{desafío} latam_

CURSO

DEVOPS SENIOR

Objetivo General del Curso

DISEÑAR ENTORNOS CLOUD NATIVE INTEGRANDO PRÁCTICAS DE KUBERNETES Y GITOPS, DE ACUERDO CON ESTÁNDARES DE SEGURIDAD Y OBSERVABILIDAD.

Objetivo específico del Módulo

CONOCER CARACTERISTICAS DE LOS KUBERNETES EN PRODUCCIÓN, SEGUN LAS PRACTICAS AVANZADAS DE GITOPS, DEVSECOPS, KUBERNETES, OBSERVABILIDAD, IAC, FINOPS Y AIOPS.

Contenidos

Objetivo General del Curso....2
Objetivo específico del Módulo....2
Módulo 5: KUBERNETES AVANZADO.....5
Capítulo 1: Kubernetes avanzado.....6
Administración de clústeres avanzados....6
Recursos avanzados y patrones....7
Seguridad avanzada....8
Rendimiento y escalabilidad....8
Observabilidad y gestión de eventos....9
Operators y CRDs (Custom Resource Definitions)....9
Pruebas, staging y GitOps....9
Buenas prácticas del DevOps senior....10
Capítulo 2: Helm.....11
¿Qué es un Helm Chart?....11
Funcionalidades clave de Helm....12
Casos de uso avanzados....13
Seguridad y buenas prácticas....14
Rol del DevOps Senior....14
Capítulo 3: Operators.....15
¿Qué es un Operator?....15
Ciclo de reconciliación....17
Herramientas para desarrollo de Operators....17
Seguridad y gobernanza....17
Rol del DevOps Senior....17
Capítulo 4: Seguridad en K8s.....18
Principios de seguridad en Kubernetes....18
1. Control de acceso: RBAC....18
2. Seguridad en pods: Pod Security Standards (PSS)....19
3. Políticas de red: NetworkPolicies....19
4. Gestión de secretos....20
6. Auditoría y detección....21
7. Firmas y verificación de imágenes....21
Rol del DevOps Senior en seguridad K8s....21
Capítulo 5: Gestión de redes.....22
1. Modelo de red de Kubernetes....22
2. CNI (Container Network Interface)....22

3. Tipos de servicios en Kubernetes 23
4. CoreDNS y resolución de nombres 23
6. Ingress e Ingress Controller 25
7. Service Mesh (visión avanzada) 25
8. Herramientas de monitoreo y troubleshooting de red 26
9. Consideraciones para producción 26

Módulo 5: KUBERNETES AVANZADO.

Capítulo 1: Kubernetes avanzado.

Kubernetes (K8s) se ha consolidado como el orquestador estándar para contenedores en entornos empresariales. En su nivel avanzado, se requiere no solo conocer los recursos básicos (pods, deployments, services), sino también dominar topologías distribuidas, patrones de alta disponibilidad, gestión eficiente de recursos, políticas de red, seguridad RBAC, controladores personalizados y afinamiento del clúster.

Administración de clústeres avanzados

1. Topología de clúster

- Nodo maestro (control plane): kube-apiserver, scheduler, controller-manager, etcd.
- Nodos trabajadores: kubelet, kube-proxy, container runtime (CRI-O, containerd).
- Implementación HA con múltiples nodos de control (HA Control Plane).
- Balanceo de carga para acceso resiliente al kube-apiserver.

2. Namespaces y aislamiento

- Uso intensivo de namespaces para entornos multi-tenant o separación de entornos (dev, qa, prod).
- Aplicación de políticas de red para aislar namespaces mediante NetworkPolicies.

Recursos avanzados y patrones

1. StatefulSets vs Deployments

- StatefulSets para servicios con identidad persistente (bases de datos, sistemas distribuidos).
- Control de orden de inicio, volúmenes persistentes por pod.

2. DaemonSets

- Despliegue de pods únicos por nodo (ej. monitoreo, logging, agentes de red).

3. Jobs y CronJobs

- Ejecución de tareas puntuales o programadas, con manejo de backoff, retries y control de concurrencia.

4. Affinity/Anti-Affinity y Tolerations

- Optimización de la colocación de pods según topología de red, zonas de disponibilidad, uso de GPU, etc.

Seguridad avanzada

1. RBAC (Role-Based Access Control)

- Roles, ClusterRoles, RoleBindings, ClusterRoleBindings.
- Principio de menor privilegio aplicado a usuarios, servicios y controladores.

2. PodSecurity Standards (PSS)

- Políticas que reemplazan PodSecurityPolicies: privileged, baseline, restricted.
- Validación en tiempo de admisión para restringir contenedores privilegiados, acceso a hostPath, etc.

3. Network Policies

- Controlan tráfico entre pods y desde/hacia el exterior, aplicando criterios por labels, puertos y protocolos.

Rendimiento y escalabilidad

1. Horizontal Pod Autoscaler (HPA)

- Escalamiento dinámico en función de CPU, memoria o métricas personalizadas (via Prometheus Adapter).

2. Vertical Pod Autoscaler (VPA)

- Recomendación y ajuste automático de requests/limits según histórico de uso.

3. Cluster Autoscaler

- Aumenta o reduce la cantidad de nodos según la demanda de recursos del clúster (en entornos cloud).

Observabilidad y gestión de eventos

- Uso de kubectl get events, describe, y logs para trazabilidad.
- Integración con Prometheus + Grafana para métricas.
- Integración con Loki o EFK/ELK Stack para logs.
- Alertas via Alertmanager y eventos críticos del scheduler.

Operators y CRDs (Custom Resource Definitions)

- Extensión de Kubernetes mediante CRDs y controladores personalizados.
- Operators: encapsulan lógica operativa de ciclo de vida (ej. backup/restore, escalamiento, upgrades).

Ejemplo: PostgreSQL Operator, Kafka Operator.

Pruebas, staging y GitOps

- Uso de herramientas como ArgoCD o Flux para despliegues GitOps.
- Gestión de ambientes paralelos mediante Helm o Kustomize.
- Validación de configuraciones mediante kubeval, conftest, OPA/Gatekeeper.

Buenas prácticas del DevOps senior

- Diseñar clústeres escalables y seguros, considerando casos de falla y planes de recuperación.
- Automatizar los flujos de despliegue y rollback con herramientas declarativas.
- Monitorear y auditar continuamente el estado de recursos y políticas.
- Establecer gobernanza clara sobre namespaces, RBAC y escalamiento.

Capítulo 2: Helm.

Helm es el gestor de paquetes oficial para Kubernetes, diseñado para simplificar el despliegue de aplicaciones complejas mediante charts reutilizables. Su uso es esencial en entornos productivos donde se requiere consistencia, trazabilidad, versionado y personalización controlada de recursos en clústeres Kubernetes.

Helm permite parametrizar manifiestos YAML, definir arquitecturas completas y facilitar upgrades y rollbacks de forma declarativa, integrándose perfectamente con flujos CI/CD y GitOps.

¿Qué es un Helm Chart?

Un Chart es una colección estructurada de archivos que describen un conjunto de recursos Kubernetes, incluyendo:

- Templates YAML: manifiestos con variables (.Values) reemplazables.
- values.yaml: archivo base de configuración.
- Chart.yaml: metadatos del chart (versión, descripción, dependencia).
- charts/: subcharts (dependencias).
- templates/: definición de recursos (Deployment, Service, Ingress, etc.).
- helpers.tpl: funciones y fragmentos reutilizables (e.g., nombres, etiquetas).

Funcionalidades clave de Helm

1. Instalación parametrizada

- helm install mi-release ./mi-chart --values prod-values.yaml
- Permite instalar la misma aplicación en diferentes entornos modificando solo los parámetros.
- Se pueden pasar valores individuales vía CLI (--set).

2. Actualización controlada

- helm upgrade mi-release ./mi-chart
- Permite aplicar cambios conservando el estado previo.
- Helm detecta diferencias entre versiones e impacta solo lo necesario.

3. Rollback automático

- helm rollback mi-release 2
- Si una actualización falla, se puede volver a una versión funcional previa.
- Ideal para entornos de producción.

4. Repositorios remotos

- Helm soporta repositorios de charts como Bitnami, Artifact Hub, JFrog, GitHub Pages.
- Se pueden subir y versionar charts privados para equipos internos.

Casos de uso avanzados

1. Despliegue de aplicaciones complejas

- Charts que despliegan toda una solución distribuida: PostgreSQL + Redis + Backend + Frontend.

Ejemplo: uso de subcharts para manejar dependencias desacopladas.

2. Personalización por entorno

- Definición de archivos values-dev.yaml, values-prod.yaml, values-qa.yaml.
- Permite cambiar réplicas, configuraciones de recursos, secretos y reglas de red sin alterar los templates.

3. Integración en pipelines

- **helm lint:** validación sintáctica.
- **helm template:** renderización local sin aplicar.
- **helm diff:** comparación entre releases.
- Paso integrado en GitHub Actions, GitLab CI, Jenkins, ArgoCD, etc.

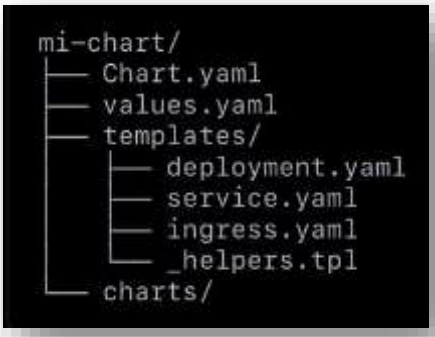
4. Uso con GitOps

- Helm Charts versionados en Git.
- Control de despliegue con ArgoCD o Flux, basados en versiones de Chart y values.yaml.
- Validación previa con herramientas como helm-unittest, kubeval, conftest.

Seguridad y buenas prácticas

- Evitar codificar secretos en values.yaml; usar herramientas como Sealed Secrets o External Secrets.
- Validar charts con políticas de admisión (OPA/Gatekeeper).
- Limitar los privilegios del ServiceAccount asociado a Helm.
- Firmar charts (helm package --sign) y verificar su integridad.

Ejemplo de estructura de un chart



Rol del DevOps Senior

El DevOps senior debe:

- Diseñar Helm Charts modulares, reutilizables y seguros.
- Mantener repositorios internos de charts auditados.
- Automatizar despliegues con validaciones previas en pipelines.
- Integrar Helm con GitOps y sistemas de observabilidad.
- Establecer una gobernanza de versiones y rollback claros para todos los entornos.

Capítulo 3: Operators.

En entornos empresariales donde las cargas de trabajo son complejas y altamente personalizadas, las herramientas tradicionales de gestión de Kubernetes (como Deployments, StatefulSets o CronJobs) pueden ser insuficientes para manejar el ciclo de vida completo de aplicaciones o servicios.

Los Kubernetes Operators emergen como una solución para encapsular conocimiento operativo experto dentro de controladores personalizados que gestionan automáticamente tareas como instalación, configuración, monitoreo, respaldo, recuperación ante fallas y escalamiento.

¿Qué es un Operator?

Un Operator es una extensión del controlador de Kubernetes que emplea Custom Resource Definitions (CRDs) y lógica de control personalizada para manejar aplicaciones complejas como si fueran recursos nativos del clúster.

Componentes principales:

- **CRD (Custom Resource Definition):** define un nuevo tipo de recurso (por ejemplo: PostgreSQLCluster, KafkaTopic, RedisFailover).
- **Controller personalizado:** proceso que observa cambios en los CRD y aplica lógica automatizada.
- **Reconciliación:** ciclo de observación → comparación → acción, basado en el estado deseado vs estado actual.

Ventajas clave de usar Operators

- Automatización total del ciclo de vida de aplicaciones: provisión, backup, failover, actualizaciones.
- Permiten construir una plataforma auto-gestionada.
- Consistencia operativa en entornos multi-tenant o multiclúster.
- Integración con herramientas de monitoreo y políticas de seguridad.

Ejemplos de Operators populares

OPERATOR	FUNCIÓN PRINCIPAL
PostgreSQL Operator (Zalando, Crunchy)	Provisionamiento, escalamiento, backups automáticos.
Prometheus Operator	Despliegue y gestión automática de Prometheus y Alertmanager.
Kafka Operator	Gestión de brokers, topics, ACLs y configuraciones.
ElasticSearch Operator	Manejo de clústeres de Elastic: Replícas, shards, upgrades.
Vault Operator	Automatización de despliegue y rotación de secretos.

Ejemplo de CRD y Custom Resource

CRD (resumen simplificado):

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: databases.mycompany.com
spec:
  group: mycompany.com
  versions:
    - name: v1
      served: true
      storage: true
  scope: Namespaced
  names:
    plural: databases
    singular: database
    kind: Database
```

Instancia de recurso personalizado:

```yaml
apiVersion: mycompany.com/v1
kind: Database
metadata:
  name: postgres-cluster
spec:
  replicas: 3
  storageSize: 20Gi
  backup: enabled
```

Ciclo de reconciliación

- Usuario aplica un recurso Database.
- El Operator detecta el cambio.
- Compara el estado actual vs el deseado.
- **Realiza acciones:** crea StatefulSet, PVCs, configuraciones, monitoreo.
- Registra estado y eventos en Kubernetes.

Herramientas para desarrollo de Operators

- **Operator SDK (Go):** framework oficial CNCF para crear Operators en Go, Ansible o Helm.
- **Kubebuilder:** scaffolding para proyectos en Go, ideal para integraciones profundas con Kubernetes.
- **Metacontroller:** controlador genérico para crear Operators sin SDK completo.
- **OperatorHub.io:** catálogo comunitario de Operators validados.

Seguridad y gobernanza

- Aplicar RBAC específico por Operator.
- Definir validaciones de esquemas (OpenAPIV3Schema) en los CRDs.
- Monitorear logs del Operator (kubectl logs <pod>) para trazabilidad.
- Auditar eventos de reconciliación (kubectl get events).

Rol del DevOps Senior

- Evaluar cuándo usar un Operator en lugar de un Helm Chart o YAML estándar.
- Diseñar estrategias de versionamiento y mantenimiento para los CRD.
- Gestionar operadores como parte de la plataforma como producto.
- Integrar lógica de negocio o monitoreo personalizado en los controladores.

Capítulo 4: Seguridad en K8s.

La seguridad en Kubernetes es un aspecto esencial en ambientes productivos debido a la naturaleza altamente distribuida de sus componentes. La superficie de ataque incluye el plano de control, los nodos, los pods, los secretos, las redes internas y los volúmenes montados. Un enfoque de seguridad DevSecOps implica establecer controles desde la definición del manifiesto hasta la ejecución del pod, garantizando aislamiento, control de acceso, auditoría, cifrado y cumplimiento normativo.

Principios de seguridad en Kubernetes

- **Defensa en profundidad:** múltiples capas de seguridad aplicadas a distintos niveles (API, contenedores, red, volúmenes).
- **Principio de menor privilegio:** limitar al máximo los permisos de usuarios, procesos y pods.
- **Zero Trust:** validar constantemente identidad y contexto, incluso dentro del clúster.
- **Automatización del cumplimiento:** validar políticas de seguridad como parte del pipeline.

1. Control de acceso: RBAC

- RBAC (Role-Based Access Control) permite definir qué acciones puede realizar un usuario o servicio sobre recursos específicos.
- **Roles / ClusterRoles:** definen permisos a nivel de namespace o del clúster completo.
- **RoleBinding / ClusterRoleBinding:** asocian roles a sujetos (usuarios, grupos, cuentas de servicio).

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: desarrollo
  name: lector-pods
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list"]
```

2. Seguridad en pods: Pod Security Standards (PSS)

Desde Kubernetes 1.25, las PodSecurityPolicies (PSP) están deprecadas. Se reemplazan por Pod Security Standards, con tres niveles:

- **privileged:** sin restricciones (para debugging o herramientas avanzadas).
- **baseline:** configuración mínima segura.
- **restricted:** mayor restricción (sin root, sin hostPath, sin privilegios).

Aplicadas mediante etiquetas en namespaces:

kubectl label ns dev pod-security.kubernetes.io/enforce=restricted

3. Políticas de red: NetworkPolicies

Las NetworkPolicies permiten definir qué pods pueden comunicarse entre sí y con el exterior. Se basa en selectors por labels y criterios como puertos y protocolos.

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

Esta política deniega todo ingreso externo al namespace, salvo que se especifique lo contrario.

4. Gestión de secretos

Opciones de manejo:

- **Secrets nativos (base64):** fáciles de usar pero no cifrados por defecto.
- **Sealed Secrets (Bitnami):** cifrado asimétrico en Git; solo el controller puede desencriptar.
- **Vault by HashiCorp:** solución robusta para rotación, expiración y políticas avanzadas.
- **External Secrets Operator:** sincroniza secretos desde proveedores externos (AWS Secrets Manager, Azure Key Vault, GCP Secret Manager).

Recomendaciones:

- Evitar montarlos como variables de entorno.
- Utilizar volumeMount con permisos restringidos (readOnly, defaultMode: 0400).
- Cifrar secretos en etcd (encryptionConfig activado en el API server).

Validación y políticas de admisión

Open Policy Agent (OPA) / Gatekeeper:

Permite validar políticas como: evitar contenedores privilegiados, limitar requests de CPU, restringir imagenes externas, etc.

Ejemplo:

```
package kubernetes.admission

violation[{"msg": msg}] {
  input.review.object.spec.containers[_].securi
  msg := "No se permiten contenedores privilegi
}
```

6. Auditoría y detección

- **Audit logs del API Server:** configurable para eventos de lectura, escritura y errores.
- **Falco:** herramienta para detección de comportamiento sospechoso en tiempo real a nivel de sistema.
- **Kube-bench:** validación automática del cumplimiento del benchmark de seguridad CIS para Kubernetes.
- **Kube-hunter:** escáner de vulnerabilidades de clústeres expuestos o mal configurados.

7. Firmas y verificación de imágenes

- Usar herramientas como Cosign o Notary v2 para firmar imágenes.
- Escanear imágenes con Trivy, Snyk o Anchore.
- Rechazar imágenes no firmadas en producción con Admission Controllers.

Rol del DevOps Senior en seguridad K8s

- Definir estándares de seguridad para los equipos de desarrollo.
- Integrar validaciones en pipelines CI/CD y flujos GitOps.
- Auditar configuraciones periódicamente y evaluar nuevas amenazas.
- Orquestar una política de control de acceso granular por equipo, entorno y aplicación.
- Automatizar rotación y cifrado de secretos.
- Monitorear continuamente el cumplimiento normativo (ISO, NIST, CIS).

Capítulo 5: Gestión de redes.

La red es un componente fundamental para el funcionamiento de cualquier clúster Kubernetes. No solo permite la comunicación entre pods, nodos y servicios, sino que también define el modelo de seguridad, enrutamiento y escalabilidad de las aplicaciones. La gestión de redes en Kubernetes implica comprender el modelo de red básico del clúster, así como la integración de plugins CNI (Container Network Interface), políticas de red, servicios, DNS, y otros componentes avanzados como Ingress Controllers y Service Mesh.

1. Modelo de red de Kubernetes

Kubernetes asume un modelo de red plano, donde:

- Cada pod tiene su propia dirección IP única.
- Los pods pueden comunicarse entre sí sin NAT, independientemente del nodo en el que estén.
- Los nodos del clúster pueden comunicarse con todos los pods.
- La resolución de nombres se realiza mediante CoreDNS.

Este modelo requiere una solución de red compatible con CNI que cumpla estas condiciones. Algunos ejemplos: Calico, Flannel, Cilium, Weave Net.

2. CNI (Container Network Interface)

- El CNI es una especificación estándar utilizada por Kubernetes para conectar contenedores a la red.
- Kubernetes no implementa la red directamente: depende de un plugin CNI.
- CNI se encarga de asignar IPs, crear puentes virtuales, enrutar paquetes y limpiar redes al eliminar pods.

Plugins CNI comunes:

- **Calico:** Foco en políticas de red y seguridad (usa BGP, IPIP, eBPF)
- **Flannel:** Sencillo, solo enruta paquetes IP
- **Cilium:** Usa eBPF para mejorar rendimiento y seguridad
- **Weave Net:** Usa malla peer-to-peer

3. Tipos de servicios en Kubernetes

Kubernetes proporciona distintos tipos de Service que actúan como puntos de entrada para acceder a uno o varios pods:

- **ClusterIP (por defecto):** acceso interno dentro del clúster
- **NodePort:** abre un puerto en cada nodo para acceder al servicio desde fuera
- **LoadBalancer:** expone el servicio a través de un balanceador externo (cloud providers)
- **ExternalName:** redirige el tráfico hacia un DNS externo

Además, Headless Services (clusterIP: None) permiten que el cliente resuelva directamente la IP del pod sin pasar por kube-proxy.

4. CoreDNS y resolución de nombres

- CoreDNS es el servicio de DNS interno que traduce nombres como svc-name.namespace.svc.cluster.local a direcciones IP.
- Cada pod tiene configurado el DNS como /etc/resolv.conf, apuntando al servicio CoreDNS del clúster.
- Permite descubrimiento de servicios dentro del clúster sin hardcodear direcciones IP.

5. Network Policies (seguridad de red)

Las Network Policies definen reglas de control de tráfico entre pods (nivel L3/L4). Por defecto, todo el tráfico está permitido, por lo que debes crear políticas explícitas si deseas segmentar:

- Se definen por pod selector, namespace selector, puertos y protocolos.
- Requieren soporte del plugin CNI (Calico y Cilium son compatibles).
- Permiten implementar microsegmentación: “el pod A solo puede comunicarse con pod B en el puerto 443”.

Ejemplo de política que permite solo tráfico entrante desde un namespace específico:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-frontend
  namespace: backend
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: frontend
```

6. Ingress e Ingress Controller

- El recurso Ingress gestiona el acceso HTTP/HTTPS externo hacia los servicios internos:
- Define reglas de enrutamiento basadas en host, path, headers.
- Se requiere un Ingress Controller (Nginx, Traefik, HAProxy, Istio).

Ejemplo de Ingress simple:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ejemplo-ingress
spec:
  rules:
  - host: app.miempresa.cl
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: mi-servicio
            port:
              number: 80
```

7. Service Mesh (visión avanzada)

Para entornos de alto rendimiento y trazabilidad, Kubernetes puede integrar un Service Mesh, como:

- Istio
- Linkerd
- Consul

Estos permiten:

- Encriptar comunicaciones mTLS entre pods.
- Control de tráfico avanzado (routing, retries, circuit breaking).
- Observabilidad (telemetría, métricas, logs, tracing).
- Desacoplar la lógica de red desde la aplicación.

Requiere inyectar sidecars (como Envoy) en los pods.

8. Herramientas de monitoreo y troubleshooting de red

- kubectl get endpoints, kubectl get svc, kubectl get netpol
- tcpdump, iftop, ss, ip netns, traceroute, ping dentro del pod
- calicoctl, cilium status, weave status según el CNI
- Plugins para visualización como Kiali, Lens, Octant

9. Consideraciones para producción

- Usar CNI que soporte políticas de red y eBPF (Calico o Cilium).
- Validar MTU de red en clusters con nodos heterogéneos.
- Habilitar mTLS en comunicaciones críticas.
- Usar Ingress Controllers con certificados válidos y TLS actualizado.
- Monitorear tráfico entre namespaces sensibles.
- Implementar auditorías de red y escaneos de puertos internos periódicos.