DevOps Guide

Terraform, GitLab CI/CD, State, Security, Observability and Incident Management

A dense operational PDF generated from the three IDEO-Lab HTML guides.

Designed for new DevOps engineers, infrastructure teams, SRE practices and production readiness.

Table of contents

How to use this guide

This document converts the interactive HTML guide into a printable operational reference. Each chapter corresponds to a card or modal from the original guides. The structure is deliberately compact: concepts, production examples, tables, checklists, diagrams and command-oriented notes are kept close together.

Use it as an onboarding handbook for new DevOps engineers, as a production-readiness checklist, or as a reference during reviews, incident preparation and GitLab/Terraform workflow design.

Part 1 - DevOps - Terraform, State, GitLab CI/CD & Gestion d'incidents (Part 1)

This part groups related operational topics from the IDEO-Lab DevOps guide. Each chapter below extracts the practical knowledge embedded in the interactive modal panels.

Step	Topic	Purpose
1.1	Vue d'ensemble DevOps	Le rôle DevOps côté infra : automatiser, fiabiliser, tracer et sécuriser.
1.2	Terraform : bases solides	Providers, resources, variables, outputs, plan/apply et cycle de vie.
1.3	Modules Terraform	Découper proprement réseau, compute, sécurité, base de données et applicatif.
1.4	State Terraform	Backend distant, locking, dérive, isolation par environnement et sécurité.
2.1	GitLab CI/CD Infra	Stages fmt, validate, plan, review, apply et rollback contrôlé.
2.2	Environnements	Dev, staging, prod : variables protégées, branches, approvals et séparation.
2.3	Secrets & sécurité CI	Variables masquées, scopes, credentials courts, policy least privilege.
2.4	Déploiement infra	Workflow propre : changement, revue, plan, fenêtre de prod, apply et vérification.
3.1	Observabilité	Logs, métriques, alertes, traces et tableaux de bord pour piloter la production.
3.2	Incidents production	Qualifier, mitiger, restaurer, communiquer, analyser et éviter la récidive.

1.1 - Vue d'ensemble DevOps

> Le rôle DevOps côté infra : automatiser, fiabiliser, tracer et sécuriser.

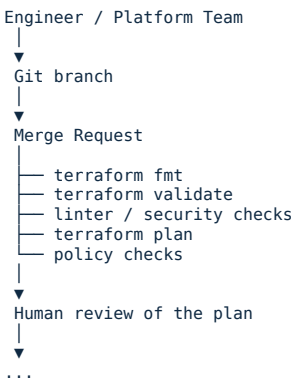
Source modal: 1.1 DevOps Infrastructure Overview

1. What infrastructure DevOps really means

Infrastructure DevOps is not only about deploying servers. It is about building platforms and delivery processes that are repeatable , auditable , secure , observable , and resilient .

The goal is simple: if production breaks, or a new environment is needed, the team must be able to rebuild or change it from code, with controlled automation, clear approvals, and full traceability.

Area	What DevOps handles
Provisioning	Create VPCs, subnets, compute, IAM, load balancers, databases, DNS, storage.
Automation	Turn infrastructure changes into reviewed code and pipelines.
Reliability	Keep services available and reduce operational risk.
Security	Apply least privilege, secrets hygiene, network controls, auditability.
Observability	Provide logs, metrics, traces, dashboards, alerts, and health checks.
Incident response	Restore service fast, communicate clearly, then drive root cause analysis.



2. What production reality looks like

New DevOps engineers often imagine infrastructure work as "create servers and deploy". In reality, production means handling multiple environments, permissions, cost controls, on-call pressure, changing business needs, compliance, and failure scenarios.

Constraint	Impact
Availability	Changes must avoid downtime or keep it minimal and controlled.
Security	Credentials, IAM roles, network exposure, and runner hardening matter.
Cost	DevOps must balance reliability with cloud spending.
Compliance	Audit trails, separation of duties, and approval flows may be mandatory.
Recovery	Backups and restore tests are part of production readiness.

```
Users
 │
 ▼
DNS / CDN / WAF
 │
 ▼
Load Balancer
 │
 ▼
Application Tier
 ├── container platform / VMs
 ├── autoscaling
 └── background workers
 │
 ▼
Data Tier
 ├── relational database
 ├── cache
...
```

3. Professional toolchain by responsibility

Master Linux basics and networking first.; Then learn Terraform structure, modules, variables, outputs, remote state.; Then learn GitLab CI/CD stages, jobs, variables, artifacts, protected environments.; Then learn observability: logs, metrics, alert rules, dashboards, SLOs.

Responsibility	Main tools	Typical usage
IaC	Terraform, Terragrunt, OpenTofu	Provision cloud resources, shared modules, environment structure.
CI/CD	GitLab CI/CD, GitHub Actions, Jenkins	Validate, plan, approve, apply, deploy, rollback.
Containers	Docker, Kubernetes, Helm	Package workloads, orchestrate services, manage releases.
Config management	Ansible, Salt, Chef	OS packages, user setup, service configuration.
Monitoring	Prometheus, Grafana, Datadog, CloudWatch	Metrics, dashboards, alerting, SLO tracking.
Logging	ELK, OpenSearch, Loki	Search logs, correlate incidents, root cause analysis.
Tracing	Jaeger, Tempo, Datadog APM	Latency analysis across services.

4. Terraform in real production

Terraform is used to describe cloud infrastructure as code. In a professional setup, code is organized into reusable modules, state is stored remotely, environments are clearly separated, and applies are done through controlled pipelines.

A team builds a three-tier web platform on AWS. Terraform modules provision the VPC, NAT gateways, ALB, ECS services, RDS, IAM roles, Route53 records, and alarms. The GitLab pipeline runs fmt , validate , security checks, plan , then an approval gate, then apply only on protected branches.

Module quality	Professional expectation
Reusable	Inputs and outputs are clear and minimal.
Safe	Default values do not expose public access unintentionally.
Readable	Variables, tags, and naming conventions are consistent.
Composable	Modules can be combined into complete environments.

```
live/
 prod/
 network/
 security/
 app/
 data/
 staging/
 network/
 security/
 app/
 data/
 modules/
```

```
vpc/
security_group/
ecs_service/
rds_postgres/
iam_role/
dns_record/
```

5. GitLab CI/CD for infrastructure

A pipeline is not only an execution engine. It is an enforcement layer for review, quality checks, security rules, approvals, and production traceability.

Stage	Goal
lint	Format and style consistency.
validate	Catch syntax and structural errors early.
security_scan	Detect risky Terraform patterns and policy violations.
plan	Show intended infrastructure changes before apply.
approval	Human control point for sensitive environments.
apply	Execute only in controlled, protected contexts.
post_deploy_checks	Verify service health, alarms, and smoke tests.

```
Stages
 1. lint
 2. validate
 3. security_scan
 4. plan
 5. review / approval
 6. apply
 7. post_deploy_checks
```

6. Incident handling: what "professional" looks like

Alert shows API error rate spike and DB connection failures.; On-call verifies a recent infrastructure change.; Terraform plan/apply history identifies a modified security group rule.; Immediate mitigation: revert or restore DB access rule.

Role	Responsibility
Incident lead	Coordinates technical response and decisions.
Communications lead	Updates stakeholders and customer-facing channels.
Operations engineer	Executes rollback, checks infra, validates mitigation.
Application engineer	Correlates infra changes with app behavior.

```
Alert fires
 │
 ▼
 Triage
 │
 ├── confirm impact
 ├── identify affected scope
 └── assign incident lead
 │
 ▼
 Mitigation
 ├── rollback
 ├── disable bad change
 ├── failover
 └── traffic control
 │
 ▼
 Recovery validation
...
```

7. How DevOps expectations differ by company type

In a SaaS company, DevOps is often expected to own cloud provisioning, CI/CD, observability, and incident response. The emphasis is on velocity, but velocity must be constrained by good delivery controls.

In a fintech environment, infrastructure changes may require formal approvals, stronger audit trails, tighter IAM policies, and detailed rollback procedures. The same Terraform skillset applies, but the governance layer is much stricter.

Company type	Typical DevOps focus	Examples of concern
Startup SaaS	Speed, automation, cost awareness, fast recovery.	Simple platform, small team, high pressure to deliver.
E-commerce	Traffic peaks, latency, checkout reliability, on-call maturity.	Black Friday readiness, scaling, payment path resilience.
Fintech / banking	Security, compliance, approval workflows, audit trail.	Separation of duties, secrets management, restricted access.

Company type	Typical DevOps focus	Examples of concern
Industrial / enterprise IT	Stability, change control, hybrid systems, documentation.	Legacy integration, strict release windows.
Media / streaming	Scalability, CDN, real-time monitoring, resilience.	Traffic spikes, regional performance, caching strategy.

8. Operational KPI, SLA, SLO, and reliability signals

A service may have an internal SLO of 99.9% availability. If incidents consume too much of the error budget, the team should pause risky changes and focus on stability improvements.

Metric	What it tells you
Availability	Percentage of time the service is reachable and working.
Error rate	How many requests fail over time.
Latency p95 / p99	User-perceived speed under load.
MTTD	Mean time to detect an issue.
MTTR	Mean time to recover service.
Deployment frequency	How often the team ships safely.
Change failure rate	How often releases or changes introduce incidents.

1.2 - Terraform : bases solides

Providers, resources, variables, outputs, plan/apply et cycle de vie.

Source modal: 1.2 Terraform : bases solides en production

1. Terraform : ce qu'il faut vraiment comprendre

Terraform est un outil d' Infrastructure as Code . Il permet de décrire l'état attendu d'une infrastructure dans des fichiers versionnés, puis de comparer cet état avec la réalité du cloud ou du provider ciblé.

Contrairement à un script Bash classique, Terraform ne se contente pas d'exécuter des commandes dans l'ordre. Il construit un graphe de dépendances , calcule les différences entre le code et l'état réel, puis propose un plan d'action.

Famille	Exemples concrets
Réseau	VPC, subnets, route tables, NAT gateway, firewall rules.
Compute	VM, autoscaling group, ECS service, Kubernetes node pool.
Sécurité	IAM roles, policies, security groups, KMS keys.
Données	RDS PostgreSQL, MySQL, Redis, buckets S3, volumes.
Routage	DNS, load balancer, listeners, certificats TLS.
Observabilité	Alarmes CloudWatch, dashboards, log groups, alert rules.

```
Code Terraform
 │
 ├── providers
 ├── variables
 ├── resources
 ├── data sources
 └── outputs
 │
 ▼
 terraform plan
 │
 ▼
 Diff entre état souhaité et état réel
 │
 ▼
 terraform apply
 │
 ▼
...
```

2. Architecture Terraform professionnelle

Un projet Terraform sérieux doit être organisé pour éviter le chaos : séparation des environnements, modules réutilisables, backend distant, conventions de nommage, tagging, sécurité et validation CI/CD.

Domaine	Contenu typique	Risque principal
network	VPC, subnets, routes, NAT, peering.	Couper tout le trafic si erreur.
security	IAM, security groups, KMS, firewall.	Ouvrir trop large ou bloquer la prod.
app	Compute, containers, load balancer, autoscaling.	Indisponibilité applicative.
data	RDS, Redis, buckets, volumes.	Perte de données ou recréation accidentelle.
monitoring	Alarmes, dashboards, logs.	Incident invisible ou alertes inutiles.

```
infra/
 ├── modules/
 │   ├── network/
 │   │   ├── main.tf
 │   │   ├── variables.tf
 │   │   ├── outputs.tf
 │   │   └── README.md
 │   ├── security_group/
 │   ├── compute_service/
 │   ├── database/
 │   ├── dns/
 │   └── monitoring/
 │
 ├── live/
 │   ├── dev/
 │   │   ├── network/
 │   │   ├── app/
 │   │   └── data/
...
```

3. Fichiers Terraform : rôle et structure

Fichier	Rôle
versions.tf	Contraintes de versions Terraform et providers.

Fichier	Rôle
providers.tf	Configuration des providers : AWS, Azure, GitLab, etc.
backend.tf	Configuration du stockage distant du state.
variables.tf	Déclaration des paramètres d'entrée.
main.tf	Ressources principales ou appel aux modules.
locals.tf	Valeurs calculées, conventions de nommage, tags communs.
outputs.tf	Valeurs exposées après création.

```
terraform/
  ├── providers.tf
  ├── versions.tf
  ├── backend.tf
  ├── variables.tf
  ├── main.tf
  ├── outputs.tf
  ├── locals.tf
  ├── terraform.tfvars
  └── README.md
```

4. Providers : la passerelle vers les plateformes

Un provider est le connecteur qui permet à Terraform de discuter avec une API externe : AWS, Azure, GCP, GitLab, Cloudflare, Kubernetes, Datadog, Vault, Hetzner, etc.

Provider	Usage courant
AWS	VPC, EC2, ECS, EKS, RDS, IAM, S3, CloudWatch, Route53.
Azure	Resource groups, VNets, AKS, Key Vault, SQL, App Services.
Google Cloud	VPC, GKE, Cloud SQL, IAM, Cloud Storage, Cloud Run.
Kubernetes	Namespaces, secrets, config maps, services, deployments.
GitLab	Projects, variables CI/CD, runners, groups, access tokens.
Cloudflare	DNS, WAF, CDN, firewall rules.
Datadog	Dashboards, monitors, alerting, SLO.

```
provider "aws" {
 region = var.aws_region
 }

 provider "cloudflare" {
 api_token = var.cloudflare_api_token
 }

 provider "gitlab" {
 token = var.gitlab_token
 }
```

5. Resources et data sources

Une resource représente un objet que Terraform doit créer, modifier ou supprimer. Exemple : une instance, un bucket, une règle réseau, une base de données.

Une data source permet de récupérer une information existante sans en devenir propriétaire. C'est utile pour référencer une AMI, un VPC existant, un secret, une zone DNS, etc.

Élément	Terraform est propriétaire ?	Exemple
Resource	Oui	Créer un security group.
Data source	Non	Lire une AMI Ubuntu existante.

```
resource "aws_security_group" "web" {
 name = "${local.name_prefix}-web-sg"
 description = "Security group for web traffic"
 vpc_id = var.vpc_id

 ingress {
 description = "HTTPS from internet"
 from_port = 443
 to_port = 443
 protocol = "tcp"
 cidr_blocks = ["0.0.0.0/0"]
 }

 egress {
 description = "Outbound traffic"
 from_port = 0
 to_port = 0
 protocol = "-1"
...
```

6. Variables, outputs et conventions

Les variables permettent de rendre le code Terraform paramétrable. Elles évitent de dupliquer le même code pour dev, staging et production.

Les outputs exposent des valeurs générées par Terraform : endpoint de base de données, DNS du load balancer, ID du VPC, ARN d'un rôle IAM, etc.

Type de donnée	Conseil
Mot de passe	Ne pas mettre en clair dans Git.
Token API	Utiliser variables CI/CD protégées ou secret manager.
Clé privée	Éviter dans Terraform si possible.
Endpoint public	Output acceptable si non sensible.

```
variable "project" {
 description = "Project name"
 type = string
 }

 variable "environment" {
 description = "Deployment environment"
 type = string

 validation {
 condition = contains(["dev", "staging", "prod"], var.environment)
 error_message = "Environment must be dev, staging, or prod."
 }
 }

 variable "instance_type" {
 description = "EC2 instance type"
 type = string
...
```

7. Cycle Terraform : init, fmt, validate, plan, apply

Commande	Rôle
terraform init	Initialise le backend, télécharge les providers et modules.
terraform fmt	Formate le code selon les conventions Terraform.
terraform validate	Vérifie la cohérence syntaxique et structurelle.
terraform plan	Affiche ce qui va changer sans modifier l'infrastructure.
terraform apply	Applique réellement les changements.
terraform destroy	Détruit les ressources gérées. À éviter en production sauf cas maîtrisé.

```
terraform init
 terraform fmt -check -recursive
 terraform validate
 terraform plan -var-file="prod.tfvars" -out="tfplan"
 terraform apply "tfplan"
```

8. State Terraform : le point le plus critique

Le state est la mémoire de Terraform. Il contient la correspondance entre les ressources déclarées dans le code et les ressources réellement créées chez le provider. Sans state fiable, Terraform ne sait plus correctement ce qu'il possède.

Sujet	Risque	Bonne pratique
State local	Perte, divergence, conflit entre développeurs.	Utiliser un backend distant.
Concurrence	Deux apply en même temps peuvent corrompre l'état.	Activer le verrouillage du state.
Sensibilité	Le state peut contenir des valeurs sensibles.	Restreindre les accès et chiffrer.
Suppression	Terraform perd la mémoire de l'infra gérée.	Versionner et protéger le backend.

```
terraform {
 backend "s3" {
 bucket = "company-terraform-state-prod"
 key = "prod/network/terraform.tfstate"
 region = "eu-west-3"
 dynamodb_table = "terraform-locks"
 encrypt = true
 }
 }
```

1.3 - Modules Terraform

Découper proprement réseau, compute, sécurité, base de données et applicatif.

Source modal: 1.3 Modules Terraform : découpage réutilisable et production-ready

1. Pourquoi créer des modules Terraform ?

Un module Terraform est une brique d'infrastructure réutilisable . Il encapsule une responsabilité claire : réseau, sécurité, compute, base de données, DNS, monitoring, load balancer, IAM ou applicatif.

Sans modules, un projet Terraform finit souvent en copier-coller massif entre dev, staging et production. Cela crée des divergences, des erreurs de configuration, des plans difficiles à relire et une maintenance fragile.

Objectif	Impact concret
Réutilisation	Le même module peut servir pour dev, staging, prod ou plusieurs projets.
Lisibilité	Le root module reste compréhensible : il appelle des briques métier.
Standardisation	Nommage, tags, sécurité, logs et conventions sont homogènes.
Contrôle	Les variables exposées limitent ce que l'utilisateur du module peut modifier.
Maintenance	Une correction dans le module peut bénéficier à plusieurs environnements.
Audit	Les décisions d'infrastructure deviennent plus faciles à relire et expliquer.

```
Root module d'un environnement
│
├── module.network
│   ├── VPC
│   ├── subnets publics / privés
│   ├── routes
│   └── NAT gateway
│
├── module.security
│   ├── security groups
│   ├── IAM roles
│   ├── policies
│   └── KMS
│
├── module.compute
│   ├── VM / ECS / EKS
│   ├── autoscaling
│   ├── load balancer
...
```

2. Architecture recommandée pour modules Terraform

Un bon découpage sépare les modules réutilisables du code live qui instancie ces modules pour un environnement précis.

Type	Rôle	Exemple
Root module	Point d'entrée exécuté par terraform plan/apply .	live/prod/app/
Child module	Brique appelée par le root module.	modules/load-balancer
Shared module	Module partagé entre plusieurs repos ou équipes.	git::ssh://...
External module	Module public ou privé publié dans un registry.	Terraform Registry, GitLab module registry.

```
infra/
├── modules/
│   ├── network/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   │   ├── locals.tf
│   │   ├── versions.tf
│   │   └── README.md
│   │
│   ├── security-group/
│   ├── load-balancer/
│   ├── compute-service/
│   ├── database-postgres/
│   ├── redis-cache/
│   ├── dns-record/
│   └── monitoring-alarm/
│
...
```

3. Le contrat d'un module : inputs, outputs, garanties

Un module Terraform sérieux doit être pensé comme une API. Il expose des inputs , retourne des outputs , applique des conventions et documente clairement ce qu'il crée.

Élément	Bonne pratique
Inputs	Peu nombreux, typés, documentés, avec defaults raisonnables.
Outputs	Limités aux valeurs nécessaires aux autres modules ou à l'exploitation.
Nommage	Prévisible : projet, environnement, composant, région.
Tags	Appliqués partout quand le provider le permet.
Sécurité	Pas d'accès public par défaut, pas de droits trop larges.

```
variable "project" {
 description = "Project name used for naming and tags."
 type = string
 }

 variable "environment" {
 description = "Environment name."
 type = string

 validation {
 condition = contains(["dev", "staging", "prod"], var.environment)
 error_message = "Environment must be dev, staging, or prod."
 }
 }

 variable "subnet_ids" {
 description = "Private subnet IDs used by the service."
 type = list(string)
...
```

4. Exemple de module réseau

Le module réseau est souvent la base de toute infrastructure cloud. Il crée le VPC, les subnets, les routes, les gateways et expose les IDs nécessaires aux modules applicatifs.

Responsabilité	Détail
VPC / réseau principal	Définir le CIDR global de l'environnement.
Subnets publics	Load balancers, NAT gateways, bastions éventuels.
Subnets privés	Applications, workers, bases de données, caches.
Routes	Accès internet, NAT, peering, transit gateway.
Outputs	VPC ID, subnet IDs, route table IDs, CIDR blocks.

```
module "network" {
 source = "../../modules/network"

 project = var.project
 environment = var.environment
 vpc_cidr = "10.20.0.0/16"
 public_subnets = ["10.20.1.0/24", "10.20.2.0/24"]
 private_subnets = ["10.20.11.0/24", "10.20.12.0/24"]
 availability_zones = ["eu-west-3a", "eu-west-3b"]

 tags = local.common_tags
 }
```

5. Exemple de module compute / applicatif

Le module compute représente la couche qui exécute l'application : VM, autoscaling group, ECS service, EKS deployment, load balancer ou worker.

Composant	Rôle
Compute	EC2, ECS service, EKS workload, instance group.
Load balancer	Entrée HTTP/HTTPS, health checks, target groups.
Autoscaling	Adapter la capacité selon CPU, mémoire, trafic.
Logs	Configurer log group, rétention, format.
Monitoring	Alarmes 5xx, latence, CPU, mémoire, tâches unhealthy.

```
module "api_service" {
 source = "../../modules/compute-service"

 project = var.project
 environment = var.environment
 vpc_id = module.network.vpc_id
 subnet_ids = module.network.private_subnet_ids
 image = var.api_image
 desired_count = 3
 min_capacity = 2
 max_capacity = 8

 enable_public_load_balancer = true
 health_check_path = "/health/"

 tags = local.common_tags
```

```
}
```

6. Exemple de module database

Les modules database sont les plus sensibles : ils gèrent des ressources avec état, des sauvegardes, des secrets, des fenêtres de maintenance et parfois des contraintes fortes de disponibilité.

Paramètre	Pourquoi c'est important
engine / version	PostgreSQL, MySQL, MariaDB et version exacte.
instance class	Dimensionnement CPU/RAM.
storage	Capacité, type disque, autoscaling éventuel.
backup retention	Durée de conservation des sauvegardes.
multi_az	Résilience en cas de panne de zone.
deletion protection	Protection contre suppression accidentelle.
maintenance window	Contrôle du moment des opérations sensibles.

```
module "postgres" {
 source = "../../modules/database-postgres"

 project = var.project
 environment = var.environment
 subnet_ids = module.network.private_subnet_ids
 allowed_sg_ids = [module.api_service.service_security_group_id]

 engine_version = "15"
 instance_class = "db.t3.medium"
 allocated_storage_gb = 100
 backup_retention_days = 14
 multi_az = true
 deletion_protection = true

 tags = local.common_tags
 }
```

7. Sécurité des modules Terraform

Les modules imposent des standards. Ils doivent donc intégrer la sécurité par défaut : accès minimal, chiffrement, absence de secrets en clair, tags d'audit et garde-fous.

Règle	Exigence
Least privilege	Ne pas créer des IAM policies trop larges par défaut.
No public by default	Un module ne doit pas exposer Internet sauf choix explicite.
Encryption	Activer chiffrement pour stockage, logs et bases si disponible.
No hardcoded secrets	Pas de mot de passe, token ou clé dans le code Terraform.
Audit tags	Tags Owner, Environment, ManagedBy, CostCenter si besoin.
Safe defaults	Les defaults doivent favoriser sécurité et stabilité.

```
variable "allowed_cidr_blocks" {
 description = "CIDR blocks allowed to access the service."
 type = list(string)
 default = []

 validation {
 condition = length(var.allowed_cidr_blocks) > 0
 error_message = "At least one allowed CIDR block must be provided."
 }
 }
```

8. Versionner les modules Terraform

Dès qu'un module est utilisé par plusieurs environnements ou plusieurs équipes, il faut le versionner proprement. Sinon, une modification du module peut casser plusieurs stacks en même temps.

Source	Usage	Risque
Local path	Monorepo, projet simple, modules internes.	Pas de version explicite si même repo.
Git tag	Module partagé et versionné.	Nécessite release discipline.
Registry	Standard entreprise ou module public.	Gouvernance et compatibilité requises.

```
module "network" {
 source = "../../modules/network"

 project = var.project
 environment = var.environment
 }
```

1.4 - State Terraform

Backend distant, locking, dérive, isolation par environnement et sécurité.

Source modal: 1.4 State Terraform : backend, locking, drift et sécurité

1. Le state est la mémoire de Terraform

Le state Terraform est le fichier qui relie le code Terraform aux ressources réellement créées chez le provider. Il contient les IDs, attributs, dépendances, métadonnées et parfois certaines valeurs sensibles.

Sans state fiable, Terraform ne sait plus correctement ce qu'il possède. Il peut alors vouloir recréer, supprimer ou modifier des ressources de manière inattendue.

Élément	Exemple	Sensibilité
IDs provider	ID d'instance, ARN IAM, ID security group.	Moyenne
Attributs ressources	IP privée, endpoint DB, DNS load balancer.	Moyenne à forte
Dépendances	Relation entre réseau, compute et database.	Faible à moyenne
Valeurs sensibles	Mot de passe, token, secret selon provider/module.	Forte
Outputs	Valeurs exposées par output .	Variable

```
Code Terraform
  │
  │ décrit l'état souhaité
  ▼
State Terraform
  │
  │ mémorise les objets gérés
  ▼
Infrastructure réelle
  │
  │ AWS / Azure / GCP / GitLab / Kubernetes
  ▼
terraform plan compare les 3 dimensions
```

2. Backend distant : standard professionnel

En local, Terraform écrit par défaut un fichier terraform.tfstate . C'est acceptable pour apprendre, mais dangereux en équipe et interdit pour une production sérieuse. Un backend distant centralise le state, protège son accès et permet le verrouillage.

Mode	Usage	Limite
Local	Formation, test personnel, prototype jetable.	Risque de perte, conflit, commit accidentel.
S3 + DynamoDB	Standard fréquent sur AWS.	Nécessite sécurisation bucket/table.
GitLab managed state	Pratique avec GitLab CI/CD.	Dépend de la plateforme GitLab.
Terraform Cloud / Enterprise	Collaboration, policy, remote runs, audit.	Solution plus structurée, parfois payante.
Azure Storage / GCS	Backends cloud natifs Azure ou GCP.	À configurer avec IAM et chiffrement.

```
terraform {
 backend "s3" {
 bucket = "company-terraform-state-prod"
 key = "prod/network/terraform.tfstate"
 region = "eu-west-3"
 dynamodb_table = "terraform-locks"
 encrypt = true
 }
 }
```

3. Locking : empêcher les apply concurrents

Le locking empêche deux exécutions Terraform d'écrire simultanément dans le même state. Sans verrouillage, deux pipelines ou deux ingénieurs peuvent modifier le même state au même moment, avec un risque de corruption, de conflit ou de plan incohérent.

Situation	Risque
Deux apply en parallèle	Écriture concurrente dans le state.
Plan obsolète appliqué	Le plan ne reflète plus la réalité.
Pipeline relancé trop vite	État partiel ou changement déjà appliqué.
Force unlock abusif	Déverrouillage alors qu'un apply est encore actif.

```
Pipeline A démarre terraform apply
  │
  ▼
Lock du state obtenu
  │
  ├── Pipeline A peut lire / écrire
  │
```

```
└── Pipeline B doit attendre
  │
  ▼
erreur ou attente selon backend
  │
  ▼
Pipeline A termine
  │
  ▼
Lock libéré
  │
...
```

4. Isolation du state par environnement et domaine

L'isolation du state réduit le blast radius. Il ne faut pas tout mettre dans un seul state global, surtout en production. Un state trop gros rend les plans illisibles, ralentit les pipelines et augmente les risques d'incident.

State	Contenu	Fréquence de changement
prod/network	VPC, subnets, routes, NAT.	Rare
prod/security	IAM, KMS, security groups globaux.	Faible à moyenne
prod/app	Compute, load balancer, autoscaling.	Moyenne à forte
prod/data	Databases, caches, buckets critiques.	Rare et sensible
prod/monitoring	Dashboards, alarms, log groups.	Moyenne

```
Un seul state global
  │
  ├── dev
  ├── staging
  ├── prod
  ├── network
  ├── compute
  ├── database
  ├── IAM
  └── monitoring
```

5. Sécurité du state Terraform

Le state peut contenir des informations sensibles ou stratégiques : endpoints privés, structure réseau, IDs IAM, parfois tokens ou mots de passe selon les ressources. Il faut donc le protéger comme un secret opérationnel.

Risque	Conséquence	Protection
Commit Git	Fuite durable dans l'historique.	.gitignore , secret scanning, hooks.
Bucket trop ouvert	Lecture complète de l'infra.	IAM strict, bucket policy, audit logs.
Pas de chiffrement	Exposition en cas de fuite stockage.	Chiffrement côté backend.
Accès CI trop large	Pipeline compromis = state compromis.	Rôles dédiés, variables protégées, environnements protégés.
Outputs sensibles	Secrets visibles dans logs ou UI.	sensitive = true et outputs minimaux.

```
.terraform/
 *.tfstate
 *.tfstate.*
 crash.log
 crash.*.log
 *.tfvars
 *.tfvars.json
 override.tf
 override.tf.json
 *_override.tf
 *_override.tf.json
```

6. Drift : quand la réalité ne correspond plus au code

Le drift apparaît quand l'infrastructure réelle ne correspond plus au code Terraform ou au state. Il est souvent causé par une modification manuelle dans la console cloud, un hotfix urgent, une ressource supprimée hors Terraform ou un import incomplet.

Cause	Exemple	Risque
ClickOps	Modification console AWS hors Terraform.	Plan inattendu.
Hotfix urgence	Ouverture temporaire d'un security group.	Dette de sécurité.
Suppression externe	Ressource supprimée par erreur hors Terraform.	Recréation ou erreur apply.
Import incomplet	Ressource existante rattachée partiellement.	Diff permanent.
Provider change	Nouvelle version modifie attributs calculés.	Plan bruyant ou instable.

```
Code Terraform
 instance_type = "t3.small"

 State Terraform
 instance_type = "t3.small"

 Cloud réel
```

```
instance_type = "t3.medium"

Résultat :
terraform plan détecte un écart
et propose de revenir à t3.small
```

7. Import, state mv, state rm : opérations sensibles

Les commandes de manipulation du state sont puissantes et dangereuses. Elles servent à rattacher des ressources existantes, renommer des ressources dans le state, ou retirer une ressource du state sans la détruire réellement.

Commande	Usage	Danger
terraform import	Rattacher une ressource existante au state.	Import incomplet si le code ne correspond pas.
terraform state mv	Déplacer ou renommer une ressource dans le state.	Erreur de mapping.
terraform state rm	Retirer une ressource du state sans la supprimer.	Terraform ne la gère plus.
terraform state show	Inspecter une ressource connue.	Peut afficher données sensibles.
terraform state pull	Exporter le state courant.	Fichier sensible localement.

```
# 1. Écrire d'abord le bloc resource correspondant
 resource "aws_security_group" "web" {
 name = "prod-web-sg"
 vpc_id = var.vpc_id
 }

 # 2. Importer la ressource existante
 terraform import aws_security_group.web sg-0123456789abcdef0

 # 3. Lancer un plan pour aligner code et réalité
 terraform plan
```

8. Incidents liés au state : scénarios réels

Stopper les applies immédiatement.; Récupérer une version précédente du backend.; Comparer avec l'infrastructure réelle.; Restaurer le state si possible.

Étape	Réaction
Détection	Un fichier .tfstate apparaît dans le repo.
Risque	Fuite d'informations internes ou secrets.
Action immédiate	Supprimer du repo, bloquer push, analyser contenu.
Action sécurité	Rotation des secrets potentiellement exposés.
Prévention	.gitignore , secret scanning, formation équipe.

```
Pipeline échoue brutalement
 |
 ▼
Lock reste présent
 |
 ▼
Nouveau plan/apply impossible
 |
 ▼
Vérifier qu'aucun apply ne tourne
 |
 ▼
force-unlock uniquement si lock orphelin
 |
 ▼
relancer terraform plan
```

2.1 - GitLab CI/CD Infra

Stages fmt, validate, plan, review, apply et rollback contrôlé.

Source modal: 2.1 GitLab CI/CD Infra : pipeline Terraform professionnel

1. GitLab CI/CD côté infrastructure

GitLab CI/CD permet d'industrialiser les changements d'infrastructure : formatage, validation, scan sécurité, génération du plan Terraform, revue humaine, approval, apply contrôlé et vérifications post-déploiement.

Le but n'est pas seulement d'automatiser. Le but est de rendre chaque changement reproductible , traçable , reviewable et limité en risque .

Besoin infra	Réponse GitLab CI/CD
Standardiser les commandes	Les mêmes jobs exécutent toujours fmt , validate , plan .
Tracer les changements	Chaque pipeline est lié à un commit, une branche, une MR et un utilisateur.
Contrôler la production	Branches protégées, environnements protégés, jobs manuels, approvals.
Sécuriser les secrets	Variables masked/protected, scopes par environnement.
Conserver les preuves	Artefacts de plan, logs de pipeline, historique d'exécution.
Réduire les erreurs humaines	Moins de commandes manuelles locales, moins de divergence entre postes.

```
Développeur / DevOps
 │
 ▼
 Branche Git
 │
 ▼
 Merge Request
 │
 ├── terraform fmt
 ├── terraform validate
 ├── scans sécurité
 ├── terraform plan
 └── artefact du plan
 │
 ▼
 Revue humaine
 │
 ▼
...
```

2. Pipeline infra recommandé

Un pipeline infrastructure sérieux doit séparer les étapes de qualité, sécurité, plan, revue, apply et vérification. Le plan doit être visible avant toute modification réelle.

Stage	But	Bloquant ?
fmt	Vérifier le format Terraform.	Oui
validate	Valider syntaxe, providers, modules.	Oui
security	Détecter ports ouverts, IAM trop larges, secrets.	Oui ou warning selon maturité.
plan	Afficher exactement ce que Terraform veut changer.	Oui
review	Relire le plan et décider.	Oui pour prod
apply	Modifier réellement l'infrastructure.	Manuel/protégé pour prod
post_check	Vérifier santé réelle après changement.	Oui si critique

```
Commit / Merge Request
 │
 ▼
 lint
 ├── terraform fmt
 └── tflint
 │
 ▼
 validate
 ├── terraform init
 └── terraform validate
 │
 ▼
 security
 ├── checkov
 ├── tfsec / trivy config
 └── secret detection
 │
...
```

3. Exemple complet de .gitlab-ci.yml pour Terraform

Exemple pédagogique : pipeline avec formatage, validation, scan sécurité, plan archivé et apply manuel en production.

Élément	Rôle
TF_IN_AUTOMATION	Indique à Terraform qu'il tourne dans une CI.
TF_INPUT=false	Évite les prompts interactifs bloquants.
artifacts	Conserve le plan binaire et le plan lisible.
when: manual	Force un clic humain pour appliquer en production.
environment	Lie le job à un environnement GitLab protégé.

```
stages:
 - lint
 - validate
 - security
 - plan
 - apply
 - post_check

 variables:
 TF_IN_AUTOMATION: "true"
 TF_INPUT: "false"
 TF_ROOT: "infra/live/prod/app"
 TF_PLAN_FILE: "tfplan"

 default:
 image: hashicorp/terraform:1.6.6
 before_script:
 - cd "$TF_ROOT"
...
```

4. GitLab Runners : exécuter les pipelines proprement

Un runner GitLab exécute les jobs CI/CD. Côté infrastructure, il est très sensible, car il peut avoir accès à des credentials cloud, au state Terraform et parfois à des environnements critiques.

Type	Usage	Risque
Shared runner	Jobs génériques, projets peu sensibles.	Moins de contrôle sur l'environnement.
Group runner	Standard d'équipe ou de département.	Attention aux droits transverses.
Project runner	Projet spécifique, meilleur cloisonnement.	Maintenance dédiée.
Protected runner	Branches/tags protégés seulement.	Recommandé pour production.
Self-hosted runner	Contrôle total réseau, IAM, packages.	Responsabilité sécurité plus forte.

```
terraform_apply_prod:
 stage: apply
 tags:
 - terraform
 - prod
 - protected
 script:
 - terraform apply -auto-approve tfplan
```

5. Variables GitLab, secrets et environnements

Les variables CI/CD servent à injecter credentials cloud, paramètres Terraform, chemins, tokens, noms d'environnement et options d'exécution. Leur mauvaise gestion est l'une des causes majeures d'incidents ou de fuites.

Type	Usage	Protection recommandée
Cloud credentials	Accès AWS, Azure, GCP.	Masked + protected + scope environnement.
Terraform vars	TF_VAR_* pour injecter variables Terraform.	Protected si prod.
Backend config	Bucket, key, région, workspace.	Variables non sensibles mais contrôlées.
Tokens outils	Checkov, Vault, GitLab API, registry.	Masked + rotation régulière.
Feature flags	Activer scan, debug, apply auto dev.	Selon contexte.

```
# Variable GitLab CI/CD
 TF_VAR_project=ideo-platform
 TF_VAR_environment=prod
 TF_VAR_aws_region=eu-west-3

 # Terraform la récupère automatiquement :
 variable "project" {
 type = string
 }
```

6. Review du plan Terraform

La revue du plan est l'étape la plus importante avant un apply. Le pipeline doit produire un plan lisible, stable et archivé pour que l'équipe puisse décider en connaissance de cause.

Point	Question à poser	Danger
Destroy	Y a-t-il des suppressions ?	Perte de service ou données.
Replace	Une ressource est-elle recréée ?	Downtime, perte d'IP, remplacement DB.
Security group	Un port est-il ouvert trop largement ?	Exposition sécurité.
IAM	Une policy donne-t-elle trop de droits ?	Escalade de privilèges.
Database	Backup, storage, deletion protection changent-ils ?	Risque data.
Scope	Le plan concerne-t-il le bon environnement ?	Erreur dev/prod.
Secrets	Une valeur sensible apparaît-elle ?	Fuite logs/artifacts.

```
Plan généré
 │
 ▼
Revue automatique
 ├── fmt
 ├── validate
 ├── security
 └── policy
 │
 ▼
Revue humaine
 ├── scope
 ├── destroy / replace
 ├── réseau / IAM
 ├── database
 └── rollback
 │
 ▼
...
```

7. Apply protégé en production

L'étape apply modifie réellement l'infrastructure. En production, elle doit être limitée, tracée, contrôlée et idéalement exécutée depuis un environnement GitLab protégé.

Garde-fou	But
Protected branch	Empêcher un apply depuis une branche non validée.
Protected environment	Limiter qui peut déployer en production.
Manual job	Forcer une décision humaine.
Approvals	Exiger validation d'un ou plusieurs responsables.
Protected variables	Rendre les credentials prod disponibles uniquement dans le bon contexte.
Runner dédié	Réduire la surface d'attaque.

```
terraform_apply_prod:
 stage: apply
 environment:
 name: production
 script:
 - terraform apply -auto-approve "$TF_PLAN_FILE"
 dependencies:
 - terraform_plan
 when: manual
 allow_failure: false
 rules:
 - if: '$CI_COMMIT_BRANCH == "main"'
 tags:
 - terraform
 - prod
 - protected
```

8. Rollback et mitigation côté infrastructure

Le rollback Terraform n'est pas toujours aussi simple qu'un rollback applicatif. Certains changements peuvent être inversés en revenant au commit précédent, mais d'autres nécessitent une mitigation spécifique, surtout pour les bases de données, réseaux et IAM.

Situation	Réaction possible	Risque
Security group trop restrictif	Revenir au commit précédent ou hotfix Terraform.	Service inaccessible.
IAM policy cassée	Restaurer policy précédente.	Jobs/applications bloqués.
Load balancer mal configuré	Revenir listener/target group.	Downtime web.
Database modifiée	Mitigation, restore ou correction manuelle contrôlée.	Risque data élevé.
Ressource supprimée	Recréation ou restauration backup.	Perte possible si state/data mal protégés.

```
Incident après apply
 │
 ▼
```

```
Identifier commit fautif
|
▼
Créer revert commit
|
▼
Pipeline terraform plan
|
▼
Relire plan de rollback
|
▼
Apply contrôlé
|
▼
...
```

2.2 - Environnements

Dev, staging, prod : variables protégées, branches, approvals et séparation.

Source modal: 2.2 Environnements : dev, staging, prod et séparation professionnelle

1. Pourquoi séparer les environnements ?

Dev, staging et production n'ont pas les mêmes objectifs, les mêmes données, les mêmes contraintes de sécurité, ni le même niveau de risque. Les mélanger dans le code, les variables, les credentials ou le state Terraform est une source classique d'incident.

Une bonne séparation permet de tester vite en dev, de valider sérieusement en staging et de protéger fortement la production.

Environnement	Objectif	Niveau de contrôle
Dev	Expérimenter, tester rapidement, casser sans impact client.	Faible à moyen, apply parfois semi-automatique.
Review app	Créer un environnement temporaire pour une merge request.	Automatique, durée limitée, ressources jetables.
Staging	Valider avant production dans un contexte réaliste.	Plan, revue, apply contrôlé.
Preprod	Répétition quasi identique à la production.	Contrôle fort, proche prod.
Prod	Servir les vrais utilisateurs et données réelles.	Plan, approval, fenêtre, rollback, monitoring.

```
Branches Git
  │
  ├── feature/*
  │   └── dev / review app
  │
  ├── release/*
  │   └── staging / preprod
  │
  └── main
  └── production

 Chaque environnement possède :
 ├── variables dédiées
 ├── credentials dédiés
 ├── state Terraform dédié
 ├── pipeline adapté
 ├── droits d'accès spécifiques
 └── politique d'approval différente
```

2. Modèle cible professionnel

Un modèle mature définit clairement le rôle de chaque environnement, son niveau de ressemblance avec la production, ses droits, sa stratégie de données et sa politique de déploiement.

Env	Infrastructure	Données	Déploiement
Dev	Réduite, économique, flexible.	Fake data ou anonymisée.	Rapide, parfois automatique.
Review	Temporaire, détruite après MR.	Jeu minimal ou fixtures.	Automatique par MR.
Staging	Proche prod mais plus petit.	Anonymisée ou snapshot contrôlé.	Manuel ou semi-automatique.
Preprod	Très proche prod.	Anonymisée, volume représentatif.	Répétition avant prod.
Prod	Haute disponibilité, monitoring complet.	Données réelles.	Manuel, protégé, audité.

```
project = "ideo-platform"
 environment = "prod"
 region = "eu-west-3"

 name_prefix = "${project}-${environment}"

 Exemples :
 ideo-platform-dev-api
 ideo-platform-staging-api
 ideo-platform-prod-api
```

3. Mapping GitLab : branches, environnements et pipelines

GitLab permet d'associer des branches, jobs, variables, runners et approvals à des environnements. C'est essentiel pour empêcher une branche non contrôlée d'accéder aux credentials de production.

Git	Environnement	Pipeline	Droits
feature/*	dev ou review app	fmt, validate, plan dev	Pas de secrets prod
develop	dev partagé	plan/apply dev possible	Credentials dev
release/*	staging	plan + apply manuel staging	Credentials staging

Git	Environnement	Pipeline	Droits
main	production	plan prod + apply manuel	Credentials prod protégés
tag v*	release production	déploiement versionné	Approval fort

```
terraform_plan_prod:
 stage: plan
 environment:
 name: production
 script:
 - terraform plan -var-file="prod.tfvars"
 rules:
 - if: '$CI_COMMIT_BRANCH == "main"'

 terraform_apply_prod:
 stage: apply
 environment:
 name: production
 when: manual
 script:
 - terraform apply tfplan
 rules:
 - if: '$CI_COMMIT_BRANCH == "main"'
```

4. Variables protégées et secrets par environnement

Les variables ne doivent pas être globales si elles donnent accès à des environnements différents. Un secret production doit être disponible uniquement dans un job production, sur branche protégée, avec runner protégé.

Variable	Scope	Protection
AWS_ACCESS_KEY_ID_DEV	dev	Non prod, droits limités.
AWS_ACCESS_KEY_ID_STAGING	staging	Protected si branche release protégée.
AWS_ACCESS_KEY_ID_PROD	production	Masked + protected + environment scoped.
TF_VAR_db_password	Selon env	Secret manager préférable.
TF_VAR_instance_type	Selon env	Non secret, mais contrôlé.

```
# dev.tfvars
 environment = "dev"
 instance_type = "t3.micro"
 min_capacity = 1
 max_capacity = 2

 # staging.tfvars
 environment = "staging"
 instance_type = "t3.small"
 min_capacity = 1
 max_capacity = 3

 # prod.tfvars
 environment = "prod"
 instance_type = "t3.medium"
 min_capacity = 2
 max_capacity = 8
```

5. State Terraform séparé par environnement

Chaque environnement doit avoir son propre state. Mélanger dev, staging et production dans un même state augmente fortement le risque d'incident et rend les plans difficiles à relire.

Éviter autant que possible que dev lise prod. Si un state doit lire un autre state, le couplage doit être volontaire, documenté et limité à des outputs non sensibles.

Domaine	Raison
network	Change rarement, impact fort.
app	Change plus souvent, impact applicatif.
data	Très sensible, protections renforcées.
monitoring	Peut évoluer sans toucher au compute.
security	IAM et règles critiques à isoler.

```
# Dev
 key = "dev/network/terraform.tfstate"
 key = "dev/app/terraform.tfstate"
 key = "dev/data/terraform.tfstate"

 # Staging
 key = "staging/network/terraform.tfstate"
 key = "staging/app/terraform.tfstate"
 key = "staging/data/terraform.tfstate"

 # Production
 key = "prod/network/terraform.tfstate"
 key = "prod/app/terraform.tfstate"
 key = "prod/data/terraform.tfstate"
```

6. Approvals et gouvernance par environnement

Les approvals permettent d'adapter le niveau de contrôle au risque. La production exige un niveau de validation plus fort que dev ou staging.

Env	Qui valide ?	Quand ?
Dev	Développeur ou DevOps responsable.	Si changement non sensible.
Staging	DevOps + tech lead si impact significatif.	Avant validation release.
Preprod	Tech lead, QA, DevOps.	Avant répétition prod.
Prod	DevOps senior, responsable plateforme, owner métier si nécessaire.	Avant apply.
Prod critique	Double approval + fenêtre de changement.	Changement réseau, DB, IAM, sécurité.

```
Merge Request
 │
 ▼
 Pipeline plan production
 │
 ▼
 Plan lisible en artifact
 │
 ▼
 Review technique
 ├── ressources créées
 ├── ressources modifiées
 ├── destroy / replace
 ├── sécurité
 └── rollback
 │
 ▼
 Approval
...
```

7. Promotion dev → staging → production

Une bonne stratégie ne consiste pas à bricoler directement en production. Le changement doit progresser de dev vers staging, puis vers production, avec validation à chaque étape.

Bénéfice	Explication
Détection précoce	Les erreurs apparaissent avant la production.
Plan plus fiable	Le changement a déjà été observé ailleurs.
Confiance équipe	Moins de surprise au moment du prod apply.
Rollback préparé	Les impacts sont mieux compris.

```
Feature branch
 │
 ├── fmt / validate / plan dev
 │
 ▼
 Merge Request
 │
 ├── review code
 ├── scan sécurité
 └── review plan
 │
 ▼
 Dev apply
 │
 ▼
 Staging apply
 │
 ├── tests fonctionnels
...
```

8. Données par environnement

La séparation des données est aussi importante que la séparation de l'infrastructure. Les données de production ne doivent pas être copiées en dev sans anonymisation et contrôle d'accès.

Env	Données recommandées	Risques
Dev local	Fixtures, données synthétiques.	Faible si pas de données réelles.
Dev partagé	Données anonymisées ou minimales.	Fuite si données réelles.
Staging	Snapshot anonymisé représentatif.	Fausse validation si données trop petites.
Preprod	Données proches prod mais anonymisées.	Conformité, confidentialité.
Prod	Données réelles.	Impact client, légal, business.

```
email:
 john.smith@example.com -> user_12345@example.test

 phone:
 +33612345678 -> +33000000000

 name:
 John Smith -> User 12345

 address:
```

```
10 Real Street -> 1 Test Street

free text:
supprimer ou remplacer si contient données personnelles
```

2.3 - Secrets & sécurité CI

Variables masquées, scopes, credentials courts, policy least privilege.

Source modal: 2.3 Secrets & sécurité CI : variables, OIDC, IAM et moindre privilège

1. Pourquoi les secrets CI/CD sont critiques

Un pipeline CI/CD infrastructure peut créer, modifier ou supprimer des ressources cloud. Les secrets qu'il utilise - clés cloud, tokens API, mots de passe, certificats, clés SSH, tokens GitLab, credentials registry - doivent être protégés comme des accès production.

Une fuite de secret dans une pipeline peut donner à un attaquant la capacité de lire un state Terraform, modifier une infrastructure, pousser une image, accéder à une base de données ou supprimer des ressources.

Secret	Exemple	Risque si fuite
Cloud credentials	AWS, Azure, GCP, Hetzner.	Contrôle infrastructure.
Terraform backend	Accès bucket state, GitLab state, Terraform Cloud token.	Lecture/modification du state.
Registry	Docker registry, GitLab registry, ECR.	Push d'images compromises.
SSH / deploy key	Clé privée serveur ou repo.	Accès serveur ou code source.
Database	Mot de passe PostgreSQL/MySQL/Redis.	Lecture ou modification de données.
API token	Cloudflare, Datadog, GitLab, Slack, Sentry.	Modification DNS, monitoring, notifications, projets.

```
Développeur
 │
 ▼
Merge Request
 │
 ├── pas d'accès secret prod
 ├── scan secret detection
 └── plan limité
 │
 ▼
Branche protégée
 │
 ▼
Runner protégé
 │
 ├── variables masked/protected/scoped
 ├── credentials courts ou OIDC
 ├── IAM least privilege
...
```

2. Variables GitLab : masked, protected, scoped

GitLab CI/CD permet de stocker des variables utilisées par les jobs. Pour les secrets, il faut utiliser les options de protection adaptées : masquage dans les logs, accès limité aux branches protégées et scope par environnement.

Option	Rôle	À utiliser pour
Masked	Masque la valeur dans les logs.	Tokens, clés, mots de passe.
Protected	Disponible seulement pour branches/tags protégés.	Credentials production.
Environment scope	Limite une variable à un environnement GitLab.	Dev, staging, production.
File variable	Crée un fichier temporaire avec le contenu secret.	Kubeconfig, certificat, clé privée.
Group variable	Partage une variable sur plusieurs projets.	Standards d'équipe.
Project variable	Limite au projet courant.	Secret spécifique applicatif.

```
# Dev
 AWS_ACCESS_KEY_ID_DEV
 AWS_SECRET_ACCESS_KEY_DEV

 # Staging
 AWS_ACCESS_KEY_ID_STAGING
 AWS_SECRET_ACCESS_KEY_STAGING

 # Production
 AWS_ACCESS_KEY_ID_PROD
 AWS_SECRET_ACCESS_KEY_PROD

 # Terraform variables
 TF_VAR_project
 TF_VAR_environment
 TF_VAR_db_password
```

3. Scopes : limiter où un secret peut être utilisé

Un secret doit être disponible uniquement dans les contextes nécessaires. Plus son scope est large, plus le risque est grand. Une variable production globale est dangereuse : elle peut être utilisée par erreur par un job qui ne devrait jamais agir sur prod.

Secret	Scope	Disponible pour
AWS_ACCESS_KEY_ID_DEV	development	Jobs dev uniquement.
AWS_ACCESS_KEY_ID_STAGING	staging	Jobs staging uniquement.
AWS_ACCESS_KEY_ID_PROD	production	Jobs production uniquement.
GITLAB_TOKEN_READONLY	global ou projet	Lecture API limitée.
REGISTRY_PUSH_TOKEN	build	Jobs build/publish.

```
feature/*
  │
  └── pas de secrets prod

 develop
  │
  └── secrets dev

 release/*
  │
  └── secrets staging

 main
  │
  └── secrets production
  ├── masked
  ├── protected
  └── environment scoped
```

4. OIDC : remplacer les clés statiques par des credentials temporaires

OIDC permet à GitLab CI/CD de demander un jeton d'identité temporaire, puis de l'échanger contre des credentials cloud temporaires. Cela évite de stocker des clés cloud longues durées dans GitLab.

Approche	Avantage	Risque
Clé statique	Simple à configurer.	Fuite durable si compromise.
OIDC	Credentials temporaires, contextuels, révocables.	Configuration initiale plus avancée.
Vault dynamique	Secrets courts et centralisés.	Nécessite plateforme Vault.

```
GitLab job
  │
  ▼
 Demande un ID token OIDC
  │
  ▼
 Cloud IAM vérifie :
  ├── projet GitLab
  ├── branche
  ├── environnement
  ├── audience
  └── claims autorisés
  │
  ▼
 Cloud émet credentials temporaires
  │
  ▼
 Terraform plan/apply
...
```

5. IAM et moindre privilège

Le compte ou rôle utilisé par la CI/CD ne doit avoir que les permissions nécessaires. Une erreur de pipeline ne doit pas pouvoir supprimer tout le compte cloud, lire tous les secrets ou modifier tous les environnements.

Rôle	Droits	Usage
plan-dev	Lecture + droits limités dev.	Plan et tests dev.
apply-dev	Écriture dev limitée.	Apply dev.
plan-prod	Lecture production.	Générer plan production.
apply-prod	Écriture production limitée aux ressources nécessaires.	Apply manuel protégé.
break-glass	Droits élevés temporaires.	Incident majeur, audit obligatoire.

```
{
 "Effect": "Allow",
 "Action": "*",
 "Resource": "*"
 }
```

6. Vault, Secret Manager et rotation

Les variables GitLab sont utiles, mais pour des organisations plus matures, un gestionnaire de secrets centralisé permet de contrôler la rotation, les accès, l'audit, les credentials dynamiques et la révocation.

Solution	Usage	Points forts
HashiCorp Vault	Secrets centralisés, credentials dynamiques.	Très puissant, multi-cloud, audit.
AWS Secrets Manager	Secrets applicatifs et rotation AWS.	Intégration AWS native.
AWS SSM Parameter Store	Paramètres et secrets simples.	Simple, économique, IAM natif.
Azure Key Vault	Secrets, certificats, clés Azure.	Intégration Azure.
GCP Secret Manager	Secrets GCP versionnés.	Intégration IAM GCP.
GitLab CI variables	Secrets pipeline simples.	Pratique, intégré CI/CD.

```
GitLab Runner
  │
  ├── s'authentifie via OIDC / rôle court
  │
  ▼
Secret Manager / Vault
  │
  ├── vérifie identité
  ├── vérifie policy
  ├── journalise l'accès
  └── retourne secret ou credential temporaire
  │
  ▼
Job CI/CD
  │
  └── utilise le secret sans le stocker dans Git
```

7. Logs CI : surface d'exposition majeure

Les logs CI sont souvent conservés, partagés, consultables par plusieurs personnes et parfois exportés vers des systèmes externes. Ils ne doivent jamais contenir de secrets.

Risque	Exemple	Prévention
Echo secret	echo $TOKEN	Interdire affichage secrets.
Mode debug	set -x	Désactiver autour des commandes sensibles.
Terraform output	Output contenant mot de passe.	sensitive = true , outputs minimaux.
Erreur outil	Stack trace imprimant config.	Tester outils, filtrer logs, limiter secrets.
Artifact involontaire	Archive contenant .env ou state.	Artifacts explicitement listés.

```
# Mauvais : affiche le secret
 script:
 - echo "$AWS_SECRET_ACCESS_KEY"

 # Mauvais : debug shell trop bavard
 script:
 - set -x
 - terraform plan

 # Mauvais : dump d'environnement
 script:
 - env
 - printenv
```

8. Terraform, state et secrets

Terraform peut manipuler des valeurs sensibles, mais il faut comprendre une limite majeure : sensitive = true masque l'affichage, mais ne garantit pas que la valeur soit absente du state. Le state doit donc être protégé.

Remote state	Chiffrement, IAM strict, audit, versioning.
Artifacts plan	Ne pas archiver des plans contenant des secrets non maîtrisés.
tfvars	Ne pas commiter les fichiers contenant secrets.
Outputs	Limiter au strict nécessaire.

```
variable "database_password" {
 description = "Database password."
 type = string
 sensitive = true
 }
```

2.4 - Déploiement infra

Workflow propre : changement, revue, plan, fenêtre de prod, apply et vérification.

Source modal: 2.4 Déploiement infra : changement, revue, fenêtre, apply et vérification

1. Déployer de l'infrastructure n'est pas "juste cliquer Apply"

Un déploiement infrastructure modifie la fondation sur laquelle repose l'application : réseau, sécurité, compute, base de données, DNS, certificats, IAM, monitoring ou stockage. Une erreur peut provoquer une indisponibilité, une faille de sécurité ou une perte de données.

Le workflow professionnel consiste à préparer le changement, générer un plan, le faire relire, choisir une fenêtre adaptée, appliquer de façon contrôlée, puis vérifier l'état réel de la production.

Étape	But	Sortie attendue
Préparation	Comprendre besoin, impact, risque.	Ticket ou change request clair.
Développement	Modifier Terraform ou pipeline.	Branche Git dédiée.
Validation	fmt, validate, security, plan.	Pipeline vert + plan lisible.
Review	Relire le plan et les risques.	Approval ou corrections.
Apply	Appliquer dans le bon contexte.	Infra modifiée proprement.
Vérification	Contrôler cloud, app, logs, métriques.	Validation post-changement.

```
Besoin métier / technique
 │
 ▼
 Change request
 │
 ├── impact
 ├── risque
 ├── rollback
 └── fenêtre éventuelle
 │
 ▼
 Branche Git
 │
 ▼
 Merge Request
 │
 ├── terraform fmt
 ├── terraform validate
...
```

2. Workflow propre de déploiement infrastructure

Créer une demande de changement ou ticket technique.; Décrire le besoin, le périmètre et le risque.; Créer une branche Git dédiée.; Modifier le code Terraform ou la configuration CI/CD.

Acteur	Responsabilité
Auteur	Prépare changement, plan, rollback et documentation.
Reviewer DevOps	Relit Terraform, plan, state, impact et sécurité.
Tech lead	Valide impact applicatif ou architecture.
Security	Valide IAM, exposition réseau, secrets si nécessaire.
Incident/on-call	Surveille après apply si changement critique.

```
MR pipeline
 │
 ├── lint
 ├── validate
 ├── security
 └── plan
 │
 ▼
 Review humaine
 │
 ▼
 Merge vers main
 │
 ▼
 Production pipeline
 │
 ├── plan final
 ├── approval
...
```

3. Change request : préparer le changement

Avant de modifier la production, il faut savoir ce qui change, pourquoi, quand, comment vérifier, comment revenir en arrière et qui doit être informé.

Niveau	Exemples	Contrôle
Faible	Tags, dashboard, alarme non critique.	Review simple.
Moyen	Autoscaling, security group interne, compute.	Review + staging.
Élevé	Load balancer, DNS, IAM, réseau.	Approval + fenêtre.
Critique	Database, state, suppression, changement CIDR.	Double approval + runbook + présence on-call.

```
Titre :
 Ajouter un nouveau target group pour l'API publique

 Objectif :
 Préparer le routage vers la nouvelle version API.

 Environnement :
 production

 Périmètre :
 Load balancer, listener rule, target group, health check.

 Risque :
 Mauvais routage ou health check incorrect.

 Plan de validation :
 - terraform plan relu
 - health check /health/
...
```

4. Plan Terraform : l'étape de vérité

Le plan Terraform est le document de décision. Il montre ce qui va être créé, modifié, remplacé ou supprimé. En production, il doit être relu comme un changement réel, pas comme un simple log technique.

Symbole	Signification	Réflexe production
+	Création.	Vérifier nommage, tags, coût, sécurité.
~	Modification en place.	Comprendre l'impact runtime.
-/+	Remplacement.	Analyse obligatoire, risque downtime.
-	Suppression.	Stop si non explicitement attendu.

```
terraform fmt -check -recursive
 terraform validate
 terraform plan -var-file="prod.tfvars" -out="tfplan"
 terraform show -no-color tfplan > plan.txt
```

5. Fenêtre de production

Tous les changements n'exigent pas une fenêtre de production stricte, mais les changements à risque doivent être faits à un moment où l'équipe peut surveiller, réagir et communiquer.

Changement	Fenêtre recommandée ?	Pourquoi
Tags ou dashboard	Non en général.	Impact faible.
Security group interne	Selon criticité.	Peut couper flux applicatifs.
Load balancer / DNS	Oui.	Impact trafic utilisateur.
IAM production	Oui.	Peut bloquer application ou pipeline.
Database	Oui, obligatoire si risque.	Données, downtime, performance.
Réseau VPC / routes	Oui.	Blast radius élevé.

```
T-30 min
  ├── vérifier pipeline vert
  ├── ouvrir dashboards
  ├── confirmer présence équipe
  └── annoncer début fenêtre

 T0
  ├── déclencher apply
  └── surveiller logs Terraform

 T+5 min
  ├── vérifier cloud resources
  ├── health checks
  └── métriques initiales

 T+15 min
  ├── vérifier erreurs 5xx
  ├── vérifier latence
...
```

6. Apply contrôlé

L'apply doit exécuter le plan validé, dans le bon environnement, avec le bon state, les bons credentials et les bons garde-fous.

Appliquer le fichier tfplan évite qu'un plan différent soit recalculé au moment de l'apply.

Contrôle	Validation
Branche	La branche autorisée est utilisée.
Environnement	Production/staging/dev clairement affiché.
State	Backend distant et locking actif.
Plan	Plan relu, archivé, non obsolète.
Credentials	Rôle CI/CD limité et prévu.
Monitoring	Dashboards ouverts.
Rollback	Procédure disponible.

```
terraform plan -out="tfplan" -var-file="prod.tfvars"
 terraform apply "tfplan"
```

7. Vérification post-apply

Un apply terminé sans erreur ne prouve pas que la production va bien. Il faut vérifier l'état réel : cloud, application, logs, métriques, alerting et parfois parcours métier.

Zone	Contrôle	Signal attendu
Cloud	Ressources créées/modifiées visibles.	État healthy / active.
Application	Healthcheck HTTP.	200 OK ou statut attendu.
Logs	Erreurs 5xx, exceptions, timeouts.	Pas d'anomalie nouvelle.
Metrics	CPU, RAM, latence, saturation, erreurs.	Stable ou attendu.
Database	Connexions, locks, réplication, espace disque.	Pas de dégradation.
Alerting	Alertes critiques nouvelles.	Aucune alerte inattendue.
Business	Login, checkout, API critique, batch.	Parcours OK.

```
curl -fsS https://example.com/health/
 curl -fsS https://example.com/api/status/

 terraform output
 terraform state list

 # Exemples Linux
 journalctl -u app.service -n 100 --no-pager
 tail -n 100 /var/log/nginx/error.log
```

8. Rollback et mitigation infrastructure

Le rollback infra doit être anticipé avant l'apply. Dans certains cas, revenir au commit précédent suffit. Dans d'autres cas, il faut une mitigation : restaurer un flux réseau, remettre une règle, basculer DNS, désactiver une route ou restaurer un backup.

Changement	Rollback typique	Attention
Security group	Restaurer règle précédente.	Ne pas rouvrir trop large.
DNS	Revenir à l'ancien record.	TTL et propagation.
Load balancer	Restaurer listener/target group précédent.	Health checks.
IAM	Restaurer policy précédente.	Propagation et permissions.
Database	Mitigation ou restore backup.	Risque data élevé.
State	Restaurer version state si nécessaire.	Opération très sensible.

```
git revert <commit_id>
 git push origin main

 # Pipeline :
 terraform plan -out=tfplan
 # Review du plan de rollback
 terraform apply tfplan
```

3.1 - Observabilité

Logs, métriques, alertes, traces et tableaux de bord pour piloter la production.

Source modal: 3.1 Observabilité : logs, métriques, traces, alertes et dashboards

1. L'observabilité n'est pas seulement du monitoring

Le monitoring répond à la question : "est-ce que ça va ?" . L'observabilité répond à une question plus profonde : "pourquoi ça ne va pas, où, depuis quand, et avec quel impact ?" .

En production, l'observabilité permet de diagnostiquer rapidement les incidents, vérifier un déploiement, comprendre les performances, réduire le MTTR et piloter la fiabilité réelle du service.

Question	Signal utile
Le service est-il disponible ?	Health checks, uptime, taux de succès HTTP.
Le service est-il lent ?	Latence p50, p95, p99, temps DB, temps externe.
Le service échoue-t-il ?	5xx, exceptions, erreurs applicatives, failed jobs.
Quelle ressource sature ?	CPU, RAM, disque, I/O, connexions DB, queue depth.
Quel changement a déclenché l'incident ?	Déploiements, Terraform apply, feature flags, logs audit.
Quel utilisateur ou service est impacté ?	Logs structurés, traces, labels, dimensions métier.

```
Production system
  │
  ├── Application
  ├── Reverse proxy / Load balancer
  ├── Database
  ├── Cache
  ├── Workers / queues
  ├── Cloud resources
  └── CI/CD / Terraform
  │
  ▼
 Telemetry
  ├── logs
  ├── metrics
  ├── traces
  ├── events
  └── audit trail
  │
...
```

2. Les piliers de l'observabilité

Ce que les systèmes déclarent : requêtes, erreurs, exceptions, événements métier, actions utilisateur, changements d'état.

Ce que les systèmes mesurent : CPU, RAM, disque, latence, 5xx, throughput, saturation, temps de réponse.

Pilier	Usage	Exemple
Events	Marquer les changements importants.	Déploiement, Terraform apply, scaling event.
Audit logs	Tracer les actions sensibles.	Qui a modifié IAM, security group, secret.
SLO / SLI	Mesurer la fiabilité du point de vue utilisateur.	99.9% de requêtes API réussies sous 300 ms.
Dashboards	Voir rapidement l'état de production.	Golden signals, infra, DB, CI/CD.
Alerting	Notifier quand une action humaine est nécessaire.	Erreur 5xx élevée pendant 5 minutes.

```
Incident API lent
  │
  ├── Metrics
  │   └── p95 latency augmente
  │
  ├── Logs
  │   └── slow queries visibles
  │
  ├── Traces
  │   └── temps passé dans PostgreSQL
  │
  ├── Events
  │   └── déploiement 10 min avant
  │
  └── Audit
      └── changement paramètre DB
```

3. Logs : rendre les événements exploitables

Les logs doivent aider à comprendre ce qui s'est passé. En production, des logs utiles sont structurés, filtrables, corrélables et suffisamment précis sans exposer de secrets.

Source	Exemples	Utilité
Application	Exceptions, erreurs métier, login, jobs.	Comprendre le comportement applicatif.
Nginx / proxy	Access logs, status code, latence, IP.	Analyser trafic et erreurs HTTP.
System	systemd, kernel, auth, disk.	Diagnostiquer serveur Linux.
Database	Slow queries, locks, connexions, erreurs.	Identifier saturation ou requêtes lentes.
CI/CD	Pipeline failed, Terraform apply, déploiements.	Relier incident à changement récent.
Cloud audit	IAM changes, SG changes, console actions.	Détecter changements manuels ou suspects.

```
{
 "level": "error",
 "service": "api",
 "environment": "prod",
 "request_id": "req-8f31",
 "trace_id": "tr-91ab",
 "user_id": "u_12345",
 "route": "/api/orders",
 "status_code": 500,
 "duration_ms": 842,
 "error": "database_timeout"
 }
```

4. Métriques : mesurer l'état réel du système

Les métriques permettent de détecter les tendances, saturations, régressions et incidents. Elles sont indispensables pour les dashboards, alertes, SLO et analyses post-incident.

Signal	Signification	Exemples
Latency	Temps de réponse ressenti.	p50, p95, p99 par endpoint.
Traffic	Volume de demandes.	RPS, jobs/min, messages queue.
Errors	Taux d'échec.	5xx, exceptions, failed jobs.
Saturation	Ressources proches de la limite.	CPU, RAM, disque, connexions DB.

```
Métrique collectée
 │
 ▼
 Dashboard
 │
 ├── tendance normale
 ├── pic temporaire
 └── anomalie durable
 │
 ▼
 Alerte si seuil + durée + impact
 │
 ▼
 Investigation logs / traces
 │
 ▼
 Action : scale, rollback, fix, mitigation
```

5. Traces distribuées : comprendre le chemin d'une requête

Les traces montrent le parcours d'une requête entre services : frontend, API, base de données, cache, services externes, workers. Elles sont essentielles dans les architectures microservices ou les applications avec beaucoup de dépendances.

Signal	Interprétation
Span très long	Service ou requête DB lente.
Erreur sur span externe	Dépendance tierce en échec.
Beaucoup de spans DB	Possible problème N+1 queries.
Trace sans request ID	Corrélation logs difficile.

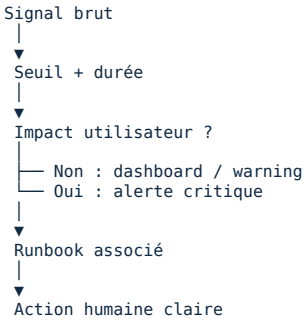
```
Request: POST /api/orders
 │
 ├── API Gateway 12 ms
 ├── Auth service 28 ms
 ├── Order API 120 ms
 │   ├── PostgreSQL query 85 ms
 │   ├── Redis cache 4 ms
 │   └── Payment API 410 ms
 └── Response total 590 ms
```

6. Alertes : prévenir sans noyer l'équipe

Une bonne alerte doit indiquer une action humaine nécessaire. Trop d'alertes créent de la fatigue, et l'équipe finit par les ignorer.

Domaine	Alerte typique	Action attendue
HTTP	5xx > 2% pendant 5 min.	Vérifier app, déploiement récent, logs.
Latence	p95 > 1s pendant 10 min.	Analyser DB, cache, dépendances.

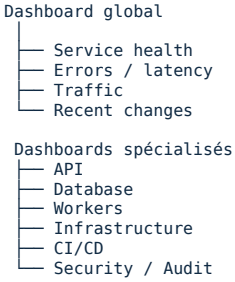
Domaine	Alerte typique	Action attendue
Disque	Disk usage > 85%.	Nettoyer, augmenter volume, vérifier logs.
Database	Connexions > 80% max.	Analyser pooling, requêtes, saturation.
Queue	Backlog augmente pendant 15 min.	Vérifier workers, erreurs, scaling.
CI/CD	Pipeline prod failed.	Vérifier état partiel, state lock, rollback.



7. Dashboards : voir vite ce qui compte

Un dashboard production doit répondre rapidement aux questions de santé, performance, capacité et impact. Il doit être lisible en incident.

Domaine	Indicateurs	Pourquoi
Vue service	Disponibilité, RPS, 5xx, p95/p99.	État utilisateur.
Application	Exceptions, endpoints lents, jobs failed.	Diagnostic métier.
Infrastructure	CPU, RAM, disque, réseau.	Saturation ressources.
Database	Connexions, locks, slow queries, replication lag.	Cause fréquente de lenteur.
Cache / queue	Hit ratio, evictions, backlog, retries.	Performance et async.
Déploiements	Dernier deploy, Terraform apply, version.	Corrélation incident/changement.



8. Observabilité en production : cas concrets

Signal	5xx passe de 0.1% à 4% après release.
Dashboard	Erreur HTTP, endpoint impacté, version déployée.
Logs	Exceptions sur route /api/orders .
Trace	Erreur lors appel payment provider.
Action	Rollback applicatif ou feature flag.

3.2 - Incidents production

Qualifier, mitiger, restaurer, communiquer, analyser et éviter la récidive.

Source modal: 3.2 Incidents production : qualifier, mitiger, restaurer et prévenir

1. Un incident production est un événement opérationnel, pas seulement un bug

Un incident production est une situation où un service réel est dégradé, indisponible, lent, instable, incorrect, non sécurisé ou incapable de remplir sa fonction métier. Il peut venir du code applicatif, de l'infrastructure, du réseau, de la base de données, d'un déploiement, d'un fournisseur externe, d'un problème sécurité ou d'une erreur humaine.

La priorité pendant l'incident n'est pas de trouver un coupable ni de rédiger une analyse parfaite. La priorité est de restaurer le service , réduire l'impact utilisateur, communiquer clairement, puis analyser à froid pour éviter la récidive.

Objectif	Résultat attendu
Qualifier	Comprendre rapidement l'impact, la gravité et le périmètre.
Stabiliser	Éviter que l'incident empire ou se propage.
Mitiger	Restaurer partiellement ou totalement le service.
Communiquer	Informer clairement les parties prenantes sans spéculation.
Restaurer	Valider le retour au nominal par des signaux mesurables.
Analyser	Identifier cause racine, facteurs contributifs et actions préventives.

```
Détection
 │
 ▼
 Qualification
 ├── impact utilisateur
 ├── service touché
 ├── gravité
 └── périmètre
 │
 ▼
 Stabilisation
 ├── stopper changement
 ├── éviter propagation
 └── geler actions risquées
 │
 ▼
 Mitigation
 ├── rollback
...
```

2. Classification de gravité : SEV / impact / priorité

La classification évite les discussions floues. Elle permet de décider qui mobiliser, quelle communication envoyer, à quelle fréquence donner des updates et quel niveau d'urgence appliquer.

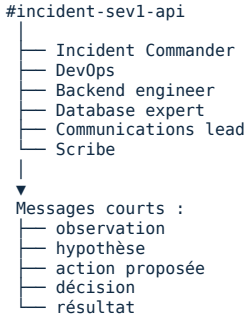
Niveau	Impact	Exemples	Réponse
SEV-1	Service critique indisponible ou perte de données.	Site down, API principale KO, DB prod inaccessible.	Mobilisation immédiate, canal incident, updates fréquents.
SEV-2	Dégradation forte mais contournement possible.	Latence élevée, fonctionnalité clé dégradée.	Réponse rapide, responsable incident nommé.
SEV-3	Impact partiel ou non critique.	Batch retardé, dashboard interne KO.	Traitement prioritaire mais sans mobilisation totale.
SEV-4	Faible impact ou anomalie préventive.	Alerte capacité, warning sécurité, bug mineur.	Suivi normal, planification correction.

```
Incident détecté
 │
 ▼
 Impact utilisateur réel ?
 │
 ├── Non : SEV-3 / SEV-4 possible
 └── Oui
 │
 ▼
 Service critique touché ?
 │
 ├── Oui : SEV-1 ou SEV-2
 └── Non : SEV-2 ou SEV-3
 │
 ▼
 Données ou sécurité impliquées ?
 │
 ├── Oui : escalade immédiate
...
```

3. Rôles pendant un incident

Un incident sérieux nécessite de la coordination. Sans rôles clairs, plusieurs personnes investiguent la même chose, personne ne communique, ou des actions contradictoires sont lancées.

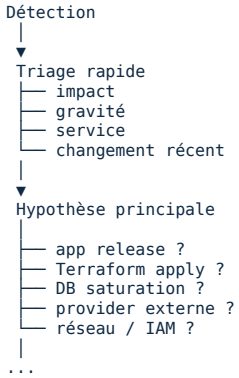
Rôle	Responsabilité	Erreur à éviter
Incident Commander	Coordonne, décide, priorise, garde la vision globale.	Tout faire soi-même.
Tech Lead Incident	Pilote l'analyse technique et les hypothèses.	Changer de piste toutes les 2 minutes.
Ops / DevOps	Vérifie infra, logs, métriques, rollback, scaling.	Appliquer des corrections sans validation.
App Engineer	Analyse code, erreurs applicatives, release récente.	Ignorer les signaux infra.
Communications Lead	Informe équipes, métier, clients si nécessaire.	Communiquer des hypothèses comme des faits.
Scribe	Note chronologie, décisions, commandes, heures.	Reconstituer après coup de mémoire.



4. Méthode en 6 étapes

Qualifier : identifier qui est impacté, quel service est touché, depuis quand, avec quelle gravité.; Stabiliser : stopper les déploiements, éviter les changements concurrents, empêcher l'incident de s'étendre.; Mitiger : rollback, restart contrôlé, scale, bypass, désactivation feature flag, correction minimale.; Communiquer : envoyer des messages courts, factuels, réguliers, sans spéculation excessive.

Action	Quand l'utiliser	Risque
Rollback app	Incident après release applicative.	Perte de correction récente.
Rollback Terraform	Incident après changement infra.	Plan de rollback à relire.
Feature flag off	Fonctionnalité nouvelle problématique.	Fonction désactivée temporairement.
Scale up	Saturation capacité.	Coût, masque cause racine.
Bypass dépendance	API externe ou service secondaire KO.	Mode dégradé.



5. Chronologie d'incident : l'outil indispensable

Une bonne chronologie permet de comprendre la séquence réelle : détection, hypothèses, actions, résultats, retour nominal. Elle évite les reconstructions approximatives après coup.

Temps	Action / observation	But
T+0	Alerte 5xx élevée sur API production.	Détection.
T+3	Dashboard confirme 5xx sur /api/orders .	Confirmer impact.
T+5	Canal incident créé, Incident Commander nommé.	Coordination.
T+8	Dernier déploiement app identifié à T-10.	Hypothèse principale.
T+12	Rollback release applicative lancé.	Mitigation.

Temps	Action / observation	But
T+18	5xx descend progressivement.	Vérifier efficacité.
T+25	Health checks et parcours métier OK.	Retour nominal.

```
[21:04] Alerte HighHttp5xxRate déclenchée.
 [21:06] Dashboard API confirme 5xx à 4.8%.
 [21:08] Incident SEV-1 déclaré. IC: Alice.
 [21:10] Dernier deploy API identifié : commit abc123.
 [21:12] Décision : rollback API vers version précédente.
 [21:16] Rollback terminé.
 [21:20] 5xx revenus sous 0.2%.
 [21:25] Health checks OK.
 [21:35] Communication : service stabilisé.
 [22:00] Début RCA.
```

6. Mitigation : restaurer vite sans aggraver

La mitigation vise à réduire l'impact avant même d'avoir une compréhension complète. Elle doit être rapide, proportionnée, réversible et mesurable.

Incident	Mitigation rapide	Vérification
Release applicative cassée	Rollback image/version précédente.	5xx, healthcheck, parcours critique.
Security group trop restrictif	Restaurer règle précédente ou correctif minimal.	Connectivité service → DB.
Saturation CPU	Scale up/out temporaire.	CPU, latence, erreurs.
DB connexions saturées	Réduire workers, ajuster pool, redémarrage contrôlé.	Connexions, locks, erreurs DB.
API externe lente	Timeout, circuit breaker, mode dégradé.	Latence et erreurs endpoint.
Disque plein	Nettoyage logs/cache, extension volume.	df -h , logs, service OK.
DNS incorrect	Restaurer record précédent.	dig , curl, trafic réel.

```
Incident actif
  │
  ▼
 Cause probable liée à un changement récent ?
  │
  ├── Oui
  │   ├── rollback app
  │   ├── revert Terraform
  │   └── désactiver feature flag
  │
  └── Non
  ├── scale temporaire
  ├── bypass dépendance
  ├── redémarrage contrôlé
  ├── restauration connectivité
  └── mode dégradé
  │
  ▼
...
```

7. Communication incident : courte, factuelle, régulière

Une communication claire réduit la panique, évite les interruptions inutiles et donne confiance dans la gestion de crise. Elle doit distinguer faits confirmés, hypothèses et prochaines actions.

Règle	Pourquoi
Être factuel	Éviter spéculation et contradictions.
Dire ce qui est impacté	Service, fonctionnalité, région, clients.
Dire ce qui est fait	Mitigation, rollback, investigation.
Donner un prochain update	Réduit les sollicitations.
Ne pas désigner de coupable	Garder une culture blameless.
Ne pas promettre sans certitude	Préserver crédibilité.

```
Incident SEV-2 en cours sur API production.
 Impact confirmé : hausse des erreurs 5xx sur /api/orders.
 Début estimé : 14:06.
 Action en cours : analyse dernier déploiement et logs applicatifs.
 Prochain update : dans 10 minutes.
```

8. RCA : Root Cause Analysis

La RCA vise à comprendre pourquoi l'incident est arrivé et pourquoi les protections existantes ne l'ont pas empêché ou détecté plus tôt. Elle ne doit pas s'arrêter au symptôme immédiat.

Niveau	Exemple
Symptôme	API retourne 500.
Cause technique immédiate	Timeout PostgreSQL sur endpoint commandes.

Niveau	Exemple
Cause technique profonde	Nouvelle requête sans index déclenche full scan.
Cause process	Pas de test de volume ni alerte slow query en staging.
Prévention	Ajouter index, test perf, alerte slow query, review SQL.

```
Pourquoi l'API était lente ?
 -> PostgreSQL répondait lentement.

 Pourquoi PostgreSQL répondait lentement ?
 -> Une requête faisait un scan complet.

 Pourquoi la requête faisait un scan complet ?
 -> Un index manquait sur une colonne filtrée.

 Pourquoi l'index manquait ?
 -> La migration n'a pas été revue côté performance.

 Pourquoi ce risque n'a pas été détecté ?
 -> Pas de test sur volume représentatif en staging.
```

Part 2 - DevOps - Terraform, State, GitLab CI/CD & Gestion d'incidents (Part 2)

This part groups related operational topics from the IDEO-Lab DevOps guide. Each chapter below extracts the practical knowledge embedded in the interactive modal panels.

Step	Topic	Purpose
3.3	Runbooks DevOps	Procédures claires pour rollback, restart, saturation, DB, réseau et pipeline bloqué.
3.3	RoadMap DevOps	Procédures claires sur la RoadMap devOps.
4.1	Cheat-sheet	Commandes Terraform, GitLab, debug pipeline et vérifications production.
1.5	Design Terraform avancé	Naming, locals, validations, count/for_each, lifecycle et dépendances.
1.6	Import & drift	Rattacher l'existant, detecter la dérive et reprendre le contrôle IaC.
1.7	Tests Terraform	Validation locale, plan de test, sandbox, Terratest et tests de modules.
1.8	Coûts & FinOps	Comprendre l'impact coût des changements infra avant apply.
2.5	Merge Request infra	Comment documenter une MR Terraform lisible et validable.
2.6	Templates GitLab CI	Factoriser les jobs CI/CD pour eviter la duplication.
2.7	Approvals & protections	Branches protegees, environnements proteges, approvals et separation des droits.

3.3 - Runbooks DevOps

Procédures claires pour rollback, restart, saturation, DB, réseau et pipeline bloqué.

Source modal: 3.3 Runbooks DevOps : procédures SRE pour production

1. Un runbook transforme la panique en procédure

Un runbook est une procédure opérationnelle claire qui explique quoi vérifier, quelles commandes lancer, dans quel ordre, avec quels risques et comment valider le retour à un état normal.

En production, le runbook évite l'improvisation. Il permet à un DevOps, un SRE, un développeur d'astreinte ou un support technique de suivre une méthode reproductible, même sous pression.

Section	Contenu attendu
Symptômes	Comment reconnaître le problème : alerte, log, métrique, comportement.
Impact	Quels utilisateurs, services ou données peuvent être touchés.
Vérifications	Commandes et dashboards à consulter avant action.
Actions	Procédure étape par étape, idéalement réversible.
Validation	Comment confirmer que le service est revenu à la normale.
Escalade	Qui appeler si la procédure ne fonctionne pas.
Rollback	Comment revenir en arrière ou mitiger.

```
Runbooks essentiels
  │
  ├── Rollback infrastructure
  ├── Rollback application
  ├── Restart service Linux
  ├── Saturation CPU / RAM / disque
  ├── Base de données lente ou saturée
  ├── DNS / certificat / load balancer
  ├── Pipeline Terraform bloqué
  ├── State lock Terraform
  ├── Secret exposé
  ├── Incident sécurité
  └── Provider cloud dégradé
```

2. Template professionnel de runbook

Critère	Bon runbook	Mauvais runbook
Clarté	Étapes numérotées, commandes exactes.	Texte vague ou théorique.
Sécurité	Précise les actions dangereuses.	Commandes destructives sans avertissement.

Critère	Bon runbook	Mauvais runbook
Validation	Explique comment mesurer le succès.	Dit seulement "vérifier".
Escalade	Indique quand appeler un expert.	Laisse l'opérateur bloqué.
Mise à jour	Versionné et revu après incident.	Obsolète depuis plusieurs mois.

```
# Runbook : titre court

 ## Objectif
 Décrire le problème que ce runbook permet de traiter.

 ## Symptômes
 - Alerte :
 - Logs :
 - Métriques :
 - Comportement utilisateur :

 ## Impact possible
 - Services touchés :
 - Utilisateurs touchés :
 - Données / sécurité :
 - Niveau de gravité probable :

 ## Pré-requis
...
```

3. Runbook : rollback infrastructure Terraform

À utiliser lorsqu'un changement Terraform a provoqué ou semble avoir provoqué une dégradation : réseau bloqué, IAM cassé, load balancer mal configuré, DNS incorrect, security group trop restrictif.

Zone	Vérification
Cloud	Ressource revenue à l'état attendu.
Application	Healthcheck OK, endpoints critiques OK.
Logs	Plus d'erreurs nouvelles liées au changement.
Metrics	5xx, latence, saturation revenus au nominal.
State	Pas de drift inattendu après rollback.

```
1. Déclarer l'incident si impact production.
 2. Identifier le dernier apply Terraform.
 3. Identifier le commit, la MR et le plan appliqué.
 4. Lire les logs du job apply.
 5. Lancer un plan refresh-only si nécessaire.
 6. Déterminer si rollback complet ou patch minimal.
 7. Revenir au commit précédent ou créer un correctif ciblé.
 8. Générer un nouveau plan.
 9. Vérifier qu'il ne détruit pas de ressource critique.
 10. Faire relire le plan.
 11. Appliquer avec approval.
 12. Vérifier logs, métriques, health checks.
 13. Documenter l'incident et la cause.
```

4. Runbook : restart contrôlé d'un service Linux

Redémarrer un service peut résoudre un blocage, mais peut aussi masquer la cause, interrompre des traitements ou aggraver l'incident. Le restart doit être contrôlé.

Situation	Pourquoi attendre
Suspicion fuite mémoire	Capturer métriques/process avant de perdre les indices.
Incident sécurité	Préserver preuves et logs.
Migration DB en cours	Risque d'interruption dangereuse.
Cluster sans redondance	Restart = downtime direct.

```
# État du service
 systemctl status app.service --no-pager

 # Logs récents
 journalctl -u app.service -n 100 --no-pager

 # Ressources système
 uptime
 free -h
 df -h

 # Processus
 ps aux | grep app
 top
```

5. Runbook : saturation serveur Linux

À utiliser en cas de CPU élevé, RAM saturée, disque plein, load average anormal, I/O wait important ou service instable sur un serveur Linux.

Symptôme	Cause possible	Action
CPU haut	Boucle, trafic élevé, job lourd.	Identifier process, scale, limiter job.
RAM saturée	Fuite mémoire, cache, trop de workers.	Identifier process, réduire workers, restart ciblé.
Disque plein	Logs, backups, uploads, cache.	Nettoyage contrôlé, rotation, extension volume.
Inodes pleins	Trop de petits fichiers.	Identifier dossier, purge contrôlée.
I/O wait élevé	Disque saturé, DB, logs intensifs.	Analyser I/O, réduire écriture, augmenter capacité.

```
# Charge système
 uptime
 top
 ps aux --sort=-%cpu | head
 ps aux --sort=-%mem | head

 # Mémoire
 free -h
 dmesg | grep -i "killed process"

 # Disque
 df -h
 df -i
 du -sh /var/log/* | sort -h
 du -sh /tmp/* | sort -h

 # Erreurs système
 journalctl -p err -n 100 --no-pager
...
```

6. Runbook : base de données lente ou saturée

Les incidents DB sont critiques : une mauvaise action peut empirer la latence, tuer des sessions utiles ou provoquer une perte de données. Il faut diagnostiquer avant d'agir.

Symptôme	Cause possible
Connexions saturées	Pool mal réglé, fuite connexion, pic trafic.
Slow queries	Index manquant, requête non optimisée, volume élevé.
Locks longs	Transaction bloquée, migration, batch.
CPU DB élevé	Requêtes lourdes, scan complet, tri massif.
Disk full	WAL/binlog, backups, tables temporaires.
Replication lag	Replica trop lent, charge écriture élevée.

```
# Ressources serveur
 top
 free -h
 df -h
 iostat -xz 1 5

 # Logs
 journalctl -u postgresql -n 100 --no-pager
 journalctl -u mysql -n 100 --no-pager
 journalctl -u mariadb -n 100 --no-pager
```

7. Runbook : réseau, DNS, load balancer et certificats

Les incidents réseau sont souvent visibles comme des timeouts, 502/503/504, erreurs TLS, DNS incorrect, target group unhealthy ou connectivité DB impossible.

Symptôme	Cause probable
502	Backend indisponible ou target unhealthy.
503	Aucun backend disponible.
504	Timeout upstream.
TLS error	Certificat expiré, mauvais SNI, chaîne incomplète.
DNS mauvais	Record incorrect, TTL, propagation.

```
# DNS
 dig example.com
 dig +short example.com
 nslookup example.com

 # HTTP / TLS
 curl -v https://example.com/
 curl -fsS https://example.com/health/
 openssl s_client -connect example.com:443 -servername example.com

 # Connectivité port
 nc -vz db.internal 5432
 nc -vz redis.internal 6379

 # Routes locales
 ip addr
 ip route
 ss -tulpn
```

8. Runbook : pipeline Terraform bloqué

À utiliser lorsqu'un pipeline infra échoue ou reste bloqué : runner indisponible, state lock, backend inaccessible, provider error, variables manquantes ou artefact de plan absent.

Symptôme	Cause probable	Action
Job pending	Runner absent ou tags incompatibles.	Vérifier runners/tags/protection.
Variable vide	Variable protected non disponible.	Vérifier branche protégée et scope.
init échoue	Backend inaccessible.	Vérifier credentials, bucket, réseau.
State lock	Apply actif ou lock orphelin.	Identifier job détenteur.
Apply sans tfplan	Artifact manquant ou expiré.	Relancer plan puis apply.
Provider error	Version/provider/API/cloud.	Lire erreur, vérifier versions et quotas.

```
1. Identifier le job exact en échec.
 2. Lire les logs depuis le début.
 3. Vérifier runner disponible et tags.
 4. Vérifier image CI et version Terraform.
 5. Vérifier variables accessibles à la branche.
 6. Vérifier terraform init et backend.
 7. Vérifier state lock.
 8. Vérifier artifact tfplan si apply.
 9. Corriger la cause minimale.
 10. Relancer uniquement le job nécessaire.
 11. Informer l'équipe si production concernée.
```

3.3 - RoadMap DevOps

Procédures claires sur la RoadMap devOps.

Source modal: 3.4 RoadMap DevOps : progression infra, CI/CD, SRE et production

1. Objectif de cette RoadMap DevOps

Cette roadmap propose une progression claire pour devenir opérationnel côté DevOps infrastructure : Linux, réseau, cloud, Terraform, GitLab CI/CD, sécurité, observabilité, incidents, runbooks et pratiques SRE.

Le but n'est pas seulement d'apprendre des outils. Le but est de construire une capacité professionnelle : livrer de l'infrastructure de façon fiable, sécurisée, automatisée, observable et récupérable en cas d'incident.

Domaine	Compétence attendue
Linux	Services systemd, logs, permissions, réseau, diagnostic CPU/RAM/disque.
Réseau	DNS, TLS, HTTP, ports, firewall, load balancer, security groups.
Cloud	Compute, VPC, IAM, stockage, base de données managée, monitoring cloud.
Terraform	Providers, modules, state, remote backend, plan/apply, drift, imports.
GitLab CI/CD	Stages, runners, variables, approvals, artifacts, environnements protégés.
Sécurité	Secrets, OIDC, IAM least privilege, scans IaC, branches protégées.
Observabilité	Logs, metrics, traces, dashboards, alerting, SLO.

```
Fondations
  │
  ├── Linux
  ├── Réseau
  ├── Git
  └── Cloud basics
  │
  ▼
Infrastructure as Code
  ├── Terraform basics
  ├── modules
  ├── remote state
  └── sécurité
  │
  ▼
Industrialisation
  ├── GitLab CI/CD
  ├── runners
...
```

2. Niveaux de progression DevOps

Compétence	Attendu
Linux	Savoir lire logs, redémarrer un service, vérifier CPU/RAM/disque.
Terraform	Comprendre init, plan, apply, variables, outputs, state.
GitLab CI/CD	Lire un pipeline, comprendre stages et jobs.
Cloud	Comprendre VM, réseau, security groups, DNS, stockage.
Incident	Savoir suivre un runbook sans improviser.

3. RoadMap 12 semaines : progression réaliste

Comprendre les fondamentaux Linux, cloud et réseau.; Créer une infrastructure Terraform propre.; Utiliser modules, remote state et locking.; Créer pipeline GitLab CI/CD infra avec plan/apply contrôlé.

Semaine	Objectif	Exercice pratique	Livrable attendu
1	Linux, Git, réseau de base.	Installer une VM, Nginx, service systemd, logs.	Serveur accessible + runbook diagnostic Linux.
2	Cloud basics.	Créer VM, security group, DNS, volume, firewall.	Mini architecture cloud documentée.
3	Terraform bases.	Créer ressources simples via Terraform.	Code Terraform init/plan/apply propre.
4	Variables, outputs, structure fichiers.	Paramétrer dev/staging/prod avec tfvars.	Projet Terraform structuré.
5	Modules Terraform.	Créer module network + compute.	Modules documentés avec inputs/outputs.
6	Remote state + locking.	Configurer backend distant par environnement.	State distant, verrouillé, non commité.

Semaine	Objectif	Exercice pratique	Livrable attendu
7	GitLab CI/CD infra.	Créer pipeline fmt/validate/plan.	MR avec plan Terraform en artefact.

```
Semaines 1-2
 Fondations Linux / Cloud

 Semaines 3-6
 Terraform solide

 Semaines 7-9
 Industrialisation CI/CD + sécurité

 Semaines 10-11
 Observabilité + incidents

 Semaine 12
 Projet complet production-ready
```

4. Parcours Terraform détaillé

Étape	Compétence	Exercice
1	Providers et resources.	Créer une VM ou un bucket.
2	Variables et outputs.	Paramétrer région, nom, taille.
3	Plan/apply.	Lire les changements + , ~ , - .
4	State.	Comprendre state list et state show .
5	Modules.	Créer module réseau et compute.
6	Remote backend.	State distant + lock.
7	Drift et import.	Importer une ressource existante.

```
Exercice 1 :
 Créer une ressource simple avec tags.

 Exercice 2 :
 Ajouter variables.tf, outputs.tf, locals.tf.

 Exercice 3 :
 Créer dev.tfvars et prod.tfvars.

 Exercice 4 :
 Créer un module network.

 Exercice 5 :
 Créer un module compute qui consomme network outputs.

 Exercice 6 :
 Configurer backend distant.

...
```

5. Parcours GitLab CI/CD Infrastructure

Bloc	À maîtriser
Syntaxe YAML	Stages, jobs, variables, rules, needs, artifacts.
Runners	Tags, runners protégés, runners dédiés infra.
Terraform pipeline	fmt, validate, security, plan, apply, post-check.
Environnements	dev, staging, prod, protected environments.
Approvals	Apply manuel, review plan, approval prod.
Secrets	Variables masked/protected/scoped, OIDC, IAM minimal.
Debug	Logs pipeline, artifacts, state lock, variables absentes.

```
Merge Request
 │
 ├── terraform fmt
 ├── terraform validate
 ├── scan sécurité
 └── terraform plan
 │
 ▼
 Review du plan
 │
 ▼
 Merge main
 │
 ▼
 Plan production
 │
 ▼
 Approval
...
```

6. Parcours Cloud, Linux et réseau

Sujet	À savoir faire
Services	systemctl status/start/stop/restart , logs systemd.
Logs	journalctl , logs Nginx, logs applicatifs.
Ressources	top , free , df , du , iostat .
Réseau	curl , dig , nc , ss , routes.
Sécurité	users, permissions, SSH, firewall, certificats.

```
systemctl status nginx
 journalctl -u nginx -n 100 --no-pager
 df -h
 du -sh /var/log/* | sort -h
 free -h
 top
 ss -tulpn
 curl -v https://example.com/
 dig example.com
 nc -vz host 443
```

7. Parcours Observabilité

Étape	Objectif	Livrable
1	Centraliser les logs.	Logs app/proxy consultables.
2	Ajouter métriques système.	CPU/RAM/disque/réseau visibles.
3	Ajouter métriques applicatives.	RPS, 5xx, p95/p99.
4	Créer dashboard global.	Vue production en 30 secondes.
5	Créer alertes critiques.	Service down, 5xx, disque, DB.
6	Ajouter runbooks aux alertes.	Alerte actionnable.
7	Corréler déploiements et incidents.	Annotations deploy/Terraform.

```
Latency:
 p50, p95, p99

 Traffic:
 requests/sec, jobs/min

 Errors:
 HTTP 5xx, exceptions, failed jobs

 Saturation:
 CPU, RAM, disk, DB connections, queue backlog
```

8. Parcours Incidents / SRE

Compétence	Objectif
Triage	Qualifier impact, gravité, périmètre.
Mitigation	Restaurer vite avec action réversible.
Rollback	Revenir à une version ou config stable.
Communication	Messages courts, factuels, réguliers.
RCA	Comprendre cause racine et facteurs contributifs.
Post-mortem	Transformer incident en amélioration.
SLO	Mesurer la fiabilité depuis le point de vue utilisateur.

```
Simulation 1 :
 Healthcheck applicatif KO.
 Objectif : diagnostiquer logs + restart contrôlé.

 Simulation 2 :
 Disque presque plein.
 Objectif : identifier dossier, mitigation, prévention.

 Simulation 3 :
 Terraform state lock.
 Objectif : identifier job détenteur, éviter force-unlock abusif.

 Simulation 4 :
 Security group DB cassé.
 Objectif : rollback règle réseau.

 Simulation 5 :
 Déploiement app provoque 5xx.
...
```

4.1 - Cheat-sheet

Commandes Terraform, GitLab, debug pipeline et vérifications production.

Source modal: 4.1 Cheat-sheet DevOps Infra : commandes, debug, audit et production

1. Cheat-sheet DevOps infra : objectif

Cette cheat-sheet regroupe les commandes et réflexes essentiels pour travailler efficacement sur une infrastructure moderne : Terraform, GitLab CI/CD, Linux, réseau, debugging pipeline, sécurité, audit et vérifications production.

Elle est pensée comme un support opérationnel : avant une merge request, avant un apply production, pendant un incident, ou pour préparer un entretien DevOps.

Domaine	Usage
Terraform	Initialiser, valider, planifier, appliquer, importer, inspecter.
State	Diagnostiquer drift, lock, remote state, ressources connues.
GitLab CI/CD	Analyser pipeline, runners, artifacts, variables et environments.
Linux	Diagnostiquer services, logs, CPU, RAM, disque, réseau local.
Réseau	Tester DNS, TLS, ports, HTTP, load balancer, connectivité.
Production	Vérifier health checks, logs, métriques, alertes et rollback.
Sécurité	Secrets, IAM, audit, logs sensibles, variables protégées.

```
Avant toute commande sensible
│
├── Quel environnement ?
│   ├── dev
│   ├── staging
│   └── prod
│
├── Quel state Terraform ?
│   ├── backend
│   ├── key
│   └── lock
│
├── Quel impact ?
│   ├── réseau
│   ├── IAM
│   ├── database
│   ├── compute
│   └── DNS
...
```

2. Commandes Terraform essentielles

Symbole	Signification	Réflexe
+	Création	Vérifier coût, tags, sécurité.
~	Modification	Comprendre l'impact.
-/+	Remplacement	Analyse obligatoire.
-	Suppression	Stop si non attendu.

```
# Initialiser le dossier Terraform
 terraform init

 # Formater le code
 terraform fmt -recursive
 terraform fmt -check -recursive

 # Valider la configuration
 terraform validate

 # Générer un plan
 terraform plan

 # Générer un plan sauvegardé
 terraform plan -out=tfplan

 # Lire un plan binaire en texte
 terraform show -no-color tfplan
...
```

3. State Terraform, drift, import et lock

Symptôme	Commande	But
Ressource inconnue	terraform state list	Voir si Terraform la gère.
Diff inattendu	terraform plan -refresh-only	Détecter drift.
Apply bloqué	Logs pipeline + lock ID	Identifier détenteur du lock.
Ressource existante	terraform import	Rattacher au state.

```
# Lister les ressources connues du state
 terraform state list

 # Afficher une ressource précise
 terraform state show aws_instance.web

 # Afficher les outputs
 terraform output

 # Exporter le state courant localement
 terraform state pull > state-backup.json

 # Pousser un state modifié
 # À éviter sauf procédure contrôlée
 terraform state push state-backup.json
```

4. GitLab CI/CD : commandes, YAML et debug rapide

Mot-clé	Usage
stages	Ordre logique du pipeline.
rules	Conditions d'exécution des jobs.
needs	Dépendances rapides entre jobs.
artifacts	Conserver plan, rapports, logs utiles.
environment	Associer job à dev/staging/prod.
when: manual	Apply manuel contrôlé.
tags	Sélectionner runner adapté.

```
# État du repo
 git status

 # Créer une branche
 git checkout -b infra/change-name

 # Voir les changements
 git diff
 git diff --staged

 # Ajouter / commit
 git add .
 git commit -m "Update Terraform infrastructure"

 # Historique
 git log --oneline -10

 # Revenir à un commit par revert
...
```

5. Linux : diagnostic production rapide

Symptôme	Commandes	Hypothèse
Service down	systemctl status , journalctl -u	Crash, config, dépendance KO.
Lenteur	top , free -h , logs app	CPU, RAM, DB, API externe.
Disque plein	df -h , du -sh	Logs, backups, uploads.
Port non ouvert	ss -tulpn , nc -vz	Service non lancé, firewall, bind.

```
# Charge et uptime
 uptime

 # CPU / process
 top
 ps aux --sort=-%cpu | head
 ps aux --sort=-%mem | head

 # Mémoire
 free -h
 dmesg | grep -i "killed process"

 # Disque
 df -h
 df -i
 du -sh /var/log/* | sort -h
 du -sh /tmp/* | sort -h

...
```

6. Réseau, DNS, HTTP, TLS et load balancer

Code	Cause fréquente	Vérification
502	Backend invalide ou indisponible.	Logs proxy + service app.
503	Aucun backend disponible.	Targets LB, healthcheck, autoscaling.
504	Timeout upstream.	App lente, DB lente, timeout proxy.
TLS error	Certificat expiré ou mauvais SNI.	openssl s_client , certbot, LB cert.
NXDOMAIN	DNS absent ou zone incorrecte.	dig , zone DNS, registrar.

```
# Résolution DNS
 dig example.com
 dig +short example.com
 nslookup example.com

 # Vérifier un record précis
 dig A example.com
 dig CNAME www.example.com
 dig TXT example.com

 # Voir TTL
 dig example.com
```

7. Vérifications production avant / pendant / après apply

Contrôle	Commande / preuve
Bon environnement	Afficher $CI_ENVIRONMENT_NAME , $TF_ROOT , backend key.
Plan relu	terraform show -no-color tfplan
Pas de destroy inattendu	Review du plan.
State distant	terraform init avec backend attendu.
Monitoring prêt	Dashboards ouverts.
Rollback connu	Runbook ou procédure.

```
# Ne pas lancer plusieurs apply
 # Surveiller logs du job
 # Noter toute ressource en erreur
 # Ne pas force-unlock sans preuve
 # Si échec partiel : relancer plan avant correction
```

8. Debug pipeline Terraform / GitLab

Symptôme	Cause probable	Vérification
Job pending	Runner indisponible ou mauvais tags.	Runner list, tags job.
Variable vide	Variable protected/scoped inaccessible.	Branche, environnement, scope variable.
terraform init échoue	Backend ou credentials.	Bucket, state, IAM, réseau.
State lock	Autre apply actif ou lock orphelin.	Job détenteur, logs pipeline.
Artifact absent	Mauvais path ou expiration.	artifacts.paths , dependencies, needs.
Plan local différent CI	Variables ou versions différentes.	Terraform version, provider lock, tfvars.

```
Job échoué
 │
 ▼
 Lire logs depuis le début
 │
 ├── image Docker
 ├── dossier courant
 ├── variables non sensibles
 ├── terraform version
 ├── terraform init
 ├── provider error
 └── backend / state
 │
 ▼
 Corriger cause minimale
 │
 ▼
 Relancer plan
...
```

1.5 - Design Terraform avancé

Naming, locals, validations, count/for_each, lifecycle et dépendances.

Source modal: 1.5 Design Terraform avancé : naming, locals, validations, for_each, lifecycle et dépendances

1. Le design Terraform avancé : écrire du code sûr, lisible et stable

Le design Terraform avancé ne consiste pas à rendre le code "plus intelligent". Il consiste à rendre l'infrastructure prévisible , lisible , contrôlée , réutilisable et moins dangereuse en production .

Un bon code Terraform permet à un autre DevOps de comprendre rapidement : quels paramètres sont exposés, quelles valeurs sont calculées, quelles ressources seront créées, quelles dépendances existent, quels garde-fous protègent les ressources critiques et comment le plan doit être relu.

Pilier	But	Exemple
Naming	Créer des noms prévisibles et cohérents.	project-env-component-region
Locals	Centraliser les valeurs calculées.	Préfixes, tags, maps, conventions.
Validations	Bloquer les mauvaises entrées tôt.	Environnement limité à dev/staging/prod.
for_each	Créer plusieurs ressources avec des clés stables.	Subnets, DNS records, security rules.
Lifecycle	Protéger ou contrôler les changements sensibles.	prevent_destroy sur database.
Dépendances	Laisser Terraform construire un graphe propre.	Références directes plutôt que depends_on abusif.

```
variables.tf
 │
 ├── inputs typés
 ├── descriptions
 ├── defaults sûrs
 └── validations
 │
 ▼
locals.tf
 │
 ├── name_prefix
 ├── common_tags
 ├── maps normalisées
 └── valeurs calculées
 │
 ▼
main.tf
 │
...
```

2. Naming Terraform : conventions lisibles et robustes

Le naming est fondamental. Il permet d'identifier rapidement à quel projet, environnement, composant, région ou équipe appartient une ressource.

Ressource	Nom recommandé
Load balancer API prod	ideo-prod-api-alb
Security group web staging	ideo-staging-web-sg
Database production	ideo-prod-main-db
Redis dev	ideo-dev-cache-redis
Bucket logs	ideo-prod-logs-eu-west-3

```
locals {
 name_prefix = "${var.project}-${var.environment}-${var.component}"

 common_tags = {
 Project = var.project
 Environment = var.environment
 Component = var.component
 ManagedBy = "terraform"
 Owner = var.owner
 }
 }
```

3. Locals : centraliser les valeurs calculées

Les locals permettent de calculer une fois des valeurs réutilisées partout : préfixes, tags, noms, maps normalisées, règles transformées, listes filtrées. Ils rendent le code plus lisible et évitent les répétitions.

Usage	Exemple
Préfixe de nom	${var.project}-${var.environment}
Tags communs	Project, Environment, Owner, ManagedBy.

Usage	Exemple
Valeur conditionnelle	Backup plus long en production.
Map normalisée	Transformer une liste en map pour for_each .
Feature flags	Activer monitoring seulement en staging/prod.

```
locals {
 name_prefix = "${var.project}-${var.environment}-${var.component}"

 common_tags = merge(
 {
 Project = var.project
 Environment = var.environment
 Component = var.component
 ManagedBy = "terraform"
 },
 var.extra_tags
 )

 is_prod = var.environment == "prod"

 default_backup_retention_days = local.is_prod ? 14 : 3
 default_deletion_protection = local.is_prod ? true : false
 }
```

4. Validations : bloquer les mauvaises entrées avant production

Les validations Terraform permettent d'éviter des erreurs de configuration avant le plan ou l'apply. Elles sont essentielles pour les modules partagés, car elles empêchent les utilisateurs du module de passer des valeurs dangereuses, incohérentes ou non supportées.

Variable	Validation recommandée
environment	Limiter à dev , staging , prod .
allowed_cidr_blocks	Interdire 0.0.0.0/0 sur ports sensibles.
backup_retention_days	Minimum plus élevé en production.
deletion_protection	Forcer true en production pour DB.
instance_type	Limiter à une liste approuvée.
domain_name	Vérifier format ou suffixe autorisé.

```
variable "environment" {
 description = "Deployment environment."
 type = string

 validation {
 condition = contains(["dev", "staging", "prod"], var.environment)
 error_message = "Environment must be dev, staging, or prod."
 }
 }
```

5. count, for_each et dynamic blocks

count et for_each permettent de créer plusieurs ressources. En production, for_each est souvent préférable car il utilise des clés stables, alors que count dépend des index de liste.

Activation simple	count = var.enabled ? 1 : 0
Ressource optionnelle	Créer ou non une ressource unique.
Liste non critique	Cas simples avec faible risque de réordonnancement.

```
resource "aws_subnet" "private" {
 count = length(var.private_subnets)

 vpc_id = var.vpc_id
 cidr_block = var.private_subnets[count.index]

 tags = merge(local.common_tags, {
 Name = "${local.name_prefix}-private-${count.index + 1}"
 })
 }
```

6. Lifecycle : protéger et contrôler les changements sensibles

Le bloc lifecycle permet d'ajouter des garde-fous sur certaines ressources : empêcher la destruction, créer avant de détruire, ignorer certains changements ou remplacer une ressource quand une dépendance change.

Très utile pour les ressources critiques : bases de données, buckets contenant des données, clés KMS, volumes persistants.

Option	Usage	Risque
prevent_destroy	Protéger DB, buckets, volumes.	Peut bloquer un changement voulu.
create_before_destroy	Réduire downtime sur ressources remplaçables.	Peut nécessiter noms uniques.
ignore_changes	Ignorer drift contrôlé.	Peut masquer un vrai problème.

Option	Usage	Risque
replace_triggered_by	Forcer remplacement si dépendance change.	À utiliser avec prudence.

```
resource "aws_db_instance" "main" {
 identifier = "${local.name_prefix}-db"

 lifecycle {
 prevent_destroy = true
 }
 }
```

7. Dépendances Terraform : laisser le graphe travailler

Terraform construit un graphe de dépendances à partir des références entre ressources. Dans la plupart des cas, il ne faut pas ajouter depends_on manuellement : une référence explicite suffit.

Ici, l'instance dépend automatiquement du security group parce qu'elle référence son ID.

Cas	Justification
Dépendance non visible dans les attributs	Terraform ne peut pas la deviner.
Ordre requis par provider	API externe nécessite une ressource prête.
Module dépendant d'un autre module	Seulement si les outputs ne créent pas déjà la dépendance.
Provisioning spécial	Cas rares, souvent à éviter.

```
resource "aws_security_group" "web" {
 name = "${local.name_prefix}-web-sg"
 vpc_id = aws_vpc.main.id
 }

 resource "aws_instance" "web" {
 ami = var.ami_id
 instance_type = var.instance_type
 vpc_security_group_ids = [aws_security_group.web.id]
 }
```

8. Design avancé des modules Terraform

Un module Terraform est une API d'infrastructure. Son design doit être stable, documenté, sécurisé par défaut et assez simple pour être utilisé sans lire tout son code interne.

Élément	Bonne pratique
Inputs	Peu nombreux, typés, décrits et validés.
Outputs	Seulement les valeurs utiles aux consommateurs.
Defaults	Sûrs par défaut, jamais trop permissifs.
README	Exemple d'usage, inputs, outputs, limitations.
Versioning	Tags Git ou registry, pas de production sur main .
Responsabilité	Un module doit avoir un périmètre clair.

```
module "api_service" {
 source = "../../modules/compute-service"

 project = var.project
 environment = var.environment
 component = "api"
 image = var.api_image
 desired_count = 3

 vpc_id = module.network.vpc_id
 subnet_ids = module.network.private_subnet_ids

 tags = local.common_tags
 }
```

1.6 - Import & drift

Rattacher l'existant, detecter la dérive et reprendre le contrôle IaC.

Source modal: 1.6 Import & drift : rattacher l'existant, détecter la dérive et reprendre le contrôle IaC

1. Import & drift : reprendre le contrôle sans casser l'existant

Dans beaucoup d'entreprises, Terraform arrive après plusieurs années de création manuelle : instances, security groups, load balancers, bases de données, DNS, buckets, rôles IAM, certificats ou règles réseau déjà existants.

L' import Terraform permet de rattacher ces ressources existantes au state, sans les recréer. Le drift désigne l'écart entre le code Terraform, le state et la réalité cloud. Bien gérer les deux permet de reprendre progressivement le contrôle IaC.

Élément	Rôle	Risque si désaligné
Code Terraform	Décrit l'état souhaité.	Terraform veut modifier ou recréer.
State Terraform	Mémorise les ressources gérées et leurs IDs.	Terraform ne sait pas ce qu'il possède.
Cloud réel	Infrastructure effectivement déployée.	Drift, surprise au prochain plan.

```
Infrastructure existante
  │
  ├── créée manuellement
  ├── créée par ancien outil
  ├── créée par console cloud
  └── créée par script historique
  │
  ▼
 Terraform import
  │
  ├── rattache au state
  ├── nécessite un bloc resource
  ├── ne crée pas la ressource
  └── ne garantit pas que le code est complet
  │
  ▼
 terraform plan
  │
...
```

2. Inventaire : cartographier avant d'importer

Un import propre commence par un inventaire. Il faut comprendre les ressources existantes, leurs dépendances, leur criticité, leurs noms, leurs tags, leurs liens réseau et leur usage réel.

Famille	Exemples	Priorité
Réseau	VPC, subnets, routes, NAT, security groups.	Très élevée
Compute	VM, autoscaling, launch templates, containers.	Élevée
Load balancing	ALB, target groups, listeners, health checks.	Très élevée
Data	RDS, buckets, volumes, snapshots, Redis.	Critique
IAM	Roles, policies, users techniques, bindings.	Critique sécurité
DNS / TLS	Zones, records, certificats, validations.	Élevée

```
Resource inventory:
 - Cloud provider:
 - Account / project:
 - Environment:
 - Resource type:
 - Resource name:
 - Provider ID:
 - Owner:
 - Criticality:
 - Dependencies:
 - Data risk:
 - Current tags:
 - Import target address:
 - Validation command:
 - Rollback / mitigation:
```

3. Terraform import classique

La commande terraform import rattache une ressource existante à une adresse Terraform. Elle ne crée pas automatiquement un code parfait : il faut écrire le bloc resource , importer l'ID, puis ajuster le code jusqu'à obtenir un plan propre.

Contrôle	Commande	But
Présence dans le state	terraform state list	Confirmer l'import.
Détails importés	terraform state show RESOURCE	Comprendre les attributs réels.
Diff restant	terraform plan	Identifier ce que le code ne décrit pas.

Contrôle	Commande	But
Risque destroy/replace	Review du plan	Éviter accident production.

```
# 1. Écrire le bloc resource minimal
 resource "aws_security_group" "web" {
 name = "prod-web-sg"
 vpc_id = var.vpc_id
 }

 # 2. Importer la ressource existante
 terraform import aws_security_group.web sg-0123456789abcdef0

 # 3. Inspecter la ressource importée
 terraform state show aws_security_group.web

 # 4. Lancer un plan
 terraform plan

 # 5. Compléter le code jusqu'à réduire le diff
 terraform plan
```

4. Import blocks et génération de configuration

Les versions récentes de Terraform permettent d'utiliser des blocs import dans le code. Cette approche rend l'import plus traçable, relisible en merge request et plus adaptée aux workflows CI/CD.

La génération de configuration peut aider à démarrer, mais le fichier généré doit être nettoyé, simplifié, commenté et adapté aux conventions du projet.

Étape	Action
Nettoyer	Retirer attributs calculés ou inutiles.
Renommer	Appliquer naming Terraform clair.
Factoriser	Déplacer tags, noms, valeurs répétées dans locals.
Paramétrer	Remplacer valeurs fixes par variables.
Protéger	Ajouter lifecycle si ressource critique.
Planifier	Répéter terraform plan jusqu'à diff maîtrisé.

```
import {
 to = aws_security_group.api
 id = "sg-0abc123456789def0"
 }

 resource "aws_security_group" "api" {
 name = "prod-api-sg"
 vpc_id = var.vpc_id
 }
```

5. Drift : détecter et traiter la dérive

Le drift apparaît quand la réalité cloud ne correspond plus au code Terraform ou au state. Il vient souvent de changements manuels dans la console, de hotfixs d'urgence, de scripts externes, de changements provider ou d'un import incomplet.

Cause	Exemple	Traitement
ClickOps	Modification console cloud.	Reporter dans Terraform ou restaurer.
Hotfix urgence	Security group ouvert temporairement.	Documenter puis codifier ou annuler.
Script externe	Script modifie tags ou scaling.	Clarifier ownership.
Autoscaling	Desired capacity modifiée.	Éventuellement ignore_changes .
Import incomplet	Attribut non décrit dans code.	Compléter resource block.
Provider upgrade	Attribut calculé différemment.	Lire changelog, pinner versions.

```
# Voir les écarts entre state et infrastructure réelle
 terraform plan -refresh-only

 # Mettre à jour le state avec la réalité observée
 terraform apply -refresh-only

 # Plan standard après refresh
 terraform plan
```

6. State operations : déplacer, retirer, inspecter

Les opérations sur le state sont puissantes et dangereuses. Elles modifient la mémoire de Terraform, pas forcément l'infrastructure réelle. Elles doivent être utilisées avec sauvegarde, revue et procédure.

Cas	But
Renommage Terraform	Changer l'adresse sans recréer la ressource.
Refactor module	Déplacer une ressource vers un module.
Correction d'adresse	Aligner state avec le nouveau design.

```
# Lister les ressources
 terraform state list

 # Inspecter une ressource
 terraform state show aws_instance.web

 # Sauvegarder localement le state
 terraform state pull > state-backup.json

 # Déplacer une ressource dans le state
 terraform state mv aws_instance.web aws_instance.app

 # Retirer une ressource du state sans la détruire
 terraform state rm aws_instance.web
```

7. Import & drift en production

En production, l'import et la correction de drift doivent être traités comme des changements sensibles. Même si l'import ne modifie pas la ressource cloud, il modifie le state et peut influencer les futurs plans.

Règle	Pourquoi
Importer par petits lots	Limiter le risque et faciliter la revue.
Lire le plan après chaque import	Détecter destroy/replace/diff inattendu.
Ne pas importer DB à la légère	Risque data et lifecycle critique.
Protéger le state	State modifié = impact futur.
Documenter chaque mapping	Adresse Terraform ↔ ID provider.
Éviter apply concurrent	Risque lock, drift ou plan obsolète.

```
Title: Import existing production API security group

 Scope:
 - Resource: aws_security_group.api
 - Provider ID: sg-0abc123456789def0
 - Environment: production
 - State: prod/network/terraform.tfstate

 Risk:
 - Bad code alignment could create unexpected diff.
 - Future plans may modify existing SG.

 Validation:
 - terraform state show
 - terraform plan
 - no unexpected destroy or replace
 - rules match current production

...
```

8. Audit : comprendre qui a changé quoi

Le drift est souvent la conséquence d'un changement manuel. Pour le traiter sérieusement, il faut identifier qui a modifié quoi, quand, pourquoi, et si cette modification doit être codifiée ou annulée.

Source	Ce qu'elle révèle
Cloud audit logs	Actions console/API, utilisateur, rôle, heure.
Git history	Changements Terraform, MR, commit, reviewer.
GitLab pipeline	Job exécuté, variables, artifact plan, statut.
Terraform state versions	Évolution du state distant.
Tickets / change requests	Justification métier ou opérationnelle.
Incident timeline	Hotfixs et actions d'urgence.

```
- Quelle ressource a changé ?
 - Le code Terraform a-t-il changé ?
 - Le state a-t-il changé ?
 - Le cloud réel a-t-il changé ?
 - Qui a fait l'action ?
 - Était-ce un hotfix ?
 - Y a-t-il un ticket ou incident associé ?
 - Faut-il garder le changement ?
 - Faut-il le reporter dans Terraform ?
 - Faut-il renforcer un garde-fou ?
```

2.6 - Templates GitLab CI

Factoriser les jobs CI/CD pour eviter la duplication.

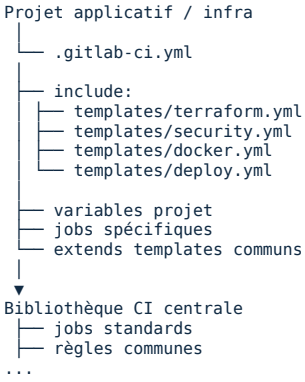
Source modal: 2.6 Templates GitLab CI : factoriser les jobs CI/CD pour éviter la duplication

1. Templates GitLab CI : industrialiser les pipelines

Les templates GitLab CI permettent de factoriser les jobs répétitifs : installation, lint, tests, scan sécurité, Terraform plan, Terraform apply, build d'image, publication d'artifacts, smoke tests et déploiements.

Sans templates, chaque projet finit avec son propre .gitlab-ci.yml , ses variantes, ses bugs, ses secrets mal déclarés et ses différences de comportement. Avec des templates bien conçus, les équipes gagnent en cohérence, sécurité, lisibilité et maintenabilité.

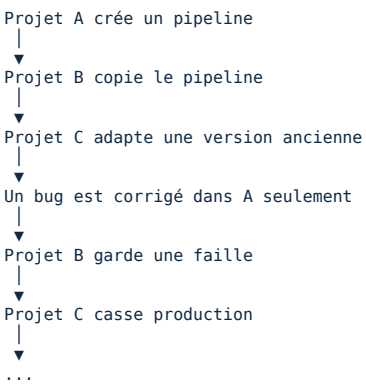
Zone	Exemples	Bénéfice
Lint / format	Terraform fmt, YAML lint, shellcheck.	Qualité homogène.
Validation	Terraform validate, tests unitaires.	Éviter erreurs basiques.
Sécurité	Checkov, Trivy, tfsec, secret scan.	Garde-fous communs.
Terraform	init, plan, show, artifacts, apply.	Workflow infra standardisé.
Déploiement	staging, production, rollback, post-checks.	Moins d'improvisation.
Artifacts	plan.txt, plan.json, rapports sécurité.	Review et audit facilités.



2. Le problème : duplication, divergence et pipelines fragiles

La duplication CI/CD devient vite une dette technique. Une correction de sécurité doit être copiée dans dix projets, un bug Terraform reste dans certains pipelines, et les nouveaux DevOps ne savent plus quel fichier est la référence.

Symptôme	Cause	Impact
Jobs similaires mais différents	Copier-coller modifié localement.	Comportements imprévisibles.
Scan sécurité absent sur certains projets	Pas de template commun.	Faille non détectée.
Terraform plan non publié	Artifacts non standardisés.	Review difficile.
Variables incohérentes	Naming non défini.	Debug lent et erreurs prod.
Fix CI à répéter partout	Aucune bibliothèque partagée.	Maintenance coûteuse.
Pipeline incompréhensible	Includes et overrides non documentés.	Onboarding difficile.



3. include : importer des templates GitLab CI

include permet d'importer des fichiers CI depuis le même repository, un autre projet GitLab, un template distant ou une bibliothèque centrale. C'est la base de la mutualisation.

Type	Usage	Risque
local	Templates dans le même repo.	Peu mutualisé entre projets.
project	Bibliothèque CI interne.	Doit être versionné proprement.
remote	Template externe ou généré.	Risque supply-chain.
template	Templates GitLab prédéfinis.	Moins adapté aux conventions internes.

```
include:
 - local: ".gitlab/ci/terraform.yml"
 - local: ".gitlab/ci/security.yml"
 - local: ".gitlab/ci/deploy.yml"
```

4. extends : réutiliser un job modèle

extends permet à un job concret d'hériter d'un job template. On définit un comportement commun dans un job caché, puis chaque projet surcharge uniquement ce qui varie : variables, rules, environment, paths ou script additionnel.

Élément	Dans template ?	Pourquoi
Image Docker standard	Oui	Cohérence des versions.
Variables génériques	Oui	Évite répétition.
Chemin projet TF_ROOT	Souvent non	Spécifique à chaque repo.
Environment production	Dans job concret	Contrôle explicite.
Rules sensibles	Template ou job dédié	Selon standard interne.
Secrets	Non	Variables GitLab protégées/scopées.

```
.terraform_base:
 image: hashicorp/terraform:1.6.6
 variables:
 TF_IN_AUTOMATION: "true"
 TF_INPUT: "false"
 before_script:
 - cd "$TF_ROOT"
 - terraform --version
 - terraform init -input=false
```

5. Jobs cachés : construire des briques réutilisables

Un job qui commence par un point, comme .terraform_base , n'est pas exécuté directement. Il sert de brique de réutilisation pour d'autres jobs.

Préfixe	Usage
.base_*	Configuration commune.
.rules_*	Conditions d'exécution.
.terraform_*	Jobs Terraform.
.security_*	Scans sécurité.
.docker_*	Build/push image.
.deploy_*	Déploiement.

```
.rules_merge_request:
 rules:
 - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'

.rules_main:
 rules:
 - if: '$CI_COMMIT_BRANCH == "main"'

.debug_context:
 before_script:
 - echo "Job=$CI_JOB_NAME"
 - echo "Branch=$CI_COMMIT_BRANCH"
 - echo "Source=$CI_PIPELINE_SOURCE"
 - echo "TF_ROOT=${TF_ROOT:-not-set}"

.terraform_base:
 image: hashicorp/terraform:1.6.6
 variables:
...
```

6. Variables : personnaliser sans dupliquer

Les variables rendent les templates flexibles. Le projet définit ce qui change : chemin Terraform, environnement, image, nom de service, stratégie de déploiement, et le template garde la logique commune.

Variable	Usage	Exemple
TF_ROOT	Dossier Terraform.	infra/live/prod/app

Variable	Usage	Exemple
TF_VAR_environment	Variable Terraform.	production
APP_NAME	Nom service.	api
DEPLOY_ENV	Environnement cible.	staging
DOCKER_IMAGE	Image applicative.	registry/app/api
SCAN_SEVERITY	Niveau blocage sécurité.	HIGH,CRITICAL

```
variables:
 APP_NAME: "api"
 TF_ROOT: "infra/live/staging/api"
 DEPLOY_ENV: "staging"
 TF_IN_AUTOMATION: "true"
 TF_INPUT: "false"
```

7. Template Terraform complet : validate, plan, apply

Terraform est l'un des meilleurs candidats à la factorisation : les commandes sont répétitives, les artifacts doivent être standards, et les garde-fous production doivent être cohérents partout.

```
.terraform_base:
 image: hashicorp/terraform:1.6.6
 variables:
 TF_INPUT: "false"
 TF_IN_AUTOMATION: "true"
 before_script:
 - echo "TF_ROOT=$TF_ROOT"
 - cd "$TF_ROOT"
 - terraform --version
 - terraform init -input=false

.terraform_validate:
 extends: .terraform_base
 stage: validate
 script:
 - terraform fmt -check -recursive
 - terraform validate

...
```

8. Sécurité des templates CI/CD

Un template CI/CD est puissant : s'il est compromis ou mal conçu, tous les projets qui l'incluent peuvent hériter du problème. Il doit donc être versionné, relu, limité et testé.

Risque	Exemple	Prévention
Supply-chain CI	Template central modifié malicieusement.	MR review, protected branch, tags.
Secrets exposés	env ou set -x .	Debug sécurisé.
Apply prod non protégé	Job manuel accessible trop largement.	Protected environment + runner.
Include sur main	Changement template impacte tout sans contrôle.	Pin sur tag stable.
Remote include non maîtrisé	Fichier externe change sans validation.	Éviter ou contrôler strictement.

```
.safe_debug_context:
 script:
 - echo "Job=$CI_JOB_NAME"
 - echo "Branch=$CI_COMMIT_BRANCH"
 - echo "Source=$CI_PIPELINE_SOURCE"
 - echo "Environment=${CI_ENVIRONMENT_NAME:-not-set}"
 - echo "TF_ROOT=${TF_ROOT:-not-set}"
 - pwd
 - ls -la
```

1.7 - Tests Terraform

Validation locale, plan de test, sandbox, Terratest et tests de modules.

Source modal: 1.7 Tests Terraform : validation locale, sandbox, plans, Terratest et tests de modules

1. Tester Terraform : éviter que le premier vrai test soit la production

Tester Terraform ne signifie pas seulement lancer terraform validate . Il faut vérifier que le code est formaté, valide, sécurisé, que le plan est cohérent, que les modules créent bien les ressources attendues, et que les outputs permettent aux autres stacks de consommer proprement l'infrastructure.

Les tests Terraform doivent être proportionnés au risque. Un module réseau, IAM, base de données ou sécurité mérite beaucoup plus de contrôles qu'un simple log group ou une ressource non critique.

Zone	Question	Signal attendu
Syntaxe	Le code Terraform est-il valide ?	terraform validate OK.
Format	Le code est-il standardisé ?	terraform fmt -check OK.
Plan	Le changement est-il celui attendu ?	Pas de destroy/replace non prévu.
Module	Le module crée-t-il les bonnes ressources ?	Resources, tags, outputs et règles corrects.
Sécurité	Le code ouvre-t-il un risque ?	Scan IaC sans blocage critique.
Runtime	L'infra fonctionne-t-elle vraiment après apply ?	Healthcheck, connectivité, logs et métriques OK.

```
Code Terraform
 │
 ▼
Tests statiques
 ├── fmt
 ├── validate
 ├── tflint
 └── checkov / trivy / tfsec
 │
 ▼
Tests de plan
 ├── plan lisible
 ├── no destroy inattendu
 ├── review humaine
 └── policy-as-code éventuelle
 │
 ▼
Sandbox apply
...
```

2. Niveaux de tests Terraform

Un bon dispositif de tests Terraform combine plusieurs couches : tests statiques rapides, revue du plan, sandbox réelle, tests de modules et vérifications post-apply.

Niveau	Outils	Quand	Bloquant ?
Format	terraform fmt	À chaque commit/MR.	Oui
Validation	terraform validate	À chaque MR.	Oui
Lint	TFLint	À chaque MR.	Selon maturité
Sécurité	Checkov, tfsec, Trivy	MR et main.	Oui sur critique
Plan	terraform plan	MR, main, staging.	Oui
Intégration	Sandbox, Terratest	Modules critiques.	Oui selon criticité
Runtime	curl, cloud CLI, tests métier	Après apply.	Oui pour prod

```
Tests post-apply / runtime
 ├── healthcheck
 ├── connectivité
 ├── DNS/TLS
 └── métriques

 Tests d'intégration
 ├── sandbox apply
 ├── Terratest
 ├── outputs
 └── destroy

 Tests de plan
 ├── no destroy
 ├── no public exposure
 ├── diff attendu
 └── review humaine

...
```

3. Validation locale : première barrière avant MR

Les validations locales doivent être rapides, répétables et faciles à lancer. Elles évitent de pousser du code Terraform cassé dans la CI/CD.

Erreur	Détection	Correction
HCL mal formaté	terraform fmt -check	terraform fmt -recursive
Variable manquante	terraform plan	Ajouter default ou tfvars.
Provider absent	terraform init	Corriger required providers.
Type invalide	terraform validate	Corriger variable/object/map.
Backend inaccessible	terraform init	Vérifier credentials et backend.

```
# Initialisation
 terraform init

 # Formatage
 terraform fmt -recursive
 terraform fmt -check -recursive

 # Validation syntaxe / providers / modules
 terraform validate

 # Voir providers utilisés
 terraform providers

 # Plan local avec variables
 terraform plan -var-file="dev.tfvars"

 # Plan sauvegardé
 terraform plan -var-file="dev.tfvars" -out=tfplan
...
```

4. Tests de plan : vérifier l'intention avant l'action

Le plan est le meilleur test avant apply. Il montre les créations, modifications, remplacements et suppressions. En production, il doit être relu comme un document de changement.

Signal	Interprétation	Réaction
+	Création.	Vérifier coût, tags, sécurité.
~	Modification.	Comprendre impact runtime.
-/+	Remplacement.	Analyse obligatoire.
-	Suppression.	Stop si non attendu.

```
# Plan standard
 terraform plan -var-file="prod.tfvars"

 # Plan sauvegardé
 terraform plan -var-file="prod.tfvars" -out=tfplan

 # Plan lisible
 terraform show -no-color tfplan > plan.txt

 # Plan JSON pour analyse automatisée
 terraform show -json tfplan > plan.json
```

5. Sandbox : tester sans mettre la production en danger

Une sandbox est un environnement isolé permettant d'appliquer réellement Terraform, de vérifier que les ressources se créent, que les outputs sont corrects et que la destruction contrôlée fonctionne.

Critère	Attendu
Isolation	Compte/projet/région ou préfixe dédié.
Coût limité	Quotas, tailles petites, auto-destroy.
Données fictives	Aucune donnée production réelle.
State séparé	Backend sandbox distinct.
Credentials limités	Impossible d'agir sur production.
TTL	Nettoyage automatique ou planifié.

```
terraform init
 terraform plan -var-file="sandbox.tfvars" -out=tfplan
 terraform apply tfplan

 # Vérifications
 terraform output
 curl -fsS https://sandbox.example.com/health/

 # Nettoyage
 terraform destroy -var-file="sandbox.tfvars"
```

6. Tests de modules Terraform

Un module Terraform est une API d'infrastructure. Il faut tester son contrat : inputs, validations, outputs, naming, tags, sécurité par défaut et comportement sur plusieurs environnements.

Élément	Test attendu
Inputs	Types corrects, validations utiles, defaults sûrs.
Outputs	Valeurs nécessaires exposées, pas de secrets inutiles.
Naming	Ressources nommées avec convention projet/env/composant.
Tags	Tags obligatoires présents partout.
Sécurité	Pas de port dangereux ouvert par défaut.
Lifecycle	Ressources critiques protégées si applicable.
Idempotence	Deuxième plan après apply ne doit pas montrer de diff inattendu.

```
modules/
 network/
 main.tf
 variables.tf
 outputs.tf
 README.md
 examples/
 basic/
 main.tf
 terraform.tfvars
 multi_az/
 main.tf
 terraform.tfvars
 tests/
 README.md
```

7. Terratest : tests d'intégration automatisés

Terratest permet d'écrire des tests en Go qui lancent Terraform, appliquent l'infrastructure, vérifient des outputs ou comportements réels, puis détruisent les ressources. C'est puissant, mais plus coûteux que les tests statiques.

Cas	Pourquoi
Module réseau critique	Vérifier subnets, routes, outputs, connectivité.
Module load balancer	Tester endpoint, healthcheck, listener.
Module database	Vérifier backup, deletion protection, private access.
Module sécurité	Contrôler règles, ports, tags, IAM.
Module réutilisé par plusieurs équipes	Éviter régression lors des évolutions.

```
go test
 │
 ▼
 terraform init
 │
 ▼
 terraform apply
 │
 ▼
 Assertions
 ├── outputs
 ├── cloud resources
 ├── HTTP endpoint
 ├── tags
 └── security rules
 │
 ▼
 terraform destroy
...
```

8. Tests sécurité IaC

Les tests sécurité IaC détectent les mauvaises pratiques avant l'apply : ports trop ouverts, stockage non chiffré, IAM trop permissif, logs désactivés, ressources publiques ou secrets potentiels.

Outil	Usage
Checkov	Scan Terraform, cloud policies, Kubernetes, CI/CD.
tfsec	Scan sécurité Terraform simple et rapide.
Trivy config	Scan IaC, containers, dépendances.
TFLint	Lint Terraform et règles provider.
OPA / Conftest	Policy-as-code personnalisée.
Sentinel	Policy-as-code dans Terraform Cloud/Enterprise.

```
# Checkov
 checkov -d .

 # tfsec
 tfsec .

 # Trivy IaC
 trivy config .

 # TFLint
 tflint --init
```

```
tflint
```

1.8 - Coûts & FinOps

Comprendre l'impact coût des changements infra avant apply.

Source modal: 1.8 Coûts & FinOps : comprendre l'impact financier avant apply

1. FinOps : relier infrastructure, usage et responsabilité financière

Un changement Terraform peut créer des coûts invisibles : instance trop grosse, stockage oublié, logs trop verbeux, NAT gateway inutile, load balancer permanent, retention excessive, snapshots jamais nettoyés ou ressources sandbox non détruites.

Le rôle DevOps ne se limite pas à déployer. Il faut aussi comprendre le coût, expliquer les compromis, taguer correctement, détecter les dérives et rendre les dépenses visibles par environnement, service, équipe et projet.

Question	Pourquoi	Signal attendu
Quelle ressource est créée ?	Chaque ressource a un modèle de coût.	Instance, DB, LB, NAT, volume, logs, bucket.
Est-ce permanent ou temporaire ?	Une sandbox oubliée coûte tous les mois.	TTL, auto-destroy ou owner clair.
Quelle taille est choisie ?	Le sizing impacte directement la facture.	Instance type, storage, IOPS, replicas.
Les tags sont-ils présents ?	Sans tags, impossible d'attribuer les coûts.	Project, Environment, Owner, CostCenter.
Y a-t-il un coût réseau ?	NAT, egress, inter-zone et LB peuvent coûter cher.	Trafic estimé et architecture validée.
Une alerte budget existe-t-elle ?	Détecter rapidement les dérives.	Budget, seuils, notification.

```
Terraform change
  │
  ▼
Plan review
  ├── ressources ajoutées
  ├── ressources modifiées
  ├── tailles / volumes
  ├── logs / retention
  ├── réseau / egress
  └── tags
  │
  ▼
Cost estimation
  ├── coût mensuel estimé
  ├── delta vs existant
  ├── risque de dérive
  └── owner coût
  │
...
```

2. Cost drivers : ce qui fait réellement monter la facture

Les coûts cloud ne viennent pas seulement des machines. Le réseau, les logs, le stockage, les backups, les bases managées et les ressources oubliées peuvent représenter une part importante.

Famille	Exemples	Risque FinOps
Compute	VM, autoscaling, containers, nodes Kubernetes.	Surdimensionnement permanent.
Database	RDS, Cloud SQL, replicas, IOPS, stockage.	Coût élevé, difficile à réduire sans risque.
Network	NAT gateway, egress, inter-zone, load balancer.	Coûts invisibles liés au trafic.
Storage	Volumes, buckets, snapshots, backups.	Accumulation silencieuse.
Logs / Observabilité	Ingestion logs, retention, métriques custom.	Explosion si logs trop verbeux.
Environnements temporaires	Review apps, sandbox, tests Terratest.	Oubli de destruction.

```
Terraform apply
  │
  ├── ressource principale visible
  │   └── instance / DB / LB
  │
  └── coûts associés parfois oubliés
  ├── stockage attaché
  ├── snapshots
  ├── logs
  ├── trafic sortant
  ├── NAT
  ├── monitoring
  ├── backups
  └── IP publiques
```

3. Tags : base de l'attribution des coûts

Les tags permettent de savoir qui consomme quoi. Sans tags cohérents, impossible de répartir les coûts par équipe, service, environnement, produit ou client.

Tag	Exemple	Usage
Project	ideo-platform	Regrouper les coûts par projet.

Tag	Exemple	Usage
Environment	dev , staging , prod	Comparer non-prod vs prod.
Owner	platform-team	Responsabilité opérationnelle.
CostCenter	engineering	Refacturation ou budget interne.
ManagedBy	terraform	Distinguer IaC vs manuel.
Criticality	low , high , critical	Prioriser optimisation et support.
ExpireAt	2026-05-31	Nettoyage sandbox/review apps.

```
locals {
 common_tags = merge(
 {
 Project = var.project
 Environment = var.environment
 Owner = var.owner
 CostCenter = var.cost_center
 ManagedBy = "terraform"
 },
 var.extra_tags
 )
}
```

4. Revue du plan Terraform sous l'angle coût

La revue du plan ne doit pas seulement chercher les suppressions et les risques sécurité. Elle doit aussi identifier les ressources qui changent la facture.

Changement	Question FinOps
Nouvelle instance	Le type est-il adapté ? Peut-elle dormir hors horaires ?
Nouvelle DB	Backup, stockage, replicas et IOPS sont-ils justifiés ?
Nouveau load balancer	Peut-il être mutualisé ? Est-il permanent ?
Nouvelle NAT gateway	Le besoin réseau justifie-t-il ce coût fixe ?
Retention logs augmentée	Le volume d'ingestion est-il connu ?
Storage augmenté	Y a-t-il une lifecycle policy ?

```
Cost impact:
- Environment: staging
- New resources: 1 load balancer, 2 instances
- Removed resources: none
- Monthly cost impact: estimated +120 EUR/month
- Main cost drivers: load balancer + compute
- Tags: Project, Environment, Owner, CostCenter OK
- Budget alert: existing staging budget
- Cleanup: not temporary
```

5. Outils FinOps et estimation de coût

Plusieurs outils permettent d'estimer ou suivre les coûts liés à Terraform. Le bon choix dépend de la maturité équipe, du cloud utilisé et du niveau d'automatisation attendu.

Outil	Usage	Moment
Infracost	Estimation du coût depuis le plan Terraform.	MR / CI.
Terraform plan JSON	Analyse automatisée du diff.	MR / audit.
Policy-as-code	Bloquer ressources trop coûteuses ou non taguées.	CI/CD.
Cloud pricing calculator	Estimation manuelle d'architecture.	Design.
Cloud cost explorer	Suivi des dépenses réelles.	Exploitation.

```
# Exemple conceptuel
terraform plan -out=tfplan
terraform show -json tfplan > plan.json

infracost breakdown \
 --path plan.json \
 --format table

infracost diff \
 --path plan.json \
 --format json \
 --out-file infracost.json
```

6. Budgets, alertes et détection d'anomalies

Les budgets et alertes coût permettent de détecter une dérive avant la fin du mois. Ils doivent être définis par environnement, projet ou équipe, avec des seuils adaptés.

Seuil	Signification	Action
50%	Consommation normale à surveiller.	Notification faible.

Seuil	Signification	Action
75%	Risque de dépassement.	Review projet/environnement.
90%	Dépassement probable.	Action corrective ou validation owner.
100%	Budget atteint.	Escalade owner / manager / FinOps.
Anomalie soudaine	Hausse anormale vs baseline.	Investigation immédiate.

```
Budget policy:
- Project: ideo-platform
- Environment: staging
- Monthly budget: 300 EUR
- Warning: 75%
- Critical: 100%
- Owner: platform-team
- Notification: Slack #finops-alerts
- Review frequency: weekly
- Action: identify top cost drivers and cleanup unused resources
```

7. Design Terraform orienté coûts

Terraform peut intégrer des garde-fous FinOps : variables validées, tailles approuvées, tags obligatoires, rétention contrôlée, ressources temporaires avec expiration et protection contre certains choix coûteux.

Pattern	But
Approved sizes	Limiter instances non standards.
Environment defaults	Éviter staging/dev trop gros.
Mandatory tags	Attribuer les coûts.
TTL temporary	Nettoyer les ressources de test.
Retention by env	Limiter logs/backups non-prod.
Lifecycle policies	Archiver ou supprimer stockage ancien.

```
variable "instance_type" {
 description = "Approved instance type."
 type = string

 validation {
 condition = contains([
 "t3.micro",
 "t3.small",
 "t3.medium"
 ], var.instance_type)

 error_message = "Instance type is not approved."
 }
}

variable "log_retention_days" {
 description = "Log retention in days."
 type = number
...
```

8. FinOps dans la CI/CD Terraform

La CI/CD peut détecter les changements coûteux avant merge : estimation Infracost, contrôle des tags, blocage des tailles non approuvées, scan de ressources publiques, et commentaire automatique dans la merge request.

Contrôle	Décision possible
Tags obligatoires manquants	Bloquer la MR.
Instance type non approuvé	Bloquer ou demander approval.
Coût mensuel supérieur seuil	Approval FinOps / lead.
NAT gateway en dev	Warning ou justification obligatoire.
Retention logs trop longue	Correction demandée.
Sandbox sans ExpireAt	Bloquer.

```
Merge Request
 │
 ▼
terraform fmt / validate
 │
 ▼
terraform plan -out=tfplan
 │
 ▼
plan.json
 │
 ├── scan sécurité
 ├── contrôle tags
 ├── estimation coût
 └── policy-as-code
 │
 ▼
MR comment
...
```

2.5 - Merge Request infra

Comment documenter une MR Terraform lisible et validable.

Source modal: 2.5 Merge Request infra : documenter une MR Terraform lisible et validable

1. La Merge Request infra est un contrat de changement

Une MR Terraform ne doit pas seulement contenir du HCL. Elle doit expliquer pourquoi le changement existe, ce qu'il modifie , quel environnement est touché , quel risque opérationnel il introduit , comment le plan a été validé et comment revenir en arrière .

Le reviewer doit pouvoir lire la MR sans deviner l'intention. Une bonne MR infra réduit les incidents parce qu'elle force la réflexion avant l'apply : impact, sécurité, coût, state, rollback, monitoring et fenêtre de production.

Question	Réponse attendue
Pourquoi ce changement ?	Besoin métier, dette technique, sécurité, fiabilité, coût, standardisation.
Quel environnement ?	Dev, staging, preprod, production, sandbox ou review app.
Quelles ressources ?	Network, compute, DB, IAM, DNS, LB, logs, monitoring, storage.
Quel impact ?	Disponibilité, sécurité, coût, performance, données, observabilité.
Quel plan Terraform ?	Add/change/destroy/replace clairement résumés.
Quel rollback ?	Revert, apply précédent, mitigation, feature flag, action manuelle contrôlée.

```
Branche infra
  │
  ▼
Modification Terraform
  │
  ▼
Pipeline MR
  ├── fmt
  ├── validate
  ├── security scan
  ├── terraform plan
  ├── cost estimate
  └── plan artifact
  │
  ▼
Description MR
  ├── contexte
  ├── impact
...
```

2. Template complet de Merge Request Terraform

Un template standard évite les oublis et donne aux nouveaux DevOps un cadre clair. Il doit être assez complet pour la production, mais pas au point de devenir bureaucratique pour les changements simples.

Type de changement	Template
Tags, dashboard, alarme non critique	Version courte.
Compute, autoscaling, logs, stockage	Template complet conseillé.
Réseau, IAM, DB, DNS, load balancer	Template complet obligatoire.
Production avec risque downtime	Template complet + approval + fenêtre.
Import ou state operation	Template complet + mapping state/provider ID.

```
## Context
Why is this change needed?
- Business / technical reason:
- Related ticket:
- Target environment:
- Affected service:

## Scope
What changes?
- Terraform root:
- Modules changed:
- Resources affected:
- Cloud account / region:

## Terraform plan summary
- To add:
- To change:
- To destroy:
...
```

3. Documenter le plan Terraform dans la MR

Le plan est le cœur de la validation. Le reviewer ne doit pas avoir à lire 800 lignes brutes pour comprendre le risque. La MR doit contenir un résumé humain du plan, avec les points sensibles.

Symbole	Signification	Réaction reviewer
+	Création	Vérifier tags, coût, sécurité, naming.
~	Modification	Comprendre impact runtime.
-/+	Remplacement	Analyser downtime, identité, données.
-	Suppression	Validation explicite obligatoire.
<=	Data source lue	Vérifier dépendances et environnement.

```
terraform fmt -check -recursive
terraform validate

terraform plan -var-file="prod.tfvars" -out=tfplan
terraform show -no-color tfplan > plan.txt
terraform show -json tfplan > plan.json
```

4. Analyse de risque dans une MR infra

L'analyse de risque permet de décider du niveau de review, de l'approval nécessaire, de la fenêtre de production et de la stratégie de rollback.

Risque	Exemples	Contrôle attendu
Faible	Tags, description, dashboard, alarme non critique.	Review simple.
Moyen	Compute size, autoscaling, log retention, non-prod.	Plan review + validation staging.
Élevé	Security group, load balancer, DNS, IAM limité.	Review senior + approval.
Critique	Database, state, KMS, network core, suppression prod.	Double approval + fenêtre + rollback documenté.

```
Risk analysis:
- Availability impact: low / medium / high
- Security impact: low / medium / high
- Data impact: none / low / medium / high
- Cost impact: none / low / medium / high
- Rollback complexity: simple / moderate / complex
- Production window required: yes / no

Risk explanation:
This change modifies ...
The main risk is ...
The mitigation is ...
```

5. Rollback et mitigation dans une MR Terraform

Une MR infra doit expliquer comment revenir en arrière ou réduire l'impact si le changement provoque un incident. Pour Terraform, un rollback n'est pas toujours un simple revert Git.

Changement	Rollback typique	Attention
Tag / metadata	Revert commit puis apply.	Faible risque.
Security group	Restaurer règle précédente.	Ne pas ouvrir trop large en urgence.
DNS	Restaurer ancien record.	TTL et propagation.
Load balancer	Restaurer listener/routing/target group.	Health checks et trafic réel.
IAM	Restaurer policy précédente.	Propagation et blast radius.
Database	Rollback souvent non trivial.	Backup, snapshot, mitigation, DBA.
State operation	Restaurer state backup ou state mv inverse.	Très sensible, review obligatoire.

```
Rollback / mitigation:
- Preferred rollback:
 Revert this MR and apply the previous Terraform plan.

- If immediate rollback is unsafe:
 Apply a minimal mitigation:
 - restore previous security group rule
 - disable new listener rule
 - switch DNS back to previous target

- Validation after rollback:
 - health endpoint OK
 - 5xx rate stable
 - latency p95 stable
 - no new critical alert

- Owner during rollback:
 Platform team / on-call DevOps
```

6. Comment reviewer une MR Terraform

Le reviewer ne doit pas seulement regarder la syntaxe. Il doit vérifier le plan, le state, les dépendances, les risques, les tags, les coûts, les secrets, la sécurité et la capacité de rollback.

Zone	À vérifier
Contexte	La raison du changement est claire.
Plan	Add/change/destroy/replace cohérents.
Environment	Bon root, bon backend, bons tfvars.
State	Pas de confusion dev/staging/prod.
Réseau	Pas d'ouverture large non justifiée.
IAM	Pas de wildcard ou privilège excessif.
Données	DB, buckets, volumes protégés.

```
Good review comments:

- The plan shows a replacement of the load balancer.
 Is this expected? What is the downtime risk?

- This security group opens port 5432 to a broad CIDR.
 Can we restrict it to the application security group?

- The new database has deletion protection disabled.
 Is this intentional for production?

- This resource is missing CostCenter and Owner tags.

- Rollback mentions revert only.
 Can you add the exact validation steps after rollback?
```

7. Approval : qui doit valider quoi ?

Toutes les MR Terraform ne nécessitent pas le même niveau d'approbation. L'approval doit dépendre de l'environnement, du type de ressource et du risque.

Changement	Reviewer requis	Approval
Dev non critique	DevOps ou pair technique.	1 approval.
Staging compute/logs	DevOps + owner service si impact.	1 approval.
Prod compute/autoscaling	DevOps senior ou SRE.	1 à 2 approvals.
Prod réseau / LB / DNS	DevOps senior + owner applicatif.	Approval renforcé.
Prod IAM / sécurité	DevOps senior + sécurité.	Approval sécurité.
Prod DB / state	DevOps senior + DBA/SRE + owner.	Double approval + fenêtre.

```
MR ouverte
  │
  ▼
Pipeline vert
  │
  ├── fmt
  ├── validate
  ├── security scan
  ├── plan
  └── cost estimate
  │
  ▼
Review technique
  │
  ▼
Review risque
  ├── réseau
  ├── IAM
...
```

8. CI/CD attendue pour une MR Terraform

La CI/CD doit produire les preuves nécessaires à la review : format, validation, scan sécurité, plan lisible, estimation coût et artifacts.

Contrôle	But
fmt	Code standardisé.
validate	Syntaxe et configuration valides.
tflint	Qualité Terraform et règles provider.
checkov / trivy / tfsec	Risques sécurité IaC.
plan artifact	Review humaine complète.
plan JSON	Contrôles automatiques et coût.
infracost	Impact financier avant merge.

```
stages:
 - lint
 - validate
 - security
 - plan
 - cost
```

```
variables:
 TF_IN_AUTOMATION: "true"
 TF_INPUT: "false"
 TF_ROOT: "infra/live/staging/app"

default:
 image: hashicorp/terraform:1.6.6
 before_script:
 - cd "$TF_ROOT"
 - terraform init

...
```

2.7 - Approvals & protections

Branches protegees, environnements proteges, approvals et separation des droits.

Source modal: 2.7 Approvals & protections : branches, environnements, droits et production

1. Approvals & protections : empêcher l'erreur humaine en production

Les protections GitLab ne sont pas de la bureaucratie. Elles servent à éviter qu'un changement non relu, non testé, mal ciblé ou exécuté par la mauvaise personne modifie la production.

Pour l'infrastructure, le risque est plus élevé qu'une simple MR applicative : un mauvais apply Terraform peut modifier un security group, casser un DNS, supprimer une ressource, changer une policy IAM, exposer un service ou bloquer une base de données.

Protection	But	Exemple
Protected branches	Empêcher push direct ou merge non autorisé.	main , release/* .
Merge request approvals	Forcer une revue humaine avant merge.	1 ou 2 approvals selon risque.
Protected environments	Limiter qui peut déployer vers production.	Seuls Maintainers/SRE peuvent lancer apply prod.
Protected variables	Limiter les secrets aux branches/tags protégés.	Credentials prod indisponibles en feature branch.
CODEOWNERS	Assigner automatiquement les bons reviewers.	Sécurité relit IAM, DBA relit DB.
Required pipeline success	Bloquer merge si CI échoue.	fmt, validate, scan, plan obligatoires.

```
Developer branch
  │
  ▼
Merge Request
  │
  ├── terraform fmt
  ├── terraform validate
  ├── security scan
  ├── terraform plan
  └── cost estimate
  │
  ▼
Approval rules
  ├── DevOps reviewer
  ├── Security reviewer if IAM/network
  ├── DBA reviewer if database
  └── SRE reviewer if prod risk
  │
...
```

2. Branches protégées : contrôler le chemin vers production

Les branches protégées empêchent les push directs, limitent qui peut merger et garantissent que les changements passent par une MR, une CI et une review.

Branche	Usage	Protection
feature/*	Travail individuel.	Pas de secrets prod, pas d'apply prod.
develop	Intégration dev.	Pipeline requis selon équipe.
release/*	Préparation staging/preprod.	Review + pipeline vert.
main	Source officielle production.	Protected, MR only, approvals, pipeline success.
hotfix/*	Correction urgente encadrée.	Review rapide mais obligatoire si prod.

```
feature/infra-change
  │
  ▼
Merge Request
  │
  ├── pipeline MR
  ├── plan artifact
  ├── review
  └── approval
  │
  ▼
main protected
  │
  ▼
Pipeline production
  │
  ├── plan final
  ├── apply manual
...
```

3. Environnements protégés : limiter qui peut déployer

Une branche protégée contrôle le code. Un environnement protégé contrôle le déploiement. Pour Terraform, cela signifie limiter qui peut lancer un apply sur staging ou production.

Environnement	Déploiement autorisé par	Niveau de contrôle
development	Developers / DevOps selon équipe.	Souple, mais secrets limités.
staging	DevOps, Maintainers.	Contrôle intermédiaire.
preproduction	DevOps senior / SRE.	Proche production.
production	SRE, Maintainers autorisés, release managers.	Strict, approval + manual apply.

```
terraform_apply_prod:
 stage: apply
 environment:
 name: production
 script:
 - terraform apply tfplan
 when: manual
 allow_failure: false
 rules:
 - if: '$CI_COMMIT_BRANCH == "main"'
 tags:
 - protected
 - terraform
 - prod
```

4. Approval rules : adapter la validation au risque

Les règles d'approval doivent refléter la criticité du changement. Un tag non critique ne nécessite pas le même niveau de validation qu'un changement IAM, database, DNS ou network core.

Type de MR	Approval minimum	Reviewer recommandé
Docs, tags, dashboard non critique	1	DevOps pair.
Compute non-prod	1	DevOps ou service owner.
Staging proche prod	1 à 2	DevOps + owner applicatif.
Production compute/autoscaling	2	DevOps senior + SRE.
Production network/DNS/LB	2	DevOps senior + service owner.
Production IAM/security	2 + sécurité	Security reviewer obligatoire.
Production DB/state	2 + expert	DBA/SRE + DevOps senior.

```
Low risk:
 1 reviewer

Medium risk:
 1 DevOps reviewer + plan artifact

High risk:
 2 reviewers + rollback documented

Critical:
 2 reviewers + expert approval + production window + post-checks
```

5. Séparation des droits : éviter le super-pouvoir permanent

La séparation des droits réduit le risque d'erreur, de compromission et d'action non autorisée. Un développeur peut contribuer au code Terraform sans avoir le droit d'appliquer en production.

Rôle	Droits GitLab	Droits infra
Developer	Créer branches/MR.	Aucun apply prod.
DevOps	Reviewer, maintenir pipelines.	Apply dev/staging selon règles.
Maintainer	Merge sur branches protégées.	Apply prod si autorisé.
SRE / Release manager	Valider et déclencher prod.	Accès prod contrôlé et audité.
Security	Review IAM/secrets/network.	Pas forcément apply, mais approval sécurité.
Break-glass	Accès urgence temporaire.	Très contrôlé, journalisé, révocable.

```
Developer
 ├── écrit Terraform
 ├── ouvre MR
 └── lit le plan

Reviewer
 ├── relit code
 ├── relit plan
 └── approuve

Authorized deployer
 ├── déclenche apply
 ├── surveille production
 └── documente résultat
```

6. Production : apply manuel, fenêtre et traçabilité

En production, l'apply Terraform doit être traité comme une opération contrôlée. Le pipeline doit empêcher l'exécution automatique non maîtrisée et fournir une trace complète.

Règle	Pourquoi
Apply manuel	Éviter modification prod automatique sans décision.
Pipeline vert	fmt, validate, plan et scans validés.
Approval avant merge	Review du plan et du risque.
Environment protégé	Seules personnes autorisées peuvent déployer.
Fenêtre si risque	Équipe disponible pour surveiller et rollback.
Post-checks	Vérifier que la prod fonctionne réellement.

```
Before apply:
- MR approved
- pipeline green
- plan reviewed
- no unexpected destroy/replace
- rollback documented
- dashboards open
- production window confirmed if needed

During apply:
- trigger manual job
- monitor job logs
- do not run concurrent apply

After apply:
- run health checks
- check logs
- check metrics
...
```

7. CODEOWNERS : router les reviews aux bonnes personnes

CODEOWNERS permet d'assigner automatiquement des reviewers selon les fichiers modifiés. Pour l'infra, c'est très utile : les changements IAM, réseau, DB ou production doivent être relus par les bonnes personnes.

Fichiers modifiés	Reviewer attendu	Pourquoi
infra/live/prod/**	SRE / platform lead	Impact production.
modules/iam/**	Security	Risque privilèges.
modules/network/**	Network / DevOps senior	Risque connectivité.
modules/database/**	DBA / SRE	Risque data.
.gitlab-ci.yml	DevOps	Risque pipeline/secrets.

```
# Global infrastructure owners
/infra/ @platform-team

# Production requires SRE review
/infra/live/prod/ @sre-team @platform-leads

# IAM and security-sensitive files
/infra/modules/iam/ @security-team @platform-team
/infra/**/iam*.tf @security-team

# Network
/infra/modules/network/ @network-team @platform-team
/infra/**/security_group*.tf @platform-team

# Databases
/infra/modules/database/ @dba-team @sre-team
/infra/**/rds*.tf @dba-team

...
```

8. Secrets, variables protégées et scopes

Les protections GitLab doivent empêcher qu'une branche non protégée, une MR externe, un runner partagé ou un job non prévu accède aux secrets production.

Option	Usage	Exemple
Masked	Masquer dans les logs.	Tokens, passwords, cloud keys.
Protected	Disponible uniquement branches/tags protégés.	Credentials production.
Environment scope	Limiter à un environnement.	production , staging .
File variable	Stocker clé/certificat/kubeconfig en fichier.	Kubeconfig, private key.

```
DEV credentials:
 protected: false or limited
 environment scope: development

STAGING credentials:
 protected: true if release branches are protected
 environment scope: staging

PROD credentials:
 protected: true
 environment scope: production
 available only from main / protected tags
```

Part 3 - DevOps - Terraform, State, GitLab CI/CD & Gestion d'incidents (Part 3)

This part groups related operational topics from the IDEO-Lab DevOps guide. Each chapter below extracts the practical knowledge embedded in the interactive modal panels.

Step	Topic	Purpose
2.8	Debug pipeline	Diagnostiquer un pipeline GitLab qui bloque, echoue ou se comporte mal.
3.5	Stratégies de déploiement	Rolling, blue/green, canary, maintenance page et rollback applicatif.
3.6	Capacité & performance	CPU, RAM, disque, connexions, queues, saturation et seuils d'alerte.
3.7	Incidents base de données	Connexions, locks, lenteurs, disque plein, backups et restauration.
3.8	Incidents réseau	DNS, TLS, Nginx, firewall, load balancer, routes et ports.
4.4	IAM approfondi	Roles, policies, assume-role, separation des privileges et break-glass.
4.5	Supply chain CI/CD	Images Docker, dépendances, runners, artifacts et provenance.
4.6	Kubernetes pour DevOps	Concepts utiles même si le poste est surtout Terraform/GitLab.
4.7	Linux hardening	Systemd, utilisateurs, SSH, firewall, logs, permissions et mises à jour.
5.5	Former les nouveaux DevOps	Progression, exercices, erreurs volontaires, revues et rituels d'equipe.
5.6	Glossaire avancé	Les termes que les nouveaux DevOps rencontrent tous les jours.
6.1	Annexes pratiques	Plans de formation, matrices de risques, procédures go/no-go et modèles opérationnels.

2.8 - Debug pipeline

Diagnostiquer un pipeline GitLab qui bloque, echoue ou se comporte mal.

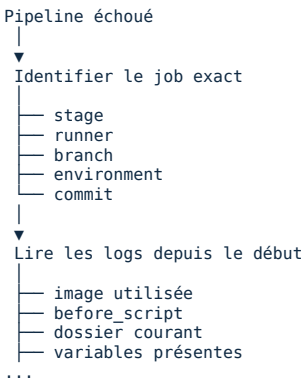
Source modal: 2.8 Debug pipeline : diagnostiquer un pipeline GitLab qui bloque, échoue ou se comporte mal

1. Déboguer un pipeline GitLab : lire une exécution comme une enquête

Un pipeline GitLab qui échoue ne doit pas être corrigé au hasard. Il faut lire le job comme un journal d'exécution : image Docker, runner utilisé, variables disponibles, dossier courant, before_script , commandes lancées, artifacts produits, credentials utilisés et message d'erreur exact.

La plupart des pannes CI/CD infra viennent de problèmes simples mais cachés : runner indisponible, tags incompatibles, variable protégée inaccessible, mauvais répertoire, mauvais environnement, artifact absent, state lock Terraform, droits cloud insuffisants, ou différence entre le contexte local et le contexte CI.

Famille	Symptôme	Cause probable
Runner	Job bloqué en pending .	Runner indisponible, tags incorrects, runner non protégé.
Variables	Credentials absents ou vides.	Variable protected/scoped inaccessible.
Chemins	cd: no such file , artifact absent.	Mauvais TF_ROOT , paths relatifs incorrects.
Terraform init	Backend inaccessible.	Credentials, bucket, state key, réseau, IAM.
Terraform plan	Erreur provider, variable manquante.	Provider lock, tfvars, droits cloud, version.
Terraform apply	Lock, drift, plan expiré.	Apply concurrent, artifact perdu, state lock.



2. Méthode terrain : diagnostic en 9 étapes

Étape	Action	Résultat attendu
1	Identifier le pipeline, le job, le commit et la branche.	Contexte exact connu.
2	Lire le log depuis le début, pas seulement la dernière ligne.	Erreur replacée dans son contexte.
3	Identifier image, runner, tags et environnement.	Exécution CI comprise.
4	Vérifier dossier courant et chemins utilisés.	Moins d'erreurs de path/artifacts.
5	Vérifier présence des variables sans afficher leur valeur.	Secrets protégés.
6	Comparer local vs CI : versions, tfvars, backend, provider.	Différences identifiées.
7	Classer la panne : runner, variable, state, provider, cloud.	Hypothèse principale claire.

```
Job failed ou pending
 │
 ▼
 Job pending ?
 │
 ├── Oui
 │   ├── runner actif ?
 │   ├── tags corrects ?
 │   ├── runner protégé ?
 │   └── quota/concurrency ?
 │
 └── Non
 │
 ▼
 Erreur avant script ?
 │
 ├── image introuvable
 ├── pull denied
...
```

3. Runner GitLab : jobs pending, tags, executor et disponibilité

Un pipeline peut être correct mais rester bloqué si aucun runner compatible n'est disponible. Le runner est l'agent d'exécution : tags, protection, executor, accès réseau et capacité sont critiques.

Symptôme	Cause probable	Correction
Job pending	Aucun runner avec les tags demandés.	Corriger tags job ou runner.
Runner offline	Service runner arrêté ou serveur KO.	Vérifier service et logs runner.
Job bloqué sur protected branch	Runner non protégé.	Utiliser runner protégé.
Image pull failed	Registry inaccessible ou credentials absents.	Vérifier registry auth.
Timeout	Job trop long ou runner saturé.	Optimiser job ou timeout/concurrency.

```
# Vérifier runners
 gitlab-runner verify
 gitlab-runner list

 # Service runner
 systemctl status gitlab-runner --no-pager
 journalctl -u gitlab-runner -n 100 --no-pager
 journalctl -u gitlab-runner -f

 # Ressources machine runner
 uptime
 free -h
 df -h
 top
```

4. Variables GitLab : protected, masked, scoped et absentes

Beaucoup de jobs échouent parce qu'une variable existe dans GitLab mais n'est pas disponible dans le contexte du job : branche non protégée, environnement différent, scope mal configuré, variable définie au mauvais niveau ou nom incorrect.

Symptôme	Cause probable	Diagnostic sûr
Variable vide	Variable protected sur branche non protégée.	Afficher présence, pas valeur.
Auth cloud failed	Credential absent ou scope incorrect.	Vérifier branch/env scope.
Backend inaccessible	Secret state backend manquant.	Contrôler variables backend.
Job local OK, CI KO	Différence de tfvars ou variables CI.	Comparer variables non sensibles.
Secret visible dans logs	Variable non masked ou debug dangereux.	Corriger variable + nettoyer logs si possible.

```
check_var() {
 local name="$1"
 local value="${!name:-}"
```

```
if [ -z "$value" ]; then
echo "$name is missing"
return 1
fi

echo "$name is present"
}

check_var "AWS_ACCESS_KEY_ID"
check_var "AWS_SECRET_ACCESS_KEY"
check_var "TF_VAR_environment"
```

5. Chemins, working directory, includes et artifacts

Les erreurs de chemins sont très fréquentes en GitLab CI : le job ne s'exécute pas dans le dossier attendu, l'artifact est généré ailleurs, ou TF_ROOT pointe vers un chemin différent entre dev, staging et prod.

Erreur	Cause probable	Correction
cd: no such file	TF_ROOT incorrect.	Afficher pwd , ls , corriger chemin.
Artifact absent	Path artifact relatif au mauvais dossier.	Aligner paths et working directory.
tfplan introuvable	Apply ne récupère pas l'artifact du plan.	Ajouter dependencies ou needs .
Include non chargé	Chemin include incorrect ou rules non déclenchées.	Vérifier YAML final et règles.
Plan OK, apply KO	Apply lancé dans un autre dossier.	Standardiser TF_ROOT .

```
echo "TF_ROOT=$TF_ROOT"
 pwd
 ls -la
 find . -maxdepth 3 -type f | sort | head -100

 cd "$TF_ROOT"
 pwd
 ls -la
```

6. Debug Terraform dans GitLab CI

Les erreurs Terraform en CI peuvent venir de la configuration, du backend, des credentials, des variables, du provider lock, de la version Terraform ou du cloud provider.

Commande	Ce qu'elle valide
terraform --version	Version Terraform réellement utilisée par CI.
terraform init	Backend, providers, modules, credentials.
terraform validate	Syntaxe et cohérence configuration.
terraform providers	Providers utilisés et provenance.
terraform plan	Droits cloud, variables, drift, diff réel.
terraform state list	Accès au state et ressources connues.

```
before_script:
 - echo "TF_ROOT=$TF_ROOT"
 - cd "$TF_ROOT"
 - terraform --version
 - terraform init -input=false

 terraform_validate:
 stage: validate
 script:
 - terraform fmt -check -recursive
 - terraform validate

 terraform_plan:
 stage: plan
 script:
 - terraform plan -input=false -out=tfplan
 - terraform show -no-color tfplan > plan.txt
```

7. State lock, apply concurrent et plan obsolète

Les problèmes de state sont sensibles. Un lock Terraform protège contre les applies concurrents. Le déverrouiller sans preuve peut corrompre une opération en cours ou créer un état incohérent.

Symptôme	Cause probable	Réflexe
State lock actif	Apply en cours ou job mort.	Identifier détenteur du lock.
Plan expiré	State changé entre plan et apply.	Relancer plan.
Apply concurrent	Deux jobs sur même state.	Bloquer concurrence CI.
Diff incohérent	Mauvais backend ou workspace.	Vérifier state key/env.
Resource already exists	Ressource créée hors state.	Import ou correction code/state.

```
terraform state list
 terraform state show RESOURCE
 terraform plan -refresh-only
 terraform state pull > state-backup.json
```

```
# Dangereux : seulement après preuve que le lock est orphelin
terraform force-unlock LOCK_ID
```

8. Logs utiles : quoi lire et quoi ajouter

Un bon log de pipeline doit aider à comprendre le contexte sans exposer de secret. Il doit afficher les informations non sensibles : branche, source du pipeline, environnement, dossier, versions, job, runner et chemins.

À éviter	Pourquoi	Alternative
env	Affiche potentiellement tous les secrets.	Afficher variables non sensibles une par une.
printenv	Risque de fuite massive.	Fonction check_var .
set -x	Trace commandes avec valeurs substituées.	Logs explicites contrôlés.
cat *.tfvars	Peut contenir secrets ou configs sensibles.	Valider présence du fichier seulement.
terraform output brut	Peut afficher outputs sensibles mal déclarés.	Afficher outputs non sensibles ciblés.

```
echo "Job: $CI_JOB_NAME"
 echo "Stage: $CI_JOB_STAGE"
 echo "Branch: $CI_COMMIT_BRANCH"
 echo "Pipeline source: $CI_PIPELINE_SOURCE"
 echo "Environment: ${CI_ENVIRONMENT_NAME:-not-set}"
 echo "TF_ROOT: $TF_ROOT"
 echo "Working directory:"
 pwd
 echo "Terraform version:"
 terraform --version
```

3.5 - Stratégies de déploiement

Rolling, blue/green, canary, maintenance page et rollback applicatif.

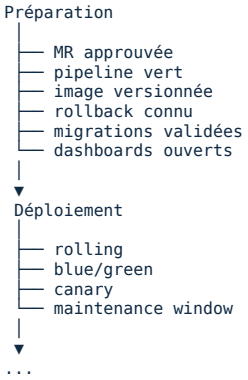
Source modal: 3.5 Stratégies de déploiement : rolling, blue/green, canary, maintenance page et rollback

1. Déployer proprement : réduire le risque client

Un déploiement production ne consiste pas seulement à pousser une nouvelle version. C'est une opération contrôlée qui doit limiter l'impact utilisateur, surveiller les signaux critiques, permettre un retour arrière rapide et laisser une trace exploitable.

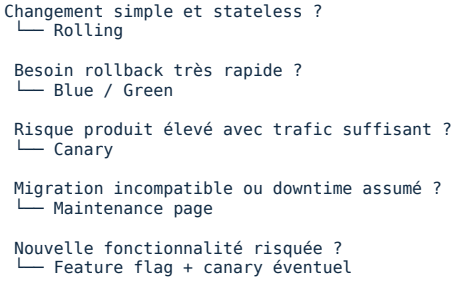
Le choix de stratégie dépend du contexte : criticité du service, nombre d'instances, présence d'un load balancer, compatibilité des migrations, tolérance au downtime, maturité monitoring, capacité de rollback et niveau de trafic.

Question	Pourquoi	Réponse attendue
Quel est le risque utilisateur ?	Détermine la stratégie.	Faible, moyen, élevé, critique.
Peut-on avoir deux versions en même temps ?	Nécessaire pour rolling/canary.	Compatibilité API, DB, cache, sessions.
Le rollback est-il testé ?	Un rollback improvisé est dangereux.	Commande, image précédente, DB compatible.
Quels signaux surveiller ?	Détecter vite une mauvaise version.	5xx, latence, erreurs métier, saturation.
Faut-il une fenêtre de maintenance ?	Changements incompatibles ou DB sensibles.	Oui/non, durée, communication.
Qui décide du go/no-go ?	Évite la confusion pendant l'opération.	Release owner, SRE, DevOps, métier.



2. Comparatif des stratégies de déploiement

Stratégie	Principe	Avantages	Risques	Cas adapté
Recreate	Stop ancienne version, start nouvelle.	Simple.	Downtime.	Petites apps internes.
Rolling	Remplacer les instances progressivement.	Peu ou pas d'interruption.	Deux versions coexistent.	Services stateless.
Blue/Green	Deux environnements, bascule trafic.	Rollback rapide par rebascule.	Coût double temporaire.	Applications critiques.
Canary	Exposer un petit % du trafic.	Risque limité.	Observabilité obligatoire.	Fort trafic, SRE mature.
Feature flags	Activer fonction sans redéployer.	Rollback fonctionnel rapide.	Dette de flags.	Nouvelles fonctionnalités.
Maintenance page	Couper proprement l'accès utilisateur.	Sûr pour changements incompatibles.	Downtime visible.	Migration DB non compatible.



3. Rolling deployment : remplacer progressivement

Le rolling deployment remplace les instances une par une ou par petits lots. Il est efficace pour les applications stateless derrière un load balancer, avec health checks fiables et compatibilité entre versions.

Condition	Pourquoi
Healthcheck fiable	Ne pas envoyer du trafic vers une instance cassée.
Sessions externalisées	Éviter perte de session si instance remplacée.
Deux versions compatibles	Ancienne et nouvelle version coexistent temporairement.
Capacité suffisante	Supporter le trafic pendant remplacement partiel.
Rollback rapide	Revenir à l'image ou release précédente.

```
Version N active
 │
 ├── instance 1 : vN
 ├── instance 2 : vN
 ├── instance 3 : vN
 └── instance 4 : vN
 │
 ▼
 Rolling update
 │
 ├── instance 1 → vN+1
 ├── healthcheck OK
 ├── instance 2 → vN+1
 ├── healthcheck OK
 └── continuer
 │
 ▼
 Version N+1 active partout
```

4. Blue / Green : basculer entre deux environnements

Le blue/green consiste à maintenir deux environnements : l'un sert le trafic, l'autre reçoit la nouvelle version. Après validation, le trafic bascule vers le nouvel environnement. Le rollback consiste souvent à rebascule le trafic vers l'ancien.

Rollback rapide	Rebasculer vers l'environnement précédent.
Validation avant trafic	Tester green avant exposition utilisateur.
Downtime réduit	La bascule peut être quasi instantanée.
Isolation	Nouvelle version préparée sans toucher blue.

```
BLUE actif
 ├── version N
 └── reçoit 100% trafic

 GREEN préparé
 ├── version N+1
 ├── smoke tests
 ├── health checks
 └── validation métier
 │
 ▼
 Bascule trafic
 ├── load balancer
 ├── DNS pondéré
 └── service mesh
 │
 ▼
 GREEN actif
...
```

5. Canary deployment : exposer progressivement

Le canary déploie la nouvelle version sur une petite part du trafic ou des utilisateurs. On augmente progressivement si les signaux restent bons. C'est puissant, mais exige une observabilité solide et des critères de décision objectifs.

Signal	Critère Go	Critère rollback
HTTP 5xx	Stable ou inférieur baseline.	Hausse anormale.
Latence p95/p99	Pas de dérive significative.	Latence supérieure seuil.
Erreurs applicatives	Pas de nouvelle exception critique.	Nouvelle erreur fréquente.
Métriques métier	Parcours clé OK.	Échec paiement, login, commande, job.
Saturation	CPU/RAM/DB stables.	Queue backlog ou DB locks.

```
Version N : 100%
 Version N+1 : 0%
 │
 ▼
 Canary 5%
 ├── surveiller 5xx
 ├── latence
 ├── erreurs métier
 └── logs
 │
 ▼
 Canary 25%
 │
 ▼
```

```
 Canary 50%
 |
 ▼
 Canary 100%
...
```

6. Maintenance page : downtime contrôlé et assumé

Une page de maintenance est adaptée quand le changement n'est pas compatible avec un trafic actif : migration majeure, modification non backward compatible, opération DB sensible, changement de stockage, ou bascule d'infrastructure difficile à rendre transparente.

Cas	Pourquoi maintenance
Migration DB non compatible	Ancien et nouveau code ne peuvent pas coexister.
Changement stockage	Risque d'écriture concurrente.
Bascule DNS/infra critique	État transitoire non stable.
Import massif ou correction data	Préserver cohérence des données.
Application mono-instance	Impossible de déployer sans interruption.

```
Before:
 - annoncer fenêtre
 - vérifier backup
 - vérifier rollback
 - préparer page maintenance
 - arrêter jobs asynchrones
 - ouvrir dashboards

 During:
 - activer maintenance
 - vérifier que trafic utilisateur est bloqué proprement
 - exécuter migration / déploiement
 - exécuter smoke tests internes

 After:
 - désactiver maintenance
 - vérifier health
 - vérifier logs
...
```

7. Rollback applicatif : restaurer vite, sans improvisation

Le rollback applicatif consiste à revenir à une version stable de l'application, de la configuration ou du routage. Il doit être préparé avant le déploiement, testé en staging et documenté dans la MR.

Type	Action	Risque
Image précédente	Redéployer l'image N-1.	DB incompatible possible.
Config précédente	Restaurer variables/config.	Secrets ou cache à recharger.
Blue/Green	Rebasculer vers blue.	DB ou sessions modifiées.
Canary	Ramener trafic canary à 0%.	Utilisateurs déjà exposés.
Feature flag	Désactiver fonctionnalité.	Dette de flag à nettoyer.
DB restore	Restaurer backup/snapshot.	Perte de données récentes possible.

```
Rollback plan:
 - Previous version: api:1.8.1
 - New version: api:1.8.2
 - Rollback command:
 kubectl rollout undo deployment/api

 - Conditions to rollback:
 - 5xx > 2% for 5 minutes
 - p95 latency > 800ms for 10 minutes
 - critical business flow failing

 - Validation after rollback:
 - health endpoint OK
 - 5xx back to baseline
 - no new critical logs
 - business smoke test OK
```

8. Déploiement et migrations de base de données

Les migrations DB sont souvent le point le plus dangereux d'un déploiement. Une stratégie de déploiement moderne impose des migrations compatibles, progressives et idéalement réversibles ou au moins mitigables.

Règle	Pourquoi
Migration backward compatible	Ancien et nouveau code coexistent.
Pas de suppression immédiate	Ancienne version peut encore lire l'ancien champ.
Backfill contrôlé	Éviter locks longs et saturation.
Index en ligne si possible	Limiter blocage DB.
Backup avant migration risquée	Mitigation en cas de problème data.

```
1. Expand
 ├── ajouter nouvelle colonne nullable
 ├── ajouter nouvelle table
 ├── ajouter index sans casser ancien code
 └── garder compatibilité

 2. Migrate
 ├── backfill progressif
 ├── double-write éventuel
 ├── lecture nouvelle donnée
 └── validation

 3. Contract
 ├── supprimer ancien champ
 ├── retirer ancien code
 └── nettoyer dette après stabilisation
```

3.6 - Capacité & performance

CPU, RAM, disque, connexions, queues, saturation et seuils d'alerte.

Source modal: 3.6 Capacité & performance : CPU, RAM, disque, connexions, queues, saturation et seuils

1. Capacité & performance : éviter la panne avant la panne

Un système ne tombe pas seulement quand une ressource atteint 100%. Il devient souvent instable avant : latence qui augmente, files d'attente qui grossissent, connexions saturées, disque lent, swap, timeouts, erreurs 5xx, workers bloqués ou base de données sous pression.

La capacité doit être suivie dans le temps : trafic normal, pics horaires, batchs de nuit, effets d'un déploiement, croissance mensuelle, saisonnalité, pics marketing, incidents externes et changements d'architecture.

Zone	Ce qu'on observe	Risque si ignoré
CPU	Utilisation, load average, steal time, throttling.	Latence, timeouts, jobs lents.
RAM	Mémoire disponible, swap, OOM killer.	Crash process, redémarrages, lenteur.
Disque	Espace, inodes, I/O wait, latence disque.	Écriture impossible, DB lente, logs bloqués.
Réseau	Débit, erreurs, connexions, saturation LB.	Timeouts, pertes, erreurs intermittentes.
Database	Connexions, locks, slow queries, cache hit ratio.	Application lente ou bloquée.
Queues	Backlog, âge des messages, workers actifs.	Retards métier, jobs jamais traités.

```
Trafic utilisateur / batchs / jobs
  │
  ▼
 Application
 ├── CPU
 ├── RAM
 ├── threads / workers
 ├── connexions DB
 └── appels externes
  │
  ▼
 Dépendances
 ├── database
 ├── cache
 ├── queue
 ├── stockage
 └── réseau
  │
...
```

2. Signaux clés de capacité et performance

Les bons signaux combinent métriques système, métriques applicatives et métriques métier. Une CPU haute seule ne suffit pas : il faut corréler avec la latence, les erreurs, la DB et les queues.

Signal	Exemple	Pourquoi
Latence	p50, p95, p99.	Montre l'expérience utilisateur réelle.
Trafic	Requests/sec, jobs/min, messages/sec.	Comprendre la charge entrante.
Erreurs	HTTP 5xx, exceptions, failed jobs.	Détecter régression ou saturation.
Saturation	CPU, RAM, DB connections, queue backlog.	Identifier la ressource limitante.

```
Production Capacity Dashboard
  │
  ├── HTTP RPS
  ├── HTTP 5xx rate
  ├── Latency p50 / p95 / p99
  ├── CPU / load average
  ├── RAM / swap / OOM
  ├── Disk usage / I/O wait
  ├── DB connections / locks / slow queries
  ├── Redis / cache hit ratio
  ├── Queue backlog / oldest message age
  ├── Worker count / failed jobs
  └── Last deployment marker
```

3. CPU, RAM, load average et saturation mémoire

Métrique	Lecture
CPU user	Temps passé dans le code applicatif.
CPU system	Temps noyau, réseau, I/O, syscalls.
I/O wait	CPU attend le disque ou stockage.
Steal time	VM pénalisée par l'hyperviseur.
Load average	Nombre de tâches en attente CPU/I/O.

```
# Charge système
```

```
uptime

# Process CPU/RAM
top
ps aux --sort=-%cpu | head
ps aux --sort=-%mem | head

# Mémoire
free -h
vmstat 1 5

# OOM killer
dmesg | grep -i "killed process"
journalctl -k | grep -i "out of memory"

# CPU détaillé si outils installés
mpstat 1 5
...
```

4. Disque, I/O, inodes, logs et saturation stockage

Le disque est une cause très fréquente d'incident : logs trop volumineux, snapshots oubliés, uploads non nettoyés, inodes épuisés, I/O wait élevé, base de données ralentie ou partition système pleine.

Symptôme	Cause probable	Action
df -h haut	Logs, uploads, backups, cache.	Identifier dossier, cleanup contrôlé.
df -i à 100%	Trop de petits fichiers.	Nettoyer cache/tmp/logs fragmentés.
I/O wait élevé	Disque lent ou saturé.	Identifier process, DB, backup, logs.
Fichier supprimé mais espace non libéré	Process garde le handle ouvert.	lsof +L1 , restart service ciblé.
Logs explosent	Erreur en boucle ou debug activé.	Corriger cause, réduire niveau log.

```
# Espace disque
df -h

# Inodes
df -i

# Gros dossiers
du -sh /* 2>/dev/null | sort -h
du -sh /var/log/* 2>/dev/null | sort -h
du -sh /tmp/* 2>/dev/null | sort -h

# Journald
journalctl --disk-usage
sudo journalctl --vacuum-time=7d

# Fichiers supprimés encore ouverts
lsof +L1

...
```

5. Database : connexions, locks, slow queries et saturation

Beaucoup de problèmes de performance applicative viennent de la base de données : trop de connexions, pool mal dimensionné, requêtes lentes, locks, index manquants, transactions longues ou saturation I/O.

Signal	Pourquoi
Connexions actives	Risque d'atteindre la limite max connections.
Requêtes lentes	Augmentent latence et consommation CPU/I/O.
Locks	Peuvent bloquer toute une chaîne applicative.
Transactions longues	Maintiennent locks et empêchent vacuum/cleanup.
Cache hit ratio	Indique si DB lit trop sur disque.
Replication lag	Risque de lecture obsolète ou failover difficile.

```
-- Connexions actives
select state, count(*)
from pg_stat_activity
group by state;

-- Requêtes longues
select pid, now() - query_start as duration, state, query
from pg_stat_activity
where state <> 'idle'
order by duration desc
limit 10;

-- Locks
select * from pg_locks where not granted;

-- Taille des tables
select relname, pg_size_pretty(pg_total_relation_size(relid))
from pg_catalog.pg_statio_user_tables
...
```

6. Queues, workers, backlog et traitement asynchrone

Les queues permettent d'absorber des pics, mais elles peuvent aussi cacher une saturation. Un système peut répondre aux utilisateurs tout en accumulant des heures de retard dans les jobs.

Métrique	Ce qu'elle dit	Risque
Backlog	Nombre de messages en attente.	Retard de traitement.
Oldest message age	Âge du plus vieux message.	SLA métier dépassé.
Throughput	Messages traités par minute.	Capacité workers insuffisante.
Failed jobs	Jobs en erreur.	Bug applicatif ou dépendance KO.
Retries	Retentatives.	Peut amplifier la charge.
Worker utilization	Workers actifs ou bloqués.	Saturation ou deadlock.

```
# Inspecter workers
 celery -A project inspect active
 celery -A project inspect reserved
 celery -A project inspect scheduled
 celery -A project inspect stats

 # Redémarrage contrôlé via systemd
 systemctl status celery.service --no-pager
 journalctl -u celery.service -n 100 --no-pager
 sudo systemctl restart celery.service
```

7. Seuils d'alerte : warning, critique et tendance

Les seuils doivent être adaptés au service. Une alerte utile combine un seuil, une durée, une sévérité, un runbook et un owner. Les alertes instantanées trop sensibles créent du bruit.

Métrique	Warning	Critique
CPU	> 80% pendant 10 min	> 90% pendant 10 min + latence
RAM	< 15% disponible	Swap forte ou OOM
Disque	> 80%	> 90% ou croissance rapide
HTTP 5xx	> 1% pendant 5 min	> 2-5% pendant 5 min
Latence p95	> baseline + 50%	> seuil SLA/SLO
DB connections	> 75%	> 90%
Queue backlog	Backlog croissant	Âge message > SLA métier

```
Alert:
 - Name: API latency p95 high
 - Condition: p95 > 800ms for 10 minutes
 - Severity: warning / critical
 - Environment: production
 - Impact: users may experience slow responses
 - Dashboard: production-api-overview
 - Runbook: runbooks/api-latency.md
 - Owner: platform-team
 - First checks:
 - last deployment
 - 5xx rate
 - DB slow queries
 - CPU/RAM
 - queue backlog
```

8. Scaling : vertical, horizontal, autoscaling et limites

Scaler peut résoudre un problème de capacité, mais pas toujours un problème de performance. Si le goulot est une requête SQL lente ou un lock, ajouter des instances applicatives peut aggraver la base.

Type	Principe	Quand
Vertical	Augmenter taille machine.	DB, instance unique, besoin rapide.
Horizontal	Ajouter replicas/instances.	Application stateless, workers, API.
Autoscaling	Scaler selon métriques.	Charge variable et métriques fiables.
Partitionnement	Découper données/charge.	Très gros volumes ou limites DB.
Cache	Réduire charge DB/app.	Lectures répétitives, données cacheables.
Optimisation code/SQL	Réduire coût par requête.	Avant scaling coûteux.

```
Saturation détectée
 │
 ▼
 Quel goulot ?
 ├── CPU app → horizontal/vertical
 ├── RAM app → leak ou vertical
 ├── DB → query/index/pool/scale DB
 ├── queue → workers/concurrency
 ├── disque → cleanup/resize/I/O
 └── réseau → LB/cache/architecture
 │
 ▼
 Tester en staging si possible
 │
```

```
▼
Apply contrôlé
│
▼
...
```

3.7 - Incidents base de données

Connexions, locks, lenteurs, disque plein, backups et restauration.

Source modal: 3.7 Incidents base de données : connexions, locks, lenteurs, disque, backups et restauration

1. Incidents DB : le cœur de l'état métier

Une base de données n'est pas un service comme les autres : elle contient l'état métier, les commandes, comptes, paiements, sessions, événements, logs applicatifs ou données clients. Un incident DB peut ralentir tout le système, bloquer les écritures, provoquer une perte de données ou rendre un rollback applicatif beaucoup plus complexe.

La première règle pendant un incident DB est de commencer par des actions read-only : observer les connexions, les locks, les requêtes lentes, l'espace disque, les transactions, les erreurs et les derniers changements. On évite les actions destructives tant que le diagnostic n'est pas clair.

Famille	Symptômes	Risque
Connexions	Max connections atteint, pool saturé, timeouts.	Application incapable de parler à la DB.
Locks	Transactions bloquées, requêtes en attente.	Écritures ou lectures critiques figées.
Lenteurs SQL	Slow queries, CPU DB haut, latence applicative.	Dégradation progressive ou saturation.
Disque plein	Écritures impossibles, WAL/binlogs énormes, backups bloqués.	Arrêt brutal ou corruption si mal géré.
Backup / restore	Backup absent, restore non testé, RPO/RTO flous.	Perte de données ou indisponibilité prolongée.
Réplication	Replica lag, réplication cassée, failover risqué.	Données obsolètes, bascule dangereuse.

```
Alerte DB / application lente
 │
 ▼
Confirmer l'impact
 ├── utilisateurs touchés ?
 ├── lectures ou écritures ?
 ├── service complet ou partiel ?
 └── depuis quand ?
 │
 ▼
Observer read-only
 ├── connexions
 ├── locks
 ├── slow queries
 ├── disque
 ├── replication lag
 └── logs DB/app
 │
...
```

2. Triage incident DB : comprendre avant d'agir

Le triage consiste à répondre rapidement à trois questions : quel est l'impact utilisateur, quelle ressource DB est sous pression, et quelle mitigation réduit le risque sans détruire d'information.

Question	Pourquoi	Exemple de réponse
Lecture ou écriture touchée ?	Le risque diffère fortement.	Les inserts commandes échouent, lectures OK.
Erreur ou lenteur ?	Pas la même urgence.	Timeouts DB côté API après 10 s.
Depuis quand ?	Corrélation avec changement.	Depuis le déploiement 14:05.
Quel service consomme ?	Identifier source de charge.	Workers export CSV saturent DB.
Dernière migration ?	Possibles locks, index, table scan.	Ajout index non concurrent sur grande table.
Backup récent vérifié ?	Avant action destructive.	Snapshot disponible à 13:30.

```
# Côté Linux / service
 systemctl status postgresql --no-pager
 systemctl status mariadb --no-pager
 journalctl -u postgresql -n 100 --no-pager
 journalctl -u mariadb -n 100 --no-pager

 # Système
 uptime
 free -h
 df -h
 df -i
 top

 # Réseau local
 ss -tulpn
```

3. Connexions DB : saturation, pools et timeouts

Une saturation de connexions peut bloquer l'application même si la DB n'est pas très chargée. Trop de connexions côté application peut aussi empirer la DB : plus de context switching, plus de locks, plus de transactions concurrentes et plus de mémoire consommée.

Cause	Symptôme	Correction
Pool applicatif trop grand	Trop de connexions simultanées.	Réduire pool par instance.
Scaling horizontal non maîtrisé	Chaque instance ajoute son pool.	Calculer total max connexions.
Connexions idle non fermées	Beaucoup de sessions idle.	Timeouts, pooling, correction app.
Transactions longues	Connexions actives longtemps.	Réduire transaction, optimiser requête.
Workers batch trop nombreux	DB saturée lors jobs asynchrones.	Limiter concurrency workers.

```
-- Connexions par état
 select state, count(*)
 from pg_stat_activity
 group by state
 order by count(*) desc;

 -- Connexions par application / utilisateur
 select usename, application_name, client_addr, state, count(*)
 from pg_stat_activity
 group by usename, application_name, client_addr, state
 order by count(*) desc;

 -- Requêtes actives longues
 select pid, now() - query_start as duration, state, wait_event_type, wait_event, query
 from pg_stat_activity
 where state <> 'idle'
 order by duration desc
 limit 20;
...
```

4. Locks : transactions bloquées et effet domino

Les locks sont l'une des causes les plus dangereuses de lenteur DB : une transaction peut bloquer une table, une ligne, un index ou une migration, puis créer une file d'attente qui finit par saturer les connexions applicatives.

Situation	Action possible	Prudence
Requête SELECT longue	Cancel si impact élevé.	Vérifier origine et utilité.
Migration bloquante	Stopper migration, rollback si possible.	Vérifier état partiel.
Transaction métier longue	Contacter owner, terminer si nécessaire.	Risque rollback transaction.
Batch bloque prod	Stopper batch, réduire concurrence.	Documenter jobs interrompus.

```
-- Sessions bloquées et bloqueurs
 select
 blocked.pid as blocked_pid,
 blocked.query as blocked_query,
 blocking.pid as blocking_pid,
 blocking.query as blocking_query,
 now() - blocked.query_start as blocked_duration
 from pg_stat_activity blocked
 join pg_locks blocked_locks
 on blocked_locks.pid = blocked.pid
 join pg_locks blocking_locks
 on blocking_locks.locktype = blocked_locks.locktype
 and blocking_locks.database is not distinct from blocked_locks.database
 and blocking_locks.relation is not distinct from blocked_locks.relation
 and blocking_locks.page is not distinct from blocked_locks.page
 and blocking_locks.tuple is not distinct from blocked_locks.tuple
 and blocking_locks.virtualxid is not distinct from blocked_locks.virtualxid
 and blocking_locks.transactionid is not distinct from blocked_locks.transactionid
...
```

5. Lenteurs SQL : slow queries, index, plans et saturation

Une lenteur DB peut venir d'une requête sans index, d'un mauvais plan d'exécution, d'un volume qui a grandi, d'une statistique obsolète, d'un cache froid, d'un lock, d'un disque lent ou d'un déploiement qui multiplie les appels.

Cause	Signal	Correction
Index manquant	Table scan, CPU/I/O élevés.	Ajouter index adapté, tester coût écriture.
N+1 queries	Explosion nombre requêtes par page.	Optimiser ORM, prefetch/select related.
Requête trop large	Beaucoup de lignes lues/retournées.	Pagination, filtres, colonnes ciblées.
Stats obsolètes	Mauvais plan d'exécution.	Analyze/vacuum selon moteur.
Lock masqué	Requête "lente" mais en attente.	Identifier wait event / bloqueur.
Cache froid	Lenteur après restart/failover.	Surveiller, préchauffer si nécessaire.

```
-- Requêtes actives les plus longues
 select pid, now() - query_start as duration, state, wait_event_type, wait_event, query
 from pg_stat_activity
 where state <> 'idle'
 order by duration desc
```

```
limit 20;

-- Si pg_stat_statements est activé
select calls,
total_exec_time,
mean_exec_time,
rows,
query
from pg_stat_statements
order by total_exec_time desc
limit 20;

-- Plan d'exécution
...
```

6. Disque plein : urgence DB à traiter avec méthode

Un disque DB plein est critique : les écritures peuvent échouer, les transactions logs grossir, les backups ne plus se produire, la réplication se bloquer, et certaines bases peuvent refuser de redémarrer correctement.

Cause	Symptôme	Action prudente
Logs applicatifs	/var/log énorme.	Rotation, vacuum journald, corriger bruit.
Backups locaux	Répertoire backup plein.	Archiver/supprimer selon rétention validée.
WAL/binlogs	Journaux transactionnels énormes.	Vérifier réplication/backup avant purge.
Table énorme	Croissance data non prévue.	Archivage, partitionnement, purge applicative.
Index trop nombreux	DB grossit vite.	Audit index inutilisés, suppression contrôlée.
Temp files	Requêtes tri/hash massives.	Optimiser requêtes, mémoire, index.

```
# Espace et inodes
df -h
df -i

# Gros répertoires
du -sh /var/lib/* 2>/dev/null | sort -h
du -sh /var/log/* 2>/dev/null | sort -h
du -sh /backup/* 2>/dev/null | sort -h

# Fichiers supprimés encore ouverts
lsof +L1

# Journaux systemd
journalctl --disk-usage
sudo journalctl --vacuum-time=7d
```

7. Backups : RPO, RTO, vérification et stratégie

Un backup n'a de valeur que s'il est restaurable. L'existence d'un fichier ou d'un snapshot ne suffit pas : il faut connaître la fréquence, le périmètre, le chiffrement, la rétention, le temps de restauration et la dernière restauration testée.

Concept	Définition	Question opérationnelle
RPO	Perte de données maximale acceptable.	Peut-on perdre 5 min, 1 h, 24 h ?
RTO	Temps maximal pour restaurer le service.	Doit-on restaurer en 15 min ou 4 h ?
Full backup	Sauvegarde complète.	Combien de temps pour restaurer ?
Incremental / WAL / binlog	Permet point-in-time recovery.	Les journaux sont-ils complets ?
Retention	Durée de conservation.	Conforme métier/légal/coût ?
Restore test	Preuve de restaurabilité.	Dernier test réussi quand ?

```
# PostgreSQL dump logique
pg_dump -Fc -f backup.dump my_database

# PostgreSQL restore logique
pg_restore -d restored_database backup.dump

# MySQL / MariaDB dump logique
mysqldump --single-transaction --routines --triggers my_database > backup.sql

# Restore MySQL / MariaDB
mysql restored_database < backup.sql
```

8. Restauration : décision lourde, procédure contrôlée

Restaurer une base de données est une décision majeure. Cela peut résoudre une corruption ou une suppression massive, mais peut aussi perdre des données récentes. La restauration doit être décidée avec le métier, l'owner applicatif, DevOps/SRE et si possible un DBA.

Type	Usage	Risque
Restore complet	DB remplacée par backup.	Perte depuis backup.
Point-in-time recovery	Revenir juste avant erreur.	Complexité et validation.
Restore parallèle	Restaurer ailleurs pour extraire données.	Plus sûr, plus lent.

Type	Usage	Risque
Table-level restore	Récupérer une table spécifique.	Cohérence référentielle.
Failover replica	Basculer sur réplica.	Replica lag ou données manquantes.

```
Before restore:
 - Confirm incident type: corruption, deletion, bad migration, outage.
 - Freeze risky writes if needed.
 - Identify last known good time.
 - Confirm available backups / WAL / binlogs.
 - Estimate data loss window.
 - Validate with business owner.
 - Prefer restore to parallel environment first.
 - Document decision and approvers.

 During restore:
 - Follow tested procedure.
 - Preserve original data if possible.
 - Monitor logs, disk, replication.

 After restore:
 - Verify application health.
 - Verify data consistency.
...
```

3.8 - Incidents réseau

DNS, TLS, Nginx, firewall, load balancer, routes et ports.

Source modal: 3.8 Incidents réseau : DNS, TLS, Nginx, firewall, load balancer, routes et ports

1. Incidents réseau : quand l'application semble morte mais ne l'est pas toujours

Les incidents réseau donnent souvent l'impression que l'application est tombée, alors que le problème peut venir d'un DNS mal propagé, d'un certificat expiré, d'un firewall trop restrictif, d'une règle Nginx incorrecte, d'un load balancer sans target healthy, d'une route cloud absente ou d'un port fermé.

Le diagnostic doit suivre le chemin réel de la requête : client → DNS → TLS → edge/load balancer → firewall → reverse proxy → application → dépendances . On évite de redémarrer les services au hasard avant d'avoir identifié où le flux se casse.

Famille	Symptômes	Risque
DNS	Domaine ne résout pas, mauvaise IP, propagation incohérente.	Service inaccessible pour tout ou partie des utilisateurs.
TLS	Certificat expiré, chaîne invalide, SNI incorrect.	Erreur navigateur, API clients bloquées.
Nginx / proxy	502, 503, 504, mauvais upstream, config cassée.	Application saine mais inaccessible.
Firewall / ports	Connexion refusée, timeout, port fermé.	Flux bloqué entre client, proxy, app ou DB.
Load balancer	Targets unhealthy, health check KO, mauvais listener.	Trafic non routé ou routé vers mauvais backend.
Routes / cloud network	Subnet isolé, route table incomplète, NAT absent.	Communication impossible entre composants.

```
Utilisateur signale une panne
 │
 ▼
 DNS
 ├── domaine résout ?
 ├── bonne IP / CNAME ?
 └── propagation correcte ?
 │
 ▼
 TLS
 ├── certificat valide ?
 ├── chaîne complète ?
 └── SNI correct ?
 │
 ▼
 Edge / Load Balancer
 ├── listener actif ?
 ├── target healthy ?
...
```

2. Chemin de la requête : cartographier avant de corriger

Avant toute action, il faut comprendre où passe le trafic. Une panne peut être visible côté utilisateur mais située très loin de l'application : DNS, CDN, WAF, load balancer, security group, reverse proxy ou service interne.

Question	Pourquoi
Tout le monde est-il impacté ?	Différencier panne globale, DNS local, opérateur, région.
HTTP ou HTTPS seulement ?	Identifier TLS, redirection, listener ou firewall.
Erreur immédiate ou timeout ?	Refus de connexion vs paquet perdu ou service lent.
IP directe fonctionne-t-elle ?	Isoler DNS du reste.
Healthcheck interne fonctionne-t-il ?	Isoler proxy/LB de l'application.

```
Browser / Client API
 │
 ▼
 DNS resolver
 │
 ▼
 CDN / WAF éventuel
 │
 ▼
 Load Balancer
 │
 ▼
 Firewall / Security Group
 │
 ▼
 Nginx / Reverse Proxy
 │
 ▼
...
```

3. DNS : résolution, propagation, TTL et mauvais records

Les incidents DNS sont trompeurs : certains utilisateurs voient l'ancien record, d'autres le nouveau, certains resolvers gardent le cache, et une erreur de CNAME/A/AAAA peut rendre un service inaccessible sans toucher l'application.

Point	Contrôle
Record A / CNAME	Pointe vers la bonne IP ou le bon load balancer.
AAAA	IPv6 ne pointe pas vers une cible cassée.
TTL	Pas trop long avant changement sensible.
Zone autoritative	La bonne zone DNS est modifiée.
Propagation	Résultats cohérents entre resolvers.

```
# Résolution simple
 dig example.com
 dig www.example.com

 # Type de record
 dig A example.com
 dig AAAA example.com
 dig CNAME www.example.com
 dig MX example.com
 dig TXT example.com

 # Trace complète
 dig +trace example.com

 # Interroger un resolver précis
 dig @1.1.1.1 example.com
 dig @8.8.8.8 example.com

...
```

4. TLS / certificats : expiration, chaîne, SNI et redirections

Une panne TLS bloque souvent tout avant même que la requête atteigne l'application. Les causes courantes : certificat expiré, mauvais domaine, chaîne intermédiaire manquante, SNI incorrect, renouvellement Certbot échoué, redirection HTTP/HTTPS mal configurée.

Point	Contrôle
Expiration	Date notAfter dans le futur.
Subject / SAN	Le domaine appelé est inclus.
Chaîne	Certificat intermédiaire fourni.
SNI	Bon certificat selon hostname.
Redirection	HTTP vers HTTPS sans boucle.

```
# Tester HTTPS avec détails
 curl -vk https://example.com/

 # Voir headers uniquement
 curl -I https://example.com/

 # Inspecter certificat
 openssl s_client -connect example.com:443 -servername example.com

 # Extraire dates du certificat
 echo | openssl s_client -connect example.com:443 -servername example.com 2>/dev/null \
 | openssl x509 -noout -dates -subject -issuer

 # Certbot
 certbot certificates
 systemctl status certbot.timer --no-pager
 journalctl -u certbot -n 100 --no-pager
```

5. Nginx / reverse proxy : 502, 503, 504 et upstreams

Nginx peut être la couche qui signale la panne sans être la cause racine. Un 502 indique souvent un upstream inaccessible ou cassé. Un 504 indique souvent un upstream trop lent ou un timeout trop court.

Code	Signification probable	Diagnostic
400	Host/header/proxy protocol incorrect.	Vérifier headers et vhost.
404	Mauvais vhost ou route app absente.	Vérifier server_name et location.
502	Upstream down/refuse/mauvais socket.	Tester app backend et logs.
503	Maintenance, overload ou upstream unavailable.	Vérifier mode maintenance / LB.
504	Timeout upstream.	Analyser lenteur app/DB/API externe.

```
# Tester configuration
 nginx -t

 # Statut service
 systemctl status nginx --no-pager

 # Recharger sans couper brutalement
```

```
sudo systemctl reload nginx

# Logs
tail -n 100 /var/log/nginx/error.log
tail -n 100 /var/log/nginx/access.log
journalctl -u nginx -n 100 --no-pager

# Voir config chargée
nginx -T | less
```

6. Firewall, security groups, ports et écoute réseau

Un service peut être en bonne santé localement mais inaccessible à cause d'un port fermé, d'une règle firewall, d'un security group, d'une ACL réseau ou d'un service qui écoute seulement sur 127.0.0.1 au lieu de l'interface attendue.

Résultat	Signification probable
Connection refused	Host joignable, port fermé ou service down.
Timeout	Firewall, route, security group ou paquet perdu.
Port écoute sur 127.0.0.1	Accessible localement seulement.
Port écoute sur 0.0.0.0	Accessible sur toutes interfaces si firewall OK.

```
# Ports ouverts localement
 ss -tulpn
 ss -ltnp

 # Tester un port depuis une autre machine
 nc -vz example.com 443
 nc -vz 10.0.1.20 8000

 # UFW
 ufw status verbose

 # iptables
 iptables -L -n -v
 iptables -S

 # nftables
 nft list ruleset

...
```

7. Load balancer : listeners, target groups et health checks

Le load balancer peut couper le trafic si ses targets sont unhealthy, si le healthcheck pointe vers le mauvais chemin, si le listener est mal configuré, si le certificat n'est pas le bon ou si les règles de routage envoient vers le mauvais backend.

Zone	Contrôle
Listener	Port 80/443 actif, certificat correct.
Rules	Host/path routing vers bon target group.
Target health	Instances/pods healthy.
Healthcheck path	Chemin répond vite et sans dépendance fragile.
Security groups	LB peut joindre les backends.
Timeouts	Adaptés au comportement app.

```
# Exemple endpoint recommandé
 GET /health/

 Expected:
 - HTTP 200
 - réponse rapide
 - pas de dépendance lourde
 - pas d'appel externe lent
 - vérification minimale app ready
```

8. Routes, subnets, NAT, gateways et réseau cloud

Les incidents réseau cloud viennent souvent d'une route manquante, d'une NAT gateway absente, d'un subnet mal associé, d'une ACL trop restrictive ou d'un peering/VPN cassé. Le symptôme est souvent un timeout plutôt qu'un refus de connexion.

Composant	Ce qu'il faut vérifier
Route table	Route vers internet, NAT, peering ou VPN.
Subnet public	Route vers internet gateway.
Subnet privé	Route vers NAT pour sorties internet si besoin.
NACL	Inbound/outbound stateless, ports éphémères.
Peering/VPN	Routes des deux côtés, CIDR non conflictuel.
DNS interne	Résolution privée, split-horizon, zones privées.

```
# Route locale
 ip route
 ip addr

 # Résolution interne
 dig internal.service.local

 # Test port interne
 nc -vz 10.0.2.15 5432
 nc -vz redis.internal 6379

 # Traceroute si disponible
 traceroute example.com
 tracepath example.com

 # Connexion HTTP interne
 curl -v http://internal-service:8080/health/
```

4.4 - IAM approfondi

Roles, policies, assume-role, separation des privileges et break-glass.

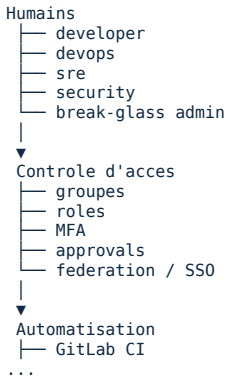
Source modal: 4.4 IAM approfondi : roles, policies, assume-role, least privilege et break-glass

1. IAM : la frontiere entre automatisation utile et risque majeur

IAM controle qui peut faire quoi, sur quelle ressource, dans quel environnement, depuis quel contexte et sous quelles conditions. Dans une equipe DevOps, IAM touche les humains, les pipelines CI/CD, Terraform, les workloads applicatifs, les comptes cloud, les secrets, les roles d'urgence et les audits de securite.

Trop peu de droits casse les pipelines et ralentit les operations. Trop de droits expose la production, les donnees et les comptes cloud. Le bon niveau consiste a donner le droit minimal necessaire , pendant le temps necessaire , avec une trace d'audit exploitable .

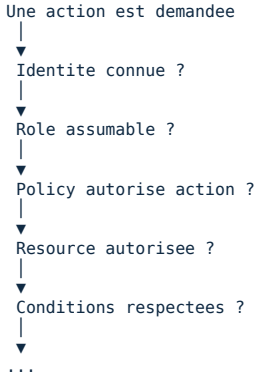
Dimension	Question	Risque si mal gere
Identite	Qui ou quoi agit ?	Utilisateur ou service non identifie.
Permission	Quelles actions sont autorisees ?	Droits trop larges ou pipeline bloque.
Ressource	Sur quoi l'action s'applique ?	Impact hors perimetre attendu.
Contexte	Depuis quelle branche, compte, IP, OIDC, MFA ?	Utilisation depuis un contexte non fiable.
Duree	Acces permanent ou temporaire ?	Credentials oublies ou compromis.
Audit	Peut-on prouver qui a fait quoi ?	Incident impossible a reconstruire.



2. Concepts IAM essentiels

Pour debugger ou concevoir IAM proprement, il faut distinguer identite, role, policy, trust policy, permission boundary, session temporaire, condition et audit trail.

Concept	Definition	Exemple
User	Identite humaine ou technique permanente.	Compte admin, service account historique.
Role	Identite assumable avec permissions.	prod-terraform-apply-role .
Policy	Document qui autorise ou refuse des actions.	s3:GetObject sur un bucket.
Trust policy	Definit qui peut assumer un role.	GitLab OIDC peut assumer role CI.
Assume-role	Obtention de credentials temporaires.	Pipeline prend role Terraform staging.
Condition	Restriction contextuelle.	Branche main, MFA, IP, tag, audience OIDC.
Permission boundary	Limite maximale meme si policy plus large.	Role projet ne peut pas toucher IAM global.



3. Roles IAM : par environnement, par usage et par niveau de risque

Un bon modele IAM evite les roles fourre-tout. On separe les roles par environnement et par usage : lecture, plan Terraform, apply Terraform, deploiement applicatif, operations DB, audit, break-glass.

Role	Usage	Risque
dev-readonly-role	Lire ressources dev, debug simple.	Faible.
staging-terraform-plan-role	Lire state et ressources pour plan.	Moyen.
staging-terraform-apply-role	Modifier infra staging.	Moyen a eleve.
prod-terraform-plan-role	Lire production et produire un plan.	Eleve en lecture sensible.
prod-terraform-apply-role	Modifier infrastructure production.	Critique.
prod-breakglass-admin-role	Urgence exceptionnelle.	Critique maximal.

```
Pattern:
 <environment>-<system>-<purpose>-role

 Examples:
 dev-api-readonly-role
 staging-terraform-plan-role
 prod-terraform-apply-role
 prod-security-audit-role
 prod-breakglass-admin-role
```

4. Policies IAM : least privilege et blast radius

Une policy professionnelle limite les actions, les ressources et les conditions. Elle evite les permissions globales, les wildcards non justifiees et les droits qui permettent une escalation indirecte.

Mauvais	Meilleur
Resource: "*"	ARN exact ou prefixe limite.
Action: "s3:*"	Actions necessaires seulement.
Policy unique massive	Policies par usage.
Pas de condition	Conditions sur env, tags, OIDC, MFA.

```
{
 "Version": "2012-10-17",
 "Statement": [
 {
 "Sid": "ReadSpecificBucket",
 "Effect": "Allow",
 "Action": [
 "s3:GetObject",
 "s3:ListBucket"
 ],
 "Resource": [
 "arn:aws:s3:::my-app-prod-artifacts",
 "arn:aws:s3:::my-app-prod-artifacts/*"
 ]
 }
 ]
 }
```

5. Assume-role : credentials temporaires et trust policy

Assume-role permet d'obtenir des credentials temporaires pour agir avec un role donne. C'est preferable aux access keys permanentes, a condition que la trust policy soit stricte et que les sessions soient auditees.

Risque	Correction
Trust trop large	Limiter principal et conditions.
Session trop longue	Reduire duree selon usage.
Role chain complexe	Documenter chemins d'assumption.
Pas de session name explicite	Forcer noms utiles pour audit.
Assume-role cross-account non controle	Limiter comptes et roles cibles.

```
Identite source
 ├── humain via SSO
 ├── GitLab OIDC
 ├── service account
 └── role intermediaire
 │
 ▼
 sts:AssumeRole
 │
 ▼
 Trust policy du role cible
 ├── principal autorise ?
 ├── conditions respectees ?
 ├── MFA / OIDC / branch ?
 └── session duration OK ?
 │
 ▼
 Credentials temporaires
...
```

6. IAM pour CI/CD : GitLab, Terraform, OIDC et roles separes

Les pipelines CI/CD sont des acteurs IAM critiques. Ils doivent pouvoir lire, planifier, deployer ou appliquer Terraform selon le contexte, mais ne doivent pas disposer d'un acces production permanent sur toutes les branches.

Job	Role IAM	Contexte
fmt / validate	Aucun role cloud.	Toutes branches.
plan dev	dev-terraform-plan-role	Branches dev.
plan prod	prod-terraform-plan-role	MR/main protegee.
apply staging	staging-terraform-apply-role	Environment staging protege.
apply prod	prod-terraform-apply-role	Manual job + protected environment.

```
terraform_plan_prod:
 stage: plan
 environment:
 name: production
 script:
 - aws sts get-caller-identity
 - terraform init
 - terraform plan -out=tfplan
 rules:
 - if: '$CI_COMMIT_BRANCH == "main"'

 terraform_apply_prod:
 stage: apply
 environment:
 name: production
 script:
 - aws sts get-caller-identity
 - terraform apply tfplan
...
```

7. Separation des privileges : humains, pipelines et production

La separation des privileges empeche qu'une seule personne ou un seul pipeline puisse tout faire : ecrire le code, approuver, merger, obtenir les secrets, appliquer en production et effacer les traces.

Acteur	Droits normaux	Droits interdits
Developer	MR, lecture dev, logs non sensibles.	Apply prod, secrets prod.
DevOps	Plan, staging apply, review infra.	Admin global permanent.
SRE	Production operations, incident response.	Contourner audit ou approvals.
Security	Review IAM, audit, policies.	Deploy applicatif sans procedure.
GitLab CI	Roles par job et environnement.	Credential permanent multi-env.

```
Terraform plan
  ├── read resources
  ├── read state
  ├── generate plan
  └── no modification

 Terraform apply
  ├── modify resources
  ├── requires protected env
  ├── manual trigger
  ├── approval
  └── audit trail
```

8. Break-glass : acces d'urgence rare, fort, temporaire et audite

Le break-glass est un acces exceptionnel permettant d'agir quand les chemins normaux sont indisponibles : incident critique, pipeline casse, IAM bloque, production inaccessible. Il ne doit jamais devenir un raccourci quotidien.

Caracteristique	Exigence
Rare	Utilise seulement en incident ou urgence validee.
Fort	Peut corriger une situation bloquante.
Temporaire	Session courte, expiration, retrait apres usage.
MFA obligatoire	Protection contre compromission simple.
Journalise	Chaque action doit etre reconstruisible.
Revu apres usage	Post-incident et verification des actions.

```
Before:
 - Confirm incident severity.
 - Open incident ticket.
 - Identify why normal path is unavailable.
 - Get approval from incident commander or senior owner.
 - Use MFA.
 - Start audit note.
```

```
During:
- Assume break-glass role.
- Execute minimal required actions.
- Avoid unrelated changes.
- Capture commands and results.

After:
- Exit session.
- Revoke temporary access if needed.
- Review audit logs.
...
```

4.5 - Supply chain CI/CD

Images Docker, dépendances, runners, artifacts et provenance.

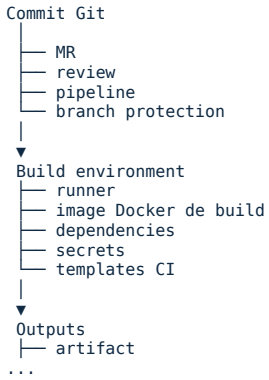
Source modal: 4.5 Supply chain CI/CD : images Docker, dépendances, runners, artifacts et provenance

1. Supply chain CI/CD : sécuriser ce qui construit et déploie

La supply chain CI/CD désigne toute la chaîne qui transforme un commit en artifact, image Docker, package, infrastructure ou déploiement production. Elle inclut le code source, les dépendances, les images de build, les templates CI, les runners, les secrets, les artifacts, les registres, les signatures et les droits cloud utilisés par les pipelines.

C'est une zone critique : si un attaquant modifie une dépendance, une image Docker, un template GitLab CI, un runner ou un artifact, il peut injecter du code, voler des secrets, modifier une image de production ou obtenir un accès indirect à l'infrastructure.

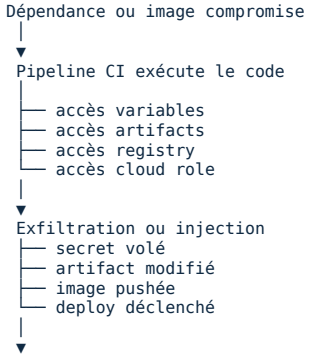
Maillon	Risque principal	Contrôle attendu
Source Git	Commit malveillant, branche non protégée.	MR, approvals, branches protégées.
Templates CI	Job modifié pour exfiltrer secrets.	Templates versionnés, review sécurité.
Images Docker	Base image compromise ou vulnérable.	Pin par digest, scan, registry contrôlé.
Dépendances	Package compromis, typosquatting, CVE.	Lockfiles, SCA, registry interne si besoin.
Runner	Runner partagé, non isolé, persistance de secrets.	Runner protégé, éphémère, tags stricts.
Artifacts	Artifact manipulé ou contenant secrets.	Expiration, contrôle contenu, accès limité.
Registry	Image remplacée ou tag mutable.	Immutabilité, signature, droits push limités.



2. Menaces supply chain : comprendre les scénarios réels

Les attaques supply chain visent rarement uniquement le code applicatif. Elles ciblent souvent les dépendances, les scripts de build, les images de base, les runners, les tokens CI, les artifacts ou les templates partagés.

Scénario	Exemple	Impact
Package compromis	Dépendance npm/pip modifiée.	Code malveillant dans build ou runtime.
Typosquatting	Package au nom proche d'un package connu.	Installation involontaire d'un malware.
Image Docker non maîtrisée	latest change sans contrôle.	Build non reproductible, vulnérabilité.
Template CI modifié	Ajout d'un curl exfiltrant secrets.	Compromission multi-projets.
Runner compromis	Secrets persistants sur machine de build.	Vol de credentials, injection artifact.
Artifact remplacé	Binary ou image modifié après build.	Déploiement d'un contenu non revu.
Registry compromise	Tag production repointé vers autre image.	Déploiement d'une image non autorisée.



...

3. Images Docker : pinning, scan, digest et registry

Les images Docker sont un maillon central. Une image de base non contrôlée peut contenir des vulnérabilités, outils inattendus, backdoors, versions obsolètes ou changements imprévus.

Pratique	Pourquoi	Exemple
Éviter latest	Tag mutable, build non reproductible.	python:3.12.3-slim
Pinner par digest	Garantit le contenu exact.	image@sha256:...
Scanner les images	Détecter CVE critiques.	Trivy, Grype, GitLab scanning.
Réduire l'image	Moins de surface d'attaque.	slim, distroless, multi-stage.
Utilisateur non-root	Limiter impact runtime.	USER appuser
Registry privé	Contrôle des images autorisées.	GitLab Registry, ECR, GCR, ACR.

```
FROM python:3.12.3-slim

 ENV PYTHONDONTWRITEBYTECODE=1
 ENV PYTHONUNBUFFERED=1

 RUN addgroup --system app && adduser --system --group app

 WORKDIR /app

 COPY requirements.txt .
 RUN pip install --no-cache-dir -r requirements.txt

 COPY . .

 USER app

 CMD ["gunicorn", "config.wsgi:application", "--bind", "0.0.0.0:8000"]
```

4. Dépendances : lockfiles, SCA, versions et registres

Les dépendances applicatives sont une porte d'entrée classique : package compromis, maintainer account piraté, dépendance abandonnée, CVE critique, typosquatting ou confusion entre registry public et privé.

Contrôle	But	Exemples
Lockfile	Reproductibilité des versions.	package-lock.json , poetry.lock .
SCA scan	Détection vulnérabilités connues.	Dependabot, GitLab, Snyk, osv-scanner.
Pin version	Éviter upgrades invisibles.	Django==4.2.11 .
Registry contrôlé	Limiter confusion dépendances.	Proxy interne, allowlist.
Review upgrades	Comprendre impact sécurité/runtime.	MR dédiée dépendances.
Secret scan	Éviter tokens dans repo ou packages.	Gitleaks, GitLab secret detection.

```
# requirements.txt
 Django==4.2.11
 gunicorn==21.2.0
 psycopg[binary]==3.1.18

 # Installation reproductible
 pip install --require-hashes -r requirements.txt
```

5. Runners GitLab : isolation, tags, privilèges et nettoyage

Le runner exécute le code du pipeline. Si le runner est partagé, persistant ou trop privilégié, un job peut lire des fichiers résiduels, exploiter Docker socket, voler des secrets ou modifier l'environnement d'autres pipelines.

Runner	Usage	Protection
shared-low-risk	Lint, tests sans secrets.	Pas de variables sensibles.
build-docker	Build images.	Registry push limité.
terraform-dev	Plan/apply dev.	Secrets dev uniquement.
terraform-prod	Plan/apply production.	Protected runner, protected env.
security-scan	Scans SAST/SCA/images.	Accès lecture, pas apply.

```
# Vérifier runners
 gitlab-runner verify
 gitlab-runner list

 # Service
 systemctl status gitlab-runner --no-pager
 journalctl -u gitlab-runner -n 100 --no-pager

 # Machine
 df -h
 free -h
 docker ps
 docker images
 docker system df
```

6. Artifacts, cache et logs : outputs utiles mais sensibles

Les artifacts et caches accélèrent les pipelines et facilitent les reviews, mais ils peuvent contenir des secrets, plans sensibles, binaires, rapports de scan, outputs Terraform, logs détaillés ou fichiers temporaires.

Output	Usage	Risque
plan.txt	Review humaine Terraform.	Noms de ressources sensibles.
plan.json	Policy-as-code.	Contenu infrastructure détaillé.
Binary/package	Release applicative.	Remplacement ou corruption.
Cache dependencies	Accélération build.	Cache poisoning.
Logs CI	Debug pipeline.	Secrets affichés par erreur.
Rapports sécurité	Review vulnérabilités.	Informations exploitables.

```
terraform_plan:
 stage: plan
 script:
 - cd "$TF_ROOT"
 - terraform plan -out=tfplan
 - terraform show -no-color tfplan > plan.txt
 - terraform show -json tfplan > plan.json
 artifacts:
 paths:
 - "$TF_ROOT/tfplan"
 - "$TF_ROOT/plan.txt"
 - "$TF_ROOT/plan.json"
 expire_in: 3 days
 when: always
```

7. Provenance : savoir exactement ce qui a été construit et déployé

La provenance permet de répondre à une question simple mais critique : "Cette image ou cet artifact vient-il bien de ce commit, construit par ce pipeline, avec ces dépendances, puis déployé dans cet environnement ?"

Élément	Exemple	Utilité
Commit SHA	CI_COMMIT_SHA	Relier image au code source.
Pipeline ID	CI_PIPELINE_ID	Retrouver logs et jobs.
Image digest	sha256:...	Identifier contenu immuable.
SBOM	Liste dépendances.	Impact CVE et audit.
Signature	Image signée.	Vérifier origine.
Release notes	Changelog, MR, ticket.	Comprendre contexte.

```
docker build \
 --label org.opencontainers.image.revision="$CI_COMMIT_SHA" \
 --label org.opencontainers.image.source="$CI_PROJECT_URL" \
 --label org.opencontainers.image.created="$CI_JOB_STARTED_AT" \
 -t "$CI_REGISTRY_IMAGE:$CI_COMMIT_SHA" .
```

8. GitLab CI : intégrer les contrôles supply chain

La protection supply chain doit être intégrée au pipeline : lint, tests, SCA, scan image, secret detection, SBOM, signature, publication registry, déploiement contrôlé et rollback.

Gate	Bloquant ?	Pourquoi
Tests unitaires	Oui	Qualité minimale.
Secret scan	Oui	Éviter fuite directe.
Dependency critical CVE	Oui ou exception validée	Risque runtime.
Image critical CVE	Oui ou exception validée	Risque container.
SBOM	Recommandé	Audit et impact CVE.
Signature	Production critique	Provenance vérifiable.

```
MR pipeline
  │
  ├── lint
  ├── unit tests
  ├── dependency scan
  ├── secret scan
  ├── build image
  ├── image scan
  └── SBOM
  │
  ▼
 Merge main
  │
  ▼
 Build release image
  ├── tag commit SHA
  ├── digest
  ├── scan
...
```

4.6 - Kubernetes pour DevOps

Concepts utiles même si le poste est surtout Terraform/GitLab.

Source modal: 4.6 Kubernetes pour DevOps : pods, deployments, services, ingress, Helm et production

1. Kubernetes : ce qu'un DevOps doit comprendre même sans être expert K8s

Kubernetes orchestre des conteneurs : il lance des applications, les redémarre si elles tombent, expose des services, gère des configurations, distribue le trafic et permet des déploiements progressifs. Même si le poste est orienté Terraform/GitLab, un DevOps doit comprendre les bases Kubernetes pour lire un pipeline, diagnostiquer un incident et dialoguer avec les équipes SRE.

Terraform sert souvent à créer l'infrastructure autour de Kubernetes : cluster managé, réseau, subnets, node pools, IAM, load balancer, registry, DNS et stockage. Kubernetes, Helm ou Kustomize servent ensuite à déployer les workloads applicatifs dans le cluster.

Sujet	Ce que le DevOps doit savoir	Exemple production
Déploiement	Comprendre rollout, replicas, images et rollback.	Version API déployée progressivement.
Réseau	Comprendre Service, Ingress, DNS interne et ports.	502 sur Ingress vers mauvais service.
Configuration	Différencier ConfigMap, Secret, variables et volumes.	Erreur prod due à une config staging.
Sécurité	RBAC, namespaces, service accounts, image policies.	Pipeline trop privilégié dans le cluster.
Observabilité	Lire logs pods, events, probes et métriques.	CrashLoopBackOff après release.
CI/CD	Déployer via Helm/Kustomize depuis GitLab.	Manual deploy production avec approval.

```
User / Browser / API client
 │
 ▼
 DNS
 │
 ▼
 Load Balancer / Ingress Controller
 │
 ▼
 Ingress
 │
 ▼
 Service
 │
 ▼
 Pods
 ├── container app
 ├── env vars
...
```

2. Concepts Kubernetes essentiels

Pour un nouveau DevOps, le plus important est de comprendre le vocabulaire opérationnel : namespace, pod, deployment, service, ingress, probes, resources, RBAC et events.

Concept	Définition simple	Commande utile
Cluster	Ensemble de nœuds qui exécutent les workloads.	kubectl cluster-info
Node	Machine qui héberge des pods.	kubectl get nodes
Namespace	Séparation logique dans le cluster.	kubectl get ns
Pod	Plus petite unité déployée.	kubectl get pods -n app
Deployment	Contrôle les replicas et les rollouts.	kubectl get deploy -n app
Service	Adresse stable devant des pods.	kubectl get svc -n app
Ingress	Routage HTTP/HTTPS externe.	kubectl get ingress -n app

```
kubectl config current-context
 kubectl get namespaces
 kubectl get all -n my-namespace
 kubectl get pods -n my-namespace -o wide
 kubectl describe pod POD_NAME -n my-namespace
 kubectl logs POD_NAME -n my-namespace
 kubectl get events -n my-namespace --sort-by=.lastTimestamp
```

3. Pods, Deployments, ReplicaSets et rollout

Un pod exécute un container. Un deployment contrôle le nombre de pods, les mises à jour, la stratégie rolling update et le rollback. En production, on ne manipule presque jamais les pods directement : on agit plutôt sur le deployment ou via Helm.

État	Signification probable	Action
Pending	Pas assez de ressources ou scheduling impossible.	describe pod , vérifier nodes/resources.
CrashLoopBackOff	Container démarre puis crash.	Lire logs et previous logs.
ImagePullBackOff	Image introuvable ou auth registry KO.	Vérifier image, tag, pull secret.
Running mais not ready	Readiness probe échoue.	Vérifier endpoint health et dépendances.
OOMKilled	Container a dépassé la limite mémoire.	Analyser mémoire, limits, fuite possible.

```
apiVersion: apps/v1
 kind: Deployment
 metadata:
 name: api
 namespace: production
 spec:
 replicas: 3
 selector:
 matchLabels:
 app: api
 strategy:
 type: RollingUpdate
 rollingUpdate:
 maxUnavailable: 1
 maxSurge: 1
 template:
 metadata:
 labels:
...
```

4. Services, Ingress, DNS interne et chemin réseau

Kubernetes sépare l'exécution des pods et leur exposition réseau. Les pods sont éphémères, les services donnent une adresse stable, et l'Ingress expose les routes HTTP/HTTPS.

Symptôme	Cause probable	Diagnostic
Ingress 404	Host/path mal configuré.	Décrire Ingress et controller logs.
502 / 503	Service sans endpoints ou pods not ready.	kubectl get endpoints
Service ne route pas	Selector ne matche aucun pod.	Comparer labels pods et selector.
TLS KO	Secret TLS absent ou mauvais.	Vérifier secret et Ingress.
DNS interne KO	Service name ou namespace incorrect.	Tester depuis un pod debug.

```
apiVersion: v1
 kind: Service
 metadata:
 name: api
 namespace: production
 spec:
 type: ClusterIP
 selector:
 app: api
 ports:
 - name: http
 port: 80
 targetPort: 8000
```

5. ConfigMaps, Secrets, variables et volumes

Kubernetes sépare la configuration du code. Les ConfigMaps stockent la configuration non sensible. Les Secrets stockent les données sensibles, mais ils doivent être protégés par RBAC, chiffrement, rotation et éventuellement un gestionnaire externe de secrets.

Objet	Usage	Exemple
ConfigMap	Configuration non sensible.	Log level, feature flag, URL publique.
Secret	Données sensibles.	Password, token, private key.
External Secret	Synchroniser depuis secret manager.	AWS Secrets Manager, Vault.

```
apiVersion: v1
 kind: ConfigMap
 metadata:
 name: api-config
 namespace: production
 data:
 DJANGO_SETTINGS_MODULE: "config.settings.production"
 LOG_LEVEL: "INFO"
 FEATURE_X_ENABLED: "false"
```

6. Helm et Kustomize : gérer les manifests applicatifs

Helm et Kustomize permettent de gérer des manifests Kubernetes sans tout dupliquer. Helm fonctionne avec des charts et des values. Kustomize applique des overlays par environnement.

Outil	Avantage	Attention
Helm	Package, release, rollback, values.	Templates complexes si mal conçus.

Outil	Avantage	Attention
Kustomize	Simple, natif kubectl, overlays.	Moins orienté package applicatif.
Raw YAML	Très lisible au début.	Duplication rapide entre environnements.

```
chart-api/
  ├── Chart.yaml
  ├── values.yaml
  ├── values-staging.yaml
  ├── values-production.yaml
  └── templates/
  ├── deployment.yaml
  ├── service.yaml
  ├── ingress.yaml
  ├── configmap.yaml
  └── hpa.yaml
```

7. GitLab CI/CD avec Kubernetes

Un pipeline Kubernetes propre construit une image, la scanne, la pousse dans un registry, déploie en staging, lance des smoke tests, puis déploie en production via job manuel protégé.

Garde-fou	But
Image taggée au commit SHA	Traçabilité.
Scan image	Bloquer CVE critiques.
Helm lint/template	Détecter erreurs YAML/chart.
Namespace explicite	Éviter déploiement mauvais environnement.
Protected environment	Production déployable uniquement par autorisés.
Rollout status	Vérifier que la release est vraiment active.

```
Commit / MR
  │
  ├── tests
  ├── build image
  ├── image scan
  └── helm lint/template
  │
  ▼
 Merge main
  │
  ├── build release image
  ├── push registry
  └── deploy staging
  │
  ▼
 Smoke tests
  │
  ▼
...
```

8. Debug production Kubernetes : méthode terrain

Le debug Kubernetes commence par le contexte, le namespace et l'objet concerné. On lit les events, les logs, les probes, le rollout et les endpoints avant d'agir.

Symptôme	Cause probable	Commande
CrashLoopBackOff	Erreur app au démarrage.	kubectl logs --previous
ImagePullBackOff	Image/tag/registry secret KO.	kubectl describe pod
Pending	Ressources insuffisantes ou node selector.	kubectl describe pod
Service 503	Pods not ready, endpoints absents.	kubectl get endpoints
Rollout bloqué	New pods non ready.	kubectl rollout status
OOMKilled	Limite mémoire trop basse ou fuite.	kubectl describe pod

```
# 1. Vérifier contexte
 kubectl config current-context

 # 2. Voir les objets
 kubectl get all -n production

 # 3. Voir les pods
 kubectl get pods -n production -o wide

 # 4. Décrire le pod
 kubectl describe pod POD_NAME -n production

 # 5. Lire les logs
 kubectl logs POD_NAME -n production
 kubectl logs POD_NAME -n production --previous

 # 6. Lire les events
 kubectl get events -n production --sort-by=.lastTimestamp
...
```

4.7 - Linux hardening

Systemd, utilisateurs, SSH, firewall, logs, permissions et mises à jour.

Source modal: 4.7 Linux hardening : systemd, utilisateurs, SSH, firewall, logs, permissions et mises a jour

1. Linux hardening : securiser la base de production

Les serveurs Linux restent au coeur de nombreuses architectures DevOps : reverse proxy Nginx, runners GitLab, workers, bases de donnees, serveurs applicatifs, bastions SSH, monitoring, backups, scripts cron et services systemd.

Le hardening Linux ne consiste pas a empiler des outils complexes. Il commence par une discipline simple : utilisateurs separes, pas de root direct, SSH durci, firewall minimal, systemd propre, logs conserves, permissions strictes, mises a jour suivies et procedures de rollback documentees.

Axe	Objectif	Exemple concret
Utilisateurs	Eviter root direct et comptes partages.	Compte deploy, sudo limite, audit login.
SSH	Reduire risque brute force et acces non maitrise.	Keys only, no root login, port controle.
Firewall	Ouvrir uniquement les ports necessaires.	80/443 publics, SSH limite, DB privee.
Systemd	Superviser proprement les services.	Restart policy, EnvironmentFile, logs journald.
Logs	Diagnostiquer et reconstruire les incidents.	journalctl, logrotate, audit auth.
Permissions	Limiter l'impact d'une compromission.	Owner correct, chmod strict, secrets non lisibles.
Updates	Corriger CVE sans casser production.	Patch window, reboot plan, rollback.

```
Internet
 │
 ▼
 Firewall / Security Group
 ├── 80 / 443 publics
 ├── SSH limite
 └── DB non exposee
 │
 ▼
 SSH hardening
 ├── no root login
 ├── keys only
 ├── users nominatifs
 └── logs auth
 │
 ▼
 Systemd services
 ├── app.service
...
```

2. Utilisateurs, groupes, sudo et separation des droits

La premiere defense d'un serveur Linux est la separation des comptes : pas de compte partage, pas de root direct, pas de sudo global sans raison, et des utilisateurs techniques dedies aux services.

Compte	Usage	Droits
root	Administration systeme exceptionnelle.	Login SSH direct interdit.
guillaume / user nominatif	Acces humain auditable.	Sudo selon besoin.
deploy	Deploiement applicatif automatise.	Acces limite aux dossiers applicatifs.
app	Execution de l'application.	Pas de shell ou shell limite.
backup	Jobs de sauvegarde.	Lecture ciblee, ecriture backup.
gitlab-runner	Execution CI/CD.	Droits strictement necessaires.

```
# Creer un utilisateur nominatif
 sudo adduser alice
 sudo usermod -aG sudo alice

 # Voir groupes
 id alice
 groups alice

 # Voir sudo autorise
 sudo -l

 # Verifier comptes avec shell
 awk -F: '$7 !~ /(nologin|false)/ {print $1, $7}' /etc/passwd

 # Verifier derniers logins
 last
 lastlog
```

3. SSH hardening : acces distant sans exposition inutile

SSH est souvent la porte d'entree principale du serveur. Il doit etre protege par des cles, une configuration stricte, une limitation des utilisateurs, des logs surveilles et idealement une restriction reseau.

Fichier / dossier	Permission	Pourquoi
~/.ssh	700	Repertoire prive.
authorized_keys	600	Cles non modifiables par autres users.
Private key locale	600	SSH refuse les cles trop ouvertes.

```
# /etc/ssh/sshd_config.d/hardening.conf

 PermitRootLogin no
 PasswordAuthentication no
 PubkeyAuthentication yes
 PermitEmptyPasswords no
 ChallengeResponseAuthentication no
 X11Forwarding no
 AllowTcpForwarding no
 ClientAliveInterval 300
 ClientAliveCountMax 2
 MaxAuthTries 3

 # Exemple : limiter aux users autorises
 AllowUsers guillaume deploy
```

4. Firewall, ports ouverts et exposition reseau

Le firewall doit suivre une logique simple : tout fermer par defaut, ouvrir uniquement ce qui est utile, limiter les sources sensibles et verifier regulierement les ports vraiment ecoutes.

Port	Exposition recommandee	Commentaire
22 SSH	IP admin uniquement.	Eviter public global.
80 HTTP	Public si redirection HTTPS.	Peut etre ferme si HTTPS only strict.
443 HTTPS	Public.	Port principal web.
3306 MySQL	Prive uniquement.	Jamais internet direct.
5432 PostgreSQL	Prive uniquement.	Security group interne.
6379 Redis	Local ou prive uniquement.	Jamais expose public.
8000 app	Local derriere Nginx.	Pas public si reverse proxy.

```
# Ports en ecoute
 sudo ss -tulpn
 sudo ss -ltnp

 # Process associe a un port
 sudo lsof -iTCP -sTCP:LISTEN -P -n

 # Test externe depuis une autre machine
 nc -vz example.com 443
 nc -vz example.com 22
```

5. Systemd : services propres, restart policy et logs

Systemd est le superviseur standard de nombreux serveurs Linux. Un service systemd bien ecrit rend l'application redemarrable, observable, parametrable et plus stable en production.

Option	But
User / Group	Eviter execution en root.
WorkingDirectory	Contexte clair du service.
EnvironmentFile	Separer config du service.
Restart=on-failure	Redemarrage automatique en cas de crash.
PrivateTmp=true	Isolation du repertoire temporaire.
NoNewPrivileges=true	Limiter escalation de privileges.

```
# /etc/systemd/system/myapp.service
 [Unit]
 Description=My Django Application
 After=network.target
 Wants=network-online.target

 [Service]
 User=app
 Group=app
 WorkingDirectory=/opt/myapp/current
 EnvironmentFile=/etc/myapp/myapp.env
 ExecStart=/opt/myapp/current/venv/bin/gunicorn config.wsgi:application --bind
127.0.0.1:8000
 Restart=on-failure
 RestartSec=5
 TimeoutStopSec=30
 KillSignal=SIGTERM

...
```

6. Logs, journald, auth logs et audit operationnel

Les logs sont essentiels pour le debug, la securite et les post-mortems. Il faut conserver assez d'historique, eviter les logs contenant des secrets, et savoir rapidement retrouver une erreur par service.

Fichier	Usage
/var/log/auth.log	SSH, sudo, authentification.
/var/log/syslog	Evenements systeme Debian/Ubuntu.
/var/log/nginx/access.log	Requetes HTTP.
/var/log/nginx/error.log	Erreurs Nginx / proxy.
journalctl -u service	Logs systemd d'un service.

```
# Logs d'un service
 journalctl -u nginx -n 100 --no-pager
 journalctl -u myapp -f

 # Logs depuis une date
 journalctl -u myapp --since "2026-04-30 10:00:00"

 # Erreurs systeme recentes
 journalctl -p err -n 100 --no-pager

 # Kernel logs
 journalctl -k -n 100 --no-pager

 # Taille journald
 journalctl --disk-usage
```

7. Permissions, ownership, secrets et dossiers applicatifs

Les permissions Linux limitent ce qu'un utilisateur ou service peut lire, modifier ou executer. Un mauvais owner, un chmod 777 ou un secret lisible par tous peuvent transformer un bug applicatif en compromission serveur.

Chemin	Owner	Mode
/opt/myapp/current	app:app	750 dossiers
Code applicatif	app:app	640 fichiers
Scripts deploy	deploy:deploy	750
/etc/myapp/myapp.env	root:app	640
~/.ssh	User	700
authorized_keys	User	600

```
# Voir permissions
 ls -la /opt/myapp
 namei -l /opt/myapp/current/.env

 # Changer owner
 sudo chown -R app:app /opt/myapp/current

 # Permissions dossiers/fichiers
 sudo find /opt/myapp/current -type d -exec chmod 750 {} \;
 sudo find /opt/myapp/current -type f -exec chmod 640 {} \;

 # Executable script
 sudo chmod 750 /opt/myapp/current/scripts/deploy.sh
```

8. Mises a jour, CVE, reboot et patch window

Les mises a jour corrigent des vulnerabilites et bugs, mais peuvent aussi casser un service si elles sont appliquees sans verification. En production, il faut un rythme, une fenetre, une verification et un plan de rollback.

Type update	Frequence	Controle
Security critical	Rapide.	Fenetre courte + validation.
Security standard	Hebdo / bihebdo.	Patch window.
Kernel	Planifie.	Reboot requis.
Major version	Projet dedie.	Staging, backup, rollback.

```
sudo apt update
 apt list --upgradable

 # Simuler upgrade
 sudo apt -s upgrade

 # Appliquer
 sudo apt upgrade

 # Security updates si unattended-upgrades configure
 sudo unattended-upgrade --dry-run --debug

 # Reboot requis ?
 test -f /var/run/reboot-required && cat /var/run/reboot-required
```

5.5 - Former les nouveaux DevOps

Progression, exercices, erreurs volontaires, revues et rituels d'equipe.

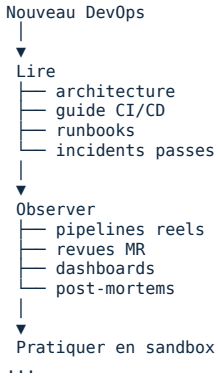
Source modal: 5.5 Former les nouveaux DevOps : progression, exercices, erreurs volontaires, revues et rituels d'équipe

1. Former un DevOps : transformer la documentation en competence terrain

Former un nouveau DevOps ne consiste pas seulement a lui donner une liste de commandes. Il faut lui apprendre a raisonner en production : contexte, risque, environnement, rollback, logs, permissions, pipeline, state, secrets, monitoring et communication.

Un bon parcours alterne theorie courte, lecture de vrais exemples, exercices en sandbox, erreurs volontaires, revues de Merge Request, simulation d'incident et observation de vrais rituels d'equipe. L'objectif est que le junior sache expliquer ce qu'il fait, pourquoi il le fait, quel est le risque et comment revenir en arriere.

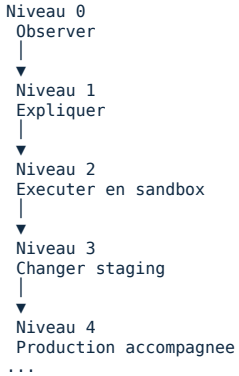
Pilier	Objectif	Exemple concret
Comprendre	Lire architecture, pipelines et runbooks.	Expliquer un pipeline Terraform de bout en bout.
Pratiquer	Executer en sandbox sans risque production.	Faire un plan Terraform sans apply.
Diagnostiquer	Observer avant d'agir.	Lire logs Nginx, systemd, GitLab runner.
Revoir	Apprendre via feedback structure.	MR commentee avec risques et rollback.
Simuler	Creer des incidents controles.	Pipeline casse, secret manquant, lock Terraform.
Documenter	Transformer l'experience en savoir collectif.	Mini RCA, runbook, note de changement.



2. Progression : de lecteur prudent a operateur autonome

Une progression saine donne de l'autonomie par paliers. Chaque palier ajoute un peu plus de responsabilite, mais aussi plus de discipline : revue, validation, trace et communication.

Niveau	Ce que le junior fait	Garde-fou
0. Observation	Lit les pipelines, runbooks, logs et MR.	Aucune action de modification.
1. Lecture active	Explique un incident ou un pipeline.	Validation senior.
2. Sandbox	Execute commandes sans impact production.	Environnement isole.
3. Staging	Propose une MR simple, lance plan/deploy staging.	MR review obligatoire.
4. Production accompagnee	Suit un runbook production avec senior.	Supervision directe.
5. Autonomie controlee	Gere changements standards et incidents simples.	Post-review et escalation claire.



3. Exercices pratiques : apprendre avec de vrais gestes DevOps

Les exercices doivent reproduire les situations reelles sans risque : pipeline qui echoue, plan Terraform a relire, service systemd a diagnostiquer, incident Nginx, secret manquant, saturation disque, healthcheck casse ou rollback staging.

Exercice	Objectif	Niveau
Lire un Terraform plan	Identifier create/update/delete et risque.	Debutant
Debug pipeline GitLab	Trouver variable manquante ou artifact absent.	Debutant/intermediaire
Service systemd KO	Lire status, journald, env file, permissions.	Intermediaire
Nginx 502	Tester upstream, logs, ports, service app.	Intermediaire
State lock Terraform	Comprendre lock, apply concurrent, force-unlock.	Intermediaire
Rollback staging	Revenir a version precedente proprement.	Intermediaire
Mini incident DB	Connexions, locks, slow query en lecture seule.	Avance

```
Mission:
 - Ouvrir plan.txt
 - Compter create / update / delete
 - Identifier ressources critiques
 - Expliquer pourquoi le changement est attendu
 - Identifier rollback ou mitigation

 Questions:
 - Y a-t-il un destroy ?
 - Le state semble-t-il correspondre au bon environnement ?
 - Les tags/naming sont-ils corrects ?
 - Le changement touche-t-il production ?
 - Qui doit approuver ?
```

4. Erreurs volontaires : apprendre sans casser la production

Les erreurs volontaires sont tres efficaces pour former les reflexes. Elles doivent etre creees dans un environnement controle, documentees, reproductibles et corrigees ensuite en groupe.

Erreur volontaire	Symptome	Competence apprise
Variable GitLab manquante	Job auth failed ou variable empty.	Variables masked/protected/scoped.
Mauvais TF_ROOT	cd: no such file .	Debug chemins CI.
Artifact Terraform absent	Apply ne trouve pas tfplan .	dependencies / needs / paths.
Nginx upstream faux	502 Bad Gateway.	Tester upstream et logs Nginx.
Permission env file incorrecte	Service systemd ne demarre pas.	Permissions et users services.
Port firewall ferme	Timeout externe.	ss, nc, UFW, security groups.
Secret expose en log sandbox	Token visible.	Rotation et hygiene logs.

```
Exercise:
 - Title: Broken GitLab artifact
 - Environment: sandbox only
 - Injected error: wrong artifact path
 - Expected symptom: apply job cannot find tfplan
 - Allowed actions: read logs, inspect job config, MR fix
 - Forbidden actions: bypass apply, recreate state
 - Success: trainee explains cause and fix
```

5. Revues : apprendre a travers les MR, plans et incidents

La revue est le meilleur outil de transmission senior → junior. Elle apprend la rigueur : pourquoi ce changement, quel risque, quel environnement, quel plan, quel rollback, quelle observation apres execution.

Point	Question de revue
Contexte	Pourquoi ce changement est-il necessaire ?
Environnement	Dev, staging ou production ?
Terraform plan	Y a-t-il un destroy ou remplacement inattendu ?
Secrets	Un secret est-il ajoute dans Git ou logs ?
IAM	Le droit ajoute est-il minimal ?
CI/CD	Le pipeline est-il reproductible et protege ?
Rollback	Comment revenir en arriere ?

```
Review comment:
 - The change is clear and limited to staging.
 - Terraform plan shows 2 creates and 0 destroy.
 - Please add the rollback note:
 "Revert this MR and re-apply previous module version."
 - Please include post-check:
 curl -fsS https://staging.example.com/health/
 - Approved after rollback note is added.
```

6. Rituels d'equipe : rendre l'apprentissage continu

La formation ne doit pas etre un evenement ponctuel. Les meilleurs apprentissages viennent des rituels courts et reguliers : revue d'incident, revue de plan, demo de pipeline, lecture de logs et partage d'un runbook.

Rituel	Frequence	But
Revue de plan Terraform	Hebdomadaire	Apprendre a lire les risques infra.
Debug pipeline live	Hebdomadaire ou bihebdo	Former aux logs GitLab et runners.
Incident review	Apres incident	Comprendre cause et prevention.
Runbook reading	Mensuelle	Verifier que la doc est utilisable.
Game day staging	Mensuelle/trimestrielle	Simuler panne sans risque utilisateur.
Office hour DevOps	Hebdomadaire	Questions juniors, dette, bonnes pratiques.

```
Duration: 30 minutes

 1. Context of the change
 2. Read Terraform plan summary
 3. Identify risky resources
 4. Discuss IAM/network/database impact
 5. Define rollback
 6. Define post-checks
 7. Capture learning points
```

7. Culture production : apprendre la prudence sans bloquer l'autonomie

Un DevOps doit apprendre que la production n'est pas seulement un environnement technique : c'est un systeme vivant avec utilisateurs, donnees, couts, securite, astreinte, support, SLA et reputation.

Reflexe	Question a se poser
Impact	Qui sera touche si cela casse ?
Fenetre	Est-ce le bon moment pour ce changement ?
Rollback	Comment revenir en arriere rapidement ?
Observation	Quels dashboards/logs regarder pendant et apres ?
Communication	Qui doit etre informe ?
Donnees	Peut-on perdre, corrompre ou exposer des donnees ?
Securite	Ce changement ouvre-t-il des droits ou ports ?

```
I am changing:
 - what: the exact component
 - where: the exact environment
 - why: the business or technical reason
 - risk: what can go wrong
 - rollback: how to revert
 - validation: how to prove it works
 - owner: who is accountable
```

8. Evaluation : verifier l'autonomie sans pieger

Evaluer un DevOps ne consiste pas a voir s'il connait toutes les commandes par coeur. Il faut verifier sa methode : diagnostic, prudence, clarte, securite, documentation, communication et capacite a demander de l'aide.

Competence	Debutant	Autonome
Contexte	Suit les instructions.	Identifie environnement, impact et risque.
Terraform	Lance fmt/validate/plan.	Interprete plan et detecte destroy inattendu.
GitLab CI	Lit un job echoue.	Diagnostique runner, variables, artifacts.
Linux	Lit systemctl et journalctl.	Trouve cause permission/env/port/service.
Securite	Sait qu'il ne faut pas exposer secrets.	Detecte fuite potentielle et propose rotation.
Production	Demande confirmation.	Prepare rollback, post-check et communication.

```
Scenario:
 - A staging deployment failed.
 - The app is returning 502.
 - GitLab pipeline was green.
 - Nginx is running.
 - The app service is failed.

 Expected trainee actions:
 1. Confirm environment.
 2. Read Nginx error logs.
 3. Check systemd service status.
 4. Read journalctl.
 5. Identify missing EnvironmentFile.
 6. Propose fix.
 7. Validate health endpoint.
 8. Write short RCA.
```

5.6 - Glossaire avancé

Les termes que les nouveaux DevOps rencontrent tous les jours.

Source modal: 5.6 Glossaire avance : les termes que les nouveaux DevOps rencontrent tous les jours

1. Glossaire DevOps : parler le meme langage en production

Le vocabulaire DevOps peut etre intimidant pour un nouveau membre d'equipe : SLA, SLO, RTO, RPO, rollback, drift, state, runner, artifact, blast radius, healthcheck, canary, IAM, OIDC, RBAC, observabilite, post-mortem.

Un glossaire utile ne doit pas etre une liste academique. Il doit relier chaque terme a une situation concrete : revue de MR, incident production, pipeline GitLab, Terraform plan, audit securite, changement reseau ou deploiement applicatif.

Famille	Exemples	Pourquoi c'est important
Production	SLA, SLO, SLI, RTO, RPO, rollback.	Relie technique, risque et utilisateur.
CI/CD	Pipeline, runner, artifact, stage, job, approval.	Comprendre comment le changement arrive en prod.
Terraform	State, plan, apply, drift, module, provider.	Eviter les erreurs d'infrastructure.
Cloud	VPC, subnet, security group, IAM, load balancer.	Comprendre le chemin reseau et les droits.
Securite	Least privilege, OIDC, RBAC, secrets, break-glass.	Reduire le blast radius et proteger les acces.
Observabilite	Logs, metrics, traces, alerting, dashboard.	Diagnostiquer avec des faits.

```
DevOps vocabulary
  │
  ├── Change
  │   ├── MR
  │   ├── pipeline
  │   ├── plan
  │   ├── deploy
  │   └── rollback
  │
  ├── Production
  │   ├── SLA / SLO / SLI
  │   ├── incident
  │   ├── mitigation
  │   ├── RCA
  │   └── post-mortem
  │
  ├── Infrastructure
  │   ├── Terraform
...
```

2. Production : SLA, SLO, SLI, RTO, RPO, rollback

Terme	Definition simple	Exemple
SLA	Engagement contractuel de service.	Disponibilite 99.9% par mois.
SLO	Objectif interne mesurable.	95% des requetes repondent sous 300 ms.
SLI	Indicateur qui mesure le service.	Latence p95, taux de 5xx, uptime.
RTO	Temps maximal acceptable pour restaurer le service.	Remettre l'API en ligne en 30 minutes.
RPO	Perte de donnees maximale acceptable.	Perdre au maximum 5 minutes de donnees.
Rollback	Retour a une version ou config precedente.	Revenir a l'image Docker N-1.
Mitigation	Action qui reduit l'impact sans forcement corriger la cause.	Desactiver une feature flag en incident.

```
SLI = ce que l'on mesure
  └── ex: taux de 5xx

 SLO = objectif interne
  └── ex: 5xx < 1% sur 30 jours

 SLA = engagement externe
  └── ex: credit client si disponibilite non respectee
```

3. CI/CD : pipeline, job, runner, artifact, environment, approval

Terme	Definition	Exemple
Pipeline	Ensemble de jobs executes pour tester, construire, deployer.	test -> build -> plan -> deploy.
Stage	Etape logique du pipeline.	validate , plan , apply .
Job	Unite d'execution dans un stage.	terraform_plan_prod .
Runner	Machine ou container qui execute les jobs.	Runner protege production.

Terme	Definition	Exemple
Artifact	Fichier produit et conserve par un job.	tfplan , plan.txt , rapport scan.
Cache	Fichiers reutilises pour accelerer les jobs.	Cache dependencies npm/pip.
Environment	Cible logique de deploiement.	staging, production.

```
Commit
 │
 ▼
Pipeline
├── lint
├── test
├── build
├── security scan
├── terraform plan
└── deploy
 │
 ▼
Artifacts
├── image
├── plan
├── reports
└── release notes
```

4. Terraform : state, plan, apply, drift, provider, module

Terme	Definition	Risque associe
Provider	Plugin qui parle a une API.	Mauvaise version = comportement different.
Resource	Objet gere par Terraform.	Destroy inattendu si mal configure.
Module	Bloc reutilisable de ressources.	Changement module impacte plusieurs envs.
State	Memoire des objets connus par Terraform.	State perdu ou corrompu = risque majeur.
Backend	Lieu ou le state est stocke.	Backend local en equipe = dangereux.
Lock	Verrou empechant deux apply concurrents.	Force-unlock abusif = corruption possible.
Plan	Simulation du changement.	Non lu = destroy cache possible.

```
Code Terraform
 │
 ▼
terraform fmt
 │
 ▼
terraform validate
 │
 ▼
terraform plan
 │
 ├── create
 ├── update
 ├── replace
 └── destroy
 │
 ▼
Review humaine
...
```

5. Cloud / Infrastructure : VPC, subnet, security group, LB, DNS

Terme	Definition simple	Exemple incident
VPC / VNet	Reseau prive cloud.	Peering absent entre deux VPC.
Subnet	Sous-reseau dans une zone.	Instance en subnet prive sans NAT.
Route table	Regles de routage reseau.	Pas de route vers internet gateway.
Security group	Firewall attache aux ressources.	Port 443 ferme par erreur.
NACL	Firewall stateless au niveau subnet.	Ports ephemeres bloques.
Load balancer	Distribue le trafic vers backends.	Targets unhealthy.
DNS record	Associe un nom a une cible.	CNAME pointe vers ancien LB.

```
User
 │
 ▼
DNS
 │
 ▼
Load Balancer
 │
 ▼
Security Group
 │
 ▼
Subnet / Route Table
 │
 ▼
Instance / Pod / Service
 │
 ▼
```

```
...
```

6. Securite : IAM, RBAC, OIDC, secrets, least privilege

Terme	Definition	Exemple
IAM	Gestion des identites et permissions.	Role Terraform apply production.
RBAC	Droits bases sur les roles.	Role Kubernetes lecture namespace.
OIDC	Federation d'identite via token court.	GitLab assume un role cloud sans secret long terme.
Least privilege	Donner uniquement les droits necessaires.	Plan role different de apply role.
Secret	Information sensible.	Password DB, token API, private key.
Break-glass	Acces d'urgence exceptionnel.	Role admin temporaire en incident critique.
Supply chain	Chaine qui produit et livre le logiciel.	Images Docker, dependencies, runners.

```
This role violates least privilege because it allows Action="*".

 The GitLab job should use OIDC instead of long-lived cloud keys.

 The secret was printed in logs, so it must be rotated.

 This break-glass access must be linked to an incident ticket.

 The blast radius is limited to staging because the role cannot assume prod.
```

7. Observabilite : logs, metrics, traces, alerts, dashboards

Terme	Definition	Exemple
Log	Evenement textuel produit par un systeme.	Erreur Nginx 502, exception Django.
Metric	Valeur numerique mesuree dans le temps.	CPU, RAM, 5xx rate, latency p95.
Trace	Suivi d'une requete entre services.	API -> DB -> Redis -> external API.
Alert	Notification quand une condition est anormale.	Disk > 90% for 10 minutes.
Dashboard	Vue graphique des signaux importants.	Production API overview.
Baseline	Comportement normal attendu.	Latence p95 normale autour de 180 ms.
Cardinality	Nombre de combinaisons de labels.	Trop de labels user_id explose les metrics.

```
Golden signals
  │
  ├── Latency
  │   └── how long requests take
  │
  ├── Traffic
  │   └── how much demand exists
  │
  ├── Errors
  │   └── how often requests fail
  │
  └── Saturation
  └── how full the system is
```

8. Incidents : triage, mitigation, RCA, post-mortem

Terme	Definition	Question associee
Triage	Evaluation rapide de l'impact et de la priorite.	Qui est touche ? Depuis quand ?
Severity	Niveau de gravite de l'incident.	Incident mineur ou critique ?
Incident commander	Personne qui coordonne la reponse.	Qui decide go/no-go ?
Mitigation	Action pour reduire l'impact rapidement.	Peut-on desactiver, scaler, rollback ?
RCA	Root Cause Analysis.	Pourquoi l'incident est arrive ?
Post-mortem	Analyse documentee apres incident.	Que change-t-on pour eviter la recidive ?
Action item	Action preventive issue du post-mortem.	Qui fait quoi et pour quand ?

```
Detection
  │
  ▼
Triage
  ├── impact
  ├── severity
  ├── owner
  └── timeline
  │
  ▼
Mitigation
  ├── rollback
  ├── scale
  ├── disable feature
  └── traffic control
  │
  ▼
Recovery
...
```

6.1 - Annexes pratiques

Plans de formation, matrices de risques, procédures go/no-go et modèles opérationnels.

Source modal: 6.1 Annexes pratiques : plans de formation, matrices de risques, Go/No-Go et modeles operationnels

1. Annexes pratiques : transformer le guide en outil operationnel

Cette annexe regroupe les modeles reutilisables pour une equipe DevOps : matrices de risques, decision Go/No-Go, plan de formation, grille de maturite, modele de changement, runbook, incident review, checklist de revue et modeles de communication.

L'objectif n'est pas d'ajouter de la bureaucratie. L'objectif est de rendre les changements plus lisibles, plus auditables, plus securises et plus faciles a transmettre aux nouveaux DevOps. Un bon modele permet de gagner du temps pendant les incidents et d'eviter les oublis critiques.

Annexe	Usage	Moment d'utilisation
Matrice de risques	Evaluer impact, probabilite, blast radius et rollback.	Avant une MR ou un apply production.
Go / No-Go	Decider si un changement peut partir.	Avant une fenetre de production.
Plan de formation	Former les nouveaux DevOps par ateliers.	Onboarding et montee en competence.
Maturite DevOps	Mesurer le niveau d'industrialisation.	Audit interne ou roadmap plateforme.
Change model	Documenter une modification infra/app.	Dans une MR, un ticket ou une release note.
Incident model	Structurer timeline, mitigation, RCA.	Pendant et apres incident.

```
Besoin de changement
 |
 ▼
Analyse de risque
 ├── environnement
 ├── impact
 ├── blast radius
 ├── rollback
 └── owner
 |
 ▼
MR / Review
 ├── plan Terraform
 ├── pipeline
 ├── secrets
 ├── IAM
 └── observabilite
 |
...
```

2. Matrice de risques operationnels DevOps

Cette matrice aide a qualifier un changement avant execution. Elle permet d'identifier les risques production, securite, donnees, couts, IAM, reseau, CI/CD, rollback et monitoring.

Niveau	Impact	Exemple	Decision
1 - Faible	Aucun impact utilisateur.	Tag, commentaire, variable dev.	Review standard.
2 - Modere	Impact limite ou staging.	Ajout ressource non critique.	Review + plan.
3 - Significatif	Production touchee mais rollback simple.	Changement Nginx ou autoscaling.	Go/No-Go recommande.
4 - Eleve	Production critique, risque client.	DB, IAM prod, security group public.	Approval senior obligatoire.
5 - Critique	Risque perte donnees, outage ou faille majeure.	Migration DB massive, restore, network core.	Fenetre dediee + rollback teste.

```
Risk score =
 impact
 × probability
 × blast radius
 × rollback difficulty

Example:
 impact = 4
 probability = 2
 blast radius = 3
 rollback difficulty = 3

 score = 72
 => senior review + Go/No-Go required
```

3. Modele Go / No-Go : decision courte, claire et tracable

Une decision Go/No-Go doit etre comprehensible par une personne qui n'a pas ecrit le changement. Elle verifie que les conditions minimales sont reunies avant de toucher un environnement sensible.

Raison	Pourquoi
Plan Terraform contient un destroy inattendu.	Risque de perte de ressource.
Backup DB non verifie.	Pas de filet de securite.
Owner applicatif absent.	Decision metier impossible.
Monitoring indisponible.	Impossible de valider l'impact.
Rollback non documente.	Risque prolonge en incident.

```
GO / NO-GO TEMPLATE

Change:
- Service:
- Environment:
- Owner:
- Reviewer:
- Ticket / MR:
- Expected impact:
- Customer visible: yes/no
- Planned window:
- Risk level: 1/2/3/4/5

Pre-check:
- Pipeline green:
- Terraform plan reviewed:
- No unexpected destroy:
- State/backend confirmed:
...
```

4. Plan de formation DevOps en ateliers progressifs

Ce plan donne une progression concrete pour nouveaux DevOps : lecture, pratique, sandbox, staging, incidents simules, revue de MR et production accompagnee.

Atelier	Objectif	Livrable
1. Architecture	Lire le schema global et le chemin requete.	Schema resume par l'apprenant.
2. GitLab CI	Comprendre stages, jobs, variables, artifacts.	Explication d'un pipeline reel.
3. Terraform plan	Lire create/update/destroy et risques.	Resume humain du plan.
4. Modules Terraform	Comprendre variables, outputs, versioning.	Mini module documente.
5. State distant	Comprendre backend, lock, drift.	Exercice state en sandbox.
6. Secrets CI	Masked, protected, scoped, OIDC.	Checklist secrets.
7. Linux ops	systemctl, journalctl, ports, disque.	Diagnostic service casse.

```
Jours 1-30
  ├── lire architecture
  ├── observer pipelines
  ├── comprendre Terraform plan
  ├── pratiquer sandbox
  └── lire runbooks

Jours 31-60
  ├── MR infra simple
  ├── debug pipeline staging
  ├── exercice Linux/Nginx
  ├── mini incident simule
  └── runbook update

Jours 61-90
  ├── deploy staging autonome
  ├── production accompagnee
  ├── Go/No-Go simple
...
```

5. Echelle de maturite DevOps infra

Cette grille permet de situer une equipe ou une plateforme, puis de definir le prochain cap. Elle peut etre utilisee en audit interne, roadmap plateforme ou plan d'amelioration continue.

Niveau	Description	Signaux visibles	Prochain cap
0. Manuel	ClickOps, changements manuels, peu de traces.	Console cloud, docs incompletes, changements non reproductibles.	Mettre Git au centre.
1. Scripted	Scripts et Terraform local, state local.	Automatisation partielle mais fragile.	Backend distant, separation environnements.
2. CI visible	fmt, validate, plan en pipeline.	Apply encore manuel depuis laptop.	Apply via GitLab avec approvals.
3. Production controlee	MR, approvals, protected env, runbooks.	Changements prod mieux traces.	Observabilite et exercices incidents.
4. Platform engineering	Modules versionnes, OIDC, policy-as-code, templates CI.	Standards reutilisables multi-equipes.	Self-service securise.
5. Excellence continue	Mesure fiabilite/cout/securite, game days, RCA suivies.	Amelioration continue pilotee par donnees.	Optimisation produit et plateforme.

```
Questions:
- Les changements infra passent-ils par MR ?
- Le state Terraform est-il distant et locke ?
- Les plans sont-ils publies en artifacts ?
- L'apply production est-il manuel et protege ?
- Les secrets CI sont-ils protected/masked/scoped ?
- Les modules sont-ils versionnes ?
- Existe-t-il des runbooks testes ?
- Les incidents produisent-ils des RCA suivies ?
- Les couts et risques sont-ils visibles avant apply ?
```

6. Modele de note de changement operationnelle

Une note de changement doit etre courte, concrete et orientee decision. Elle doit permettre a un reviewer de comprendre le changement sans deviner le contexte.

Claire	Pas besoin d'appeler l'auteur pour comprendre.
Courte	Lisible pendant une fenetre de changement.
Verifiable	Contient commandes, dashboards ou criteres.
Reversible	Rollback explicite avant execution.
Tracable	Liee a MR, ticket, pipeline, owner.

```
CHANGE NOTE

Title:
Service:
Environment:
Owner:
Reviewer:
Ticket / MR:
Planned date:

Context:
- Why is this change needed?
- What problem does it solve?

Scope:
- Included:
- Excluded:

...
```

7. Modele de runbook operationnel

Un runbook doit aider quelqu'un sous pression. Il doit etre actionnable, ordonne, et contenir des commandes de lecture avant les commandes de modification.

Critere	Question
Lisible	Un junior peut-il le suivre ?
Prudent	Les actions read-only viennent-elles d'abord ?
Verifiable	Y a-t-il des post-checks ?
Escalable	Qui appeler si cela ne suffit pas ?
Maintenu	Date de derniere revue presente ?

```
RUNBOOK TEMPLATE

Title:
Service:
Environment:
Owner:
Last reviewed:
Severity:

Symptoms:
- What alert fires?
- What user impact?
- What logs or metrics confirm?

First checks:
1. Check environment/context.
2. Check recent deployments.
3. Check service health.
...
```

8. Modele incident : timeline, mitigation, RCA, actions

Le modele incident sert a garder une trace propre pendant la crise et a produire une RCA exploitable. Il separe les faits, les hypotheses, les actions et les decisions.

What happened?	Facts and timeline.
Why did it happen?	Technical and process causes.
Why was it not caught earlier?	Testing, alerting, review gaps.
What reduced the impact?	Good practices to keep.
What will prevent recurrence?	Action items with owners.

```
INCIDENT TEMPLATE

Incident title:
Severity:
Start time:
Detection time:
End time:
Incident commander:
Services affected:
Customer impact:

Current status:
- Investigating / Mitigating / Monitoring / Resolved

Timeline:
- HH:MM alert fired
- HH:MM first triage
- HH:MM mitigation started
...
```

Final operational checklist

Control	Expected state
1	Terraform state is remote, versioned, locked and restricted.
2	Every production infrastructure change has a plan artifact, review and approval.
3	GitLab protected variables and protected environments are configured for production.
4	Secrets are masked, scoped and never written to logs or artifacts.
5	Runbooks exist for rollback, restart, saturation, network, database and pipeline failures.
6	Dashboards and alerts are in place before production incidents occur.
7	Incident handling separates facts from hypotheses and mitigation from root-cause analysis.
8	Post-mortems produce action items with owners, due dates and follow-up.
9	Cost impact is reviewed before infrastructure apply.
10	New DevOps engineers train on sandbox exercises, controlled failures and guided reviews.