GitLab
Plateforme DevSecOps de bout en bout
Présentation, architecture, exploitation, automatisation, CI/CD

Association LinuxMaine
Thierry GAYET
Thierry.Gayet@gmail.com

Association LinuxMaine

2026

Plan général

Partie 1 — Git & outils SCM

- Histoire de git (Linus, 2005, distribué)
- Comparatif Gitolite / **GitLab CE & EE** / GitHub / Bitbucket
- Matrice CE / Free / Premium / Ultimate
- Migration CE → EE

Partie 2 — Architecture & admin GitLab

- Architecture interne (Puma, Sidekiq, Gitaly, Workhorse, KAS, ...)
- Refcard scripts `gitlab-*.sh`
- Prérequis matériels (sizing GitLab + runners)
- 5 modes d'installation (Omnibus, Docker, Compose, K8s, Ansible/OpenTofu)
- Admin Area : users, PAT, groupes / sous-groupes, invitations
- Matrice complète des 7 rôles
- SMTP, Backup/restore, Audit Events
- SSO Keycloak (OIDC, mapping de groupes)
- Monitoring : Zabbix, Prometheus + Loki + Grafana, alertes
- Runners : niveaux, registration, `config.toml`, maintenance

Partie 3 — Mon premier repo

- Création + settings General, Repository, MR, CI/CD
- Branch protection, push rules, codeowners, MR approvals
- **Mirroring** (GitLab → GitHub, GitLab → GitLab, Pull Premium, diagnostic)
- Webhooks détaillés + Container Registry + Packages + Pages + Releases
- Analytics / Monitor / Operate

Partie 4 — CI/CD

- Anatomie d'un `.gitlab-ci.yml`
- **8 exemples pédagogiques progressifs** : Hello World → Build/test → Matrice → Cache/artifacts → Rules → DAG/needs → Review apps → Blue/green
- Patterns de réutilisation (templates, includes, extends, parent-child)
- Cross-pipeline artifacts + chaînes `strategy:depend`
- Pipeline DevSecOps complet (SAST/DAST/SCA/Secrets/SBOM)
- **Tous les types de tokens** (PAT, Project/Group, Deploy, Trigger, Job, Runner, Email, Feed, OAuth, Impersonation) + rotation
- **Tous les GitFlows** : DevOps, DevSecOps, GitOps, MLOps, LLMOps, AIOps, DataOps, FinOps, SecOps, NetOps, ChatOps, ChaosOps, PlatformOps — avec variables CI dédiées
- Variables CI & secrets (4 dimensions) + Vault JWT

Partie 5 — API GitLab

- Portée : ce qui est exposé + limites (rate limits, complexité)
- REST v4 : auth (PAT, OAuth, Job token, session), endpoints, pagination
- GraphQL : query / mutation / introspection, GraphiQL
- Bibliothèques par langage + exemples *(depuis `api/`)*
- Webhooks receiver (Express.js)

Partie 6 — Annexes

- Lexique français (40+ termes)
- Bibliographie FR + EN
- Refcards `.gitlab-ci.yml`, `.gitignore`, ...
- Liens + docs officielles + chaînes YouTube FR (Stéphane Robert, Xavki, Geoadmin, ...)

Comment lire ce module

Structure pédagogique

1. **Cours** : concept + vocabulaire
2. **À retenir** : 3–5 points clés
3. **TP associé** : pratique guidée
4. **Checkpoint** : validation
5. Section suivante

Conventions

✓ ce qui est validé
× ce qui est interdit
[i] pour info
[!] attention danger

Pré-requis matériel

- Linux Debian 12 / Ubuntu / Fedora
- 4 vCPU, 8 Go RAM, 30 Go disque
- Docker Engine + Compose v2
- Connexion Internet

Parcours TP — 23 ateliers

TP	Sujet	Apprentissages
01	Premiers pas avec git	init/add/commit/branch/merge/conflict
02	Git distribué	remote/push/pull/fetch + rebase vs merge
03	Architecture GitLab	gitlab-ctl, services internes, logs
04	Scripts wrappers du lab	gitlab-start/stop/status/ssh.sh
05	GitLab en Docker	gitlab-ce:latest, volumes, reconfigure
06	Provisioning Ansible	playbook + rôles common/docker/gitlab
07	Provisioning OpenTofu	provider gitlabhq, backend HTTP, lock
08	Users + groupes + rôles	Guest/Reporter/Developer/Maintainer/Owner
09	Keycloak SSO (OIDC)	realm, client, mapping de groupes
10	Monitoring stack	Prometheus + Loki + Grafana + Zabbix
11	Runner Docker	register, executor, premier job
12	Premier repo	branch protection, push rules, settings
13	Workflow Merge Request	review, approvals, squash, codeowners
14	.gitlab-ci.yml de base	stages, jobs, rules, needs, artifacts, cache
15	Pipeline DevSecOps	SAST, DAST, SCA, Secret Detection, SBOM
16	Secrets & Vault	masked/protected, JWT GitLab → Vault
17	Review apps	1 environnement par MR + `on_stop`
18	Multi-project pipeline	`trigger:project:` + `strategy:depend`
19	Parent-child monorepo	`rules:changes:` + 3 sous-pipelines conditionnels
20	GraphQL	queries, mutations, introspection (GraphiQL)
21	Cross-language API	même cas dans 3 langages depuis `api/`
22	Rotation tokens	`POST /access_tokens/:id/rotate` + schedule
23	Pipeline MLOps	DVC + MLflow + quality gate + model registry

Tous les TP sont dans `labs/tpXX-*/README.md`.

1
Partie 1 — Git & outils SCM

Partie 1 — Ce que nous allons voir

Au programme

1. Histoire de git (2005, BitKeeper, Linus)
2. Concepts fondamentaux : DAG, SHA-1, distribué
3. Outils SCM concurrents :
   - Gitolite
   - GitLab
   - GitHub
   - Bitbucket

Ce que vous saurez à la fin

- Pourquoi git a gagné (vs SVN, Mercurial, Perforce)
- Différencier hébergement self-hosted vs SaaS
- Choisir une forge selon les contraintes
- Comparer matrices fonctionnelles

Avant git : SVN, CVS, Perforce, BitKeeper

Le contexte historique (1986–2005)

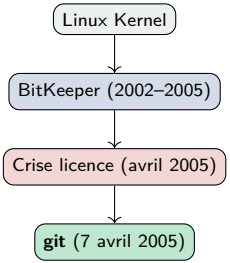
- **CVS** (1986) : versionnement fichier par fichier, serveur central
- **SVN** (Subversion, 2000) : améliore CVS, atomicité, branchements lourds
- **Perforce** : commercial, gros monorepos (Google, AAA games)
- **BitKeeper** (2002) : gratuit pour l'OSS dont *Linux kernel*, distribué

⋆ À retenir

Tous *centralisés* sauf BitKeeper. Coût élevé pour les branches. Un commit = un round-trip réseau.

La naissance de git — avril 2005

- Avril 2005 : BitMover (BitKeeper) retire la licence gratuite pour Linux
- Linus Torvalds n'aime aucune alternative existante
- Il code **git en 10 jours** avec 4 objectifs :
  1. Rapide (perf indispensable sur le kernel)
  2. **Distribué** (pas de serveur unique)
  3. Forte intégrité (SHA-1 partout)
  4. Workflow non-linéaire massif (1000s de branches)
- Premier commit auto-hébergé : `e83c5163316f89bfb`, 7 avril 2005
- Junio Hamano prend la maintenance dès juillet 2005



Concepts fondamentaux de git

Objets internes

- `blob` : contenu d'un fichier
- `tree` : répertoire
- `commit` : snapshot + métadonnées
- `tag` : référence nommée
- tous adressés par **SHA-1** (40 hex chars)

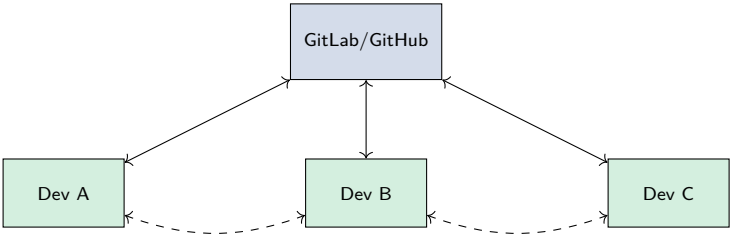
DAG (Directed Acyclic Graph)

- Chaque commit pointe vers ≥ 1 parent
- Merge = commit avec 2+ parents
- Branche = pointeur mobile vers un commit
- HEAD = pointeur vers la branche courante

⋆ À retenir

git n'est pas un système de *deltas* comme SVN/CVS — c'est un système de *snapshots* adressés par contenu. Toute la magie en découle.

git distribué : un game-changer



- **Chaque** clone est un dépôt complet (full history + objets).
- Travail **offline** possible (commit, branch, log).
- Push/pull vers *n'importe quel* remote, pas seulement le central.
- Le « serveur » est une convention sociale, pas une obligation technique.

[TP associé]

TP01 — Premiers pas : init, add, commit, branch, merge, conflit.

Gitolite — le minimaliste self-hosted

- Auteur : Sitaram Chamarty (2010+)
- Écrit en Perl, ∼10000 LoC
- Pas d'UI web — **config par fichiers texte** (push sur un repo `gitolite-admin`)
- ACL fines (par branche, par tag, regex)
- Authentification SSH uniquement
- Idéal pour : équipe technique homogène, infra minimaliste

Use case typique

- Lab universitaire
- Petite équipe ops
- Infra durcie (bastion + SSH only)

Forces

Léger, scriptable, zéro overhead.

Limites

Pas de MR/PR, pas de CI, pas d'UI.

GitLab — la plateforme intégrée

- Auteur originel : Dmitriy Zaporozhets (2011, Ukraine)
- Stack : Ruby on Rails (web) + Go (Gitaly, Workhorse) + Vue.js
- **4 éditions** :
  - Community Edition (CE, MIT)
  - Enterprise Edition (EE, source-available)
  - GitLab.com (SaaS)
  - GitLab Dedicated (mono-tenant managé)
- Couvre tout le cycle : *Plan* → *Code* → *Build* → *Test* → *Secure* → *Deploy* → *Monitor* → *Govern*

Forces

- All-in-one (issues, MR, CI/CD, registry, pages, security)
- Self-hosted possible (CE)
- AutoDevOps, Auto-DAST/SAST

Limites

- Lourd (8+ Go RAM)
- Features Premium derrière paywall EE

GitLab CE vs EE — les 4 éditions en détail

Édition	Licence	Distribution	Public visé
CE (Community Edition)	MIT (FOSS)	Paquet apt/dnf séparé, image Docker `gitlab/gitlab-ce`, sources sur GitLab.com	Self-hosted gratuit, équipes maîtrisant leur infra
EE (Enterprise Edition)	*Source-available* (gratuit jusqu'à Free, sinon abonnement)	Paquet `gitlab-ee`, image `gitlab/gitlab-ee`, sources visibles	Self-hosted avec features avancées
GitLab.com	SaaS	gitlab.com, mutualisé, géré par GitLab Inc.	Open source publics, équipes sans infra
GitLab Dedicated	SaaS managé mono-tenant	Instance EE déployée et opérée par GitLab Inc. pour vous	Grandes entreprises (souveraineté, isolation)

⋆ À retenir

Le binaire EE contient TOUTES les features — CE et les 4 tiers Free/Premium/Ultimate/Dedicated sont activés par la licence. Donc on installe quasi toujours `gitlab-ee` en prod (même pour le tier Free), ça permet d'upgrader plus tard sans réinstall.

GitLab CE vs EE : matrice fonctionnelle

Feature	CE	Free EE	Premium	Ultimate
Repos illimités, MR, issues, wiki	✓	✓	✓	✓
Pipelines CI/CD, runners shared	✓	✓	✓	✓
Container Registry, Pages, Package Registry	✓	✓	✓	✓
SSO (OIDC, SAML, Kerberos)	✓	✓	✓	✓
Push mirror (GitLab → ext.)	✓	✓	✓	✓
Pull mirror (ext. → GitLab)			✓	✓
Merge request approvals (règles)			✓	✓
Push rules (regex commit msg, size)			✓	✓
LDAP group sync			✓	✓
Multi-project pipelines		∼	✓	✓
Code Owners ($\geq$2 approvers)			✓	✓
Epics, Roadmap, Iterations			✓	✓
SAST		∼	∼	✓
Dependency Scanning / SCA				✓
DAST				✓
Container Scanning				✓
License Scanning / Compliance				✓
Security Dashboard, Vulnerability Mgmt				✓
Compliance Frameworks (SOC2, ISO27001)				✓
Geo (réplication multi-DC)			✓	✓
Streaming Audit Events				✓

Migration CE → EE — mode d'emploi

Avant la migration

- Backup complet : `sudo gitlab-backup create STRATEGY=copy`
- Backup des secrets : `/etc/gitlab/gitlab.rb` + `/etc/gitlab/gitlab-secrets.json`
- Vérifier la version : la version EE doit être **strictement identique** à la CE en place

Migration sur Debian/Ubuntu

```
sudo apt update
sudo apt install -y gitlab-ee      # remplace gitlab-ce
sudo gitlab-ctl reconfigure
sudo gitlab-rake gitlab:check
```

Migration sur Fedora/RHEL

```
sudo dnf install -y gitlab-ee
sudo gitlab-ctl reconfigure
```

⋆ À retenir

GitHub — le réseau social du code

- Lancé en 2008 par Wanstrath, Hyett, Preston-Werner, Chacon
- Racheté par Microsoft en 2018 (7,5 Mds $)
- Stack : Ruby on Rails + MySQL (à l'origine) + Go (microservices)
- Modèle : SaaS dominant + GitHub Enterprise Server (self-hosted)
- Actions = CI/CD intégrée (depuis 2019)
- Copilot = LLM intégré (depuis 2022)

Forces

- Marketplace énorme
- Communauté open source
- UI/UX raffinée
- Free pour public unlimited

Limites

- Hosting US, souveraineté
- Cher en Enterprise self-hosted

Bitbucket — l'écosystème Atlassian

- Racheté par Atlassian en 2010
- 2 produits distincts depuis 2021 :
  - Bitbucket **Cloud** (SaaS)
  - Bitbucket **Data Center** (self-hosted, ex-Server)
- Mercurial supprimé en 2020 (git-only depuis)
- Bitbucket Pipelines = CI/CD intégrée
- Intégration native Jira + Confluence + Trello

Forces

- Suite Atlassian si déjà utilisée
- Free pour ≤ 5 utilisateurs
- Self-hosted (Data Center)

Limites

- Écosystème plugins étroit
- Adoption en baisse en EU

Matrice comparative

Critère	Gitolite	GitLab CE	GitHub	Bitbucket
Licence	GPL-2	MIT (CE), EE source-avail.	Propriétaire	Propriétaire
Self-hosted	✓	✓	EE seulement	Data Center seulement
Cloud / SaaS	×	gitlab.com	github.com	bitbucket.org
UI web	×	✓	✓	✓
Merge / Pull Requests	×	✓	✓	✓
CI/CD intégrée	×	✓ Pipelines	✓ Actions	✓ Pipelines
Container Registry	×	✓	✓ GHCR	∼
Issues / Wiki	×	✓	✓	∼
SAST / DAST natif	×	✓ (Ultimate)	∼ CodeQL	×
SSO (SAML/OIDC)	×	✓ (CE/EE)	✓ (Enterprise)	✓
Free private repos (cap)	N/A	illimité (CE)	illimité	5 users max
Open source du produit	✓	✓ (CE)	×	×

[TP associé]

TP02 — Travail distribué via git : pousser sur GitLab, simuler 2 collaborateurs.

Partie 1 — Ce que vous savez maintenant

Acquis

- ✓ Origines de git (Linus, 2005, distribué, SHA-1)
- ✓ Modèle objets/DAG vs SVN (snapshot vs delta)
- ✓ 4 outils SCM majeurs et leurs trade-offs
- ✓ Critères de choix selon le contexte

Architecture interne de GitLab, gitflows (DevOps/DevSecOps/MLOps), installation et administration complète (users, groupes, rôles, SSO, monitoring, runners).

2
Partie 2 — Architecture & administration GitLab

Partie 2 — Ce que nous allons voir

Au programme (1/2)

1. GitFlows : DevOps, DevSecOps, MLOps, LLMOps, AIOps
2. Architecture interne GitLab (Puma, Sidekiq, Gitaly, Workhorse, PostgreSQL, Redis)
3. Refcard des scripts `gitlab-*.sh`
4. Prérequis matériel + conseils sizing
5. Installation
   - Astuces générales
   - Docker
   - Omnibus (apt/dnf)
   - Ansible (notre lab)
   - OpenTofu

Au programme (2/2)

6. Administration détaillée :
   - Création users / repos / groupes
   - Matrice des **rôles** (Guest → Owner + Admin)
   - Keycloak SSO (OIDC)
   - Sonde Zabbix
   - Stack Prometheus + Loki + Grafana
   - Paramétrage runners

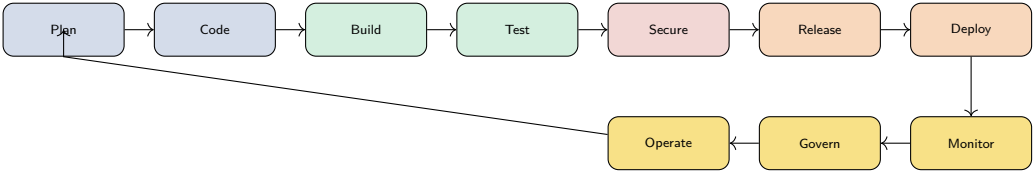
De DevOps à AIOps : panorama

Famille	Objectif central	Outils typiques (en plus de GitLab)
DevOps	Fluidifier dev → prod (CI/CD, IaC, Observabilité)	Docker, Ansible, Terraform, Kubernetes, Prometheus
DevSecOps	Intégrer la sécurité *à chaque étape*	SAST, DAST, SCA, IaC scan, SBOM, Sigstore, Vault
MLOps	Industrialiser les modèles ML (data, train, eval, deploy)	DVC, MLflow, Kubeflow, Seldon, Triton, Feast
LLMOps	Variante MLOps pour LLM / RAG / fine-tuning	LangChain, vLLM, Hugging Face, pgvector, Pinecone
AIOps	IA *pour* les ops (détection anomalies, root cause)	Loki, Grafana ML, Splunk MLTK, BigPanda, Moogsoft
GitOps	Le *repo git* comme seule source de vérité de l'état infra	Argo CD, Flux, Atlantis, Crossplane
FinOps	Optimisation des coûts cloud	OpenCost, Kubecost, Infracost, CloudHealth

⋆ À retenir

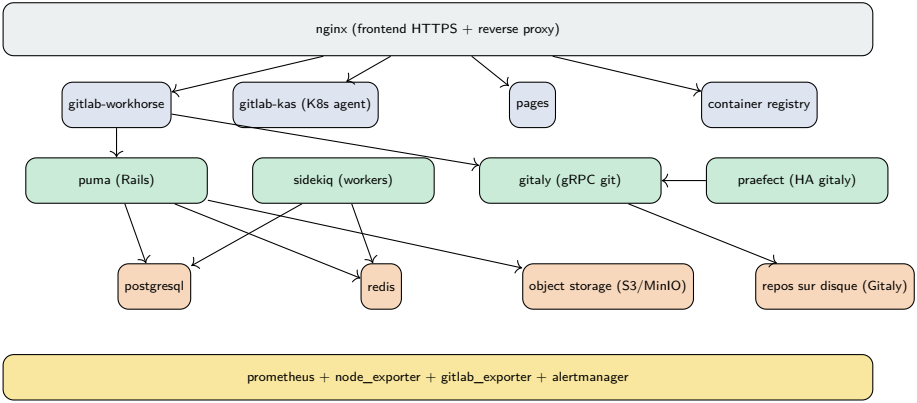
Tous partent du même socle : **git** + **CI/CD** + **automatisation déclarative** (IaC).

Le cycle DevSecOps en 8 étapes



GitLab couvre les **10 étapes** nativement (au moins partiellement). Les outils externes restent utiles pour des cas spécialisés (e.g. Vault, Argo CD).

Vue logique : les briques de GitLab Omnibus



⋆ À retenir

Workhorse (Go) décharge Puma (Ruby) des requêtes lourdes (uploads, LFS, websockets). **Gitaly** (Go) est le seul à toucher au disque pour git.

Tableau des services Omnibus

Service	Port	Rôle
nginx	80, 443	Reverse proxy frontend
gitlab-workhorse	socket Unix	Décharge Puma (uploads, LFS, websockets, git-http-backend)
puma	socket Unix	Serveur Rails (UI + API REST + GraphQL)
sidekiq	–	Workers asynchrones (mail, jobs CI, mirroring)
gitaly	8075	Backend git (clone/push/fetch) via gRPC
praefect	2305	Routeur HA pour 3+ gitaly
gitlab-kas	8150	Agent Kubernetes Server
pages	8090	GitLab Pages (sites statiques)
container registry	5000	Docker registry intégré
postgresql	5432	Données métier
redis	6379	Cache + queues Sidekiq + sessions
prometheus	9090	Métriques internes
node-exporter	9100	Métriques OS
gitlab-exporter	9168	Métriques métier (jobs CI, pipelines)
postgres-exporter	9187	Métriques PG
redis-exporter	9121	Métriques Redis

[TP associé]

TP03 — Inspecter en live l'architecture sur la VM via `gitlab-ctl status`.

Refcard — les 4 scripts wrappers du lab

Script	Option	Effet
`gitlab-start.sh`	(aucune)	boot VM QEMU + tunnel SSH + provisioning Ansible si marker absent
	`--reset`	supprime disque + provisioning FROM SCRATCH (∼20 min)
	`--update`	force la ré-exécution du playbook
	`--restart`	arrêt + redémarrage VM sans toucher au disque
`gitlab-status.sh`	(aucune)	8 checks complets (QEMU, cloud-init, SSH, VM, Python, Ansible, GitLab)
	`--quick`	version rapide (SSH + Ansible uniquement)
`gitlab-ssh.sh`	(aucune)	session SSH interactive root@127.0.0.1:2222
`gitlab-stop.sh`	(aucune)	arrêt propre VM + tunnel SSH

```
$ ./gitlab-start.sh            # première fois ou après reboot
$ ./gitlab-status.sh           # vérifier l'état
$ ./gitlab-ssh.sh              # entrer dans la VM
$ ./gitlab-start.sh --reset    # repartir from scratch
$ ./gitlab-stop.sh             # éteindre proprement
```

[TP associé]

TP04 — Maîtriser ces 4 scripts (modes `--reset`, `--update`, `--restart`, `--quick`).

Sizing : GitLab seul (sans CI lourde)

Profil	vCPU	RAM	Disque	Cas d'usage
Lab / démo	2	4 Go	20 Go	Découverte, formations
Petite équipe	4	8 Go	50 Go	1–20 users, $\leq$100 projets
Équipe moyenne	8	16 Go	200 Go	20–100 users, $\leq$1000 projets
Grosse PME	16	32 Go	1 To SSD	100–500 users, mirroring, registry
Grande organisation	32+	64+ Go	5+ To NVMe	500+ users, HA (3 Gitaly + praefect)

- Disque : **SSD obligatoire** dès 50+ utilisateurs (random IO Gitaly).
- PostgreSQL externe recommandé dès 50+ utilisateurs.
- Redis externe recommandé dès 200+ utilisateurs.
- Object Storage S3/MinIO recommandé pour artifacts/uploads/LFS dès 100 utilisateurs.

Sizing : runners CI/CD

Profil runner	Resources	Notes
Léger (lint, tests)	2 vCPU, 4 Go RAM	Executor Docker, image `alpine`
Standard	4 vCPU, 8 Go RAM	Executor Docker, build/test/SAST
Build lourd	8 vCPU, 16+ Go RAM, SSD	Compilations C++/Rust, builds Docker multi-stage
GPU (ML)	8 vCPU, 32 Go RAM, GPU	Training, inference, executor docker–nvidia

⋆ À retenir

Règle simple : **N runners parallèles = max(concurrent jobs)**. Calculer le quota concurrent dans `config.toml` en fonction de la RAM / CPU réelle de l'hôte.

Vue d'ensemble : 5 façons d'installer GitLab

Méthode	✓ Avantages	× Inconvénients	Pour qui ?
Omnibus (apt/dnf)	Simple, supporté officiel	Mono-host, MAJ apt nécessaire	PME, prod simple
Docker (single)	Portable, isolation, snapshots	Pas pour HA, volumes critiques	Lab, démo, dev
Docker Compose	Multi-services intégrés	Toujours mono-host	Petite équipe
Helm chart Kubernetes	HA natif, scale horizontal	Complexité K8s	Grande org
Ansible / OpenTofu	Reproductible, IaC	Demande une orchestration claire	Plateforme as code

Astuces d'installation (Omnibus)

```
# Debian / Ubuntu
curl -fsSL https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script.deb.sh |
    sudo bash
sudo EXTERNAL_URL="https://gitlab.example.com" apt install -y gitlab-ce

# Fedora / RHEL / Rocky
curl -fsSL https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script.rpm.sh |
    sudo bash
sudo EXTERNAL_URL="https://gitlab.example.com" dnf install -y gitlab-ce
```

- **NE PAS** oublier `EXTERNAL_URL` au moment de l'install (sinon URL=hostname local).
- Certificats : Let's Encrypt automatique si DNS public + port 80 ouvert.
- Mot de passe initial : `/etc/gitlab/initial_root_password` (effacé sous 24h).
- Toute modif de `gitlab.rb` → `sudo gitlab-ctl reconfigure`.

[TP associé]

TP05 — Installation Docker complète + premier login + reconfigure.

Provisioning Ansible (le lab)

- Playbook `provisionning/playbooks/site.yml`
- 4 rôles :
  - `common` : apt, timezone, /etc/hosts
  - `docker` : Docker Engine + Compose
  - `gitlab` : Omnibus CE + `gitlab.rb` templaté
  - `gitlab_runner` : binaire + register 2 runners
- Marker idempotent : `/etc/ansible_provisioned`
- Vault chiffré : `group_vars/vault.yml`

Commande type

```
cd provisionning/
ansible-playbook \
  -i inventory/hosts.yml \
  playbooks/site.yml \
  --ask-vault-pass
```

[TP associé]

TP06 — Lire le playbook + lancer en dry-run puis réel + vérifier idempotence.

Provisioning OpenTofu (IaC déclaratif)

- OpenTofu = fork open-source de Terraform (fondation Linux Foundation, 2024).
- Provider officiel : `gitlabhq/gitlab` (v17+).
- Idéal pour : créer/maintenir users, groupes, projets, deploy keys, runners, variables CI **à partir d'un code git**.
- Backend distant possible côté GitLab : *Terraform state* en natif.

```
provider "gitlab" {
  base_url = "http://localhost:8080/api/v4"
  token    = var.gitlab_token
}

resource "gitlab_group"   "acme"   { name = "ACME" path = "acme" visibility = "internal" }
resource "gitlab_project" "lab"    { name = "lab-app" namespace_id = gitlab_group.acme.id }
resource "gitlab_user"    "alice"  { name = "Alice" username = "alice" email = "a@example.com" password = var.alice_pw }
```

[TP associé]

TP07 — Provisionner groupe + projet + user via OpenTofu, backend state distant GitLab.

Vue d'ensemble de l'Admin Area

L'Admin Area est accessible uniquement aux comptes ayant l'**Access level Administrator**. Bouton « clé/écusson » dans la barre du haut, ou directement via `/admin`.

Section	Contenu
Overview	Compteurs (users, projets, groupes), version GitLab, ressources
Overview → Users	Liste + création + edit + impersonate + delete
Overview → Groups	Liste tous les groupes (transferts admin, suppression forcée)
Overview → Projects	Tous projets de l'instance, archivage masse
Overview → Runners	Runners shared + group + project (registration token instance)
Overview → Jobs	Pipeline jobs sur toute l'instance
Monitoring	Background jobs, health check, audit events, requests profiles
Messages	Broadcast messages (bandeau à tous les users)
System Hooks	Webhooks instance-wide (audit, sync externe)
Settings → General	Sign-up, sign-in restrictions, terms of service, default project visibility
Settings → Repository	Storage path, default branch, commit signing
Settings → CI/CD	Shared runner tags, registration tokens, container registry
Settings → Network	Outbound restrictions, IP rate limiting
Settings → Reporting	Spam, abuse
Settings → Preferences	Email server, MR diff limits

Admin Area → Overview : tableau de bord avec compteurs (users, projets, runners actifs) et version GitLab installée.

Création d'un utilisateur (workflow complet)

Chemin : Admin Area → Users → bouton `New user` en haut à droite.

Champ	Notes
Name	Nom affiché (modifiable par le user après)
Username	Identifiant unique *immuable* (slug URL, ne sera pas réutilisable même après suppression)
Email	Doit être unique. Si SMTP configuré → mail de bienvenue
Access level	`Regular` (défaut) · `Admin` (DSI/SRE) · `Auditor` (read-only)
Validate user account	[x] par défaut sur GitLab 16+ (sinon état `pending approval`)
External	[x] pour les sous-traitants (ne peut voir que les projets explicitement partagés)
Projects limit	Quota de projets personnels créables (défaut 100 000)
Note (interne)	Visible des admins uniquement

Après création :

- `Edit` → `Reset password` : génère un mot de passe temporaire, expire à 24h.
- Lien direct pour réinitialisation envoyé par email (si SMTP).
- Bloquer : `Block` (réversible) · Désactiver : `Deactivate` (inactif 90j+).
- Supprimer : `Delete` (avec/sans contributions). [!] irréversible.

Admin Area → Users → New user : formulaire avec les champs Name / Username / Email / Access level / External / Projects limit.

Personal Access Tokens (PAT) et Project/Group tokens

Personal Access Tokens (par utilisateur)

Profile → Preferences → Access Tokens.
Scopes typiques : `api` (tout l'API REST), `read_api`, `read_user`, `read_repository`, `write_repository`, `read_registry`, `write_registry`, `sudo` (admin only), `ai_features`, `k8s_proxy`. Préfixe `glpat-` dans le token. Expiration max 1 an (configurable).

Project / Group Access Tokens

Settings → Access Tokens du projet/groupe.
Crée un *bot user* associé. Idéal pour CI vers un autre projet, sans dépendre d'un user humain. Préfixe `glpat-` également.

Deploy Tokens / Deploy Keys

Settings → Repository → Deploy tokens (HTTP) ou Deploy keys (SSH). Tokens read-only ou read+write sur registry/packages/repo, sans user attaché. Idéal pour pipelines externes ou déploiement en prod.

Profile → Access Tokens : formulaire de création (Token name, Expires at, Scopes).

Création d'un groupe et de sous-groupes

Chemin : icône GitLab en haut → Groups → `New group`.

Champ	Notes
Group name	Nom affiché
Group URL slug	`acme` dans `gitlab.example.com/acme` (URL immuable après création)
Visibility level	Private (membres seulement) / Internal (users connectés) / Public (web entier)
Role	Rôle implicite des futurs membres (Developer recommandé)

Sous-groupes : depuis le groupe, bouton `New subgroup`. Hiérarchie illimitée :

`acme/backend/payments/api-v2`

Permissions cascadantes : un user Maintainer du parent l'est sur tous les descendants. Un user Developer d'un sous-groupe n'a aucun droit sur le parent.

Templates de groupe (Settings → General → Custom project templates) : permet de standardiser la création de projets dans le groupe à partir d'un projet modèle hébergé dans un sous-groupe `templates/`.

Groups → New group : formulaire (name, URL slug, visibility, parent group si sous-groupe).

Inviter des membres dans un groupe ou projet

Chemin : Group/Project → Manage → Members → `Invite members`.

Champ	Notes
GitLab member or Email	Multi-sélection (autocomplete sur users connus, sinon email externe)
Select a role	Guest / Reporter / Planner / Developer / Maintainer / Owner
Access expiration	Date d'expiration de l'invitation (utile pour stagiaires, prestataires)

Invitation par email externe : le destinataire reçoit un lien d'inscription. Pratique pour onboarder un prestataire sans compte existant. L'invitation crée un *pending member* jusqu'à acceptation.

Inviter un groupe entier (Share with group) : tous les membres du groupe X reçoivent le rôle dans le projet/groupe Y. Idéal pour partager un repo `infra/` avec le groupe `secops/`.

2FA enforcement : Admin → Settings → General → Sign-in restrictions → [x] *Two-factor authentication required for all users*.

Members → Invite members : champ d'autocomplete + sélecteur de rôle + date d'expiration optionnelle.

Configuration SMTP (email transactionnel)

Indispensable pour : invitations, reset password, notifications MR/issues, mentions @user, daily digest. Sans SMTP, GitLab fonctionne mais les emails ne partent pas.

```
Puis : sudo gitlab-ctl reconfigure Test : sudo gitlab-rails console > Notify.test_email(
'vous@example.com','sujet','corps').deliver_now
```

[!] Pour Microsoft 365 / Google Workspace : utiliser App Passwords (2FA bloque l'authentification basique).

Backup et restauration

Chaque instance prod doit avoir un plan de backup testé.

```
3 Backup partiel sudo gitlab-backup create SKIP=registry,artifacts,uploads,builds,lfs,packages
4 Restauration sudo gitlab-ctl stop puma sidekiq sudo gitlab-backup restore
        BACKUP=1737800000_2026_05_24_18.0.0 sudo gitlab-ctl restart sudo gitlab-rake gitlab:check SANITIZE=true
```

Ce qui n'est PAS dans le backup

- `/etc/gitlab/gitlab.rb` — configuration
- `/etc/gitlab/gitlab-secrets.json` — secrets (clés de chiffrement, 2FA tokens)

Ces 2 fichiers doivent être sauvegardés *séparément* et copiés sur la machine de restauration.

[Aller plus loin]

Automatiser via cron + offload S3 : `gitlab_rails['backup_upload_connection']`.

Audit Events et Compliance

Chemin : Admin → Monitoring → Audit Events.
(Niveau projet/groupe : Settings → Audit Events.)

Événements tracés automatiquement :

- Login / logout / 2FA enable/disable
- Création / suppression / transfert de projet ou groupe
- Modification du rôle d'un membre
- Changement de visibilité (Private → Public !)
- Push sur branche protégée, force-push
- Modification de Settings sensibles (CI/CD, integrations, hooks)
- Création / révocation de PAT / Deploy tokens

Export : CSV téléchargeable, ou push vers SIEM via *Streaming Audit Events* (Premium/Ultimate, webhook + signature HMAC).

⋆ À retenir

Obligatoire pour SOC2, ISO27001, PCI-DSS. Activer le streaming dès le déploiement, rétention minimum 1 an, immutable storage côté SIEM.

Matrice des rôles GitLab (1/2)

Action	Guest	Reporter	Developer	Maintainer	Owner
Voir le projet	✓	✓	✓	✓	✓
Créer issue	✓	✓	✓	✓	✓
Voir code source		✓	✓	✓	✓
Voir MR, pipelines, jobs		✓	✓	✓	✓
Cloner / pull		✓	✓	✓	✓
Push branche non-protégée			✓	✓	✓
Push branche protégée				✓	✓
Force-push				✓	✓
Créer MR			✓	✓	✓
Approuver MR (si éligible)			✓	✓	✓
Merger MR			∼	✓	✓
Modifier `.gitlab-ci.yml`			✓	✓	✓
Annuler / retry un job CI			✓	✓	✓

Légende : ✓ autorisé · ∼ selon config (allowed_to_merge) · vide = refusé.

Matrice des rôles GitLab (2/2)

Action	Guest	Reporter	Developer	Maintainer	Owner
Gérer runners projet				✓	✓
Gérer variables CI				✓	✓
Configurer Settings projet				✓	✓
Configurer branches protégées				✓	✓
Configurer push rules				✓	✓
Ajouter / retirer un membre				✓	✓
Modifier rôle d'un membre				∼	✓
Archiver le projet					✓
Transférer le projet (autre namespace)					✓
Supprimer le projet					✓
Modifier visibilité / nom du groupe					✓
Supprimer le groupe					✓

∼ Maintainer peut modifier le rôle des membres jusqu'au niveau Maintainer max.

Description des rôles — résumé synthétique

Guest	*Visiteur* — accès uniquement aux issues et au wiki public. Aucun accès au code. Utilisé pour les parties prenantes métier (PO, support client).
Reporter	*Lecteur* — accès lecture au code, MR, pipelines. Peut télécharger artefacts. Pas de modification. Utilisé pour QA, auditeurs internes.
Planner	*Gestion ticketing* — nouveau rôle (intermédiaire Guest/Reporter) focalisé sur issues, epics, milestones, labels. Sans accès code. Idéal Product/PM.
Developer	*Contributeur* — push sur branches non protégées, création de MR, approbations, modif CI. Rôle standard pour la majorité d'une équipe dev.
Maintainer	*Lead technique* — approbations + merge sur branches protégées, gestion runners, variables CI, settings projet, members. Tech Lead, DevOps Engineer.
Owner	*Propriétaire* — tout droit sur le groupe : visibilité, suppression, transfert, gestion du rôle des autres. Engineering Manager, Director.
Admin	*Instance-level* — droits totaux *cross-tenant* sur toute l'instance (impersonate, instance settings, broadcast, runners shared). DSI, SRE.

[TP associé]

TP08 — Mettre la matrice en pratique : alice=Developer, bob=Maintainer, charlie=Owner.

Authentification : socles disponibles dans GitLab

Méthode	Notes
Locale (DB GitLab)	Toujours active. Utilisée pour 'root' et users non-SSO
LDAP	Active Directory, OpenLDAP. Sync automatique groups → groupes GitLab
OmniAuth :	30+ providers (OIDC, SAML, GitHub, Google, Microsoft, GitLab.com. . .)
OpenID Connect	**Recommandé** (moderne, signed JWT). Keycloak, Authentik, Auth0, Okta
SAML 2.0	Entreprise legacy. ADFS, Shibboleth
Kerberos	Single Sign-On Windows / GSSAPI
SCIM	Provisioning automatique users + groupes (Premium/Ultimate)
Smart Card / X.509	PKI client TLS (cas industriels durcis)

Configuration Keycloak (OIDC) — extrait gitlab.rb

```
gitlab_rails['omniauth_enabled'] = true
gitlab_rails['omniauth_allow_single_sign_on'] = ['openid_connect']
gitlab_rails['omniauth_block_auto_created_users'] = false
gitlab_rails['omniauth_auto_link_user'] = ['openid_connect']
gitlab_rails['omniauth_providers'] = [
  {
    name: 'openid_connect',
    label: 'Keycloak',
    args: {
      name: 'openid_connect',
      scope: ['openid','profile','email'],
      response_type: 'code',
      issuer: 'https://keycloak.example.com/realms/linuxmaine',
      discovery: true,
      uid_field: 'preferred_username',
      client_options: {
        identifier:   'gitlab',
        secret:       '<CLIENT_SECRET_DEPUIS_KEYCLOAK>',
        redirect_uri: 'https://gitlab.example.com/users/auth/openid_connect/callback'
      }
    }
  }
]
```

[TP associé]

TP09 — Démarrer Keycloak Docker + créer realm + client + login GitLab via OIDC.

Stratégie d'observabilité 3-piliers

Pilier	Outil typique	Ce qu'on regarde
Métriques	Prometheus + Grafana	CPU, RAM, queues Sidekiq, pipelines/min, GC Ruby
Logs	Loki + Promtail (ou Splunk, ELK)	nginx access, puma errors, sidekiq, gitaly
Traces	Jaeger / Tempo (OpenTracing)	traçage distribué (requêtes longues, slow MR)
Alerting	Alertmanager + Slack/Teams/PagerDuty	seuils sur métriques + matchers labels
Health checks	Zabbix / Nagios / Icinga	checks HTTP, port, certs, services

Sonde Zabbix — HTTP checks essentiels

Endpoint	Vérification
`GET /api/v4/version`	HTTP 200 + JSON {"version":"...", "revision":"..."}
`GET /-/health`	contient "GitLab OK"
`GET /-/readiness?all=1`	JSON {"status":"ok","master_check":[...]} sur PG, redis, gitaly
`GET /-/liveness`	contient "ok" (très léger, idéal pour load balancer)
`GET /-/metrics`	métriques Prometheus brutes (auth header requis)
`GET /api/v4/runners/all`	liste runners (auth admin) — détecte runners offline

Item Zabbix — exemple version GitLab

Type : `HTTP agent` · URL : `{HOST.URL}/api/v4/version`
Request method : GET · Response type : Text · Update interval : 1m
Preprocessing : *JSONPath* $.version \rightarrow$ `String`
`Trigger : {last(/GitLab/version.string)}=""` $\rightarrow$ severity *High*

Zabbix UI → Items : capture du formulaire de configuration de l'item HTTP agent sur /api/v4/version.

Stack Prometheus + Grafana + Loki (Docker)

Composant	Port	Rôle
Prometheus	9091	Scrape les exporters de la VM GitLab (toutes 15s)
Grafana	3000	Dashboards (admin / linuxmaine en lab)
Loki	3100	Indexation des logs (étiquettes)
Promtail	–	Tail `/var/log/gitlab/` → push Loki
Alertmanager	9093	Routage des alertes (Slack, email, PagerDuty)

Targets Prometheus sur la VM GitLab :

- :9100 *node-exporter* — CPU, RAM, disque, réseau
- :9168 *gitlab-exporter* — pipelines, MR, queues Sidekiq
- :9187 *postgres-exporter* — requêtes PG, locks
- :9121 *redis-exporter* — mémoire, commandes/s, hit ratio
- :9090 *prometheus interne* — WAL, scrape duration
- :9252 *gitlab-runner* — jobs, durée, file d'attente

Grafana → Dashboards → Import ID 9628 (GitLab Omnibus) : tuiles CPU/RAM, Sidekiq queue size, pipelines/min, HTTP latency.

Alertes Prometheus + Alertmanager

```yaml
# alerts.yml
groups:
- name: gitlab
  rules:
  - alert: GitLabDown
    expr: probe_success{instance="http://localhost:8080/-/liveness"} == 0
    for: 2m
    labels: { severity: critical }
    annotations:
      summary: "GitLab non joignable depuis 2 min"
      runbook:  "https://wiki.example.com/runbooks/gitlab-down"

  - alert: SidekiqQueueGrowing
    expr: sidekiq_queue_size{queue="default"} > 1000
    for: 10m
    labels: { severity: warning }
    annotations:
      summary: "Queue Sidekiq default > 1000 jobs depuis 10 min"

  - alert: RunnerStarvation
    expr: sum(gitlab_runner_jobs{state="pending"}) > 5
    for: 5m
    labels: { severity: warning }
```

⋆ À retenir

Toute alerte doit avoir un *runbook* (URL wiki) qui décrit diagnostic + remédiation. Sans runbook, l'oncall réinvente la roue à 3h du matin.

[TP associé]

Niveaux et workflow d'un runner

Niveau	Visibilité	Cas d'usage
Instance	Tous les projets de l'instance	Build léger partagé, contrôle DSI
Group	Groupe + tous descendants	Build dédié à une BU / un produit
Project	1 seul projet	Runner sensible (deploy prod), GPU dédié

Workflow d'enregistrement (GitLab 16+) :

1. Admin/Group/Project → CI/CD → Runners → `New instance/group/project runner`
2. Choisir Linux/Mac/Windows + Tags + [x] Run untagged + Lock to current project
3. GitLab génère un **token éphémère** (préfixe `glrt-...`) à utiliser *une fois* pour register
4. Sur la machine cible : `gitlab-runner register -token glrt-...`
5. Le binaire se connecte, s'enregistre, écrit `config.toml`

Admin → Runners → New instance runner : formulaire avec OS, Tags, Run untagged jobs, Lock to projects, Maximum job timeout.

Configuration runner — /etc/gitlab-runner/config.toml

```
concurrent     = 4                          # jobs en parallèle (global)
check_interval = 0                          # 0 = défaut (3s)
log_level      = "info"
log_format     = "json"                     # idéal pour Loki/Splunk

[session_server]
  session_timeout = 1800

[[runners]]
  name         = "docker-runner-1"
  url          = "http://localhost:8080"
  token        = "glrt-xxxxxxxxxxxxxxxxx"   # rotation possible
  executor     = "docker"
  shell        = "bash"
  output_limit = 4096                       # KB de logs/job (anti-runaway)

  [runners.docker]
    image           = "alpine:3.20"          # image par défaut si job ne précise pas
    privileged      = false                  #     true uniquement pour build images
    pull_policy     = ["if-not-present"]
    disable_cache   = false
    volumes         = ["/cache","/var/run/docker.sock:/var/run/docker.sock"]
    shm_size        = 0
    cpus            = "2.0"
    memory          = "4g"
    network_mode    = "bridge"

  [runners.cache]
    Type     = "s3"                          # cache distribué (sinon "local")
    Path     = "runner-cache"
    Shared   = true
    [runners.cache.s3]
      ServerAddress  = "minio.local:9000"
      AccessKey      = "MINIO_KEY"
      SecretKey      = "MINIO_SECRET"
```

Options de paramétrage avancées

Option	Effet	Quand l'utiliser
`executor`	shell / docker / docker-windows / ssh / kubernetes / instance	Docker = standard ; K8s pour HA/scale
`tags`	Étiquettes matchées par les jobs	Cibler hardware (gpu, arm64, build-heavy)
`run_untagged`	Accepter jobs sans tag	Runner shared default
`locked`	Verrouiller au projet courant	Runner dédié à un projet sensible
`access_level`	`not_protected` / `ref_protected`	Runners prod réservés branches protégées
`concurrent`	N jobs en parallèle	Ajuster selon RAM/CPU
`docker_privileged`	Conteneur privilégié	Build d'images (kaniko préférable)
`pull_policy`	`always|if-not-present|never`	`always` pour reproductibilité
`output_limit`	KB de logs/job	Anti-runaway (boucle infinie de print)
`maximum_timeout`	Temps max d'un job (override projet)	Quota anti-runner-monopoly
`request_concurrency`	N jobs requêtés simultanément au master	Réduire le throttling sur petit GitLab

Maintenance d'un runner

Commande	Effet
`gitlab-runner status`	État systemd du service
`gitlab-runner list`	Runners enregistrés (lit `config.toml`)
`gitlab-runner verify`	Ping chaque runner vers son master GitLab
`gitlab-runner verify -delete`	Supprime les runners orphelins (master non joignable)
`gitlab-runner restart`	Redémarre le service
`gitlab-runner unregister -all-runners`	Désenregistre tout
`gitlab-runner exec docker test-job`	Exécuter un job CI localement (debug `.gitlab-ci.yml`)
`journalctl -u gitlab-runner -f`	Logs systemd live

Côté GitLab UI

Admin/Group/Project → CI/CD → Runners :

- [!] point vert/gris : online/offline (timeout > check_interval × 2)
- `Pause` : ne distribue plus de nouveaux jobs
- `Edit` : modifier description, tags, access_level, lock projects
- `Remove` : suppression définitive

Admin → Runners : liste avec points verts, tags, online/offline, jobs en cours, possibilité de pause/edit/remove par runner.

Partie 2 — Ce que vous savez maintenant

Acquis

- ✓ GitFlows (DevOps → AIOps) et leurs outils
- ✓ Architecture GitLab : 15+ services internes
- ✓ Refcard scripts du lab
- ✓ Sizing pour GitLab + runners
- ✓ 5 modes d'installation (Omnibus, Docker, Compose, K8s, Ansible/OpenTofu)
- ✓ Administration : users, groupes, 7 rôles (matrice complète)
- ✓ SSO Keycloak (OIDC) avec mapping de groupes
- ✓ Monitoring 3-piliers (métriques, logs, traces) + Zabbix
- ✓ Configuration des runners (executor, tags, niveaux)

Créer son premier repo : settings détaillés (branch protection, push rules, MR approvals, mirroring, webhooks).

3
Partie 3 — Mon premier repo

Partie 3 — Ce que nous allons voir

Au programme

1. Créer un nouveau projet
2. Settings détaillés :
   - General
   - Repository (branches protégées, push rules)
   - Merge requests
   - CI/CD
   - Integrations & Webhooks
   - Monitor, Analytics, Packages
3. Workflow Merge Request complet

À la fin vous saurez

- Standardiser un repo (compliance)
- Configurer la review et le merge
- Mettre en place codeowners
- Déclencher un build externe via webhook
- Choisir entre merge / squash / rebase

Créer un projet : 4 voies

Méthode	Cas d'usage
Create blank project	Démarrer de zéro, init local possible
Create from template	Template officiel (Rails, Express, Spring, Hugo...) ou template d'entreprise
Import project	Depuis GitHub, Bitbucket, repo URL git, archive
Fork	Copier un projet existant tout en gardant le lien

Champs critiques au create

- **Project URL** : namespace (groupe ou user) + slug
- **Visibility** : Private (membres uniquement) · Internal (tous users connectés) · Public (web)
- **Initialize with README** : facilite le 1er clone
- **Default branch** : `main` (par défaut depuis 15.0)
- **Project deployment target** : K8s, ECS, Cloud Run, none

New Project form (icône + en haut)

Settings → General

Section	Quoi y mettre
Naming, topics, avatar	Description claire, topics pour la recherche
Visibility, project features, permissions	Activer/désactiver issues, MR, wiki, pages, CI, registry...
Badges	Build status, coverage, version (auto-injectés dans README)
Service Desk	Convertit emails entrants en issues
Advanced	Archive, transfert, suppression, export

Settings → Repository

Section	Détail
Default branch	Renommer si héritage ancien (`master` → `main`)
Protected branches	**Le plus important.** Définir qui peut push/merge sur main/develop/release/*
Protected tags	Empêcher la suppression / réécriture des tags de release
Push rules (EE)	Regex sur commit messages, branches, sign-off, max file size, fichiers interdits
Mirroring	Push/pull mirror vers un repo distant (sync GitHub, sauvegarde)
Deploy keys	Clés SSH read-only pour pipelines externes
Deploy tokens	Tokens HTTP pour container registry/packages
Repository graph	DAG visuel (utile review historique)

[TP associé]

TP12 — Créer `acme/backend/lab-app` + protéger main + push rules.

Mirroring d'un repo externe — vue d'ensemble

Chemin : Project → Settings → Repository → *Mirroring repositories*.

Direction	Sens	Édition
Push	GitLab → repo distant	Disponible en **Free** (CE incluse)
Pull	repo distant → GitLab	**Premium** (Ultimate inclus)

Cas d'usage typiques :

- Sauvegarde croisée vers GitHub/Bitbucket/un autre GitLab.
- Présence open source sur GitHub + dev privé interne sur self-hosted GitLab.
- Sync multi-cloud (HA région) ou multi-datacenter.
- Migration progressive d'une plateforme à une autre.
- Mirror miroir d'un repo upstream public (forks, kernels, distros).

Fréquence de sync :

- **Push** : déclenchée automatiquement à chaque push GitLab + retry toutes les 5 min en cas d'échec.
- **Pull** : toutes les 30 min par défaut (1 min minimum en Premium).
- Trigger manuel : bouton *Update now* (icône).

Settings → Repository → Mirroring repositories : section avec champ Git repository URL + Mirror direction (Push/Pull) + Authentication method + bouton Mirror repository.

Push mirror : GitLab → GitHub (cas le plus courant)

Pré-requis côté GitHub :

1. Créer un repo destination (vide ou existant) sur `github.com`.
2. GitHub → Settings → Developer settings → Personal access tokens → *Fine-grained* : générer un PAT avec scope `Contents: read+write` sur le repo cible.
   Format : `github_pat_xxxxxxxxxxxxxxxx`

Configuration côté GitLab : Settings → Repository → Mirroring repositories.

Champ	Valeur
Git repository URL	`https://<USERNAME>:<GITHUB_PAT>@github.com/<owner>/<repo>.git`
Mirror direction	`Push`
Authentication method	`None` (déjà dans l'URL) — ou `Password` avec PAT séparé
Mirror only protected branches	[x] recommandé (évite de leaker des branches WIP)
Keep divergent refs	[ ] par défaut (force-push si divergence détectée côté GitHub)
Mirror branches matching regex	ex. `^(main|release/.+)$` pour cibler

Test : *Update now* (icône cercle) → vérifier statut « Last successful update : il y a quelques secondes ».
En cas d'échec, l'erreur s'affiche en survol (auth refusée, repo absent, divergence).

Mirroring repositories : ligne avec URL https://...@github.com/..., statut Last update + icône Update now + bouton crayon Edit + corbeille Delete.

Push mirror : GitLab → GitLab (instance externe)

Cas typique : miroir GitLab self-hosted vers `gitlab.com`, ou entre deux instances internes.

Auth recommandée : Deploy Token côté destination

1. Sur le GitLab **destination** : projet cible → Settings → Repository → Deploy tokens.
2. Créer un token avec scope `write_repository` (+ `read_repository`).
3. Noter le `username` et le `token` générés (token non révélé ensuite !).

Sur le GitLab source : Settings → Repository → Mirroring repositories.

Champ	Valeur
Git repository URL	`https://gitlab-dest.example.com/<groupe>/<projet>.git`
Mirror direction	`Push`
Authentication method	`Username and password`
Username	celui du Deploy Token (ex. `gitlab+deploy-token-12`)
Password	le Deploy Token

Alternative : SSH avec clé déployée

- URL : `git@gitlab-dest.example.com:<groupe>/<projet>.git`
- Auth : SSH public key (GitLab génère la paire, push la clé publique côté destination Deploy Keys).
- Plus robuste pour des sync fréquents et long-terme (pas de rotation PAT à gérer).

⋆ À retenir

Ne JAMAIS utiliser le mot de passe d'un user humain dans l'URL/auth : rotation impossible, fuite de logs, MFA bloquant. Toujours un PAT, Deploy Token, ou SSH key dédiée.

Pull mirror : repo externe → GitLab (Premium)

Cas typique : héberger un mirroir lecture-seule d'un repo amont (kernel, distribution, framework) pour servir de cache local, ou pour scanner avec les outils de sécurité internes.

Configuration (Settings → Repository → Mirroring repositories) :

Champ	Valeur
Git repository URL	URL HTTPS ou SSH du repo source
Mirror direction	`Pull`
Authentication method	None (public), Password (PAT), SSH
Trigger pipelines for mirror updates	[x] pour lancer la CI à chaque sync (attention au quota minutes !)
Only mirror protected branches	[x] pour limiter aux branches utiles
Override divergent refs	[x] : force le reset si divergence locale

Limitations :

- Le repo GitLab devient *read-only* sur les branches mirroirisées (rejets de push direct).
- Les MR ne sont pas synchronisées (seulement le code git).
- Issues / wikis / settings ne sont pas mirrorés.
- Pour synchroniser bidirectionnel + MR/issues : voir `gitlab-merge-request-importer` ou `github-mirror`.

[Aller plus loin]

Pour synchroniser *tous* les repos d'une organisation GitHub vers GitLab, utiliser l'outil officiel `Import → GitHub` (API) ou le script `api/python3/gitlab-miror-all-repos/` (lab).

Diagnostiquer un mirroring qui échoue

Côté UI : Settings → Repository → Mirroring repositories : hover sur l'icône d'état → message d'erreur (auth, network, divergence).
Côté API :

```
# Lister les mirrors d'un projet
curl -sH "PRIVATE-TOKEN: $TOKEN" \
  $GITLAB_URL/api/v4/projects/$PROJECT_ID/remote_mirrors | jq .

# Forcer une sync immédiate (push mirror)
curl -X POST -H "PRIVATE-TOKEN: $TOKEN" \
  $GITLAB_URL/api/v4/projects/$PROJECT_ID/remote_mirrors/$MIRROR_ID/sync

# Pour pull mirror : déclencher import refresh
curl -X POST -H "PRIVATE-TOKEN: $TOKEN" \
  $GITLAB_URL/api/v4/projects/$PROJECT_ID/mirror/pull
```

Erreurs courantes :

- `13:Repository or revision not found` : URL ou auth invalide.
- `remote rejected (pre-receive hook declined)` : push rules côté destination bloquent (regex commit, taille).
- `non-fast-forward` : divergence ; activer *Override divergent refs*.
- `Permission denied (publickey)` : clé SSH non déployée côté destination.
- `Connection timed out` : firewall / proxy sortant.

[!] Côté VM GitLab : `tail -f /var/log/gitlab/sidekiq/current | grep -i mirror`

Settings → Merge requests

Option	Recommandation
Merge method	`Merge commit with semi-linear history` (équilibré) ou `Fast-forward`
Squash commits	`Encourage` (à activer par défaut dans la MR)
Merge options	[x] Pipeline must succeed · [x] All discussions resolved
Merge suggestions	[x] Suggested reviewers (basé sur les codeowners)
Approval rules	Définir N approbations obligatoires par sujet (sécurité, qualité…)
Merge request templates	`.gitlab/merge_request_templates/*.md`

Settings → CI/CD

Section	Quoi configurer
General pipelines	Auto-cancel, skip outdated, public/private pipelines, timeout
Variables	Secrets : masked + protected + file type (cf. Partie 4)
Runners	Activer shared, instance, project, group ; tags
Pipeline triggers	Tokens pour déclencher la CI depuis un script ou outil externe
Deploy freezes	Bloquer les deploys sur des plages horaires
Auto DevOps	Pipeline magique en 1 clic (utile pour démos)
Secret detection / SAST	Bouton ON/OFF des templates

Settings → Integrations et Webhooks

Élément	**Cas d'usage**
Webhooks	POST HTTP vers URL externe à chaque push/MR/pipeline/issue
Integrations	50+ intégrations : Jira, Slack, Microsoft Teams, Mattermost, Discord, Prometheus, Jenkins, Datadog, Pivotal, YouTrack...
Mattermost slash commands	`/gitlab issue create...` depuis le chat
Slack app	Notifs + slash commands
Kubernetes agent (kas)	GitOps pull-based pour cluster K8s

Webhooks — déclencher un service externe

Chemin : Settings → Webhooks → *Add new webhook*.

Champ	Notes
URL	POST HTTP(S) vers l'endpoint externe
Secret token	Envoyé en header `X-Gitlab-Token` (à valider côté receveur)
Trigger events	Push, Tag push, Merge request, Issue, Note, Pipeline, Job, Deployment, Releases, Feature flags, Confidential issue/note, Subgroup events
SSL verification	[x] recommandé (sauf en dev local)
Per-branch filter	Regex sur ref name pour limiter à certaines branches

Validation côté serveur (extrait Express.js) :

```
app.post('/webhook', (req, res) => {
  const token = req.headers['x-gitlab-token'];
  if (token !== process.env.GITLAB_WEBHOOK_SECRET) return res.sendStatus(401);
  const event = req.headers['x-gitlab-event'];   // "Push Hook", "Merge Request Hook", ...
  console.log(`Event: ${event}`, req.body);
  res.sendStatus(200);
});
```

Outils de debug :

- *Test* (bouton à droite de chaque webhook) : envoie un sample event.
- *Recent events* : 20 derniers envois avec statut HTTP + body de la réponse.
- Inspecter en local : `ngrok http 3000` pour exposer un dev serveur.

Settings → Webhooks : formulaire + table des hooks actifs +

Settings → Analytics, Monitor, Operate

height**Section**	**Contenu**
Analytics → Value Stream	Lead time, MR throughput, deployment frequency (DORA metrics)
Analytics → Repository	Commits par contributeur, lignes par auteur, langages détectés
Analytics → CI/CD	Durée pipelines / jobs, success rate, queue time
Analytics → Code Review	Délai moyen revue, MR ouvertes par auteur
Monitor → Incidents	Suivi d'incident lié à une alerte ou un déploiement
Monitor → Alerts	Alertes Prometheus / OpsGenie / Datadog intégrées
Monitor → Error Tracking	Intégration Sentry-compatible
Operate → Environments	Liste des environnements (staging, prod, review apps)
Operate → Feature flags	Toggles déployables sans nouveau release
Operate → Kubernetes clusters	Liaison via GitLab Agent (KAS) pour GitOps
Operate → Terraform states	Backend HTTP intégré pour state files
Plan → Iterations / Roadmap	Cadencement agile (Premium pour boards multi-projets)

Workflow MR end-to-end

1. `git switch -c feature/foo` + commits sur la branche
2. `git push -u origin feature/foo` → GitLab affiche un lien direct *Create MR*
3. Remplir titre (Conventional Commits), description (template auto-inséré), assignee, reviewer, labels, milestone
4. Pipeline CI démarre sur la MR (badge en haut)
5. Reviewer fait des commentaires inline, des *suggestions* appliquables en 1 clic
6. Auteur itère (commits supplémentaires sur la branche → MR mise à jour)
7. Threads *Resolved*, approbations atteintes, pipeline ✓
8. Merger (squash + delete source branch en 1 bouton)

[TP associé]

TP13 — Workflow MR complet à 2 personas : alice (auteur) et bob (reviewer).

bFichier .gitlab/CODEOWNERS à la racine ou dans docs/ :

```
Owners par d
éfaut * @team/leads
Backend Python src/api/**/*.py @alice @team/backend requirements.txt @alice @team/backend
Frontend Vue.js frontend/ @bob @team/frontend
Pipelines CI/CD : Maintainer obligatoire .gitlab-ci.yml @charlie @team/devops .gitlab/ @charlie
Sécurité :  2 approbateurs (groupe SecOps) [Security] @team/secops src/auth/ @team/secops src/crypto/ @team/secops
```

bFichier .gitlab/CODEOWNERS à la racine ou dans docs/ :


```
Owners par d
éfaut * @team/leads
Backend Python src/api/**/*.py @alice @team/backend requirements.txt @alice @team/backend
Frontend Vue.js frontend/ @bob @team/frontend
Pipelines CI/CD : Maintainer obligatoire .gitlab-ci.yml @charlie @team/devops .gitlab/ @charlie
Sécurité :  2 approbateurs (groupe SecOps) [Security] @team/secops src/auth/ @team/secops src/crypto/ @team/secops
```

bFichier `.gitlab/CODEOWNERS` à la racine ou dans `docs/` :

```
Owners par d
éfaut * @team/leads
Backend Python src/api/**/*.py @alice @team/backend requirements.txt @alice @team/backend
Frontend Vue.js frontend/ @bob @team/frontend
Pipelines CI/CD : Maintainer obligatoire .gitlab-ci.yml @charlie @team/devops .gitlab/ @charlie
Sécurité :  2 approbateurs (groupe SecOps) [Security] @team/secops src/auth/ @team/secops src/crypto/ @team/secops
```

bFichier `.gitlab/CODEOWNERS` à la racine ou dans `docs/` :

```
Owners par d
éfaut * @team/leads
Backend Python src/api/**/*.py @alice @team/backend requirements.txt @alice @team/backend
Frontend Vue.js frontend/ @bob @team/frontend
Pipelines CI/CD : Maintainer obligatoire .gitlab-ci.yml @charlie @team/devops .gitlab/ @charlie
Sécurité :  2 approbateurs (groupe SecOps) [Security] @team/secops src/auth/ @team/secops src/crypto/ @team/secops
```

⋆ À retenir

Les codeowners sont automatiquement ajoutés comme *eligible approvers* sur toute MR touchant les patterns. Combiné aux approval rules, c'est la base d'une review tracée (compliance SOC2, ISO27001).

Partie 3 — Ce que vous savez maintenant

Acquis

- ✓ Créer un projet (4 voies)
- ✓ Configurer General, Repository, MR, CI/CD, Integrations, Webhooks
- ✓ Protected branches + push rules + codeowners
- ✓ Workflow MR : push → review → approbations → merge

CI/CD : anatomie d'un `.gitlab-ci.yml`, stages, jobs, rules, needs, artifacts, cache, parent-child pipelines, et gestion des secrets (masked, protected, Vault).

4
Partie 4 — CI/CD

Partie 4 — Ce que nous allons voir

Au programme

1. Anatomie d'un `.gitlab-ci.yml`
2. Stages, jobs, dependencies (`needs`)
3. Artifacts & cache
4. `rules`, `when`, `only/except` (legacy)
5. `include`, `extends`, parent-child pipelines
6. Variables et secrets :
   - types (Variable / File)
   - flags (masked, protected, hidden)
   - HashiCorp Vault via JWT GitLab

À la fin vous saurez

- Écrire un pipeline build/test/deploy
- Optimiser avec cache + needs (DAG)
- Sécuriser un secret (4 niveaux)
- Récupérer un secret depuis Vault sans le stocker côté GitLab
- Déboguer un pipeline rouge

Structure de base

```yaml
stages: [build, test, deploy]

default:                        # appliqué à tous les jobs sauf override
  image: alpine:3.20
  tags: [docker]
  before_script:
    - apk add --no-cache curl
  retry:
    max: 2
    when: [runner_system_failure, stuck_or_timeout_failure]

variables:                      # variables globales (peuvent être override par job)
  GREETING: "hello"

build-app:
  stage: build
  script:
    - echo "$GREETING from build"
    - mkdir -p dist && echo "v1" > dist/version
  artifacts:
    paths: [dist/]
    expire_in: 1 day

test-unit:
  stage: test
  needs: [build-app]            # DAG : démarre dès build-app fini, sans attendre fin du stage
  script: cat dist/version
```

Cache : npm, pip, gradle, cargo...

```yaml
# Exemple Node.js      cache les node_modules invalidé par hash du lockfile
test-node:
  image: node:20-alpine
  cache:
    key:
      files: [package-lock.json]
    paths:
      - node_modules/
      - .npm/
  script:
    - npm ci --cache .npm --prefer-offline
    - npm test
```

Différence cache vs artifacts

- **cache** : performance (re-construit si manquant, partagé entre jobs/runs sur même runner)
- **artifacts** : output de build (toujours uploadé, téléchargeable depuis l'UI, expirable)

Rules : la machine à conditionner

```yaml
deploy-prod:
  stage: deploy
  script: ./deploy.sh
  rules:
    # ne PAS exécuter sur les MR
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      when: never
    # sur tag vX.Y.Z, déclenche manuellement
    - if: $CI_COMMIT_TAG =~ /^v[0-9]+\.[0-9]+\.[0-9]+$/
      when: manual
      allow_failure: false
    # sinon, ne pas créer le job
    - when: never
  environment:
    name: production
    url: https://app.example.com
```

⋆ À retenir

`rules` évalué *de haut en bas*, première règle qui matche gagne. Préférer `rules` à `only`/`except` (legacy).

Parent-child pipelines + include

```yaml
# Pipeline parent : déclenche un child pipeline pour chaque sous-projet
include:
  - template: 'Security/SAST.gitlab-ci.yml'           # template GitLab
  - project: 'acme/ci-templates'                       # cross-project
    ref:     'main'
    file:    '/templates/python.yml'
  - local:    '.gitlab/ci/deploy.yml'                  # local

trigger-frontend:
  trigger:
    include: frontend/.gitlab-ci.yml
    strategy: depend                                   # parent attend le child

trigger-backend-pipeline:
  trigger:
    project: acme/another-project
    branch: main
    strategy: depend
```

Progression pédagogique : du Hello World au déploiement multi-env

N°	Exemple	Concept introduit
1	Hello World	1 stage, 1 job, image alpine
2	Build + test (Python)	Plusieurs stages séquentiels, dépendances entre jobs
3	Build + test parallèle multi-langage	`parallel:matrix`, factorisation via `extends`
4	Cache et artifacts	`cache`, `artifacts`, dépendances entre stages
5	Conditional jobs avec `rules`	`rules:`, branches/tags, MR-only
6	DAG : pipelines accélérés via `needs`	Pipeline DAG, gains de temps
7	Environnements + déploiement	`environment:`, *review apps* dynamiques par MR
8	Blue/green + canary	Déploiement progressif, `manual`, `when:`

⋆ À retenir

Démarrer petit, ajouter UN concept à la fois. Ne jamais coller un pipeline de 500 lignes copié d'Internet : on ne le maîtrise pas le jour où ça casse.

Exemple 1 — Hello World

```yaml
# .gitlab-ci.yml minimal --- prouver que la CI est branchée
hello:
  image: alpine:3.20
  tags: [docker]
  script:
    - echo "Hello from runner $(hostname)"
    - cat /etc/os-release
```

- **1 seul job**, **1 seul stage** (`test` par défaut).
- Sans `stages:`, GitLab utilise `[.pre, build, test, deploy, .post]` par défaut.
- Sans `tags:`, le job ira sur un runner shared sans tag.

⋆ À retenir

Avant tout pipeline complexe, valider qu'un `echo` passe en vert.

Exemple 2 — Build + test Python séquentiel

```yaml
stages: [build, test]

default:
  image: python:3.12-slim
  tags: [docker]
  before_script: [pip install -q --no-cache-dir -r requirements.txt]

build-package:
  stage: build
  script:
    - python -m build          # produit dist/*.whl
  artifacts:
    paths: [dist/]
    expire_in: 1 day

test-unit:
  stage: test
  script:
    - pip install -q dist/*.whl
    - pytest -q --junitxml=junit.xml
  artifacts:
    reports:
      junit: junit.xml         # GitLab affiche les tests dans la MR
```

⋆ À retenir

`artifacts:reports:junit` fait apparaître les tests dans l'onglet *Tests* de la MR (vert/rouge/passed/failed/skipped).

Exemple 3 — Matrice parallèle multi-langage

```yaml
.test-template:
  stage: test
  script: [./run-tests.sh]

# 6 jobs en parallèle (3 versions Python  2 OS)
test:
  extends: .test-template
  image: python:$PY_VERSION-$BASE_OS
  parallel:
    matrix:
      - PY_VERSION: ["3.10", "3.11", "3.12"]
        BASE_OS:    [alpine, slim]

# Variante : 3 langages totalement différents
test-go:
  extends: .test-template
  image: golang:1.23-alpine
  script: [go test ./...]

test-rust:
  extends: .test-template
  image: rust:1.81-slim
  script: [cargo test]
```

⋆ À retenir

`extends` factorise. `parallel:matrix` génère N jobs à partir d'un produit cartésien — excellent pour le compat-matrix testing.

Exemple 4 — Cache + artifacts (build Node.js)

```yaml
stages: [install, build, test]

default:
  image: node:20-alpine
  tags: [docker]

# Cache des node_modules invalidé par hash du package-lock.json
.node-cache: &node-cache
  cache:
    key:
      files: [package-lock.json]
    paths: [node_modules/, .npm/]
    policy: pull-push           # défaut

install:
  stage: install
  <<: *node-cache
  script: [npm ci --cache .npm --prefer-offline]

build:
  stage: build
  needs: [install]
  cache:
    <<: *node-cache
    policy: pull               # ne réécrit pas le cache
  script: [npm run build]
  artifacts:
    paths: [dist/]
    expire_in: 1 week

test:
  stage: test
  needs: [install]
  cache: { <<: *node-cache, policy: pull }
  script: [npm test -- --coverage]
```

Cache vs artifacts — aide-mémoire

	cache	**artifacts**
Stockage	Sur le runner (ou objet S3 si configuré shared)	Côté GitLab (object storage)
Cycle de vie	Best-effort, peut disparaître	Garantis pour le pipeline
Partage	Entre runs sur même clé/runner	Toujours téléchargeable depuis l'UI
Idéal pour	node_modules, /.cargo, Maven .m2, pip wheels	dist/, binaries, rapports JUnit, SBOM
Invalidation	Changement de la clé (hash lockfile)	Expire-in (`1 day`, `1 week`, ...)
Performance	Pull rapide, accélère install	Téléchargé entre jobs (passage explicite)

⋆ À retenir

Règle simple : **cache** = ce qu'on *peut* reconstruire, **artifacts** = ce qu'on *produit* (livrable, rapport).

Exemple 5 — Conditional jobs avec rules

```yaml
# Run linters sur toute MR
lint:
  stage: test
  script: [npm run lint]
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"

# Build artefacts uniquement sur main et tags
build:
  stage: build
  script: [npm run build]
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
    - if: $CI_COMMIT_TAG

# Deploy prod uniquement sur tag SemVer, en manuel
deploy-prod:
  stage: deploy
  script: ./deploy.sh production
  environment: production
  rules:
    - if: $CI_COMMIT_TAG =~ /^v[0-9]+\.[0-9]+\.[0-9]+$/
      when: manual
      allow_failure: false
    - when: never
```

⋆ À retenir

`rules:` évalué de haut en bas, première règle qui match gagne. `when: never` en dernier = job non créé sauf condition explicite.

Exemple 6 — DAG via `needs` (pipeline accéléré)

```
stages: [install, build, test, package, deploy]

install:           { stage: install,  script: [npm ci] }
build-frontend:    { stage: build,    needs: [install], script: [npm run build:fe] }
build-backend:     { stage: build,    needs: [install], script: [npm run build:be] }
test-frontend:     { stage: test,     needs: [build-frontend], script: [npm run test:fe] }
test-backend:      { stage: test,     needs: [build-backend],  script: [npm run test:be] }
package-frontend:  { stage: package,  needs: [test-frontend, build-frontend], script: [./pkg-fe.sh] }
package-backend:   { stage: package,  needs: [test-backend,  build-backend],  script: [./pkg-be.sh] }
deploy-staging:    { stage: deploy,   needs: [package-frontend, package-backend], script: ./deploy.sh }
```

Sans `needs` (séquentiel par stage) : durée = max(install) + max(build) + max(test) + max(package) + max(deploy).
Avec `needs` (DAG) : chemin critique seulement, jobs indépendants en parallèle.

[Aller plus loin]

Visualiser le DAG : Pipelines → Pipeline → onglet *Show dependencies*. Gain typique sur un monorepo : 30 à 60% du temps total.

Exemple 7 — Environnements et review apps

```yaml
stages: [build, deploy]

build:
  stage: build
  image: docker:24
  services: [docker:24-dind]
  script:
    - docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA .
    - docker push    $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA

# Review app = 1 environnement par MR, URL dynamique
deploy-review:
  stage: deploy
  needs: [build]
  script:
    - kubectl apply -f k8s/review-${CI_COMMIT_REF_SLUG}.yaml
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    url:  https://$CI_COMMIT_REF_SLUG.review.example.com
    on_stop: stop-review
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"

stop-review:
  stage: deploy
  script: [kubectl delete namespace review-$CI_COMMIT_REF_SLUG]
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    action: stop
  when: manual
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
```

Exemple 8 — Blue/green + canary (déploiement progressif)

```yaml
stages: [build, deploy-canary, validate, deploy-prod, rollback]

build: { stage: build, script: ./build.sh }

# 1. Canary : déploie sur 10% du trafic
deploy-canary:
  stage: deploy-canary
  script: ./deploy.sh canary 10
  environment: { name: production, action: prepare }
  rules: [{ if: '$CI_COMMIT_BRANCH == "main"' }]

# 2. Validation manuelle (sanity check, alertes)
validate-canary:
  stage: validate
  script: ./smoke-tests.sh https://canary.example.com
  needs: [deploy-canary]
  allow_failure: false

# 3. Bascule complète vers le nouveau (blue/green swap)
deploy-prod:
  stage: deploy-prod
  needs: [validate-canary]
  script: ./deploy.sh blue-green-swap
  environment: production
  when: manual

# 4. Rollback en 1 clic si problème détecté en prod
rollback-prod:
  stage: rollback
  script: ./rollback.sh
  environment: production
  when: manual
  allow_failure: true
```

Composer ses pipelines : 4 patterns de réutilisation

Pattern	Outil GitLab	Cas d'usage
Templates intégrés	`include: template: 'X.gitlab-ci.yml'`	SAST, DAST, Dependency-Scanning fournis par GitLab
Includes *local*	`include: local: '.gitlab/ci/lint.yml'`	Découper un gros .gitlab-ci.yml en fichiers
Includes *project*	`include: project: file:`	Centraliser les pipelines d'une organisation
`extends:`	`extends: .base-job`	Factorisation de jobs similaires
Anchors YAML	`&anchor *anchor`	Réutilisation au sein d'un même fichier
Parent-child pipelines	`trigger: include:`	Monorepo : 1 sous-pipeline par sous-projet
Multi-project pipelines	`trigger: project: branch:`	Coordination de releases inter-projets

[TP associé]

TP18 — Mettre en place un multi-project pipeline (déclencher la CI d'un projet downstream).

[TP associé]

TP19 — Monorepo : parent + 3 enfants conditionnels (frontend/backend/infra) avec `rules:changes:`.

Cross-pipeline artifacts — 3 portées

Portée	Syntaxe	Cas d'usage
Même pipeline	`needs: [job-name]`	DAG classique (le plus courant)
Même projet, autre pipeline	`needs:project:ref:job:`	Réutiliser un build issu du dernier `main`
Autre projet, autre pipeline	`needs:project:ref:job:artifacts:true`	Récupérer un binaire d'un projet upstream
Job parent d'un child pipeline	`needs:pipeline:job:`	Le child accède aux artifacts du parent

⋆ À retenir

`needs:project:` (cross-project) ne télécharge **que les artifacts**, pas tout le repo. Le pipeline appelant n'a pas besoin de cloner le projet source. Auth via `CI_JOB_TOKEN` (scope cross-project à autoriser côté projet cible : Settings → CI/CD → Job token permissions).

Récupérer un artifact d'un autre projet

```yaml
# Projet B : pipeline qui consomme l'artefact du projet A
deploy-from-A:
  image: alpine:3.20
  stage: deploy
  needs:
    - project: acme/backend/lab-app
      job:     build-package
      ref:     main
      artifacts: true            # télécharge ce que build-package a produit
  script:
    - ls dist/                   # binaires venus du projet A
    - ./deploy-from-tarball.sh dist/*.tar.gz
```

Pré-requis (côté projet source A)

Settings → CI/CD → *Job token permissions* → `Limit access` → ajouter `acme/ops/deploy-system` (projet B) dans la *allowlist*.

[Aller plus loin]

Variante *tag* : `ref: 'v1.2.3'` pour récupérer l'artefact correspondant à une release stable, pas la dernière main mouvante.

Récupérer un artifact d'un pipeline antérieur (même projet)

```yaml
# Pipeline régulier : utilise le dernier build de main (économise du temps)
e2e-tests:
  stage: test
  needs:
    - project: $CI_PROJECT_PATH    # même projet
      job:     build-image
      ref:     main
      artifacts: true
  script:
    - docker run --rm -v "$PWD/dist:/app" registry/e2e:latest

# Si on veut le dernier artefact tag-relatif
deploy-staging:
  stage: deploy
  needs:
    - project: $CI_PROJECT_PATH
      job:     build-package
      ref:     latest-stable        # branche ou tag dédié
      artifacts: true
```

⋆ À retenir

Le job référencé doit avoir un `artifacts:paths:` qui n'est pas expiré. Pour conserver longtemps : `artifacts:expire_in: 1 year` ou *Keep* manuellement le pipeline depuis l'UI (icône pin).

Chaîne de pipelines en dépendance — pattern `strategy:depend`

Scénario : `lib` → `service-A` → `service-B` → `deploy-orchestrator`. Chaque maillon attend le précédent.

```yaml
# Projet 'lib' (.gitlab-ci.yml)
publish-lib:
  stage: publish
  script: [./publish-to-package-registry.sh]
  rules: [{ if: '$CI_COMMIT_TAG' }]

trigger-service-A:
  stage: trigger
  needs: [publish-lib]
  trigger:
    project: acme/services/service-A
    branch:  main
    strategy: depend           # attend la fin du downstream
  variables:
    UPSTREAM_LIB_VERSION: $CI_COMMIT_TAG
    UPSTREAM_PIPELINE_ID:  $CI_PIPELINE_ID
```

```yaml
# Projet 'service-A' : à la fin, déclenche service-B
trigger-service-B:
  stage: trigger
  trigger:
    project: acme/services/service-B
    branch:  main
    strategy: depend
  variables:
    UPSTREAM_SERVICE_A_IMAGE: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA
```

Orchestration centrale : pipeline « pilote »

Plutôt qu'une longue chaîne, un pipeline pilote orchestre N enfants en parallèle ou séquentiel et attend tout le monde :

```
# Pipeline pilote : déclenche 3 projets en parallèle
stages: [smoke, build-all, deploy-all]

smoke-tests:
  stage: smoke
  script: [curl -f https://staging.example.com/health]

trigger-frontend:
  stage: build-all
  needs: [smoke-tests]
  trigger: { project: acme/frontend, branch: main, strategy: depend }

trigger-backend:
  stage: build-all
  needs: [smoke-tests]
  trigger: { project: acme/backend,  branch: main, strategy: depend }

trigger-mobile:
  stage: build-all
  needs: [smoke-tests]
  trigger: { project: acme/mobile,   branch: main, strategy: depend }

trigger-deploy:
  stage: deploy-all
  needs: [trigger-frontend, trigger-backend, trigger-mobile]
  trigger: { project: acme/ops/deploy, branch: main, strategy: depend }
```

⋆ À retenir

Sans `strategy:depend`, l'upstream est marqué « success » dès le *lancement* du downstream (fire-and-forget). Avec `depend`, l'upstream attend la fin et reflète le statut downstream.

Visualisation et gestion d'une chaîne de pipelines

- **UI Pipelines** → vue arborescente : pipelines parents/enfants visibles avec flèches « Downstream pipeline », statut propagé.
- **Onglet « Pipeline graph »** → rendu DAG complet, hover sur job pour voir `needs`.
- **Onglet « Test summary »** → agrégation JUnit des enfants.
- Une chaîne complète apparaît dans le *breadcrumb* :
  `lib #42` → `service-A #17` → `service-B #9` → `deploy #5`

Gestion

- Cancel d'un parent → propage l'annulation aux downstreams.
- Retry uniquement du downstream : icône cercle sur la boîte downstream.
- Variables transmises : visibles dans le pipeline downstream sous *Variables* avec source « `trigger` ».

[TP associé]

TP18 — Multi-project pipeline avec `strategy:depend` et passage de variables.

Trigger en dépendance — 3 modes possibles

Mode	Effet sur l'upstream
`strategy: depend`	Attend la fin du downstream, reflète son statut
Pas de `strategy`	Fire-and-forget : success dès le lancement
`allow_failure: true`	Le downstream peut planter sans casser l'upstream

Trigger conditionnel (avec `rules`) :

```
trigger-deploy:
  trigger:
    project: acme/ops/deploy
    branch:  main
    strategy: depend
  rules:
    - if: $CI_COMMIT_TAG =~ /^v[0-9]+\.[0-9]+\.[0-9]+$/  # uniquement sur SemVer
  variables:
    DEPLOY_VERSION: $CI_COMMIT_TAG
    DEPLOY_ENV:     production
```

Trigger via API HTTP (depuis script externe) :

```
curl -X POST \
  -F "token=$TRIGGER_TOKEN" \
  -F "ref=main" \
  -F "variables[ENV]=staging" \
  https://gitlab.example.com/api/v4/projects/$PROJECT_ID/trigger/pipeline
```

[Aller plus loin]

`TRIGGER_TOKEN` créé via Settings → CI/CD → Pipeline triggers. Permet de déclencher la CI depuis Jenkins, ArgoCD, un cron externe, un webhook GitHub, etc.

Pipeline DevSecOps complet (recap)

```yaml
include:
  - template: Security/SAST.gitlab-ci.yml
  - template: Security/Secret-Detection.gitlab-ci.yml
  - template: Security/Dependency-Scanning.gitlab-ci.yml
  - template: Security/Container-Scanning.gitlab-ci.yml
  - template: Security/DAST.gitlab-ci.yml
  - template: Jobs/SAST-IaC.gitlab-ci.yml
  - template: Jobs/Code-Quality.gitlab-ci.yml
  - template: Jobs/License-Scanning.gitlab-ci.yml
  - local:    '.gitlab/ci/build.yml'
  - local:    '.gitlab/ci/deploy.yml'

stages: [build, test, security, package, deploy]

variables:
  DAST_WEBSITE: https://staging.example.com
  CS_IMAGE:     $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
```

[TP associé]

TP15 — Activer tous les scanners DevSecOps et observer les rapports dans la MR.

Panorama des tokens dans GitLab

Token	Préfixe	Portée	Usage typique
Personal Access Token (PAT)	`glpat-`	Par utilisateur, multi-projets	Scripts personnels, dev local
Project Access Token	`glpat-`	1 seul projet, bot user dédié	CI cross-projet sans dépendre d'un humain
Group Access Token	`glpat-`	1 groupe + descendants	Toolchain transverse à une BU
Deploy Token	`gldt-`	1 projet, scopes `read_repo` / `registry` / `packages`	CI externe, déploiement, miroir push
Deploy Key (SSH)	paire SSH (clé)	1 projet, read-only ou r+w	Cloner sans user attaché, pull mirror SSH
Pipeline Trigger Token	40 char hex	1 projet	Déclencher la CI depuis l'extérieur (cron, webhook GitHub)
Job Token (CI)	`CI_JOB_TOKEN`	Expire fin du job, scope projet	Cross-project depuis CI (auth API + registry + packages)
Runner Authentication Token	`glrt-`	1 runner	Communication runner ↔ GitLab
Runner Registration Token	`glrt-` éphémère	Création d'un runner (one-shot)	Nouvel enregistrement (GitLab 16+)
Incoming Email Token	32 char	Email → création issue/MR	Service Desk, contributions par mail
Feed Token	`glft-` ou hex	Flux RSS / iCal personnel	Lecteur RSS, agenda extérieur
OAuth 2 Access Token	64 char base64	App OAuth déclarée	Apps tierces (Jira, Slack, login GitLab social)
Impersonation Token (admin)	`glpat-`	Token PAT créé par admin pour un autre user	Audit, support, automation au nom d'un user

Personal Access Token (PAT) — scopes en détail

Chemin : Profile → Preferences → Access Tokens.

Scope	Permet
`api`	Accès complet à la REST API + GraphQL + Container/Package registry
`read_api`	Lecture seule sur tout l'API
`read_user`	Profil user uniquement
`read_repository`	Clone / pull en HTTP
`write_repository`	+ push
`read_registry`	Pull d'images depuis le Container Registry
`write_registry`	+ push d'images
`read_observability`	Métriques exporters (Premium)
`write_observability`	Push de métriques OTLP (Premium)
`ai_features`	GitLab Duo, Code Suggestions
`k8s_proxy`	Proxy K8s via Agent for Kubernetes
`sudo` (admin only)	Imperonate un autre user dans l'API

⋆ À retenir

Principe de moindre privilège : créer un token par usage, le plus restreint possible. `api` ne devrait être utilisé que pour les scripts ayant vraiment besoin d'écrire partout.

Project / Group Access Token — bots dédiés

Crée un **utilisateur bot** (de type `project_bot_*` ou `group_bot_*`) lié au projet/groupe avec un rôle (Guest/Reporter/Developer/Maintainer/Owner).

Chemins :

- Project Token : `Project Settings` → `Access Tokens`
- Group Token : `Group Settings` → `Access Tokens`

Avantages vs PAT humain :

- Pas de dépendance à un user qui quitte l'entreprise.
- Rôle (Guest/Reporter/Developer/Maintainer/Owner) cumulé avec scopes API.
- Rotation simple (1 clic + script).
- Audit : les actions apparaissent sous le nom du bot.

⋆ À retenir

Pour toute intégration externe (CI cross-projet, monitoring, déploiement) : **toujours** un Project ou Group Access Token, jamais un PAT humain.

Deploy Token vs Deploy Key vs Trigger Token

Type	Auth	Quand l'utiliser
Deploy Token	user+password HTTP	Pull mirror, déploiement vers prod, package registry pull
Deploy Key	paire SSH (public/private)	Cloner repo en CI *externe* (Jenkins, CircleCI...)
Trigger Token	token 40 char	Déclencher pipeline via API HTTP depuis cron / outil ext.

```
23 Pull mirror SSH avec Deploy Key git clone git@gitlab.example.com:acme/repo.git  cl
24 é publique configurée côté projet
25 Docker pull authentifié via Deploy Token docker login -u <deploy_token_user> -p <deploy_token> registry.example.com
```

CI_JOB_TOKEN — le token magique de la CI

Caractéristiques :

- Généré automatiquement par GitLab à chaque job CI.
- Disponible dans la variable `$CI_JOB_TOKEN`.
- **Expire à la fin du job** (sécurité : pas de fuite long-terme).
- Scope : projet du job + projets explicitement autorisés.

```yaml
# Auth registry depuis CI (push image)
build:
  script:
    - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" "$CI_REGISTRY"
    - docker build -t "$CI_REGISTRY_IMAGE:$CI_COMMIT_SHA" .
    - docker push    "$CI_REGISTRY_IMAGE:$CI_COMMIT_SHA"

# Appel API d'un autre projet (cross-project)     projet B doit avoir autorisé A
publish-package:
  script:
    - curl -H "JOB-TOKEN: $CI_JOB_TOKEN" \
        --upload-file dist/my-lib-1.0.0.tar.gz \
        "$CI_API_V4_URL/projects/42/packages/generic/my-lib/1.0.0/my-lib-1.0.0.tar.gz"
```

Autoriser un projet B à utiliser `CI_JOB_TOKEN` de A :
Projet A → Settings → CI/CD → *Job token permissions* → Add project B.

Rotation, audit et sécurité des tokens

Bonnes pratiques (toutes catégories)

- **Expiration** : 90 jours max recommandé (1 an par défaut), forcer le renouvellement.
- **Stockage** : variables CI *masked* + *protected*, ou Vault dynamique.
- **Rotation** : automatiser (script + API `POST /projects/:id/access_tokens/rotate`).
- **Audit** : Admin → Monitoring → Audit Events trace création/révocation/usage.
- **Révocation immédiate** : un seul clic dans Settings → Access Tokens.
- **Détection des fuites** : Secret Detection scan le code + l'historique.

À NE JAMAIS faire

- × Token sans expiration (ou expiration > 1 an).
- × Commit du token dans un repo (même privé).
- × Échanger un token par mail / Slack non chiffré.
- × Donner scope `api` quand `read_repository` suffit.
- × Utiliser le PAT du `root` dans une CI (→ Project Access Token à la place).

[TP associé]

TP22 — Rotation automatique d'un Project Access Token via cron + API.

La famille des « *Ops » — vue complète

Famille	Objectif	Outils typiques (en plus de GitLab)
DevOps	Fluidifier dev → prod (CI/CD, IaC, observabilité)	Docker, Ansible, Terraform, K8s, Prometheus
DevSecOps	Sécurité intégrée à chaque étape	SAST, DAST, SCA, IaC scan, SBOM, Sigstore, Vault
GitOps	Repo git = source de vérité de l'infra	Argo CD, Flux, Atlantis, Crossplane
MLOps	Industrialisation des modèles ML	DVC, MLflow, Kubeflow, Seldon, Triton, Feast
LLMOps	Variante MLOps pour LLM / RAG / fine-tuning	LangChain, vLLM, Hugging Face, pgvector, Pinecone
AIOps	IA *pour* les ops (anomaly detection, RCA)	Loki, Grafana ML, Splunk MLTK, BigPanda, Moogsoft
DataOps	Industrialiser les pipelines de données	Airflow, dbt, Dagster, Prefect, Great Expectations
FinOps	Optimisation des coûts cloud	OpenCost, Kubecost, Infracost, CloudHealth
SecOps	Opérations sécurité (SOC, IR)	Wazuh, SOAR, MISP, TheHive, Cortex
NetOps	Gestion réseau as code	Netbox, Nornir, Ansible Network, Containerlab
ChatOps	Opérations depuis Slack/Teams/Mattermost	Hubot, StackStorm, GitLab slash commands
ChaosOps	Tests de résilience par injection de pannes	Chaos Mesh, Litmus, Gremlin
PlatformOps	Internal Developer Platform (IDP)	Backstage, Port, Humanitec, Crossplane

Variables CI/CD essentielles — DevOps

Variable	Usage
`CI_COMMIT_SHA`	SHA complet du commit, immuable
`CI_COMMIT_BRANCH`	Branche déclenchant le pipeline
`CI_COMMIT_TAG`	Tag si la pipeline vient d'un tag
`CI_PIPELINE_SOURCE`	`push|merge_request_event|schedule|api|trigger|web`
`CI_PROJECT_DIR`	Racine du checkout sur le runner
`CI_REGISTRY_IMAGE`	URL du registry image du projet
`CI_ENVIRONMENT_NAME`	Nom de l'environnement (de `environment:`)
`KUBECONFIG`	(à fournir) kubeconfig pour déployer
`DEPLOY_ENV`	(custom) staging/production

Conseils DevOps

- 1 pipeline réutilisable pour tous les environnements (varier par `rules`).
- Tester l'IaC en *plan* sur chaque MR, *apply* après merge.
- Coverage des tests, security scans, performance benchmarks : tous artifacts dans la MR.

Variables CI/CD essentielles — DevSecOps

Variable	Usage
`SECRET_DETECTION_*`	Config Gitleaks (historic scan, exclude paths)
`SAST_EXCLUDED_PATHS`	Patterns à ignorer (vendor/, build/, test fixtures)
`SAST_EXCLUDED_ANALYZERS`	Désactiver des analyzers (semgrep, brakeman, ...)
`DS_EXCLUDED_PATHS`	Dependency Scanning : paths à exclure
`DAST_WEBSITE`	URL cible (staging ou review app)
`DAST_AUTH_URL`	URL de login (DAST authentifié)
`CS_IMAGE`	Image à scanner (`$CI_REGISTRY_IMAGE:$CI_COMMIT_SHA`)
`KICS_EXCLUDE_PATHS`	IaC scan : exclure des paths
`VAULT_ID_TOKEN`	JWT GitLab vers HashiCorp Vault (id_tokens)
`COSIGN_PASSWORD`	Pour signer images avec Sigstore

Conseils DevSecOps

- Vault dynamique pour TOUS les secrets de prod.
- Shift-left : scans à chaque MR, pas seulement en pre-prod.
- SBOM signé (Sigstore cosign) à chaque release.
- Quality gates : Vulnerability-Check + License-Check obligatoires.

Variables CI/CD essentielles — MLOps / LLMOps

Variable	Usage
`MLFLOW_TRACKING_URI`	Serveur MLflow pour le tracking
`DVC_REMOTE_URL`	S3/Azure Blob/GCS où sont les datasets versionnés
`MODEL_REGISTRY_URL`	Hugging Face Hub, MLflow Models, Seldon
`HF_TOKEN`	Authentification Hugging Face
`WANDB_API_KEY`	Tracking expériences avec Weights & Biases
`GPU_MEM_FRACTION`	Quota GPU partagé (PyTorch/TF)
`TRITON_URL`	Inference Server NVIDIA Triton
`LANGCHAIN_TRACING_V2`	Tracing LLM (LangSmith)
`OPENAI_API_KEY`	Modèles propriétaires (LLMOps)
`VECTOR_DB_URL`	pgvector / Pinecone / Weaviate / Qdrant

Conseils MLOps / LLMOps

- Versionner les datasets avec DVC + commit du `.dvc` file.
- Runner GPU dédié avec tags `gpu`, `cuda12`.
- Évaluation systématique sur dataset de test *figé* (régression de qualité).
- Pour LLMOps : Eval offline + canary online + observabilité de coûts API.

[TP associé]

TP23 — Pipeline MLOps : DVC pull + train + MLflow log + push model registry.

Variables CI/CD essentielles — AIOps, GitOps, FinOps

AIOps : la CI alimente plutôt qu'utilise.

- Push de métriques OTLP / Prometheus depuis les jobs.
- Anomaly detection sur durée pipeline, taux d'échec, queue Sidekiq.
- Auto-remediation : un webhook AIOps peut déclencher un `retry` ou un rollback.

GitOps : le repo est la source de vérité de l'infra.

- Variables : `ARGOCD_SERVER`, `FLUX_GIT_BRANCH`, `KUSTOMIZE_PATH`.
- Pas de `kubectl apply` en CI : la CI *modifie le repo*, Argo/Flux applique.
- Pull mode (cluster pull) plus sécurisé que push mode.

FinOps :

- Variables : `INFRACOST_API_KEY`, `KUBECOST_URL`, `TF_VAR_budget_alert`.
- Sur chaque MR : commenter le delta de coût mensuel estimé.
- Quota runners : limiter les minutes CI par groupe pour éviter les dérives.

Variables CI/CD essentielles — ChatOps, DataOps, ChaosOps

ChatOps :

- Variables : `SLACK_WEBHOOK_URL`, `TEAMS_WEBHOOK_URL`, `MATTERMOST_TOKEN`.
- Slash commands : `/gitlab deploy production`, `/gitlab rollback`.
- Approval gates dans le chat (boutons interactifs).

DataOps :

- Variables : `AIRFLOW_URL`, `DBT_PROFILES_DIR`, `DAGSTER_URL`.
- Tests de qualité de données (Great Expectations) en CI.
- Versionner les DAGs et les modèles dbt comme du code.

ChaosOps :

- Variables : `CHAOS_MESH_NAMESPACE`, `LITMUS_AGENT_TOKEN`.
- Game days : pipeline `chaos-experiment` qui injecte une faute en staging.
- Toujours coupler avec runbook + observabilité prête.

Synthèse : choisir le bon « *Ops » pour son contexte

Si vous voulez...	**Adopter en priorité**
Livrer du code en prod plus vite	DevOps (base)
Sécuriser cette livraison	DevSecOps (en couche sur DevOps)
Industrialiser des modèles ML	MLOps + DVC + MLflow
Déployer des LLM en prod	LLMOps (couche au-dessus de MLOps : eval, tracing, vector DB)
Détecter automatiquement les anomalies de prod	AIOps (Grafana ML, Loki, alertmanager + ML)
Gérer l'infra K8s comme du code	GitOps (Argo CD + repo `infra-state/`)
Optimiser les coûts cloud	FinOps (Infracost dans la CI, dashboards)
Tester la résilience	ChaosOps (chaos engineering en staging puis prod canary)
Standardiser DX pour les équipes	PlatformOps + IDP (Backstage, Port)

⋆ À retenir

Aucun de ces « Ops » ne s'oppose — ils se cumulent. On commence presque toujours par DevOps, on ajoute DevSecOps, puis spécialisations selon métier (ML, data, infra).

Les 4 dimensions d'une variable CI

Niveau	Instance / Group / Project / Pipeline trigger / Defined-in-yaml
Type	`Variable` (texte injecté) ou `File` (écrit dans un fichier temporaire, son chemin est dans $VAR)
Visibilité	`Visible` · `Masked` (caché des logs) · `Hidden` (non-lisible depuis l'UI)
Scope	`All envs` · `Protected refs only` (branches/tags protégés)

Recommandations sécurité

- Tout secret : **Masked + Protected** (au minimum).
- Préférer `File` pour kubeconfig, clés TLS, JSON service-account.
- Préfixer les noms : `K8S_*`, `AWS_*`, `DB_*`.
- × Jamais : `echo $SECRET` dans un script (apparaît dans les logs).
- Rotation : utiliser une source externe (Vault, AWS Secrets Manager) plutôt que stocker à long terme dans GitLab.

HashiCorp Vault via JWT (id_tokens)

```yaml
deploy-prod:
  stage: deploy
  image: alpine:3.20
  id_tokens:
    VAULT_ID_TOKEN:                                  # JWT signé par GitLab
      aud: https://vault.example.com
  script:
    - apk add --no-cache curl jq
    - export VAULT_ADDR=https://vault.example.com
    - |
      VAULT_TOKEN=$(curl -s --request POST \
        --data "{\"jwt\":\"$VAULT_ID_TOKEN\",\"role\":\"gitlab-lab-app\"}" \
        $VAULT_ADDR/v1/auth/jwt/login | jq -r .auth.client_token)
    - export VAULT_TOKEN
    - API_TOKEN=$(curl -s -H "X-Vault-Token: $VAULT_TOKEN" \
        $VAULT_ADDR/v1/secret/data/ci/lab-app | jq -r .data.data.api_token)
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
```

⋆ À retenir

Le JWT GitLab est signé et contient (project_id, branch, user_id…). Vault peut binder l'accès à un secret précis pour un (projet, branche) précis. **Aucun secret long-lived n'est stocké dans GitLab.**

[TP associé]

TP16 — Démarrer Vault dev, configurer auth JWT, récupérer un secret depuis la CI.

Partie 4 — Ce que vous savez maintenant

Acquis

- ✓ Anatomie d'un pipeline (stages, jobs, rules, needs, artifacts, cache)
- ✓ Optimisations DAG (needs) et templates (include/extends)
- ✓ Variables CI à 4 dimensions
- ✓ Secrets éphémères via JWT → Vault

API GitLab : REST v4, GraphQL, webhooks, bibliothèques par langage et exemples concrets dans 10 langages.

5
Partie 5 — API GitLab

Partie 5 — Ce que nous allons voir

Au programme

1. Portée de l'API GitLab (qu'est-ce qui est exposé ?)
2. REST API v4 : auth, endpoints, pagination, rate limit
3. GraphQL API : avantages, introspection
4. Webhooks (rappel) + Server-sent events
5. Bibliothèques clientes par langage
6. Exemples concrets en 10 langages (depuis `api/`)

À la fin vous saurez

- Lister/créer/modifier projets, users, MR, issues via REST
- Écrire une query GraphQL pour un dashboard custom
- Choisir le bon client par langage
- Gérer la pagination, les tokens, les limites
- Construire un script ou microservice qui interagit avec GitLab

Qu'est-ce qui est exposé par l'API ?

Quasi 100 % des fonctionnalités UI sont accessibles par API. Les entités principales :

Entité	Opérations REST principales
Projects	`GET/POST/PUT/DELETE /projects`, fork, transfer, archive
Groups + subgroups	`CRUD`, members, sharing
Users	`CRUD`, impersonate (admin), block, deactivate
Repository files / branches	`GET/POST/PUT/DELETE /projects/:id/repository/files/...`
Commits + diffs	`GET /projects/:id/repository/commits/:sha` + cherry-pick + revert
Merge requests	`CRUD`, approve, merge, rebase, cancel, notes (comments)
Issues + epics	`CRUD`, links, milestones, labels, time tracking
Pipelines + Jobs	list, retry, cancel, artifacts download, trace logs, trigger
Runners	list, register, pause, unregister
Container Registry	repositories, tags, manifests, cleanup
Package Registry	npm, Maven, NuGet, PyPI, Conan, Cargo, Composer, Generic, Helm, Debian/RPM
Releases / Tags	`CRUD`, assets (binaires liés)
Webhooks / System hooks	`CRUD`
Access Tokens	list, create, revoke (par user/project/group)
Environments / Deployments	list, create, stop, list-jobs
Feature flags	toggles déployables sans release
Compliance / Audit Events	lecture + Streaming (Premium)
Geo + Praefect (HA)	status replication, switch primary

⋆ À retenir

Si une opération existe dans l'UI, elle existe en API (et parfois *uniquement* en API : impersonate, broadcast).

Limites et restrictions

Limite	Détail
Rate limit anonyme	500 req/min par IP (Admin → Network → Rate limits)
Rate limit authentifié	2 000 req/min par user
Rate limit search global	30 req/min par user
Webhooks	Timeout 10 s, retry 4 fois
GraphQL : profondeur max	15 niveaux (configurable Admin)
GraphQL : complexité max	250 par requête (10 000 pour admin)
Token PAT/Project/Group : durée max	1 an par défaut (configurable Admin Settings)

Headers utiles renvoyés :

- `RateLimit-Remaining`, `RateLimit-Reset` : backoff
- `X-Total`, `X-Per-Page`, `X-Page` : pagination
- `Link` : header RFC 5988 (next, prev, first, last)

REST API : authentification (4 méthodes)

Méthode	Header / paramètre
Personal Access Token (PAT)	`PRIVATE-TOKEN: glpat-...`
OAuth 2 (flow standard)	`Authorization: Bearer <access_token>`
Job token (depuis CI)	`JOB-TOKEN: $CI_JOB_TOKEN` (scope projet, expire fin du job)
Session cookie	`_gitlab_session=...` (uniquement même origine UI)

```
# Lister les projets accessibles
curl -sH "PRIVATE-TOKEN: glpat-xxxxxxxxxxxxxxxxxxx" \
  https://gitlab.example.com/api/v4/projects | jq '.[].path_with_namespace'

# Créer une issue
curl -X POST -H "PRIVATE-TOKEN: glpat-xxx" \
     -d "title=Bug critique&labels=bug,prio:high" \
     https://gitlab.example.com/api/v4/projects/42/issues

# Pagination via Link header
curl -sI -H "PRIVATE-TOKEN: glpat-xxx" \
  "https://gitlab.example.com/api/v4/projects?per_page=100&page=1" | grep -i link
```

REST API : endpoints les plus utiles

```
# Metadata
GET    /api/v4/version
GET    /api/v4/user                              # user courant (vérification token)
GET    /api/v4/metadata                          # version + features actives

# Projects
GET    /api/v4/projects?owned=true&per_page=20
POST   /api/v4/projects                          # create
GET    /api/v4/projects/:id                      # :id numérique OU "namespace%2Fpath"
PUT    /api/v4/projects/:id
DELETE /api/v4/projects/:id

# Repository
GET    /api/v4/projects/:id/repository/branches
GET    /api/v4/projects/:id/repository/files/:path/raw?ref=main
POST   /api/v4/projects/:id/repository/commits   # multi-fichiers atomique

# CI/CD
GET    /api/v4/projects/:id/pipelines
POST   /api/v4/projects/:id/pipeline?ref=main
GET    /api/v4/projects/:id/jobs/:job_id/trace
POST   /api/v4/projects/:id/jobs/:job_id/retry

# Issues / MR
GET    /api/v4/issues?scope=assigned_to_me&state=opened
POST   /api/v4/projects/:id/merge_requests
PUT    /api/v4/projects/:id/merge_requests/:iid/merge
```

GraphQL : query unique, payload sur mesure

Endpoint unique : `POST /api/graphql` avec body JSON `{ "query": "..." }`. Mêmes tokens que REST.

```
# Liste des MR ouvertes + labels + 5 derniers commentaires (1 requête !)
query {
  currentUser {
    name; username
    assignedMergeRequests(state: opened) {
      nodes {
        title; webUrl; sourceBranch; targetBranch
        labels { nodes { title color } }
        notes(first: 5) { nodes { author { username } body createdAt } }
      }
    }
  }
}
```

⋆ À retenir

En REST, il aurait fallu : `/user` + `/merge_requests` + N × `/labels` + N × `/notes`. En GraphQL : **1 requête**, exactement les champs voulus, 1 round-trip.

GraphQL : mutations et introspection

```
# Mutation : créer une issue
mutation {
  createIssue(input: {
    projectPath: "acme/backend/lab-app",
    title:       "Bug API GraphQL",
    description: "Détaillé ici...",
    labels:      ["bug", "api"]
  }) {
    issue { iid webUrl }
    errors
  }
}

# Introspection : découvrir le schéma à la volée
query { __schema { types { name fields { name type { name } } } } }
```

Outils recommandés :

- **GraphiQL** intégré : `/-/graphql-explorer` sur votre instance.
- Insomnia / Postman : support GraphQL natif (introspection auto).
- Bibliothèques : `gql` (Python), `apollo-client` (JS), `graphql-client` (Ruby).

[TP associé]

TP20 — Écrire une query GraphQL listant les pipelines failed + auteur + commit.

Bibliothèques par langage — panorama

Voir le dépôt `api/` pour les samples détaillés (10 langages, chacun avec `samples/01_list_projects.*` et `02_create_issue.*`).

Langage	Bibliothèque	PDF / Doc
Python 3	`python-gitlab`	✓ PDF readthedocs inclus
Go	`gitlab.com/gitlab-org/api/client-go` (officiel)	pkg.go.dev
Rust	crate `gitlab`	docs.rs
C / C++	cpr + nlohmann/json (HTTP brut)	docs.libcpr.org + json.nlohmann.me
C#	`NGitLab` (Ubisoft)	github.com/ubisoft/NGitLab
Ruby	gem `gitlab` (NARKOZ)	rubydoc.info
Node.js	`@gitbeaker/rest`	gitbeaker.com
TypeScript	`@gitbeaker/rest` (typage natif)	gitbeaker.com
Java / Kotlin	`gitlab4j-api`	gitlab4j.github.io/gitlab4j-api
PHP	`m4tthumphrey/php-gitlab-api`	github

Exemple Python (api/python3/samples/)

```python
import os, gitlab
gl = gitlab.Gitlab(url=os.environ["GITLAB_URL"],
                   private_token=os.environ["GITLAB_TOKEN"])
gl.auth()

for p in gl.projects.list(iterator=True, per_page=20):
    print(f"{p.id:>5}  {p.path_with_namespace}")

proj = gl.projects.get(42)
issue = proj.issues.create({
    "title":       "Issue via python-gitlab",
    "description": "Créée par script.",
    "labels":      ["bug", "demo"],
})
print(f"✓ #{issue.iid}     {issue.web_url}")
```

Exemple Go (api/go/samples/)

```go
package main
import (
    "fmt"; "log"; "os"
    gitlab "gitlab.com/gitlab-org/api/client-go"
)

func main() {
    git, err := gitlab.NewClient(
        os.Getenv("GITLAB_TOKEN"),
        gitlab.WithBaseURL(os.Getenv("GITLAB_URL")+"/api/v4"),
    )
    if err != nil { log.Fatal(err) }
    projects, _, _ := git.Projects.ListProjects(
        &gitlab.ListProjectsOptions{ ListOptions: gitlab.ListOptions{PerPage: 20} })
    for _, p := range projects {
        fmt.Printf("%5d  %s\n", p.ID, p.PathWithNamespace)
    }
}
```

Exemples Node.js / Ruby / Java

```
// Node.js     @gitbeaker/rest
import { Gitlab } from '@gitbeaker/rest';
const api = new Gitlab({ host: process.env.GITLAB_URL, token: process.env.GITLAB_TOKEN });
const projects = await api.Projects.all({ perPage: 20, maxPages: 1 });
projects.forEach(p => console.log(p.id, p.path_with_namespace));

# Ruby     gem 'gitlab'
g = Gitlab.client(endpoint: "#{ENV['GITLAB_URL']}/api/v4",
                  private_token: ENV['GITLAB_TOKEN'])
g.projects(per_page: 20).each { |p| puts "#{p.id}  #{p.path_with_namespace}" }

// Java     gitlab4j-api
try (GitLabApi api = new GitLabApi(System.getenv("GITLAB_URL"),
                                   System.getenv("GITLAB_TOKEN"))) {
    for (Project p : api.getProjectApi().getProjects(1, 20)) {
        System.out.printf("%5d  %s%n", p.getId(), p.getPathWithNamespace());
    }
}
```

[TP associé]

TP21 — Implémenter le même cas (list + create issue) dans 3 langages différents.

Cas pratique : Webhook receiver Express.js

```
import express from 'express';
const app = express();
app.use(express.json());

app.post('/hooks/gitlab', (req, res) => {
  const token = req.headers['x-gitlab-token'];
  if (token !== process.env.GITLAB_WEBHOOK_SECRET) return res.sendStatus(401);
  const event = req.headers['x-gitlab-event'];
  switch (event) {
    case 'Push Hook':
      console.log(`Push on ${req.body.ref} by ${req.body.user_name}`); break;
    case 'Merge Request Hook':
      const mr = req.body.object_attributes;
      console.log(`MR #${mr.iid} ${mr.action}: ${mr.title}`); break;
    case 'Pipeline Hook':
      console.log(`Pipeline ${req.body.object_attributes.status}`); break;
  }
  res.sendStatus(200);
});

app.listen(3000, () => console.log('Webhook receiver on :3000'));
```

Partie 5 — Ce que vous savez maintenant

Acquis

- ✓ Portée de l'API GitLab (entités, limites, rate limits)
- ✓ Auth REST (PAT, OAuth, Job token, session)
- ✓ Endpoints REST les plus utiles + pagination Link header
- ✓ GraphQL : query, mutation, introspection (GraphiQL)
- ✓ Bibliothèques recommandées pour 10 langages + exemples
- ✓ Webhooks receiver minimal

Annexes : lexique français, bibliographie FR+EN, refcards (`.gitlab-ci.yml`, `.gitignore`, ...), liens utiles (chaînes YouTube FR, doc officielle).

6
Partie 6 — Annexes

Lexique français (1/2) — A à L

Terme	Définition
AIOps	IA appliquée aux opérations IT (détection d'anomalies, root cause analysis).
Artefact	Sortie d'un job CI (binaire, rapport, package) téléchargeable et passable au job suivant.
Cache	Données réutilisables entre exécutions de jobs (ex. `node_modules/`).
CI/CD	Intégration / Livraison / Déploiement continus.
CODEOWNERS	Fichier listant qui doit reviewer telle ou telle partie du code.
DAG	Directed Acyclic Graph. Dépendances entre commits (git) ou entre jobs CI.
DAST	Dynamic Application Security Testing : scan de l'app en exécution.
DevOps	Culture/pratiques unissant développement et opérations.
DevSecOps	DevOps avec la sécurité intégrée à chaque étape du cycle.
Executor	Type d'env où s'exécute un job runner : shell, docker, ssh, kubernetes.
Fork	Copie d'un projet vers un autre namespace tout en conservant la liaison.
Gitaly	Service interne GitLab (Go) qui sert toutes les opérations git via gRPC.
GitFlow	Modèle de branchement Driessen (2010) : develop / feature / release / hotfix / main.
GitOps	Repo git comme seule source de vérité de l'infrastructure souhaitée.
IaC	Infrastructure as Code (Terraform, OpenTofu, Ansible, Pulumi).
Inventory	Fichier listant les hôtes cibles d'Ansible.
KAS	Kubernetes Agent Server : pont sécurisé entre GitLab et un cluster K8s.
LFS	Large File Storage : extension git pour les gros binaires.
LLMOps	Variante MLOps focalisée sur les Large Language Models.

Lexique français (2/2) — M à W

Terme	Définition
Merge Request	Demande de fusion d'une branche dans une autre (équivalent Pull Request).
MLOps	Industrialisation des modèles ML (data, train, eval, deploy, monitor).
Monorepo	Un seul repo git pour plusieurs projets/microservices.
Namespace	Conteneur logique pour un projet : un user ou un groupe.
OmniAuth	Module Rails d'auth fédérée (OIDC, SAML, GitHub...) utilisé par GitLab.
Pipeline	Ensemble ordonné de jobs CI/CD déclenché par un événement git.
Praefect	Routeur HA placé devant 3+ instances Gitaly.
Puma	Serveur Rails par défaut de GitLab (remplace Unicorn).
Push rule	Règle serveur (Premium) qui peut refuser un push selon des critères.
Runner	Agent qui exécute les jobs CI. Shared, group ou project.
SAST	Static Application Security Testing : scan du code source.
SBOM	Software Bill of Materials : nomenclature de toutes les dépendances.
SCA	Software Composition Analysis : scan des dépendances (CVE, licences).
Secret Detection	Scan du code et de l'historique à la recherche de credentials accidentels.
Sidekiq	Workers Ruby asynchrones (Redis-backed) de GitLab.
Sigstore	Écosystème de signature de software : cosign, rekor, fulcio.
SSO	Single Sign-On : login unique fédéré (SAML, OIDC, Kerberos).
Stage	Groupement de jobs CI exécutés en parallèle, séquentiellement entre stages.
Tag	Référence git nommée et immuable (souvent pour versions v1.2.3).
Trunk-Based	Branchement minimal : tout le monde commit sur main, behind feature flags.
Webhook	POST HTTP envoyé par GitLab à une URL externe lors d'un événement.
Workhorse	Service Go devant Puma : uploads, LFS, WebSocket.

Ouvrages — Français (git & fondamentaux)

Titre	Auteur	Éditeur	ISBN
Pro Git (traduction française, libre)	Scott Chacon, Ben Straub	Apress / git-scm	978-1-4842-0076-6
Git — maîtrisez la gestion de vos versions de code	Sébastien Castiel	D-BookeR	978-2-8227-0353-7
Git et GitHub — du débutant au pro	Antoine Augusti	Editions ENI	978-2-409-03611-9
Maîtrisez Git et GitHub	Théo Champion	Eyrolles	978-2-416-00854-0
Pratique de Git — 13 ateliers progressifs	Sébastien Rohaut	Editions ENI	978-2-7460-9802-9
Mémento Git	Sébastien Castiel	Eyrolles	978-2-212-67790-3

Ouvrages — Français (DevOps, GitLab CI, sécurité)

Titre	Auteur	Éditeur	ISBN
Apprendre Linux et DevOps (PDF libre, blog)	Stéphane Robert	auto-publié	— (blog.stephane-robert.info)
Le guide pratique de DevOps	Patrice Debray	Editions ENI	978-2-409-03734-5
GitLab et DevOps — mise en pratique	Christophe Pinto, Mickaël Baron	ENI	978-2-409-04026-0
Intégration continue avec GitLab CI/CD	Patrice Debray	Editions ENI	978-2-409-04147-2
Apprendre la programmation Ansible	Patrice Debray	Editions ENI	978-2-409-03362-0
Containers Docker — du développement à la production	Pierre Mavro, Jean-Baptiste Aubin	Eyrolles	978-2-212-67900-6
Kubernetes en production	Sébastien Goasguen	Eyrolles	978-2-212-67896-2
Sécurité DevSecOps	Akram El Korashy	Editions ENI	978-2-409-04220-2
L'écosystème open source HashiCorp	Romain Boulanger	Editions ENI	978-2-409-03922-6

Certains ISBN reflètent l'édition la plus récente connue à 2025 — vérifier auprès de l'éditeur pour la dernière révision.

Ouvrages — Anglais (git, GitLab, DevOps)

Titre	Auteur(s)	Éditeur	ISBN
Pro Git (2e éd., libre online)	Scott Chacon, Ben Straub	Apress	978-1-4842-0076-6
Professional Git	Brent Laster	Wrox / Wiley	978-1-119-28497-8
Version Control with Git (3rd ed.)	Prem Kumar Ponuthorai, Jon Loeliger	O'Reilly	978-1-492-09151-0
GitLab Quick Start Guide	Adam O'Grady	Packt	978-1-78934-878-9
Mastering GitLab 12	Joost Evertse	Packt	978-1-78953-440-2
GitLab Cookbook	Jeroen van Baarsen	Packt	978-1-78328-833-5
Hands-On Auto-DevOps with GitLab CI	Joost Evertse	Packt	978-1-80056-957-1
Learning DevOps (2nd ed.)	Mikael Krief	Packt	978-1-80181-896-5
Practical DevOps (2nd ed.)	Joakim Verona	Packt	978-1-78839-257-3
The DevOps Handbook (2nd ed.)	Kim, Humble, Debois, Willis	IT Revolution Press	978-1-950508-40-2
Accelerate	Forsgren, Humble, Kim	IT Revolution Press	978-1-942788-33-1
Site Reliability Engineering (« SRE Book »)	Beyer, Jones, Petoff, Murphy (Google)	O'Reilly	978-1-491-92912-4

Ouvrages — Anglais (sécurité, IaC, cloud)

Titre	Auteur(s)	Éditeur	ISBN
Container Security	Liz Rice	O'Reilly	978-1-492-05670-1
Kubernetes Security and Observability	Brendan Creane, Amit Gupta	O'Reilly	978-1-098-10713-5
Securing DevOps	Julien Vehent	Manning	978-1-617-29429-3
DevSecOps	Glenn Wilson	Packt	978-1-80323-947-9
The Phoenix Project (récit, fondateur du mouvement DevOps)	Kim, Behr, Spafford	IT Revolution Press	978-0-988-26259-2
The Unicorn Project (suite, dev perspective)	Gene Kim	IT Revolution Press	978-1-942788-76-8
Terraform: Up and Running (3rd ed.)	Yevgeniy Brikman	O'Reilly	978-1-098-11674-8
Ansible: Up and Running (3rd ed.)	Bas Meijer, René Moser, Lorin Hochstein	O'Reilly	978-1-098-10915-3
Ansible for DevOps (2nd ed.)	Jeff Geerling	Leanpub	— (PDF en ligne)
Cloud Native DevOps with Kubernetes (2nd ed.)	John Arundel, Justin Domingus	O'Reilly	978-1-098-11682-3
Building Microservices (2nd ed.)	Sam Newman	O'Reilly	978-1-492-03402-0
Designing Data-Intensive Applications	Martin Kleppmann	O'Reilly	978-1-449-37332-0

Ouvrages — MLOps, LLMOps, GitOps

Titre	Auteur(s)	Éditeur	ISBN
Introducing MLOps	M. Treveil et al. (Dataiku)	O'Reilly	978-1-098-10968-9
Designing Machine Learning Systems	Chip Huyen	O'Reilly	978-1-098-10796-8
Machine Learning Engineering in Action	Ben Wilson	Manning	978-1-617-29871-0
Practical MLOps	Noah Gift, Alfredo Deza	O'Reilly	978-1-098-10301-4
Reliable Machine Learning	C. Adams, R. Cohen, T. Kunz, K. Pop	O'Reilly	978-1-098-10643-5
Building LLMs for Production	Louis-François Bouchard, Louie Peters	Towards AI	979-8-218-31900-9
LLM Engineer's Handbook	Paul Iusztin, Maxime Labonne	Packt	978-1-83644-708-2
GitOps and Kubernetes	Yuen, Matyushentsev, Ekenstam, Suen	Manning	978-1-617-29727-0
Continuous Delivery in Java	Daniel Bryant, Abraham Marín-Pérez	O'Reilly	978-1-491-98602-8
Streaming Systems	Tyler Akidau, Slava Chernyak, R. Lax	O'Reilly	978-1-491-98367-6

Refcard `.gitlab-ci.yml` essentiel

```yaml
stages: [build, test, deploy]
default: { image: alpine:3.20, tags: [docker] }
variables: { FOO: "bar" }

job-name:
  stage: build
  image: node:20
  needs: [other-job]            # DAG
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
  script:
    - echo "$CI_COMMIT_SHORT_SHA"
  artifacts:
    paths: [dist/]
    expire_in: 7 days
    reports:
      junit: report.xml         # rapports natifs (junit, codequality, sast, dast, container_scan)
  cache:
    key: { files: [package-lock.json] }
    paths: [node_modules/]
  environment:
    name: staging
    url: https://staging.example.com
  retry: { max: 2, when: [runner_system_failure] }
  timeout: 30 minutes
```

Refcard .gitignore générique multi-langage

```
# OS
.DS_Store
Thumbs.db
*~

# IDE
.idea/
.vscode/
*.swp

# Build / dist
dist/
build/
target/
out/

# Logs / temp
*.log
tmp/

# Secrets / env
.env
.env.*
!.env.example
*.pem
*.key

# Langages
node_modules/          # JS
__pycache__/           # Python
*.py[cod]
.venv/
*.class                # Java
*.jar
*.war
```

Refcard `.gitattributes` – normaliser fins de ligne, LFS, diff

```
# Normalisation des fins de ligne
*               text=auto eol=lf
*.bat           eol=crlf
*.cmd           eol=crlf

# Marqueur binaire (jamais merge)
*.png           binary
*.pdf           binary
*.zip           binary

# git-lfs (gros fichiers)
*.psd           filter=lfs diff=lfs merge=lfs -text
*.glb           filter=lfs diff=lfs merge=lfs -text
*.mp4           filter=lfs diff=lfs merge=lfs -text
data/**/*.csv   filter=lfs diff=lfs merge=lfs -text

# diff custom
*.json          diff=json
*.lock          merge=ours
*.generated.*   linguist-generated=true
```

Refcard .editorconfig (cross-IDE)

```
root = true

[*]
charset = utf-8
end_of_line = lf
insert_final_newline = true
trim_trailing_whitespace = true
indent_style = space
indent_size = 4

[*.{js,ts,jsx,tsx,vue,json,yml,yaml,html,css}]
indent_size = 2

[Makefile]
indent_style = tab

[*.md]
trim_trailing_whitespace = false
```

Documentation officielle

- GitLab Docs : https://docs.gitlab.com/
- GitLab CI Reference : https://docs.gitlab.com/ci/yaml/
- gitlab-runner : https://docs.gitlab.com/runner/
- Pro Git book : https://git-scm.com/book/fr/v2
- OpenAPI Specification : https://swagger.io/specification/
- OpenTofu Registry : https://search.opentofu.org/
- Provider `gitlabhq/gitlab` : https://registry.terraform.io/providers/gitlabhq/gitlab/latest/docs
- HashiCorp Vault Docs : https://developer.hashicorp.com/vault/docs

Communautés francophones & chaînes vidéo

- **Stéphane Robert** (blog référence FR) : https://blog.stephane-robert.info/
- **Xavki** (chaîne YouTube DevOps FR très complète) : https://www.youtube.com/@xavki
- **Cocadmin** (DevOps + sécurité, FR) : https://www.youtube.com/@cocadmin
- **Ippon Talks** (conférences FR) : https://www.youtube.com/@IpponTechnologies
- **NumberlyTech** (REX FR) : https://medium.com/numberly-engineering
- **GitLab Unfiltered** (chaîne officielle, EN) : https://www.youtube.com/@GitLabUnfiltered
- **TechWorld with Nana** (DevOps EN, très clair) : https://www.youtube.com/@TechWorldwithNana

Communautés & événements

- **LinuxMaine** : https://linuxmaine.org/ (organisateur de ce module)
- Slack GitLab Community : https://gitlab.com/gitlab-com/community-edition-slack
- GitLab Forum : https://forum.gitlab.com/
- DevSecCon : https://www.devseccon.com/
- DevOps'D Day Marseille, ParisDevOps Meetup, NantesDevOps Meetup
- CNCF Conferences (KubeCon, GitOpsCon)

Merci de votre attention.

Questions, retours, propositions de TP ?

Thierry GAYET — Thierry.Gayet@gmail.com
Association **LinuxMaine** — 2026
Sources LaTeX et TP : `labs/slides/` et `labs/labs/`