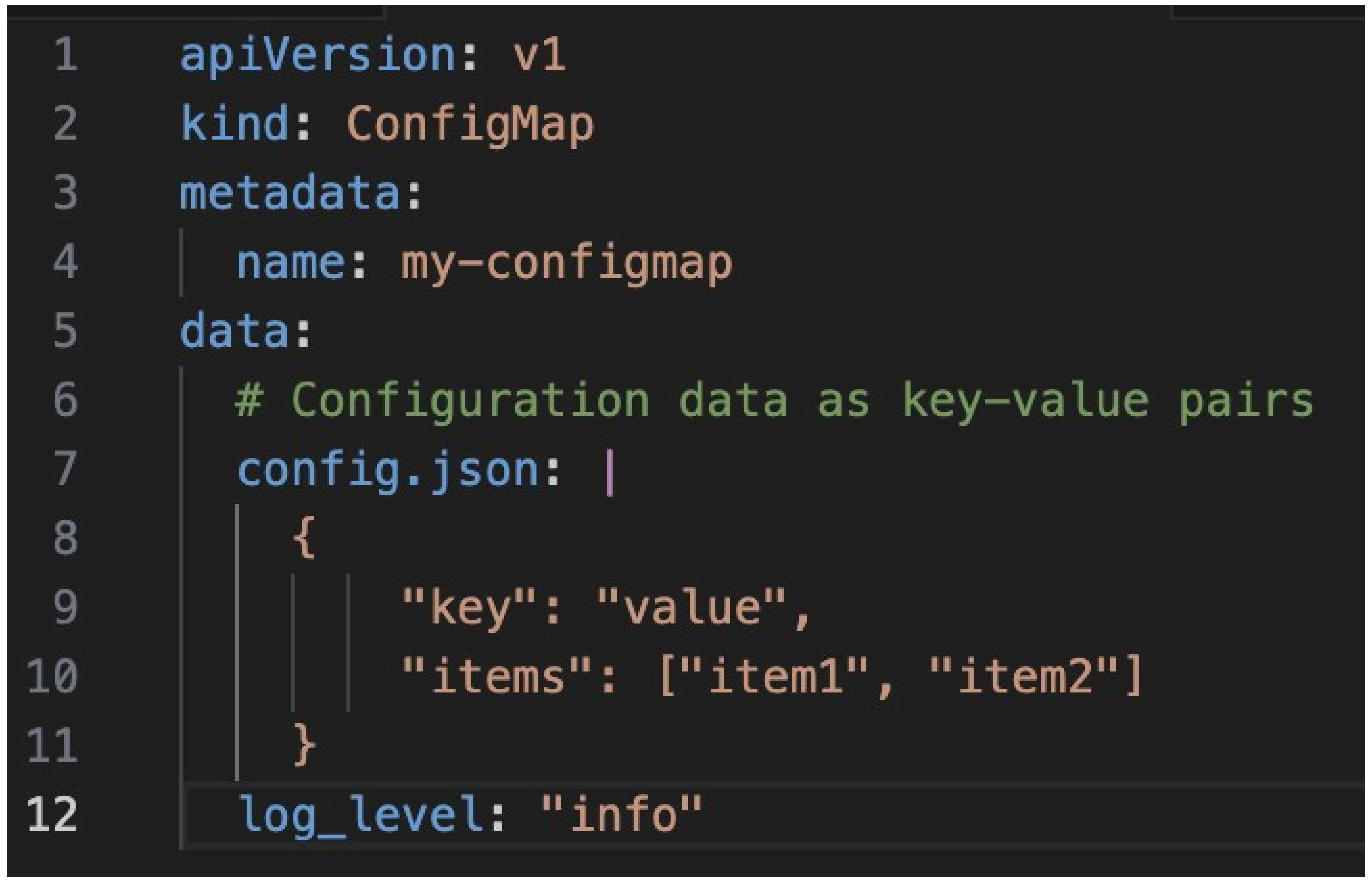


```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  password: bXlTdHJvbmdQYXNzd29yZA==  # Base64 encoded value of 'myStrongPassword'

```

Resource: ConfigMap

Centralizing Configuration Management

- A ConfigMap is used to store non-confidential data in key-value pairs.
- ConfigMaps can be mounted as volumes inside pods or their key-value pairs can be used as environment variables



```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  # Configuration data as key-value pairs
  config.json: |
    {
      "key": "value",
      "items": ["item1", "item2"]
    }
  log_level: "info"
```

Namespaces

Organizing Cluster Resources Efficiently

- **Logical Partitioning**: Namespaces in Kubernetes provide a way to divide cluster resources between multiple users through logical partitioning.
- **Organizational Tool**: They serve as an organizational tool to group, isolate, and manage objects like Pods, Deployments, and Services within a cluster.
- **Resource Management**: Namespaces help in managing resource quotas, limiting the amount of resources (CPU, memory) that a group or project can use, thereby preventing one team or project from monopolizing cluster resources.
- **Access Control**: They facilitate fine-grained access control by allowing administrators to set permissions on a per-namespace basis, enhancing security.

Using Kubernetes

kubectl

Command-Line Interface (CLI) for Kubernetes

- **Cluster Management**: Enables users to manage various aspects of Kubernetes clusters, including launching applications, scaling deployments, and rolling out updates to applications or configurations.
- **Resource Inspection**: Offers commands to view the state of cluster resources, such as Pods, Deployments, Services, and more, helping users troubleshoot issues and understand cluster activity.

kubectl get nodes

Show the clusters nodes

- The "kubectl get nodes" command lists all the nodes in a Kubernetes cluster.
- It shows the status, roles, ages, and version for each node in a concise table format.
- This command helps administrators monitor the health and capacity of the cluster.

kubectl apply -f <manifest yaml file>

Deploy a Kubernetes Resource

- Used to create or update resources in a Kubernetes cluster based on the definitions provided in a YAML or JSON manifest file.
- This command is idempotent, meaning it can be run multiple times without changing the result beyond the initial application. It ensures that the actual state of the cluster matches the desired state specified in the manifest file.
- This command is declarative, meaning you declare the desired state of your resources, and Kubernetes works to maintain that state.
- Aside: many available tools for customising / preprocessing manifests
    - e.g. kustomize, helm, jsonnet (tanka / kubecfg), cuelang, …

kubectl get pods

Show information about pods

- The "kubectl get pods" command is utilized to list all the pods in a Kubernetes cluster or within a specific namespace.
- This command provides a snapshot of all current pods, showing their names, status, number of restarts, and age.
- By default, it lists pods in the default namespace unless another namespace is specified. The command can also be customized with various flags to filter, sort, or format the output, making it a versatile tool for managing and inspecting pods.

kubectl logs <pod name>

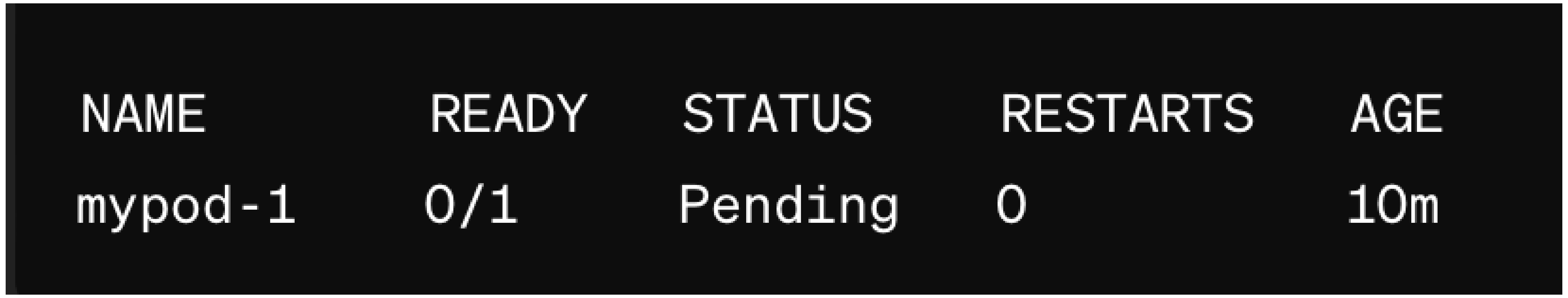
Show logs from a container

- The “kubectl logs” command is used to retrieve logs from containers running inside pods in a Kubernetes cluster.
- This command will display the stdout and stderr streams of the container running in the specified pod.

Troubleshooting

Identifying Kubernetes Scheduling Issues

- When a pod isn't being scheduled in Kubernetes, the output from **kubectl get pods** will typically show the status of the pod as **Pending**
- This indicates that the pod has been accepted by the Kubernetes system but hasn't been assigned to a node for execution.
- Example of what the output might look like:



```
NAME        READY   STATUS    RESTARTS   AGE
mypod-1     0/1     Pending   0          10m
```

Troubleshooting

Resolving Kubernetes Scheduling Issues

- To get more detailed information about why the pod isn't being scheduled, you can use the **kubectl describe pod <pod-name>** command.
- This will provide events and messages from the scheduler that can help identify issues such as insufficient resources that are preventing the pod from being scheduled.

Questions / Comments