{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

CREAR INFRAESTRUCTURAS MODULARES, SEGURAS Y FUNCIONAL CON TERRAFORM AVANZADA, SEGUN LAS PRACTICAS AVANZADAS DE GITOPS, DEVSECOPS, KUBERNETES, OBSERVABILIDAD, IAC, FINOPS Y AIOPS.

Contenidos

Módulo 7: INFRAESTRUCTURA COMO CÓDIGO AVANZADA.

Capítulo 1: Terraform modular.

En entornos DevOps donde se gestiona infraestructura dinámica y de gran escala, la adopción de Terraform Modular es una práctica indispensable para aplicar principios de reutilización, escalabilidad, trazabilidad y automatización en el diseño de la infraestructura como código (IaC). Este enfoque permite a los equipos construir componentes de infraestructura que puedan ser instanciados múltiples veces, parametrizados, testeados y versionados, mejorando la eficiencia operativa y reduciendo la probabilidad de errores.

Fundamentos del enfoque modular

Terraform modular permite agrupar recursos en unidades lógicas llamadas módulos, que actúan como bloques reutilizables. Estos módulos encapsulan una parte de la infraestructura —como una VPC, un clúster EKS, una base de datos RDS o un bucket S3— y exponen una interfaz declarativa para su uso mediante variables de entrada y outputs.

Este enfoque rompe con la práctica de definir todo el main.tf en un solo archivo plano, que se vuelve inmanejable a medida que el sistema crece.

Estructura básica de un módulo

Todo módulo debe tener como mínimo los siguientes componentes:

- **main.tf:** define los recursos y relaciones internas.
- **variables.tf:** declara las entradas requeridas o con valores por defecto.
- **outputs.tf:** expone los valores clave que serán utilizados por otros módulos o niveles superiores.

Por ejemplo, un módulo vpc puede aceptar como entrada un bloque CIDR y retornar el ID de la VPC creada.

La definición modular puede utilizarse desde el proyecto raíz de Terraform mediante el bloque:

```
module "vpc" {
  source     = "./modules/vpc"
  cidr_block = "10.0.0.0/16"
}
```

Esto permite que múltiples entornos (dev, staging, prod) utilicen la misma lógica con diferentes parámetros.

Modularización por dominio y por entorno

Una práctica avanzada es modularizar por dominio de infraestructura (red, cómputo, seguridad, almacenamiento) y reutilizar estos módulos en distintos entornos.

La estructura típica se puede organizar así:

```
terraform/
├── modules/
│   ├── vpc/
│   ├── eks/
│   ├── rds/
│   └── s3/
├── environments/
│   ├── dev/
│   └── prod/
└── backend.tf
```

Cada entorno (dev, prod) utiliza los mismos módulos, lo que garantiza consistencia y gobernanza.

Comunicación entre módulos

Los módulos se conectan mediante el paso de variables y la lectura de outputs. Por ejemplo:

- El módulo vpc expone vpc_id como output.
- El módulo eks requiere vpc_id como variable de entrada.

```
module "eks" {
  source  = "./modules/eks"
  vpc_id  = module.vpc.vpc_id
}
```

Esta arquitectura desacopla componentes y mejora la mantenibilidad.

Versionamiento de módulos y reutilización remota

Terraform permite utilizar módulos remotos desde:

- Terraform Registry (terraform-aws-modules/vpc/aws)
- GitHub (por tag o branch)
- Repositorios internos

Esto facilita la reutilización entre equipos y proyectos. Al versionar los módulos, se puede controlar con precisión cuándo aplicar cambios y bajo qué condiciones.

```
module "vpc" {
  source  = "git::https://github.com/org/modulc
}
```

Testing, validación y control de calidad

Al modularizar, es más fácil aplicar pruebas automatizadas y validaciones específicas para cada componente. Algunas prácticas recomendadas incluyen:

- terraform validate para comprobar la sintaxis.
- tflint para aplicar reglas de estilo.
- checkov para validar seguridad.
- Testing de integración con Terratest (Go) para módulos críticos.
- Documentación automática con terraform-docs.

Además, cada módulo puede incluir pruebas CI/CD específicas que se ejecutan ante cualquier cambio en su repositorio o subcarpeta.

Buenas prácticas en diseño modular

- Diseñar módulos pequeños y cohesionados (single responsibility).
- Evitar lógica condicional excesiva dentro del mismo módulo.
- Definir variables bien tipadas y con validaciones (type, default, description).
- Exponer solo los outputs necesarios.
- Documentar cada módulo con README estructurado y ejemplos de uso.
- Usar locals para simplificar la lógica interna.
- Versionar todos los módulos antes de ser consumidos por entornos productivos.

Integración en pipelines y GitOps

El enfoque modular es ideal para pipelines CI/CD y flujos GitOps, ya que permite:

- Ejecutar planes de forma aislada por módulo
- Desplegar entornos por rama (main → prod, develop → dev)
- Controlar cambios de infraestructura con revisión de código y PRs
- Ejecutar pipelines automáticos con validaciones y políticas

Herramientas como Atlantis, Spacelift, Terraform Cloud o ArgoCD permiten trabajar con módulos versionados integrados en pipelines de infraestructura declarativa.

El enfoque modular con Terraform no solo representa una mejora de organización del código, sino que permite a las empresas y equipos DevOps escalar su infraestructura de forma segura, mantenible, reutilizable y auditable.

Cuando se aplica correctamente, Terraform Modular se convierte en un pilar estratégico para la gestión de múltiples entornos, control de versiones, trazabilidad, automatización y cumplimiento normativo.

Capítulo 2: Testing.

En el contexto de infraestructura como código (IaC), el testing es una práctica esencial para asegurar la calidad, estabilidad, seguridad y mantenibilidad de los recursos desplegados mediante Terraform. Cuando se trabaja con módulos reutilizables, la aplicación de pruebas automatizadas cobra aún más relevancia, ya que un error o una mala práctica en un módulo puede replicarse en múltiples entornos o sistemas.

El testing en Terraform modular permite detectar errores antes del despliegue, validar que los módulos cumplen con las expectativas, aplicar políticas de seguridad y verificar que las salidas generadas son consistentes.

Niveles de testing en Terraform

Terraform admite distintos niveles de pruebas que, en conjunto, permiten una validación exhaustiva:

1. Linting (formato y estilo)
2. Validación sintáctica
3. Testing funcional de los módulos
4. Análisis de seguridad y cumplimiento
5. Testing de integración (infraestructura real o simulada)

Cada nivel aporta una capa diferente de garantía para entornos productivos y pipelines CI/CD.

Linting: estilo y buenas prácticas

Antes de realizar cualquier prueba funcional, es necesario mantener la coherencia del código. Herramientas como:

- **terraform fmt:** estandariza el formato de los archivos .tf
- **tflint:** analiza errores comunes, uso incorrecto de recursos, variables mal tipadas, etc.
- **terraform-docs:** genera automáticamente documentación estructurada de los módulos

Estas herramientas aseguran legibilidad, coherencia y documentación, y pueden integrarse fácilmente en pipelines automatizados.

Validación estática

Terraform permite validar la sintaxis del código sin aplicar cambios reales:

- **terraform validate:** asegura que la configuración es válida, aunque no aplica cambios.
- **terraform plan:** muestra las acciones que Terraform realizaría, ayudando a identificar problemas en tiempo de ejecución antes del apply.

Estas validaciones deben ejecutarse en cada módulo individual, especialmente cuando se trabaja con múltiples entornos o ramas.

Análisis de seguridad y cumplimiento

La seguridad es crítica en entornos productivos. Existen herramientas especializadas para analizar configuraciones peligrosas o fuera de política:

- **Checkov:** escanea módulos Terraform y detecta configuraciones inseguras (puertos abiertos, claves hardcodeadas, roles mal definidos).
- **Tfsec:** realiza análisis estático con enfoque en buenas prácticas de seguridad por proveedor (AWS, Azure, GCP).
- **OPA / Rego:** define políticas organizacionales como "no se permiten buckets públicos", que deben cumplirse antes del apply.

Estas herramientas deben ejecutarse tanto en los módulos como en los entornos que los consumen.

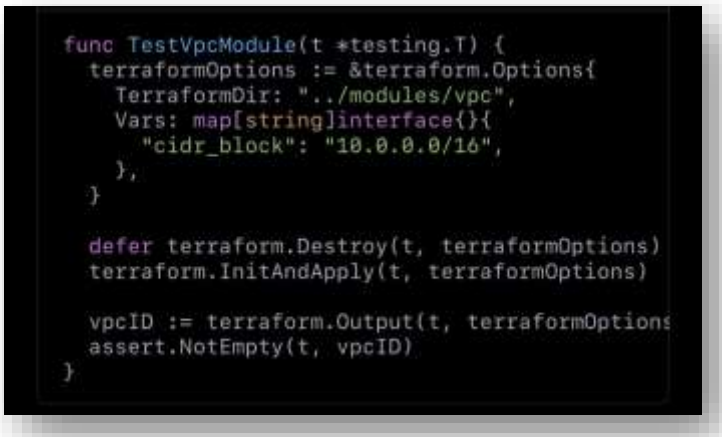
Testing funcional con Terratest

Para pruebas automatizadas sobre módulos, la herramienta más robusta y comúnmente usada es Terratest.

Terratest es una librería de testing escrita en Go que permite:

- Aplicar un módulo Terraform completo en un entorno aislado
- Verificar que los recursos fueron creados correctamente
- Consultar salidas (outputs)
- Ejecutar validaciones post-despliegue (por ejemplo, acceso HTTP, existencia de recursos)

Ejemplo básico de prueba con Terratest:



```
func TestVpcModule(t *testing.T) {
  terraformOptions := &terraform.Options{
    TerraformDir: "../modules/vpc",
    Vars: map[string]interface{}{
      "cidr_block": "10.0.0.0/16",
    },
  }

  defer terraform.Destroy(t, terraformOptions)
  terraform.InitAndApply(t, terraformOptions)

  vpcID := terraform.Output(t, terraformOptions
  assert.NotEmpty(t, vpcID)
}
```

Capítulo 3: Sentinel.

Sentinel es el motor de políticas como código (Policy-as-Code) desarrollado por HashiCorp. Su propósito es permitir que las organizaciones apliquen reglas personalizadas y automáticas para gobernar el uso de infraestructura como código, especialmente en Terraform Cloud y Terraform Enterprise. A diferencia de herramientas que realizan análisis estático previo al despliegue, Sentinel se ejecuta durante la ejecución de Terraform y puede impedir que una operación avance si no cumple con las políticas definidas.

En organizaciones que requieren trazabilidad, cumplimiento normativo, o control sobre costos y seguridad, Sentinel permite a los equipos de plataforma y seguridad definir políticas centralizadas que se aplican de manera transversal, sin necesidad de que los desarrolladores modifiquen su flujo de trabajo.

Funcionalidad y propósito

Sentinel actúa como un sistema de validación de políticas organizacionales en tiempo de ejecución. Estas políticas permiten inspeccionar y controlar el contenido de un terraform plan o apply, validando atributos, configuraciones, etiquetas, tamaños de recursos, ubicación regional, uso de recursos públicos, y cualquier otra condición que la organización considere crítica.

El objetivo es automatizar el cumplimiento de políticas de infraestructura sin depender de revisiones manuales, y sin necesidad de integraciones complejas con herramientas externas.

Niveles de aplicación de políticas

Las políticas Sentinel pueden aplicarse con distintos niveles de rigidez:

- El modo advisory permite ejecutar el plan aunque se incumpla la política, pero muestra una advertencia visible.
- El modo soft-mandatory detiene la ejecución, pero permite a ciertos usuarios autorizados forzar el avance.
- El modo hard-mandatory impide completamente que la operación continúe si no se cumple la política.

Esto permite a las organizaciones implementar las políticas de forma progresiva, comenzando con advertencias y avanzando hacia bloqueos obligatorios una vez validadas.

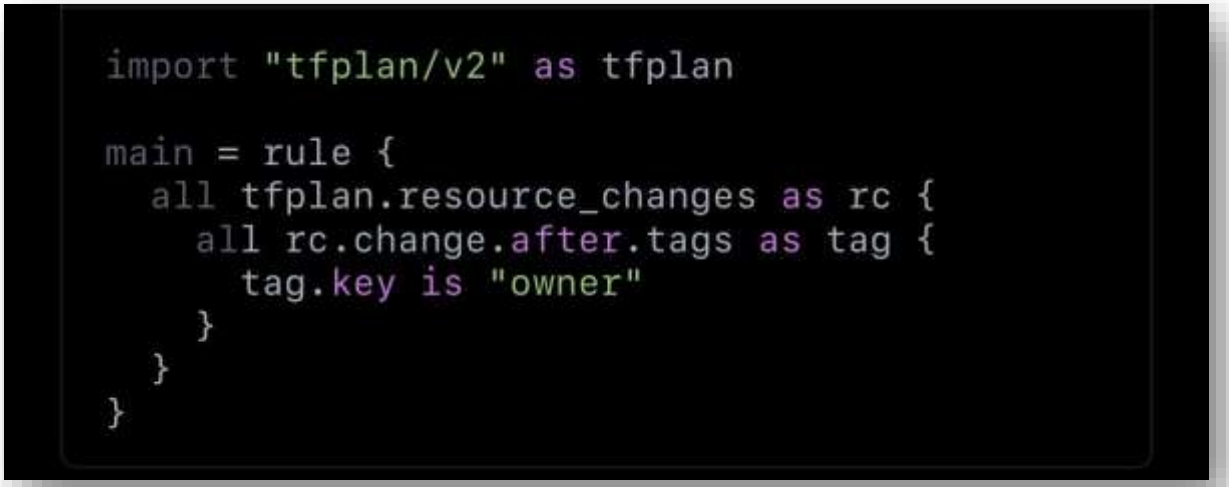
Estructura de una política Sentinel

Una política Sentinel se compone de:

Importaciones: módulos como tfplan, tfstate, o tfconfig que permiten acceder al contenido del plan, estado o configuración de Terraform.

- **Reglas:** expresiones lógicas que evalúan condiciones específicas.
- **Función principal (main):** define el criterio de aprobación o rechazo.
- **Operadores lógicos:** combinaciones de condiciones, iteraciones y filtros que permiten expresar lógica compleja.

Por ejemplo, se puede crear una política que bloquee toda instancia que no tenga la etiqueta owner:



```
import "tfplan/v2" as tfplan

main = rule {
  all tfplan.resource_changes as rc {
    all rc.change.after.tags as tag {
      tag.key is "owner"
    }
  }
}
```

Si alguna instancia en el plan no tiene esa etiqueta, la ejecución se bloquea (dependiendo del modo de enforcement).

Integración con Terraform Cloud

El uso de Sentinel requiere una cuenta en Terraform Cloud o Terraform Enterprise, donde se habilita el motor de políticas como parte del flujo de trabajo.

Las políticas se agrupan en conjuntos llamados Policy Sets, los cuales se pueden asociar a uno o varios workspaces. Cuando un usuario ejecuta un plan en uno de estos workspaces, Terraform evalúa las políticas activas contra el contenido del plan generado.

Esto garantiza que todas las ejecuciones cumplan con las normas organizacionales, independientemente de qué desarrollador, entorno o repositorio haya iniciado la operación.

Aplicaciones prácticas

Sentinel permite definir reglas como:

- Rechazar cualquier recurso que no tenga ciertas etiquetas requeridas.
- Impedir la creación de buckets S3 con acceso público.
- Permitir solo ciertos tamaños de instancias en producción.
- Bloquear el uso de claves o contraseñas sin rotación automática.
- Validar que los recursos solo se creen en regiones autorizadas.
- Forzar la creación de recursos con cifrado habilitado.

Estas políticas pueden ser tan simples o complejas como lo requiera la organización. Sentinel proporciona un lenguaje declarativo potente y expresivo, que permite inspeccionar incluso estructuras profundamente anidadas dentro de un plan de Terraform.

Validación y pruebas de políticas

Aunque Sentinel se ejecuta dentro del entorno de Terraform Cloud, también es posible probar las políticas localmente con la herramienta Sentinel CLI, utilizando archivos simulados (mock data) que representan el plan, configuración o estado.

Esto permite validar que las políticas funcionan correctamente antes de activarlas en producción, evitando bloqueos inesperados.

El flujo común es:

1. Escribir la política .sentinel
2. Crear archivos .json que simulen planes válidos e inválidos
3. Ejecutar sentinel test para validar que las reglas se comportan como se espera

Mejores prácticas en su implementación

Para aplicar Sentinel de forma eficaz:

- Comenzar en modo advisory para monitorear impactos sin bloquear despliegues.
- Versionar todas las políticas mediante control de versiones (Git).
- Estandarizar una librería organizacional de políticas reutilizables (por ejemplo: políticas de tagging, regiones válidas, seguridad de red).
- Integrar revisiones de políticas en los flujos CI/CD de infraestructura.
- Capacitar a los equipos en el lenguaje y lógica de Sentinel para facilitar la colaboración y la adopción.

Además, es recomendable combinar Sentinel con herramientas previas al plan como Checkov o Tfsec, para obtener una validación integral desde el diseño hasta la ejecución.

Sentinel representa una capa de gobernanza automatizada sobre Terraform, especialmente útil en entornos organizacionales donde múltiples equipos trabajan sobre una infraestructura compartida. Su capacidad para bloquear ejecuciones en base a reglas declarativas personalizadas permite mantener estándares de seguridad, cumplimiento y arquitectura, sin frenar la velocidad de entrega de los equipos DevOps.

Al integrar Sentinel como parte de una estrategia modular de Terraform, la organización adquiere control granular y centralizado sobre sus recursos, elevando el nivel de seguridad, trazabilidad y calidad de toda su infraestructura como código.

Capítulo 4: Integración IaC + GitOps.

La gestión moderna de infraestructura exige no solo automatización, sino también control de versiones, trazabilidad y revisión estructurada. Para ello, la combinación entre Infraestructura como Código (IaC) y el enfoque GitOps permite implementar una estrategia robusta, repetible y auditable para la provisión y mantenimiento de entornos cloud, híbridos o on-premise.

Integrar IaC con GitOps significa que los cambios en la infraestructura se gestionan desde Git, donde cada modificación es trazada, revisada y aplicada automáticamente a través de pipelines controlados. Este modelo reduce riesgos, acelera despliegues, mejora la colaboración y fortalece la gobernanza organizacional.

Principios de GitOps aplicados a IaC

El enfoque GitOps se basa en los siguientes pilares fundamentales:

- Declaración del estado deseado en archivos versionados dentro de Git.
- Auditoría completa mediante historial de commits y pull requests.
- Automatización del ciclo de aplicación usando herramientas CI/CD o agentes GitOps.
- Reversión instantánea de cambios mediante rollback a versiones anteriores.
- Revisión colaborativa antes de cualquier modificación (merge request, pull request).

Al integrar esto con Terraform, se logra que la infraestructura nunca se modifique manualmente, sino solo a través de Git, lo que permite controlar entornos de forma programática y segura.

Repositorio como fuente única de verdad

La integración GitOps comienza al definir que el repositorio Git es el único lugar autorizado para describir el estado de la infraestructura. Desde ahí:

- Se declaran módulos Terraform, entornos y configuraciones (.tf, terraform.tfvars)
- Se almacenan los backends, configuraciones de state remoto y versiones de módulos
- Se documentan los flujos, outputs, decisiones arquitectónicas y condiciones de entorno

Este enfoque convierte al repositorio en el núcleo operativo de la infraestructura, facilitando inspección, gobernanza y colaboración entre equipos.

Flujo GitOps para Terraform

El ciclo de vida típico al integrar Terraform con GitOps es el siguiente:

1. Un desarrollador o ingeniero de plataforma realiza cambios en una rama (feature/infra-update).
2. Se abre un pull request, donde se revisa el código y se ejecutan pruebas automatizadas:
- terraform fmt, validate, tflint, checkov, terraform plan
3. Tras la aprobación del PR, el código se mergea a main o prod.
4. Un pipeline CI/CD detecta el cambio y ejecuta el terraform apply, aplicando la infraestructura declarada.
5. El estado (terraform.tfstate) se almacena en un backend remoto (S3, Terraform Cloud, etc.).
6. Toda la ejecución queda registrada en el historial de Git, la salida del pipeline y los logs del proveedor cloud.

Este flujo permite aplicar cambios reproducibles, con validación previa, y con trazabilidad completa.

Herramientas y enfoques de automatización

La implementación de GitOps con Terraform puede realizarse con múltiples herramientas, dependiendo del entorno técnico:

- **CI/CD clásico:** herramientas como GitHub Actions, GitLab CI/CD, Jenkins, CircleCI pueden ejecutar terraform plan y apply ante cada cambio en Git.
- **GitOps Agents:** herramientas como Atlantis o Spacelift permiten ejecutar los cambios desde Git, validando en PR y aplicando al aprobar.
- **Terraform Cloud/Enterprise:** integran repositorios Git y ejecutan planes/applies directamente, con políticas y seguridad integradas.

Cualquiera sea la herramienta, el patrón es claro: Git origina la ejecución, no el usuario manualmente.

Gestión de entornos y ramas

Una ventaja crítica de GitOps es la separación clara entre entornos mediante ramas, carpetas o workspaces. Algunas prácticas comunes incluyen:

- **main o prod:** rama de despliegue productivo.
- **develop:** rama para testing e integración.
- **Ramas por entorno:** env/dev, env/staging, env/prod.
- **Carpetas por entorno:** terraform/environments/dev, terraform/environments/prod.

Cada rama o carpeta puede usar los mismos módulos, pero con distintos valores (.tfvars), garantizando consistencia con flexibilidad.

Beneficios clave de integrar IaC + GitOps

- **Auditoría total:** cada cambio tiene autor, motivo, fecha y revisión previa.
- **Automatización confiable:** evita errores humanos en apply manuales.
- **Despliegues repetibles:** un commit genera el mismo resultado en cualquier entorno.
- **Rollback simplificado:** volver a un commit anterior revierte la infraestructura.
- **Seguridad fortalecida:** se puede limitar el acceso a aplicar cambios a través de control de acceso a Git.
- **Escalabilidad organizacional:** múltiples equipos pueden contribuir sin colisionar ni sobrescribir recursos críticos.

Riesgos y mitigaciones

Como toda automatización, la integración de IaC con GitOps requiere consideraciones clave:

- **Evitar cambios manuales fuera de Git:** si se hace, debe detectarse mediante drift detection o políticas de control de estado.
- **Validar siempre antes de aplicar:** terraform plan debe ejecutarse en PRs, con resultados visibles para el equipo.
- **Controlar permisos sobre Git:** restringir quién puede hacer merge a ramas productivas.
- **Proteger el state remoto:** asegurar versiones, bloqueo concurrente y cifrado en backends.
- **Versionar módulos y dependencias:** evitar cambios inesperados al consumir módulos sin control de versión.

Estas medidas permiten mantener el enfoque GitOps como una práctica segura, trazable y escalable.

La integración de IaC con GitOps representa un salto cualitativo en la forma en que las organizaciones administran su infraestructura. Al basarse en principios de declaración, revisión, automatización y control de versiones, permite transformar un proceso manual y propenso a errores en un flujo reproducible, auditable y colaborativo.

En entornos DevOps maduros, GitOps no es una alternativa, sino una extensión natural y necesaria del uso de Terraform modular, y constituye uno de los pilares fundamentales de una operación confiable, segura y escalable en entornos multi-equipo y multi-entorno.