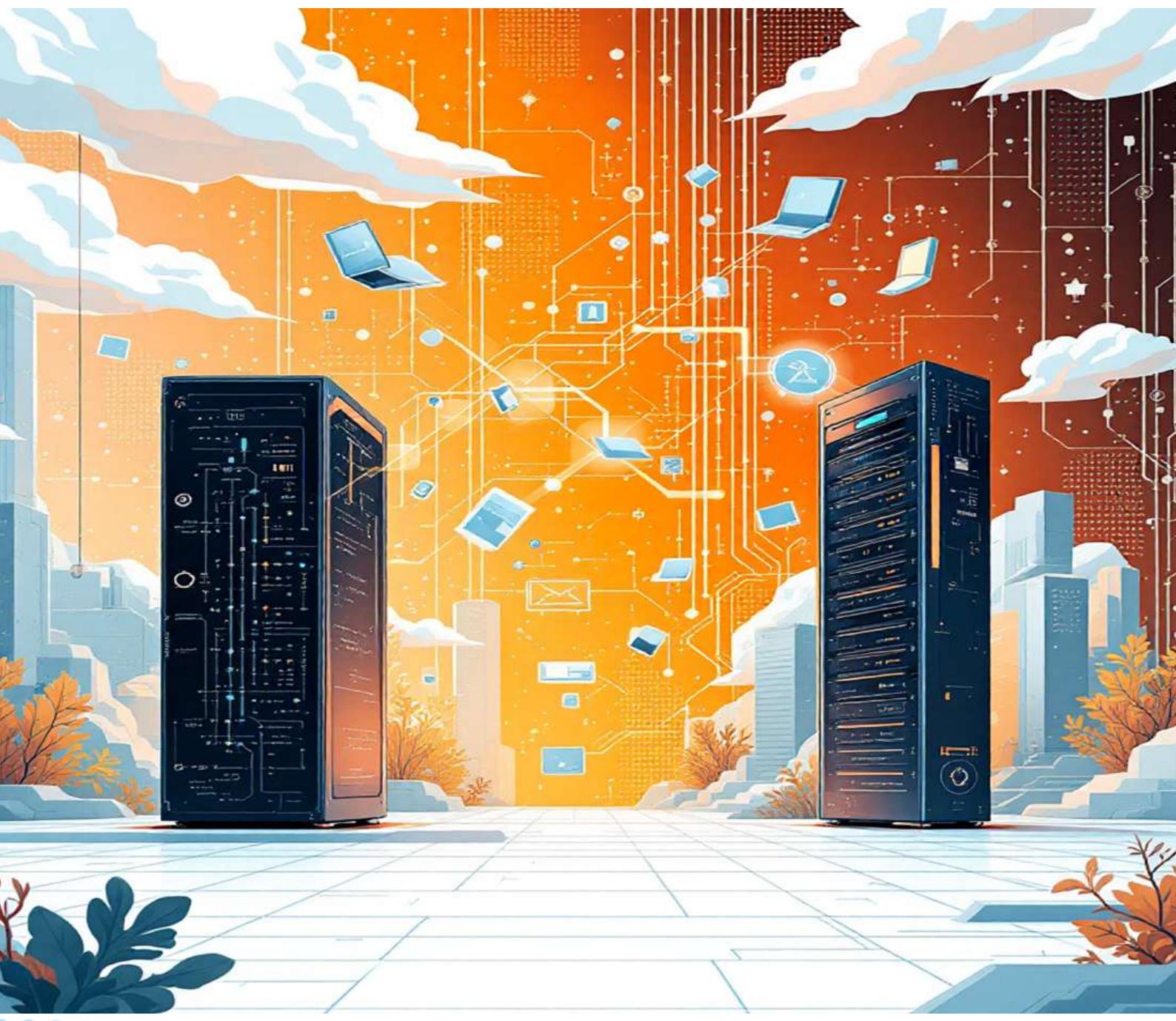


CURSO DEVOPS SENIOR



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

IDENTIFICAR CONCEPTOS DE DEVOPS ESTRATÉGICO Y GITOPS AVANZADO Y EVOLUCIÓN DEL ROL DEVOPS, 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 1: DEVOPS ESTRATÉTICO Y GITOPS.	4
Capítulo 1: DevOps moderno.....	5
Capítulo 2: GitOps patterns.....	7
Principales patrones GitOps	7
Consideraciones estratégicas.....	8
Valor para perfiles senior.....	8
Capítulo 3: ArgoCD avanzado.....	9
Observabilidad y auditoría	10
Consideraciones para perfiles senior	11
Capítulo 4: Gestión de secretos.	12
Tipos de secretos gestionados.....	12
Herramientas y enfoques avanzados de gestión de secretos	12
Buenas prácticas en entornos GitOps	13
Observabilidad y trazabilidad	14
Rol del perfil senior.....	14
Capítulo 4: DR (Disaster Recovery)	15
¿Qué es Disaster Recovery?	15
Objetivos clave de un plan de DR	15
Conceptos fundamentales	15
Tipos de desastres que aborda DR.....	16
Estrategias de Disaster Recovery	16
DR en entornos modernos (Cloud, Kubernetes, GitOps).....	16
Buenas prácticas de Disaster Recovery	17
Ejemplo de flujo DR en nube con GitOps.....	17
Herramientas útiles para DR	18

Módulo 1: DEVOPS ESTRATÉTICO Y GITOPS.



Capítulo 1: DevOps moderno.

En la actualidad, el enfoque DevOps ha dejado de ser una simple práctica técnica para convertirse en una estrategia organizacional transversal. El DevOps moderno no se limita a la automatización del ciclo de vida del software, sino que se integra con el negocio, los equipos de desarrollo, operaciones, seguridad y control financiero, generando una cultura de colaboración, trazabilidad y entrega continua de valor.

A diferencia de sus primeras implementaciones, centradas en herramientas específicas y scripts personalizados, el DevOps moderno se sustenta en principios de escalabilidad, orquestación automatizada y gobierno del cambio, alineado con marcos de referencia como CALMS (Culture, Automation, Lean, Measurement, Sharing) y prácticas como GitOps, FinOps y DevSecOps.

Las características clave del DevOps moderno incluyen:

- **Automatización integral:** desde el aprovisionamiento de infraestructura hasta la entrega continua (CI/CD), usando herramientas como Ansible, Terraform, Jenkins, GitHub Actions o GitLab CI.
- **Contenerización y orquestación avanzada:** uso extensivo de contenedores (Docker) y plataformas de orquestación como Kubernetes, con foco en resiliencia, autoescalado y eficiencia.
- **Observabilidad distribuida:** implementación de sistemas de monitoreo proactivos (Prometheus, Grafana, Jaeger) que permiten trazabilidad completa del sistema y métricas accionables.
- **Seguridad embebida (DevSecOps):** políticas de seguridad que se integran desde el inicio del ciclo de desarrollo (shift-left), incluyendo escaneos de vulnerabilidades, control de secretos, análisis SAST/DAST y cumplimiento normativo.
- **Infraestructura como Código (IaC):** gestión declarativa y versionada de los entornos mediante herramientas como Terraform, Pulumi o CloudFormation, lo que permite entornos reproducibles, auditables y coherentes.
- **Cultura de colaboración ágil y responsabilidad compartida:** énfasis en la integración efectiva entre áreas, eliminando silos y fomentando la toma de decisiones conjunta y ágil.

En el contexto senior, dominar DevOps moderno implica comprender la arquitectura completa de la plataforma, influir en las decisiones de gobernanza tecnológica y liderar procesos de transformación digital con foco en eficiencia operativa y reducción del time-to-market.

Además, el DevOps actual se apoya en el análisis continuo de métricas de valor y productividad como DORA (Deployment Frequency, Lead Time for Changes, Change Failure Rate, Time to Restore), permitiendo medir el impacto directo de las prácticas DevOps en el negocio.

Capítulo 2: GitOps patterns.

GitOps es una evolución natural de los principios de DevOps y la Infraestructura como Código (IaC). Su premisa fundamental es utilizar Git como única fuente de verdad para la gestión de infraestructura, configuraciones de aplicaciones y políticas de seguridad. A través de este enfoque declarativo, reproducible y auditável, GitOps permite automatizar la entrega y operación de sistemas complejos en entornos distribuidos.

En ambientes empresariales, el uso de GitOps Patterns permite estandarizar procesos de despliegue, segmentar responsabilidades y mejorar la trazabilidad. Estos patrones no solo garantizan consistencia en los entornos, sino que también facilitan la integración con flujos CI/CD, prácticas de compliance, auditoría continua y recuperación ante fallos.

Principales patrones GitOps

1. Repository-per-Environment (un repositorio por entorno)

- Se utiliza un repositorio distinto para cada entorno (dev, QA, stage, prod).
- **Ventajas:**
- Mayor aislamiento entre entornos.
- Mayor control sobre los accesos por entorno.
- **Riesgo:** duplicación de configuraciones y más esfuerzo en sincronización.

2. Single Repository with Environment Folders

- Un único repositorio, con carpetas separadas por entorno.
- **Ventajas:**
- Gestión centralizada y versionada.
- Menor duplicidad.
- **Reto:** requiere estrictas políticas de ramas y validación automatizada para evitar errores de despliegue cruzado.

3. Progressive Delivery Pattern

- Integración con herramientas de feature flags, canary deployments o blue-green deployments.
- Permite entregar cambios a subconjuntos de usuarios o servicios, monitorear impacto y luego ampliar el despliegue.
- Comúnmente integrado con Argo Rollouts, Flagger, o sistemas de control de tráfico en Kubernetes.

4. Drift Detection and Reconciliation

- Uso de operadores como ArgoCD o FluxCD para comparar el estado deseado en Git con el estado actual en el clúster.

Al detectar desviaciones, el sistema puede:

- Notificar al equipo (modo observación),
- Reconciliar automáticamente (modo forzado).
- Esto asegura integridad operativa, incluso ante intervenciones manuales o fallas parciales.

5. Multi-Tenant GitOps

- Implementación de GitOps en entornos compartidos (multi-cluster, multi-equipo).
- **Requiere:**
- Control granular de acceso (RBAC por equipo).
- Estrategias de segmentación (por namespace, carpeta o cluster).
- Automatización de permisos y CI adaptativo.
- Ideal para plataformas internas (PaaS) y organizaciones con múltiples áreas de desarrollo.

Consideraciones estratégicas

Implementar GitOps Patterns exige:

- Políticas claras de revisión de cambios (pull request, approvals).
- Validación previa con linters, validadores de schema (Kustomize, Helm, JsonSchema).
- Integración con pipelines CI que generen artefactos confiables y versionados.
- Automatización de control de acceso y auditoría de cambios (e.g., OPA/Gatekeeper).
- Alta madurez en IaC y monitoreo para lograr observabilidad de los estados deseados vs. reales.

Valor para perfiles senior

Un profesional senior en DevOps no solo debe implementar GitOps, sino diseñar la arquitectura de gobernanza de los repositorios, definir estrategias de escalamiento (GitOps-as-a-Service) y alinear estos patrones con la seguridad, trazabilidad y agilidad organizacional.

Capítulo 3: ArgoCD avanzado.

ArgoCD es una herramienta de despliegue continuo declarativo para Kubernetes basada en GitOps. En contextos avanzados, ArgoCD no solo actúa como motor de sincronización entre Git y el clúster, sino como un componente central de control, seguridad y visibilidad del pipeline operativo. Su uso estratégico permite a los equipos DevOps senior orquestar entornos complejos, multicluster y con políticas estrictas, asegurando reproducibilidad, trazabilidad y recuperación frente a fallos.

Arquitectura de ArgoCD en escenarios complejos

En una implementación avanzada, ArgoCD se compone de los siguientes elementos clave:

- **API Server:** expone la interfaz de usuario, la CLI (argocd) y API RESTful.
- **Repository Server:** accede a los repositorios Git y realiza análisis de manifiestos.
- **Application Controller:** compara el estado deseado con el real y ejecuta la reconciliación.
- **Dex o SSO:** integra autenticación corporativa (OIDC, LDAP, GitHub, etc.) para control de acceso.

En sistemas multicluster, se puede optar por una arquitectura “Hub-and-Spoke” donde un ArgoCD central gestiona múltiples clústeres (spokes), o instancias dedicadas por clúster, coordinadas por herramientas como ArgoCD ApplicationSets.

Funcionalidades avanzadas

1. App of Apps pattern

- Permite declarar una jerarquía de aplicaciones desde una sola fuente (repositorio raíz).
- Mejora la gestión modular de ambientes y facilita el versionamiento centralizado.
- Se utiliza comúnmente para plataformas internas (PaaS) o clusters con múltiples equipos.

2. ApplicationSets

- Automatiza la generación de aplicaciones ArgoCD a partir de fuentes dinámicas:
- Listas de clústeres
- Parámetros Helm
- Directorios
- Pull Requests (preview environments)
- Ideal para gestionar entornos multicluster o despliegues repetitivos a gran escala.

3. Sync Waves y Hooks personalizados

- Control de orden de despliegue entre componentes usando syncWave en los manifiestos.
- Integración con PreSync, Sync, PostSync y Skip hooks para tareas específicas (migraciones, validaciones, backups, etc.).

4. Automated Sync Policies

- Configuración de políticas de sincronización automática:
- On commit (auto-sync)
- Auto-prune (limpia recursos obsoletos)
- Self-heal (reversa cambios manuales en el clúster)
- Se utiliza junto con notificaciones vía Slack, Teams o Webhooks para integrarse al ecosistema organizacional.

5. Integración con OPA/Gatekeeper y RBAC

- Definición de políticas de validación de manifiestos previas al despliegue.
- Limitación de acciones según rol o proyecto (lectura, sincronización, eliminación).

Observabilidad y auditoría

- **Audit logs:** seguimiento detallado de cada sincronización, falla y cambio manual.
- Health checks personalizados: definición de estados saludables o degradados para recursos no estándar.
- **Dashboards de control:** integración con Prometheus/Grafana para métricas como:
 - Tiempo medio de sincronización
 - Fallas por aplicación o equipo
 - Detección de desviaciones

Integraciones comunes en entornos empresariales

- **Sistemas CI:** Jenkins, GitHub Actions, GitLab CI para generación de artefactos y actualizaciones automáticas de los manifiestos.
- **Notificaciones:** ArgoCD Notifications para alertas sobre despliegues, errores y actividades específicas.
- **Sistemas de seguridad y compliance:** integración con firmas de imágenes (cosign), escáneres (Trivy, Gype) y sistemas de aprobación externa.

Consideraciones para perfiles senior

El dominio avanzado de ArgoCD implica más que su instalación y operación: requiere la capacidad de diseñar una arquitectura GitOps segura, escalable y auditabile, coordinando múltiples entornos, equipos y estrategias de recuperación. También exige una comprensión profunda de Kubernetes, IaC, gestión de cambios y observabilidad operativa.

Capítulo 4: Gestión de secretos.

La gestión de secretos es uno de los pilares críticos en entornos DevOps modernos, especialmente cuando se adopta una filosofía GitOps, donde el código y la infraestructura se encuentran versionados en repositorios Git. En este contexto, el desafío radica en mantener la seguridad, confidencialidad y trazabilidad de credenciales, tokens, certificados, claves de acceso y otros datos sensibles, sin comprometer la automatización y el versionado declarativo.

En ambientes senior y altamente regulados, una mala gestión de secretos puede derivar en exposición de información confidencial, accesos no autorizados, o incluso pérdida de integridad operacional. Por ello, se requiere una arquitectura robusta, auditible y automatizada.

Tipos de secretos gestionados

- Credenciales de bases de datos y servicios internos
- Tokens de acceso a APIs o sistemas de terceros
- Claves SSH, JWT, GPG y certificados TLS
- Contraseñas para servicios legacy
- Parámetros cifrados o datos sensibles en variables de entorno

Herramientas y enfoques avanzados de gestión de secretos

1. Sealed Secrets (Bitnami)

- Convierte secretos de Kubernetes en manifiestos cifrados y seguros para versionar en Git.
- Cifrados con una clave pública del controlador (sealed-secrets-controller).
- Solo pueden ser desencriptados por el clúster destino.

2. Vault by HashiCorp

- Solución enterprise para almacenamiento, rotación y políticas de acceso de secretos.
- Capaz de generar secretos dinámicos bajo demanda (e.g., credenciales de bases de datos temporales).
- Integra autenticación con tokens, Kubernetes ServiceAccounts, LDAP, AWS IAM, etc.
- Posee control de acceso basado en políticas (ACL) y soporte para auditoría avanzada.

3. External Secrets Operator (ESO)

- Permite sincronizar secretos almacenados en proveedores externos hacia Kubernetes:
- AWS Secrets Manager
- Azure Key Vault
- Google Secret Manager
- HashiCorp Vault
- Desacopla los secretos del repositorio Git y los maneja vía integración declarativa.

4. SOPS (Secrets OPerationS)

- Herramienta de Mozilla para cifrar archivos YAML/JSON con claves GPG o KMS.
- Compatible con Helm y GitOps.
- Ideal para pipelines CI/CD que versionan plantillas con datos sensibles cifrados.

5. Secret Management con Helm y Kustomize

- Inyección de secretos vía plantillas con values.yaml en Helm Charts o secretGenerator en Kustomize.
- Requiere integración con herramientas como SOPS o ESO para cumplir estándares de seguridad.

Buenas prácticas en entornos GitOps

- Nunca almacenar secretos en texto plano en Git, incluso en repos privados.
- Implementar escáneres de secretos (e.g., Gitleaks, truffleHog) para detectar fugas accidentales.
- Aplicar políticas de control de acceso a secretos según el principio de mínimo privilegio.
- Automatizar la rotación periódica de credenciales sensibles.
- Usar firmas y verificación de integridad para garantizar que los secretos no han sido alterados (e.g., cosign, Notary V2).
- Documentar e integrar un plan de respuesta ante compromiso de secretos, incluyendo revocación inmediata y auditoría forense.

Observabilidad y trazabilidad

En entornos avanzados, la gestión de secretos debe incluir:

- Registro de accesos a secretos (quién accede, cuándo y desde dónde).
- Alertas ante accesos anómalos o no autorizados.
- Integración con SIEM y herramientas de monitoreo para correlación de eventos de seguridad.

Rol del perfil senior

Un DevOps senior debe ser capaz de:

- Diseñar la arquitectura de gestión de secretos adecuada al nivel de exposición y regulación del entorno.
- Integrar la solución con flujos GitOps, CI/CD y plataformas multi-cloud.
- Auditar, automatizar y documentar todos los procesos relacionados con secretos, asegurando cumplimiento normativo y resiliencia ante filtraciones.

Capítulo 4: DR (Disaster Recovery)

¿Qué es Disaster Recovery?

Disaster Recovery (DR) es el conjunto de políticas, herramientas y procedimientos diseñados para restaurar sistemas de TI, datos y operaciones críticas tras un evento disruptivo como fallas de hardware, errores humanos, ciberataques, desastres naturales o caídas masivas de servicios.

Es una parte esencial de la Continuidad del Negocio (BC - Business Continuity) y se enfoca principalmente en recuperar la infraestructura tecnológica y los servicios digitales en un plazo aceptable, minimizando pérdidas.

Objetivos clave de un plan de DR

- Minimizar el tiempo de inactividad.
- Reducir la pérdida de datos.
- Restaurar servicios críticos de forma controlada.
- Mantener la continuidad operativa del negocio.

Conceptos fundamentales

Concepto	Definición breve
RTO (Recovery Time Objective)	Tiempo máximo permitido para recuperar un sistema tras el desastre.
RPO (Recovery Point Objective)	Máximo tiempo aceptable de pérdida de datos medido desde el último backup.
BCP (Business Continuity Plan)	Plan general para mantener el negocio en marcha tras una crisis.
DRP (Disaster Recovery Plan)	Plan específico para recuperar sistemas de TI tras un incidente.

Tipos de desastres que aborda DR

- **Físicos:** incendio, terremoto, inundación, corte eléctrico.
- **Lógicos:** corrupción de datos, fallos en actualizaciones, errores humanos.
- **Cibernéticos:** ransomware, DDoS, accesos no autorizados.
- **Operativos:** caída de datacenter, mal diseño de arquitectura, sabotaje interno.

Estrategias de Disaster Recovery

Estrategia	Descripción
Backups tradicionales	Copias regulares almacenadas en medios externos o en la nube.
Sitio frío	Infraestructura inactiva lista para activarse en caso de desastre (requiere tiempo de montaje).
Sitio caliente	Infraestructura duplicada activa o casi activa, lista para asumir la carga.
Failover automático	Redireccionamiento instantáneo a otra región o sistema redundante (cloud, contenedores, etc.).
Infraestructura como código (IaC)	Uso de Terraform, Ansible o similares para recrear entornos rápidamente.

DR en entornos modernos (Cloud, Kubernetes, GitOps)

Cloud DR:

- Uso de snapshots, multi-región, storage replicado.
- Servicios como AWS Backup, Azure Site Recovery, GCP Disaster Recovery.

Kubernetes DR:

- Backup de etcd, estado de clúster y volúmenes persistentes.
- Soluciones como Velero, Stash o Kasten K10.
- Replicación de manifiestos vía GitOps (ArgoCD/FluxCD) y almacenamiento replicado.

GitOps como soporte para DR:

- Repositorio Git como fuente de verdad para restaurar configuraciones.
- Automatización del redeployment con ArgoCD o Flux.
- Compatibilidad con políticas de reconciliación y sincronización en sitios secundarios.

Buenas prácticas de Disaster Recovery

- Definir RTO/RPO realistas según criticidad del negocio.
- Mantener backups cifrados, redundantes y verificados.
- Automatizar pruebas de recuperación de manera periódica.
- Documentar y mantener actualizado el plan de DR.
- Entrenar al equipo técnico y definir roles claros.
- Simular escenarios de desastre y ajustar el plan.

Ejemplo de flujo DR en nube con GitOps

- Fallo detectado en clúster primario (zona 1).
- Failover automático a clúster secundario (zona 2).
- ArgoCD en el clúster secundario sincroniza manifiestos desde Git.
- Volúmenes restaurados desde snapshot.
- Servicios se levantan en minutos.
- Notificación a DevOps y usuarios impactados.

Herramientas útiles para DR

- Backups & Restore: Velero, Restic, AWS Backup, GCP Snapshots.
- Automatización: Terraform, Ansible, Pulumi.
- Observabilidad: Prometheus, Grafana, Loki, para detectar fallas rápidamente.
- Notificación y respuesta: PagerDuty, Opsgenie, Slack Alerts.

Un buen plan de Disaster Recovery no sólo depende de herramientas, sino de una cultura proactiva, documentación sólida, automatización y entrenamiento constante. Con la adopción de GitOps, IaC y herramientas de nube, es posible diseñar soluciones DR más rápidas, confiables y auditables, alineadas con las exigencias modernas del negocio digital.