Manuel d'administration

Gitrust — forge Git self-hosted écrite en Rust

Équipe Gitrust

© Gitrust

Sommaire

1. Manuel d'administration 3
1.1 Manuel d'administration 3
1.2 Tutoriels 6
1.3 Guides pratiques 44
1.4 Référence 135
1.5 Explications 204

© Gitrust

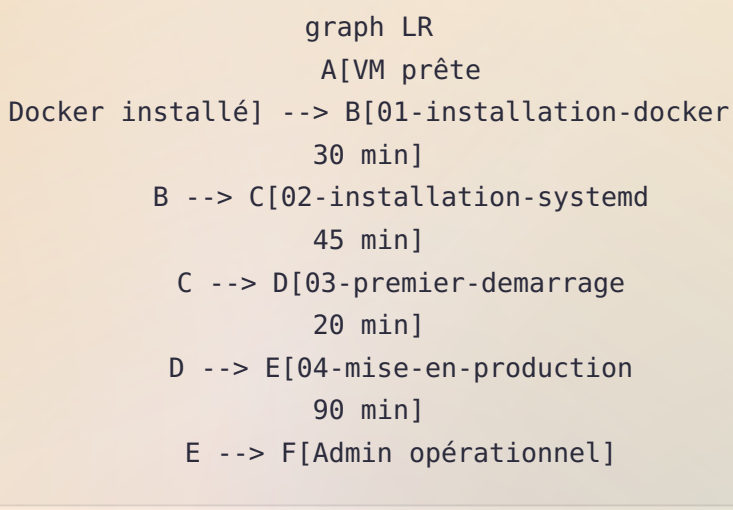
1. Manuel d'administration

1.1 Manuel d'administration

Ce manuel s'adresse aux opérateurs et self-hosters qui déploient, configurent et maintiennent une instance gitrust en production. Il couvre l'intégralité du cycle de vie d'une instance : premier démarrage, mise en production, sauvegarde, mise à jour et dépannage. Il suppose des connaissances en administration Linux, en réseau et en TLS — il n'explique pas ces prérequis.

1.1.1 Parcours d'apprentissage

Le parcours administrateur est structuré en 4 tutoriels qui représentent environ 3 heures au total. L'étayage est dégressif : le premier tutoriel fournit tout clé en main, le dernier demande de composer soi-même à partir des recettes disponibles.



> ***Checkpoint de parcours*** *: avant de passer au tutoriel 02, vous devez avoir une instance gitrust accessible sur* `http://localhost:4000` *avec un compte administrateur fonctionnel. Si ce n'est pas le cas, relisez le tutoriel 01.*
>
> ***Checkpoint de parcours*** *: avant le tutoriel 04, vous devez avoir une instance démarrée avec une configuration SMTP et un reverse-proxy TLS opérationnel. Si ce n'est pas le cas, relisez le tutoriel 03.*

© Gitrust

1.1.2 Tutoriels — apprendre en faisant

Les tutoriels sont guidés pas-à-pas avec des sorties verbatim attendues. Suivez-les dans l'ordre — chacun suppose le précédent complété.

- 01 — Installation Docker : stack docker-compose complète — ~30 min
- 02 — Installation systemd : binaire natif + service — ~45 min
- 03 — Premier démarrage : admin bootstrap, migrations — ~20 min
- 04 — Mise en production — ~90 min

1.1.3 How-to — recettes pour les tâches courantes

Les how-to répondent à « comment faire X » sans explication de fond. Consultez-les dans n'importe quel ordre selon votre besoin.

- Sauvegarder et restaurer
- Mettre à jour l'instance
- Configurer SMTP
- Configurer OAuth (Google, GitHub)
- Configurer un CI runner distant
- Intégrer Dependency-Track (SBOM)
- Durcir avec Fail2ban
- Tuner le rate limiting
- Forcer la 2FA globalement
- Dépanner SSH
- Dépanner la CI
- Gérer les utilisateurs depuis /admin
- Auditer l'instance

1.1.4 Référence — informations techniques exactes

La référence documente de façon exhaustive les paramètres, schémas et comportements. Elle est destinée à la consultation ponctuelle.

- Variables d'environnement (.env)
- Paramètres dynamiques (app_settings)
- Schéma de base de données
- Ports et services (4000 HTTP, 2222 SSH, 5432 PG)
- Conformité ANSSI PA-074

© Gitrust

1.1.5 Explication — comprendre le pourquoi

Les explanations construisent les modèles mentaux nécessaires pour prendre de bonnes décisions d'exploitation. Lisez-les quand vous voulez comprendre les choix d'architecture, pas seulement les appliquer.

- Architecture globale de gitrust
- Modèle de déploiement : mono-machine vs HA
- CI Dagger : pourquoi et comment
- Stratégie de sauvegarde 3-2-1 adaptée gitrust

© Gitrust

1.2 Tutoriels

1.2.1 Installation Docker : démarrez gitrust en 30 minutes avec docker-compose

Objectifs

À la fin de ce tutoriel, vous saurez :

- **O1. Déployer** la stack gitrust minimale (application + PostgreSQL) avec docker-compose
- **O2. Vérifier** que les trois services démarrent correctement et sont en bonne santé
- **O3. Créer** le premier compte administrateur et accéder à l'interface web

Pré-requis

- **Technique** : une VM ou machine Linux (Debian 12 / Ubuntu 22.04 ou supérieur), Docker Engine ≥ 24 installé (`docker --version` répond), Docker Compose plugin installé (`docker compose version` répond), ports 4000 et 2222 libres sur la machine
- **Image Docker gitrust** : à construire localement à partir du code source (l'image n'est pas encore publiée sur un registre public). Suivez la section Construire l'image gitrust ci-dessous avant le `docker compose up`.
- **Pédagogique** : aucun tutoriel gitrust préalable requis — c'est le point de départ du parcours administrateur
- **Temps estimé** : ~30 minutes

Construire l'image gitrust

Image non encore publiée

gitrust fournit un `Dockerfile` multi-stage (Rust + Tailwind/DaisyUI) dans le dépôt source, mais aucune image officielle n'est encore publiée sur un registre public (Docker Hub, ghcr.io, quay.io). L'installation Docker officielle est donc **expérimentale** : il faut cloner le dépôt et construire l'image localement. Le support Docker production sera stabilisé dans une prochaine release.

```
git clone https://demo.gitrust.eu/gitrust/gitrust.git
cd gitrust
docker build -t gitrust:latest .
```

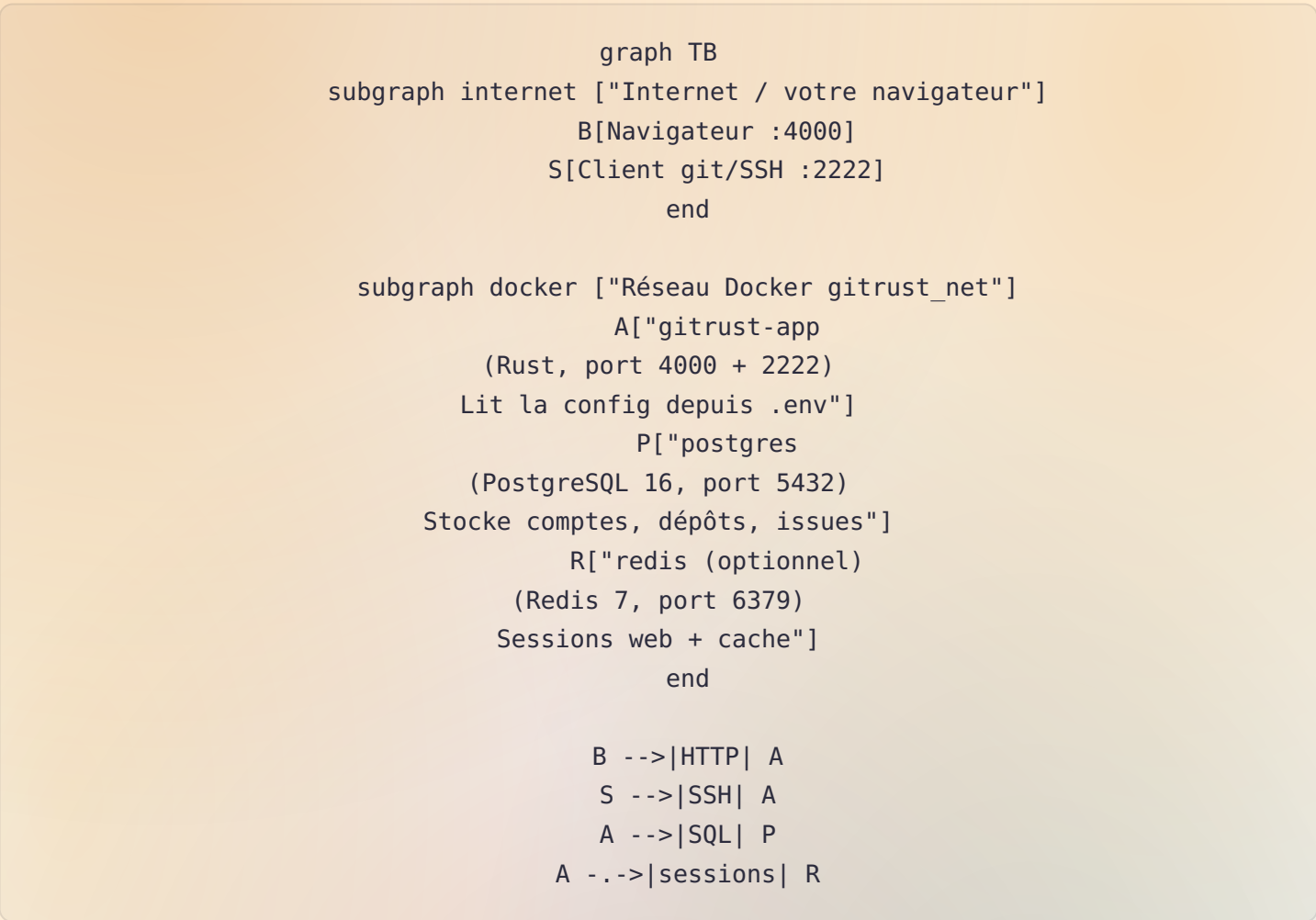
À la fin, `docker image ls | grep gitrust` doit afficher l'image locale `gitrust:latest`. Le build prend ~8-15 min (compilation Rust en release + build CSS Tailwind). Si vous rencontrez une erreur, ouvrez une issue sur le dépôt source — le pipeline Docker n'a pas encore été testé en production.

© Gitrust

Vue d'ensemble

Avant de lancer la moindre commande, comprenons ce que docker-compose va orchestrer et pourquoi.

gitrust est composé de **trois services principaux** dans sa configuration minimale. Pensez-y comme trois pièces d'un appartement qui s'appellent entre elles via un réseau interne :



gitrust-app est le cœur : serveur HTTP (`:4000`) + serveur SSH (`:2222`) compilé en Rust. Il lit sa configuration depuis un fichier `.env` et écrit les dépôts Git bare sur un volume Docker. **PostgreSQL** stocke toutes les données relationnelles (comptes, organisations, dépôts, issues, PRs). **Redis** est optionnel dans la configuration minimale — il gère les sessions web et le cache quand il est présent.

Dans ce tutoriel, nous utilisons la stack **minimale** (app + PostgreSQL) suffisante pour évaluer gitrust et démarrer une petite instance. La stack production avec Redis et SMTP se trouve dans `template/docker/docker-compose.production.yml` .

Étape 1 : Créez le répertoire de travail et le fichier de configuration

Créez un répertoire dédié à votre instance gitrust et positionnez-vous dedans :

© Gitrust

```
mkdir -p /opt/gitrust && cd /opt/gitrust
```

Créez le fichier de configuration `.env` avec les valeurs minimales. **Changez impérativement** les valeurs marquées `CHANGE_ME` :

```
cat > .env << 'EOF'
# === gitrust — configuration minimale ===
# Changez toutes les valeurs CHANGE_ME avant de démarrer

# Clé secrète de l'application (générez avec : openssl rand -hex 32)
SECRET_KEY=CHANGE_ME_openssl_rand_hex_32

# URL publique de l'instance (sans slash final)
APP_URL=http://localhost:4000

# Base de données PostgreSQL
DATABASE_URL=postgres://gitrust:CHANGE_ME_db_password@postgres:5432/gitrust
POSTGRES_USER=gitrust
POSTGRES_PASSWORD=CHANGE_ME_db_password
POSTGRES_DB=gitrust

# Serveur SSH (port exposé à l'extérieur)
SSH_PORT=2222

# Mode de l'application
RUST_LOG=info
APP_ENV=production
EOF
```

Générez maintenant une clé secrète robuste et remplacez la valeur dans `.env` :

```
SECRET=$(openssl rand -hex 32)
sed -i "s/CHANGE_ME_openssl_rand_hex_32/$SECRET/" .env
```

Choisissez également un mot de passe pour PostgreSQL et remplacez les deux occurrences :

```
DB_PASS=$(openssl rand -hex 16)
sed -i "s/CHANGE_ME_db_password/$DB_PASS/g" .env
```


© Gitrust

Sortie attendue (vérification) :

```
grep SECRET_KEY .env
# → SECRET_KEY=4a7f2c9e1b8d3f6a0e5c2b9d7f4a1e8c3b6d9f2a5e8c1b4d7f0a3e6c9b2d5f8a
```

Checkpoint : exécutez `grep CHANGE_ME .env` — la commande doit ne rien retourner. Si elle retourne des lignes, ces valeurs n'ont pas été remplacées et le démarrage échouera.

Étape 2 : Créez le fichier docker-compose.yml

Créez le fichier `docker-compose.yml` suivant dans `/opt/gitrust/`. Ce contenu correspond à la stack minimale gitrust — il est reproduit ici intégralement pour que vous puissiez démarrer sans dépendance externe :

```
cat > docker-compose.yml << 'EOF'
# docker-compose.minimal.yml — Stack gitrust minimale (app + PostgreSQL)
# Cas d'usage : évaluation, développement, petite instance (<10 utilisateurs)
# Pour la production : utilisez docker-compose.production.yml (+ Redis + SMTP)
#
# Usage :
#   docker compose up -d
#   docker compose logs -f gitrust
#   docker compose down

version: "3.9"

services:
  # ── Base de données ──────────────────────────────────────────────
  postgres:
    image: postgres:16-alpine
    container_name: gitrust_postgres
    restart: unless-stopped
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      # Données persistantes — ne supprimez pas ce volume sans sauvegarde
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
      interval: 10s
```


© Gitrust

```
      timeout: 5s
      retries: 5
    networks:
      - gitrust_net

  # ── Application gitrust ──────────────────────────────────────────
  gitrust:
    image: gitrust:latest
    container_name: gitrust_app
    restart: unless-stopped
    depends_on:
      postgres:
        condition: service_healthy
    env_file:
      # Lit toutes les variables depuis .env — ne commitez jamais ce fichier
      - .env
    ports:
      # Interface web — exposée sur toutes les interfaces de la machine hôte
      - "4000:4000"
      # Serveur SSH Git — exposé sur le port configuré dans .env
      - "${SSH_PORT:-2222}:2222"
    volumes:
      # Dépôts Git bare — données critiques, sauvegardez ce volume
      - git_repos:/var/lib/gitrust/repositories
      # Clés SSH du serveur — générées au premier démarrage, persistez-les
      - ssh_host_keys:/etc/gitrust/ssh
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:4000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 30s
    networks:
      - gitrust_net

volumes:
  # postgres_data : données relationnelles (comptes, dépôts, issues, PRs)
  postgres_data:
  # git_repos : dépôts Git bare (le contenu réel des commits)
  git_repos:
  # ssh_host_keys : clés SSH hôte (fingerprint de votre serveur)
  ssh_host_keys:
```


© Gitrust

```
networks:
  gitrust_net:
    driver: bridge
EOF
```

Checkpoint : vérifiez que les deux fichiers sont présents dans `/opt/gitrust/` :

```
ls -la /opt/gitrust/
```

Sortie attendue :

```
total 16
drwxr-xr-x 2 root root 4096 avr 17 10:00 .
drwxr-xr-x 8 root root 4096 avr 17 09:55 ..
-rw------- 1 root root  612 avr 17 10:00 .env
-rw-r--r-- 1 root root 2048 avr 17 10:00 docker-compose.yml
```

Étape 3 : Démarrez la stack et suivez les logs

Démarrez tous les services en arrière-plan :

```
docker compose up -d
```

Sortie attendue :

```
[+] Running 4/4
 ✔ Network gitrust_gitrust_net    Created                    0.1s
 ✔ Volume "gitrust_postgres_data" Created                    0.0s
 ✔ Container gitrust_postgres     Started                    0.8s
 ✔ Container gitrust_app          Started                    1.2s
```

Suivez les logs de l'application pour observer le démarrage (les migrations s'exécutent automatiquement) :

```
docker compose logs -f gitrust
```


© Gitrust

Sortie attendue (les 15-20 premières secondes) :

```
gitrust_app  | 2026-04-17T10:00:15Z INFO  gitrust > Starting gitrust v0.9.0
gitrust_app  | 2026-04-17T10:00:15Z INFO  gitrust > Connecting to database...
gitrust_app  | 2026-04-17T10:00:16Z INFO  gitrust > Running database migrations...
gitrust_app  | 2026-04-17T10:00:17Z INFO  gitrust > Applied 42 migrations successfully
gitrust_app  | 2026-04-17T10:00:17Z INFO  gitrust > Generating SSH host keys...
gitrust_app  | 2026-04-17T10:00:17Z INFO  gitrust > SSH host key fingerprint: SHA256:xx
gitrust_app  | 2026-04-17T10:00:17Z INFO  gitrust > HTTP server listening on 0.0.0.0:40
gitrust_app  | 2026-04-17T10:00:17Z INFO  gitrust > SSH server listening on 0.0.0.0:222
gitrust_app  | 2026-04-17T10:00:17Z INFO  gitrust > gitrust is ready
```

Appuyez sur **Ctrl+C** pour quitter le suivi des logs (les containers continuent de tourner).

Checkpoint : vérifiez l'état de santé des services :

```
docker compose ps
```

Sortie attendue :

```
NAME              IMAGE                          COMMAND                  SERVICE    CR
gitrust_app       gitrust:latest "/usr/local/bin/gitr…"  gitrust    2 minutes ago   Up
gitrust_postgres  postgres:16-alpine             "docker-entrypoint.s…"  postgres   2 m
```

Les deux containers doivent afficher `(healthy)`. Si l'un affiche `(starting)`, attendez 30 secondes et relancez la commande. S'il affiche `(unhealthy)`, consultez les logs avec `docker compose logs gitrust` ou `docker compose logs postgres`.

Étape 4 : Créez le premier compte administrateur

Ouvrez votre navigateur et rendez-vous sur `http://localhost:4000`.

Vous devriez voir la page d'accueil gitrust avec un formulaire d'inscription. Comme l'instance est vierge, **le premier compte créé devient automatiquement administrateur**.

Remplissez le formulaire : - **Pseudo** : votre identifiant administrateur (ex. `admin`) - **E-mail** : votre adresse e-mail - **Mot de passe** : choisissez un mot de passe robuste (≥ 12 caractères)

Cliquez sur **Créer mon compte**.

© Gitrust

Sortie attendue dans les logs (visible avec `docker compose logs -f gitrust`) :

```
gitrust_app  | 2026-04-17T10:05:00Z INFO  gitrust::auth > New user registered: admin (i
gitrust_app  | 2026-04-17T10:05:00Z INFO  gitrust::admin > First user promoted to admin
```

Checkpoint final : connectez-vous avec vos identifiants. Après la connexion, vérifiez que le menu en haut à droite affiche un lien **Administration** (ou naviguez vers `http://localhost:4000/admin`). Cette page n'est accessible qu'aux administrateurs — sa présence confirme que votre compte a bien le rôle admin.

L'URL de votre instance est `http://localhost:4000` et votre compte administrateur est opérationnel.

Récapitulatif

- √ **O1 accompli** : la stack docker-compose est déployée — `docker compose ps` affiche deux containers `(healthy)`
- √ **O2 accompli** : les trois services (app, PostgreSQL, réseau interne) démarrent correctement — les logs montrent « gitrust is ready » et les migrations sont appliquées
- √ **O3 accompli** : le premier compte administrateur est créé et la page `/admin` est accessible

© Gitrust

Et si ça ne marche pas

Symptôme	Cause probable	Correction
`Error: port 4000 is already in use` au démarrage	Un autre service occupe le port 4000 sur la machine hôte	Identifiez le service avec `ss -tlnp | grep 4000` ou `lsof -i :4000`. Arrêtez-le ou changez le port dans `docker-compose.yml` : remplacez `"4000:4000"` par `"4001:4000"` et accédez sur `:4001`
Le container `gitrust_app` affiche `(unhealthy)` ou redémarre en boucle	La variable `DATABASE_URL` est incorrecte ou `SECRET_KEY` contient des caractères spéciaux non échappés	Vérifiez avec `docker compose logs gitrust | grep ERROR`. Assurez-vous que `SECRET_KEY` est bien une chaîne hexadécimale sans guillemets. Rechargez avec `docker compose down && docker compose up -d`
La page `http://localhost:4000` affiche « Connection refused »	Le container n'est pas encore en `(healthy)` ou le port n'est pas exposé	Attendez 30 secondes et rechargez. Si le problème persiste, vérifiez que votre pare-feu local (`ufw status`) autorise le port 4000
`FATAL: password authentication failed for user "gitrust"` dans les logs postgres	Le mot de passe dans `DATABASE_URL` ne correspond pas à `POSTGRES_PASSWORD`	Supprimez le volume PostgreSQL (`docker compose down -v`) et relancez — **attention : supprime toutes les données**

Prochaine étape

→ 02 — Installation systemd : binaire natif + service : découvrez l'installation sans Docker pour un déploiement production avec hardening AppArmor (~45 min)

Ou si vous souhaitez rendre cette instance accessible depuis Internet :

→ 04 — Mise en production : reverse-proxy TLS, SMTP, sauvegarde automatique (~90 min)

© Gitrust

1.2.2 Installer gitrust comme service systemd (binaire natif)

Objectifs

À la fin de ce tutoriel, vous saurez :

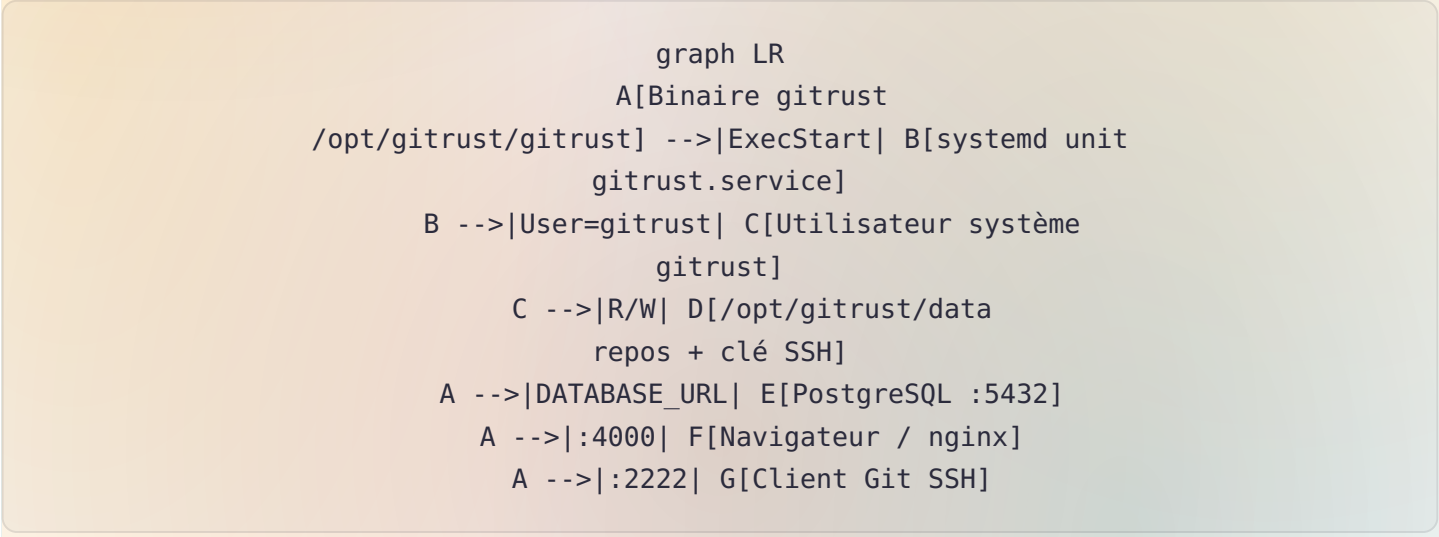
- O1. Créer un utilisateur système `gitrust` dédié et structurer l'arborescence `/opt/gitrust`
- O2. Configurer PostgreSQL pour gitrust et vérifier la connexion
- O3. Écrire et activer un fichier `gitrust.service` avec durcissement systemd
- O4. Démarrer gitrust et confirmer que le serveur HTTP et SSH répondent

Pré-requis

- **Technique** : Debian 12 ou Ubuntu 22.04 LTS (ou dérivé) ; accès root ou sudo ; `curl` , `rsync` , `psql` installés ; Docker **non** requis (c'est précisément l'alternative binaire natif)
- **Pédagogique** : Tutoriel 01-installation-docker lu ou notions équivalentes d'administration Linux
- **Temps estimé** : ~45 minutes

Vue d'ensemble

L'installation binaire natif place gitrust comme un service systemd classique, au même titre que nginx ou PostgreSQL. Il n'y a aucun conteneur : le binaire Rust s'exécute directement sous un utilisateur système dédié, avec un durcissement systemd (PrivateTmp, ProtectSystem, NoNewPrivileges).



Modèle mental : pensez à gitrust comme à nginx — un processus unique, lancé par systemd, propriétaire de ses données, sans jamais tourner en root.

© Gitrust

Étape 1 : Créer l'utilisateur et l'arborescence

Créez un utilisateur système sans shell de login ni répertoire home personnel :

```
sudo useradd \
  --system \
  --home-dir /opt/gitrust \
  --shell /usr/sbin/nologin \
  --comment "Gitrust service" \
  gitrust
```

Créez l'arborescence de travail :

```
sudo mkdir -p /opt/gitrust/data/repos
sudo mkdir -p /opt/gitrust/static
sudo chown -R gitrust:gitrust /opt/gitrust
sudo chmod 750 /opt/gitrust
```

Sortie attendue (vérification) :

```
$ ls -la /opt/gitrust/
total 20
drwxr-x--- 4 gitrust gitrust 4096 ...
drwxr-xr-x 3 gitrust gitrust 4096 ...  data/
drwxr-xr-x 2 gitrust gitrust 4096 ...  static/
```

Checkpoint O1 : `id gitrust` doit retourner une ligne avec `uid=NNN(gitrust) gid=NNN(gitrust)`. Si la commande échoue, relisez l'étape 1.

Étape 2 : Installer PostgreSQL et créer la base gitrust

Installez PostgreSQL si ce n'est pas fait :

```
sudo apt-get update
sudo apt-get install -y postgresql postgresql-contrib
sudo systemctl enable --now postgresql
```

Créez l'utilisateur et la base de données :

© Gitrust

```
sudo -u postgres psql <<'SQL'
CREATE USER gitrust WITH PASSWORD 'CHOISIR_UN_MOT_DE_PASSE_FORT';
CREATE DATABASE gitrust OWNER gitrust;
GRANT ALL PRIVILEGES ON DATABASE gitrust TO gitrust;
SQL
```

Vérifiez la connexion :

```
psql "postgres://gitrust:CHOISIR_UN_MOT_DE_PASSE_FORT@localhost:5432/gitrust" \
  -c "SELECT version();"
```

Sortie attendue :

```
                                                 version
-----------------------------------------------------------------------------------------
 PostgreSQL 15.x on x86_64-pc-linux-gnu, ...
(1 row)
```

Checkpoint O2 : Si `psql` retourne `version()`, la base est accessible. Conservez le mot de passe PostgreSQL pour l'étape suivante.

Étape 3 : Déployer le binaire gitrust

Sur votre machine de développement (ou sur le serveur si vous compilez localement) :

```
# Depuis la racine du dépôt gitrust
cargo build --release
```

Puis transférez vers le serveur :

```
# Depuis la machine de développement
rsync -avz target/release/gitrust  user@VOTRE_SERVEUR:/opt/gitrust/gitrust
rsync -avz static/                 user@VOTRE_SERVEUR:/opt/gitrust/static/
```

Sur le serveur, rendez le binaire exécutable et attribuez-le à l'utilisateur `gitrust` :

```
```


© Gitrust

```
sudo chown gitrust:gitrust /opt/gitrust/gitrust
sudo chmod 750 /opt/gitrust/gitrust
sudo chown -R gitrust:gitrust /opt/gitrust/static/
```

> ***Alternative** : si vous utilisez `deployment/deploy.sh` (fourni dans le dépôt source), il effectue les étapes 1 à 5 automatiquement. Ce tutoriel décompose les étapes pour que vous compreniez ce que le script fait.*

Étape 4 : Créer le fichier de configuration `.env`

Créez `/opt/gitrust/.env` avec les valeurs minimales de production :

```
sudo -u gitrust tee /opt/gitrust/.env <<'EOF'
# === BASE DE DONNÉES ===
DATABASE_URL=postgres://gitrust:CHOISIR_UN_MOT_DE_PASSE_FORT@localhost:5432/gitrust

# === SERVEUR HTTP ===
SERVER_HOST=127.0.0.1
SERVER_PORT=4000
RUST_LOG=info

# === SERVEUR SSH ===
SSH_PORT=2222
SSH_LISTEN_ADDR=0.0.0.0
SSH_HOST_KEY_PATH=/opt/gitrust/data/ssh_host_ed25519_key
SSH_PUBLIC_HOST=VOTRE_DOMAINE_OU_IP

# === DÉPÔTS GIT ===
GIT_REPOS_BASE_PATH=/opt/gitrust/data/repos

# === FICHIERS STATIQUES ===
STATIC_FILES_PATH=/opt/gitrust/static

# === JWT (OBLIGATOIRE — générer avec : openssl rand -hex 64) ===
JWT_SECRET=REMPLACER_PAR_OPENSSL_RAND_HEX_64
JWT_EXPIRATION_MINUTES=15
JWT_ISSUER=gitrust

# === SESSIONS ===
SESSION_TIMEOUT_MINUTES=30
SESSION_BACKEND=seaorm

# === COOKIES ===
```


© Gitrust

```
COOKIE_SECURE=true
COOKIE_SAME_SITE=Lax
APP_DEBUG=false

# === ADMIN INITIAL ===
ADMIN_USERNAME=admin
ADMIN_EMAIL=admin@VOTRE_DOMAINE
ADMIN_PASSWORD=REMPLACER_PAR_MOT_DE_PASSE_FORT

# === EMAIL (optionnel au démarrage initial, configurer ensuite) ===
# SMTP_HOST=smtp.example.com
# SMTP_PORT=587
EMAIL_BASE_URL=https://VOTRE_DOMAINE
EOF
```

Sécurisez le fichier (il contient des secrets) :

```
sudo chmod 600 /opt/gitrust/.env
sudo chown gitrust:gitrust /opt/gitrust/.env
```

Générez le vrai secret JWT :

```
openssl rand -hex 64
```

Copiez la sortie et remplacez `REMPLACER_PAR_OPENSSL_RAND_HEX_64` dans `/opt/gitrust/.env`.

Checkpoint : `sudo -u gitrust stat /opt/gitrust/.env` doit afficher `Mode: 0600` et `Uid: gitrust`.

Étape 5 : Créer le fichier `gitrust.service`

Créez le fichier d'unité systemd :

```
sudo tee /etc/systemd/system/gitrust.service <<'EOF'
[Unit]
Description=Gitrust - Self-hosted Git platform
Documentation=https://demo.gitrust.eu/docs/
After=network.target postgresql.service
Requires=postgresql.service

[Service]
Type=simple
```


© Gitrust

```
User=gitrust
Group=gitrust
WorkingDirectory=/opt/gitrust
EnvironmentFile=/opt/gitrust/.env
ExecStart=/opt/gitrust/gitrust
Restart=on-failure
RestartSec=5
TimeoutStartSec=30
TimeoutStopSec=30

# --- Durcissement systemd ---
NoNewPrivileges=true
ProtectSystem=strict
ProtectHome=true
PrivateTmp=true
PrivateDevices=true
ProtectKernelTunables=true
ProtectKernelModules=true
ProtectControlGroups=true
RestrictRealtime=true
RestrictSUIDSGID=true
LockPersonality=true

# Répertoires en écriture autorisés
ReadWritePaths=/opt/gitrust/data

# Si gitrust écoute sur un port < 1024 (ex: 80) :
# AmbientCapabilities=CAP_NET_BIND_SERVICE

[Install]
WantedBy=multi-user.target
EOF
```

Rechargez systemd et activez le service :

```
sudo systemctl daemon-reload
sudo systemctl enable gitrust
```

Sortie attendue :

© Gitrust

```
Created symlink /etc/systemd/system/multi-user.target.wants/gitrust.service → /etc/syst
```

Checkpoint O3 : `systemctl is-enabled gitrust` doit afficher `enabled`.

Étape 6 : Premier démarrage et vérification

Démarrez le service :

```
sudo systemctl start gitrust
```

Attendez 3 secondes, puis vérifiez le statut :

```
sudo systemctl status gitrust
```

Sortie attendue :

```
● gitrust.service - Gitrust - Self-hosted Git platform
     Loaded: loaded (/etc/systemd/system/gitrust.service; enabled; vendor preset: enabl
     Active: active (running) since ... ago
   Main PID: XXXX (gitrust)
      Tasks: N (limit: NNNN)
     Memory: NNM
        CPU: NNNms
     CGroup: /system.slice/gitrust.service
             └─XXXX /opt/gitrust/gitrust
```

Vérifiez que les ports répondent :

```
# HTTP
curl -s -o /dev/null -w "%{http_code}" http://localhost:4000/
# Attendu : 200 ou 302

# SSH (vérification du fingerprint)
ssh-keyscan -p 2222 -H localhost 2>/dev/null | head -1
# Attendu : [localhost]:2222 ssh-ed25519 AAAA...
```

Consultez les logs pour les migrations automatiques :

© Gitrust

```
sudo journalctl -u gitrust -n 50 --no-pager
```

Cherchez les lignes :

```
INFO gitrust_core::migrations: Running migrations...
INFO gitrust_core::migrations: Applied migration m20260325_000001_create_repositories
...
INFO gitrust_core::migrations: All migrations applied successfully
INFO rustwarden_core::services::default_admin_service: Admin user 'admin' created
```

Checkpoint O4 : `curl http://localhost:4000/` retourne du HTML (code 200 ou 302). Le service est opérationnel.

Récapitulatif

- √ O1 accompli en créant l'utilisateur système `gitrust` et l'arborescence `/opt/gitrust`
- √ O2 accompli en configurant PostgreSQL et en vérifiant la connexion avec `psql`
- √ O3 accompli en rédigeant `gitrust.service` avec les directives de durcissement systemd
- √ O4 accompli en démarrant le service et en confirmant la réponse HTTP et SSH

© Gitrust

Et si ça ne marche pas

Symptôme	Cause probable	Correction
`Active: failed` immédiatement après `start`	`.env` manquant ou mal formé ; `DATABASE_URL` incorrecte	`journalctl -u gitrust -n 30 --no-pager` pour voir l'erreur exacte ; vérifier que `/opt/gitrust/.env` existe et que `DATABASE_URL` est correct
`Permission denied` dans les logs	Répertoire `data/` non accessible par l'utilisateur `gitrust`	`sudo chown -R gitrust:gitrust /opt/gitrust/data`
`Address already in use` sur le port 4000	Un autre processus utilise le port	`sudo ss -tlnp | grep 4000` pour identifier le conflit
`JWT_SECRET trop court` / refus de démarrage	Le secret JWT est la valeur d'exemple ou fait moins de 32 bytes	Générer avec `openssl rand -hex 64` et mettre à jour `.env`
Les migrations échouent (`migration error`)	Base de données inaccessible ou droits insuffisants	Vérifier `DATABASE_URL` et que l'utilisateur PG a `GRANT ALL ON DATABASE gitrust`
`curl: (7) Failed to connect`	Le service n'est pas encore démarré ou écoute sur `127.0.0.1` uniquement	Tester depuis le serveur (pas depuis l'extérieur) ; vérifier `SERVER_HOST=127.0.0.1`

Prochaine étape

→ 03 — Premier démarrage : bootstrap admin, clé SSH hôte, vérification santé

© Gitrust

1.2.3 Premier démarrage : bootstrap admin, clé SSH hôte et vérification santé

Objectifs

À la fin de ce tutoriel, vous saurez :

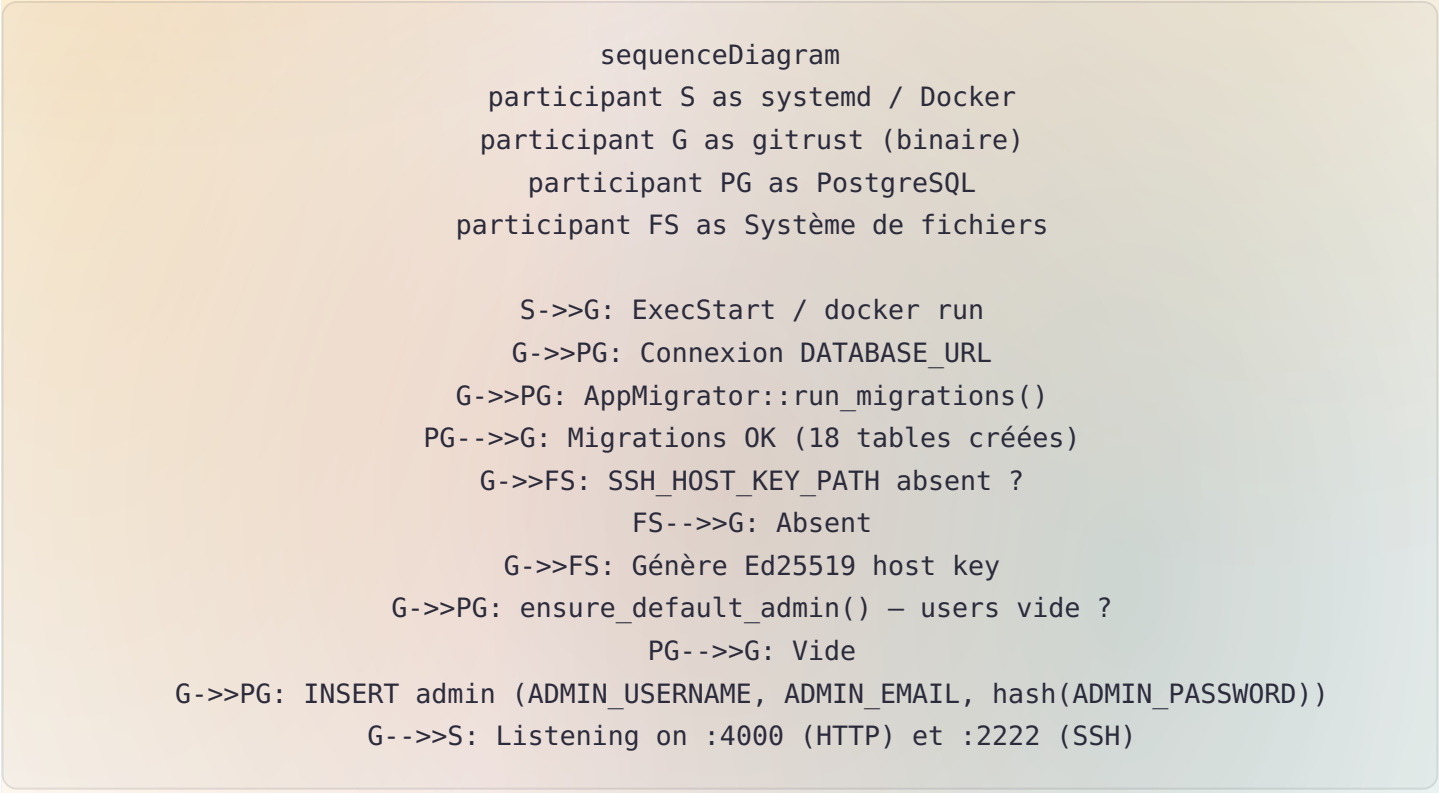
- O1. Expliquer comment gitrust crée automatiquement le compte administrateur initial à partir des variables `ADMIN_*`
- O2. Identifier et vérifier la clé SSH hôte Ed25519 générée automatiquement au premier démarrage
- O3. Confirmer que les trois services (HTTP :4000, SSH :2222, PostgreSQL :5432) sont sains via des commandes de vérification
- O4. Vous connecter à l'interface d'administration `/admin` pour la première fois

Pré-requis

- **Technique** : gitrust installé et démarré via 01-installation-docker ou 02-installation-systemd ; `curl`, `ssh-keyscan`, `psql` disponibles
- **Pédagogique** : Tutoriel 01 ou 02 complété — gitrust doit être `active (running)`
- **Temps estimé** : ~20 minutes

Vue d'ensemble

Le premier démarrage de gitrust effectue automatiquement trois opérations critiques, dans cet ordre :



© Gitrust

Modèle mental : c'est le même principe qu'un routeur domestique — à la première mise sous tension, il crée un mot de passe admin par défaut (ici fourni par vous via `.env`) et ne recommence jamais s'il existe déjà des données.

Étape 1 : Vérifier les migrations automatiques

Les migrations sont appliquées à chaque démarrage, mais seulement si elles n'ont pas encore été jouées. Elles sont idempotentes.

Consultez les logs :

```
# Installation systemd
sudo journalctl -u gitrust --since "10 minutes ago" --no-pager | grep -E "migrat|admin|

# Installation Docker
docker compose logs gitrust 2>&1 | grep -E "migrat|admin|listen"
```

Sortie attendue (premier démarrage) :

```
INFO gitrust_core::migrations: Running migrations...
INFO gitrust_core::migrations: Applied migration m20260305_000001_initial_schema
INFO gitrust_core::migrations: Applied migration m20260306_000002_create_app_settings_t
...
INFO gitrust_core::migrations: Applied migration m20260327_000013_create_pr_comments
INFO gitrust_core::migrations: All migrations applied successfully (18 applied)
INFO rustwarden_core::services::default_admin_service: Admin user 'admin' created
INFO gitrust_web: HTTP server listening on 127.0.0.1:4000
INFO gitrust_ssh: SSH server listening on 0.0.0.0:2222
```

Sortie attendue (démarrage suivant) :

```
INFO gitrust_core::migrations: Running migrations...
INFO gitrust_core::migrations: No pending migrations
```

Checkpoint O1 : si vous voyez `Admin user 'admin' created`, le bootstrap admin a fonctionné. Si vous voyez `Admin user already exists, skipping`, un compte admin existait déjà — c'est normal si ce n'est pas le tout premier démarrage.

Étape 2 : Vérifier la clé SSH hôte

gitrust génère une clé Ed25519 à `SSH_HOST_KEY_PATH` si elle n'existe pas. Cette clé identifie votre instance auprès des clients Git.

© Gitrust

Vérifiez que la clé existe :

```
# Systemd
sudo -u gitrust ls -la /opt/gitrust/data/ssh_host_ed25519_key*

# Docker (le volume est monté dans /data dans le conteneur)
docker compose exec gitrust ls -la /data/ssh_host_ed25519_key*
```

Sortie attendue :

```
-rw------- 1 gitrust gitrust  399 ...  /opt/gitrust/data/ssh_host_ed25519_key
-rw-r--r-- 1 gitrust gitrust   93 ...  /opt/gitrust/data/ssh_host_ed25519_key.pub
```

Affichez le fingerprint SHA256 (à conserver précieusement — vos utilisateurs vérifieront ce fingerprint lors du premier `git clone`) :

```
# Systemd
sudo -u gitrust ssh-keygen -l -f /opt/gitrust/data/ssh_host_ed25519_key.pub

# Docker
docker compose exec gitrust ssh-keygen -l -f /data/ssh_host_ed25519_key.pub
```

Sortie attendue :

```
256 SHA256:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX gitrust-host-key (ED25519)
```

> ***Important*** *: notez ce fingerprint SHA256 dans votre documentation interne. Vos utilisateurs devront le comparer lors du premier* `ssh-keyscan` *ou lors du message* `The authenticity of host ... can't be established`*. Si vous régénérez cette clé après que des utilisateurs ont cloné des dépôts, leur* `~/.ssh/known_hosts` *sera en conflit et tous les push/pull échoueront.*

Checkpoint O2 : les deux fichiers existent avec les bonnes permissions (clé privée `600`, clé publique `644`).

Étape 3 : Vérifier la santé des trois services

HTTP :4000

© Gitrust

```
curl -v http://localhost:4000/ 2>&1 | grep -E "^< HTTP|Location"
```

Sortie attendue :

```
< HTTP/1.1 302 Found
< location: /login
```

Une redirection vers `/login` signifie que gitrust répond et que le routeur fonctionne.

SSH :2222

```
ssh-keyscan -p 2222 -H localhost 2>/dev/null
```

Sortie attendue :

```
[localhost]:2222 ssh-ed25519 AAAA... (votre clé hôte)
```

Si aucune sortie, le serveur SSH n'écoute pas encore — consultez les logs.

PostgreSQL :5432

```
# Depuis l'hôte
psql "postgres://gitrust:VOTRE_MOT_DE_PASSE@localhost:5432/gitrust" \
  -c "SELECT COUNT(*) AS nb_tables FROM information_schema.tables WHERE table_schema='p
```

Sortie attendue :

```
 nb_tables
-----------
        18
(1 row)
```

18 tables = toutes les migrations ont été appliquées.

Checkpoint O3 : les trois commandes répondent comme attendu. Si l'une échoue, consultez la section « Et si ça ne marche pas ».

© Gitrust

Étape 4 : Premier accès à l'interface d'administration

Ouvrez votre navigateur sur `http://VOTRE_SERVEUR:4000/login` (ou `https://` si vous avez déjà configuré un reverse-proxy TLS).

Connectez-vous avec : - **Identifiant** : valeur de `ADMIN_USERNAME` dans votre `.env` (par défaut : `admin`) - **Mot de passe** : valeur de `ADMIN_PASSWORD` dans votre `.env`

Après connexion, accédez au tableau de bord d'administration :

`http://VOTRE_SERVEUR:4000/admin`

Vous devriez voir : - Le nombre d'utilisateurs (1 — l'admin initial) - Le nombre de dépôts (0) - L'état des services CI

> ***Sécurité** : changez immédiatement le mot de passe admin via `/admin/users` ou le profil utilisateur si vous avez utilisé un mot de passe faible dans `.env`. Ensuite, vous pouvez supprimer ou commenter `ADMIN_PASSWORD` dans `.env` — gitrust ne recrée le compte admin que si la table `users` est vide.*

Checkpoint O4 : vous êtes connecté à `/admin` et le tableau de bord s'affiche sans erreur.

© Gitrust

Récapitulatif

- √ O1 accompli en lisant les logs de démarrage qui confirment les migrations et la création du compte admin
- √ O2 accompli en vérifiant l'existence et le fingerprint SHA256 de la clé SSH hôte Ed25519
- √ O3 accompli en testant HTTP (:4000 → 302), SSH (:2222 → fingerprint) et PostgreSQL (:5432 → 18 tables)
- √ O4 accompli en se connectant à `/admin` avec les identifiants `ADMIN_*` du `.env`

Et si ça ne marche pas

Symptôme	Cause probable	Correction
`HTTP/1.1 Connection refused` sur le port 4000	gitrust n'a pas démarré ou écoute sur un autre port	`systemctl status gitrust` ou `docker compose ps` ; vérifier `SERVER_PORT` dans `.env`
Logs : `JWT_SECRET trop court` ou `invalid JWT secret`	Le secret JWT est la valeur exemple du template	Générer avec `openssl rand -hex 64` et redémarrer
`ssh-keyscan` ne retourne rien	Le port SSH est bloqué par un firewall ou gitrust n'a pas démarré le serveur SSH	`sudo ufw allow 2222/tcp` (si ufw actif) ; vérifier `SSH_PORT` dans `.env`
`Admin user already exists` mais impossible de se connecter	Le mot de passe dans `.env` n'est pas celui du compte en base (déjà modifié)	Réinitialiser via `psql` : `UPDATE users SET password_hash = '...' WHERE username = 'admin'` — ou utiliser la procédure de reset
18 tables manquantes (moins de 18 en base)	Migrations interrompues	`journalctl -u gitrust -n 100` pour voir l'erreur ; souvent un problème de droits PG ou de connexion interrompue

Prochaine étape

→ 04 — Mise en production : reverse-proxy TLS, SMTP, sauvegarde

Ou, si vous voulez configurer les fonctionnalités avancées maintenant : - Configurer SMTP - Configurer OAuth - Sauvegarder l'instance

© Gitrust

1.2.4 Mise en production sur une machine Linux avec HTTPS Let's Encrypt

Contexte

Procédure de référence pour déployer une instance gitrust sur une machine Linux à IP fixe, avec patte externe routée et HTTPS via Let's Encrypt. Reproductible et testée en conditions réelles.

Adaptez les valeurs suivantes à votre environnement :

Placeholder	Remplacez par
`<your-server-ip>`	L'IP fixe de votre serveur (ex: `192.168.1.57` sur un LAN, ou une IP publique)
`<your-domain>`	Votre FQDN principal (ex: `gitrust.example.com`)
`<acme-email>`	L'email pour Let's Encrypt (ex: `contact@example.com`)
`<admin-user>`	L'utilisateur SSH admin sur le serveur cible

Les trois services gitrust

#	Service	Bind interne	Exposition publique	Variable
1	Web HTTP (UI SSR + API)	`127.0.0.1:4000`	`:443` HTTPS via Nginx	`SERVER_PORT`
2	SSH Git (russh)	`127.0.0.1:2222`	`:22` via Nginx stream	`SSH_PORT`
3	Worker CI/CD (Dagger)	tâche Tokio in-process	— (déclenché par push)	`CI_ENABLED`

PostgreSQL = dépendance, conteneur Docker bind loopback `127.0.0.1:5432`.

Topologie réseau

```
Internet
    |
    v
<your-server-ip> (patte externe)
    |
    +-- :22   Nginx stream  --> 127.0.0.1:2222 (gitrust SSH)
    +-- :80   Nginx HTTP    --> 301 HTTPS (sauf /.well-known/acme-challenge)
    +-- :443  Nginx HTTPS   --> 127.0.0.1:4000 (gitrust HTTP)
    +-- :2022 sshd système  (accès admin, déplacé depuis :22)
```


© Gitrust

```
    Loopback uniquement :
    +-- 127.0.0.1:4000  gitrust web
    +-- 127.0.0.1:2222  gitrust SSH
    +-- 127.0.0.1:5432  postgres docker
```

Risque critique — ordre d'exécution

Le déplacement de `sshd` système de `:22` vers `:2022` **doit** être fait avant la configuration `nginx stream`, et **doit** être validé depuis une **deuxième session SSH** avant fermeture de la première. Sinon : lockout total (besoin d'accès console physique).

Pré-requis

Poste de build (machine locale avec le code source)

```
rustup show && command -v rsync envsubst npx
# Installer si besoin : sudo apt install rsync gettext-base
```

Cible `<your-server-ip>`

- Debian/Ubuntu récent
- Accès SSH avec `sudo`
- Docker + plugin compose (`docker compose version`)
- Snap ou apt pour `certbot` + plugin nginx
- DNS `<your-domain>` → IP publique de la patte externe (vérifier `dig +short <your-domain>`)
- Module `stream` Nginx disponible (paquet `libnginx-mod-stream` sur Debian)

Phase 0 — Générer `.env.production` (poste de build, une seule fois)

```
JWT=$(openssl rand -hex 64)
ADMIN_PWD=$(openssl rand -base64 24)
PG_PWD=$(openssl rand -base64 24)

cat > .env.production <<EOF
# ---- Base de données ----
DATABASE_URL=postgres://gitrust:${PG_PWD}@127.0.0.1:5432/gitrust

# ---- Serveur HTTP (derrière Nginx HTTPS) ----
SERVER_HOST=127.0.0.1
```


© Gitrust

```
SERVER_PORT=4000

# ---- Serveur SSH (derrière nginx stream :22) ----
SSH_PORT=2222
SSH_LISTEN_ADDR=127.0.0.1
SSH_PUBLIC_HOST=<your-domain>
SSH_HOST_KEY_PATH=/opt/gitrust/data/ssh_host_ed25519_key

# ---- Chemins (réécrits par deploy.sh) ----
GIT_REPOS_BASE_PATH=/opt/gitrust/data/repos
STATIC_FILES_PATH=/opt/gitrust/static

# ---- Sécurité — HTTPS actif ----
JWT_SECRET=${JWT}
JWT_EXPIRATION_MINUTES=15
REFRESH_TOKEN_EXPIRATION_DAYS=7
COOKIE_SECURE=true            # HTTPS via Nginx
COOKIE_SAME_SITE=Lax
APP_DEBUG=false
RUST_LOG=info

# ---- Admin initial ----
ADMIN_USERNAME=admin
ADMIN_EMAIL=<acme-email>
ADMIN_PASSWORD=${ADMIN_PWD}

# ---- Inscription & UI ----
ALLOW_REGISTRATION=false
APP_NAME=Gitrust
DEFAULT_LOCALE=fr

# ---- Email ----
EMAIL_VALIDATION_REQUIRED=false
EMAIL_BASE_URL=https://<your-domain>

# ---- CI/CD Dagger (worker 3) ----
CI_ENABLED=true
CI_MAX_CONCURRENT=4
CI_DEFAULT_TIMEOUT=3600
CI_WORKSPACE_PATH=/tmp/gitrust-ci
CI_REMOTE_HOST=localhost

# ---- Worker import ----
```


© Gitrust

```
IMPORT_MAX_CONCURRENT=2
IMPORT_TIMEOUT_SECS=1800
DB_WORKER_POOL_SIZE=4
EOF

chmod 600 .env.production
echo "ADMIN_PASSWORD = ${ADMIN_PWD}"
echo "POSTGRES_PASSWORD = ${PG_PWD}"
```

> ***Critique** : noter `ADMIN_PASSWORD` et `POSTGRES_PASSWORD` dans un gestionnaire de mots de passe.*

Phase 1 — Préparer la cible : déplacer sshd système sur :2022

À faire en gardant deux sessions SSH simultanées.

1.pre. Diagnostic de l'état initial

1.pre.1 — Trouver le port SSH ouvert depuis le poste de build

```
# Depuis votre poste (sans être connecté) :
for p in 22 2022 2222 2200 22022; do
  timeout 3 bash -c "echo >/dev/tcp/<your-server-ip>/$p" 2>/dev/null \
    && echo "Port $p : OUVERT" \
    || echo "Port $p : fermé/filtré"
done
```

1.pre.2 — Se connecter sur le port détecté

```
# Cas le plus probable : sshd est sur :22 par défaut
ssh <admin-user>@<your-server-ip>
```

1.pre.3 — Une fois connecté, vérifier l'état sshd actuel

```
# Sur quel(s) port(s) sshd écoute-t-il ?
sudo ss -tlnp | grep sshd

# Contenu de la config
sudo grep -riE '^[[:space:]]*Port[[:space:]]+[0-9]+' /etc/ssh/sshd_config /etc/ssh/sshd
```


© Gitrust

```
# Statut du service
sudo systemctl status ssh --no-pager
```

1a. Session 1 : modifier la config sshd

```
ssh <admin-user>@<your-server-ip>

# Éditer /etc/ssh/sshd_config
sudo tee /etc/ssh/sshd_config.d/99-gitrust.conf <<'EOF'
Port 2022
EOF

# Vérifier la syntaxe
sudo sshd -t

# Si OK, reload (ne déconnecte pas la session courante)
sudo systemctl reload ssh
sudo ss -tlnp | grep -E ':22\b|:2022\b'
# Attendu : sshd écoute sur 22 ET 2022
```

1b. Session 2 : valider le nouveau port

Depuis le poste de build, **dans un autre terminal** :

```
ssh -p 2022 <admin-user>@<your-server-ip> 'echo "OK port 2022"'
# Doit afficher : OK port 2022
```

1c. (Plus tard, après Phase 5) Fermer le port 22 du sshd système

Une fois `nginx stream :22` actif et testé :

```
ssh -p 2022 <admin-user>@<your-server-ip>
sudo sed -i 's/^Port 22$/# Port 22 (déplacé vers 2022, :22 utilisé par nginx stream)/'
sudo systemctl restart ssh
```

1d. Firewall

```
ssh -p 2022 <admin-user>@<your-server-ip> bash <<'EOF'
sudo ufw allow 80/tcp comment 'HTTP ACME + redirect'
sudo ufw allow 443/tcp comment 'HTTPS Gitrust'
sudo ufw allow 22/tcp comment 'SSH Git via nginx stream'
```


© Gitrust

```
sudo ufw allow 2022/tcp comment 'SSH admin système'
sudo ufw status numbered
EOF
```

> *Mettre à jour `~/.ssh/config` côté poste dev :*
>
> ```
> Host gitrust-host
>     HostName <your-server-ip>
>     User <admin-user>
>     Port 2022
> ```
>
> *Le reste du plan utilise `ssh gitrust-host`.*

Phase 2 — PostgreSQL (Docker, loopback)

```
scp -P 2022 database/docker-compose.yml <admin-user>@<your-server-ip>:/tmp/pg-compose.y

ssh gitrust-host bash <<EOF
sudo mkdir -p /opt/gitrust/database
sudo mv /tmp/pg-compose.yml /opt/gitrust/database/docker-compose.yml
echo 'POSTGRES_PASSWORD=${PG_PWD}' | sudo tee /opt/gitrust/database/.env >/dev/null
sudo chmod 600 /opt/gitrust/database/.env
cd /opt/gitrust/database && sudo docker compose up -d
sudo docker compose ps
EOF
```

Remplacer `${PG_PWD}` par la valeur réelle. Vérifier que le port n'est pas exposé publiquement :

```
ssh gitrust-host 'sudo ss -tlnp | grep 5432'
# Attendu : 127.0.0.1:5432 — PAS 0.0.0.0
```

Phase 3 — Build + déploiement gitrust via `deploy.sh`

3a. Préparer `deployment/deploy.conf`

© Gitrust

```
cp deployment/deploy.conf.example deployment/deploy.conf
# Éditer DEPLOY_TARGET=gitrust-host, DEPLOY_REMOTE_PATH=/opt/gitrust, DEPLOY_FQDN_INITI
$EDITOR deployment/deploy.conf
chmod 600 deployment/deploy.conf
```

3b. Validation du build seul (optionnel)

```
BUILD_ONLY=1 ./deployment/deploy.sh
```

3c. Déploiement

```
# Sans argument : utilise DEPLOY_TARGET du deploy.conf (gitrust-host via ~/.ssh/config,
./deployment/deploy.sh
```

Ce que fait `deploy.sh` : `cargo build --release`, `tailwindcss --minify`, adapte `gitrust.service` + `.env`, rsync, crée user `gitrust`, installe systemd, restart.

À la fin : gitrust écoute sur `127.0.0.1:4000` (HTTP) et `127.0.0.1:2222` (SSH).

```
ssh gitrust-host 'sudo systemctl status gitrust --no-pager && sudo ss -tlnp | grep -E "
# Attendu : 127.0.0.1:4000 et 127.0.0.1:2222 (PAS 0.0.0.0)
```

Phase 4 — Nginx HTTP + Let's Encrypt

4a. Installer nginx + certbot

```
ssh gitrust-host bash <<'EOF'
sudo apt update
sudo apt install -y nginx libnginx-mod-stream certbot python3-certbot-nginx
sudo systemctl enable --now nginx
EOF
```

4b. Déployer la conf Nginx

```
scp -P 2022 deployment/nginx-gitrust.conf <admin-user>@<your-server-ip>:/tmp/gitrust-ng

ssh gitrust-host bash <<EOF
sudo sed -i 's/gitrust.nuage.ebii/<your-domain>/g' /tmp/gitrust-nginx.conf
```


© Gitrust

```
sudo mv /tmp/gitrust-nginx.conf /etc/nginx/sites-available/gitrust
sudo ln -sf /etc/nginx/sites-available/gitrust /etc/nginx/sites-enabled/gitrust
sudo rm -f /etc/nginx/sites-enabled/default
sudo mkdir -p /var/www/html/.well-known/acme-challenge
sudo nginx -t && sudo systemctl reload nginx
EOF
```

4c. Émettre le certificat Let's Encrypt

```
ssh gitrust-host 'sudo certbot --nginx \
    -d <your-domain> \
    --email <acme-email> \
    --agree-tos \
    --no-eff-email \
    --redirect'
```

4d. Vérifier le renouvellement automatique

```
ssh gitrust-host 'sudo systemctl list-timers | grep certbot'
ssh gitrust-host 'sudo certbot renew --dry-run'
```

4e. Test HTTPS

```
curl -I https://<your-domain>/
# Attendu : HTTP/2 200 (ou 302 vers /login)

# Vérifier HSTS
curl -sI https://<your-domain>/ | grep -i strict-transport
```

Phase 5 — Exposer SSH Git sur :22 via Nginx stream

5a. Ajouter le bloc stream

```
ssh gitrust-host bash <<'EOF'
sudo nginx -V 2>&1 | grep -o with-stream

sudo tee -a /etc/nginx/nginx.conf <<'NGINX'

# --- Gitrust SSH (proxy :22 -> :2222) ----------------------------------------
```


© Gitrust

```
stream {
    upstream gitrust_ssh {
        server 127.0.0.1:2222;
    }
    server {
        listen 22;
        listen [::]:22;
        proxy_pass gitrust_ssh;
        proxy_timeout 1h;
        proxy_connect_timeout 30s;
        error_log /var/log/nginx/gitrust-ssh.log;
    }
}
NGINX

sudo nginx -t && sudo systemctl restart nginx
sudo ss -tlnp | grep -E ':22\b|:2222\b'
# Attendu : nginx :22, gitrust :2222 (loopback)
EOF
```

5b. Désactiver sshd système sur :22 (Phase 1c)

```
ssh -p 2022 <admin-user>@<your-server-ip> bash <<'EOF'
sudo sed -i 's/^Port 22$/# Port 22 déplacé/' /etc/ssh/sshd_config 2>/dev/null || true
sudo systemctl restart ssh
sudo ss -tlnp | grep -E ':22\b|:2022\b'
# Attendu : nginx :22, sshd :2022 — plus de sshd:22
EOF
```

5c. Test clone Git

```
# Dans l'UI gitrust : Settings -> SSH keys -> coller ~/.ssh/id_ed25519.pub
# Créer un repo test admin/test-deploy via UI

git clone git@<your-domain>:admin/test-deploy.git /tmp/test-deploy
cd /tmp/test-deploy
echo "# test" > README.md
git -c user.email=<acme-email> -c user.name=admin add . && \
  git -c user.email=<acme-email> -c user.name=admin commit -m "test deploy" && \
  git push origin main
```


© Gitrust

Phase 6 — Vérifications globales

```
# Service systemd
ssh gitrust-host 'sudo systemctl status gitrust --no-pager'

# Logs (recherche d'erreurs)
ssh gitrust-host 'sudo journalctl -u gitrust -n 100 --no-pager | grep -iE "error|warn|f

# Ports d'écoute (vue d'ensemble)
ssh gitrust-host 'sudo ss -tlnp'
# Attendu :
#   :22    nginx
#   :80    nginx
#   :443   nginx
#   :2022  sshd système
#   127.0.0.1:2222  gitrust
#   127.0.0.1:4000  gitrust
#   127.0.0.1:5432  docker-proxy

# Renouvellement TLS automatique
ssh gitrust-host 'sudo systemctl status certbot.timer --no-pager'

# Test E2E
curl -I https://<your-domain>/login
ssh -T git@<your-domain> || true   # Banner SSH russh attendu
```

Phase 7 — Durcissement : Fail2ban

Voir `admin/how-to/durcir-avec-fail2ban.md` pour la configuration complète.

7a. Installation

```
ssh gitrust-host bash <<'EOF'
sudo apt install -y fail2ban
sudo systemctl enable --now fail2ban
EOF
```

7b. Configuration minimale rapide

© Gitrust

```
ssh gitrust-host sudo tee /etc/fail2ban/jail.local <<'EOF'
[DEFAULT]
backend  = systemd
bantime  = 1h
findtime = 10m
maxretry = 5
ignoreip = 127.0.0.1/8 ::1
banaction          = ufw
banaction_allports = ufw

[sshd]
enabled  = true
port     = 2022
filter   = sshd
backend  = %(sshd_backend)s
logpath  = %(sshd_log)s
maxretry = 3

[nginx-http-auth]
enabled  = true
port     = http,https
filter   = nginx-http-auth
logpath  = /var/log/nginx/gitrust.error.log

[gitrust-login]
enabled  = true
port     = http,https
filter   = gitrust-login
logpath  = /var/log/nginx/gitrust.access.log
maxretry = 5

[gitrust-ssh]
enabled      = true
port         = 22,2222
filter       = gitrust-ssh
backend      = systemd
journalmatch = _SYSTEMD_UNIT=gitrust.service
maxretry     = 5

[recidive]
enabled   = true
logpath   = /var/log/fail2ban.log
banaction = %(banaction_allports)s
```


© Gitrust

```
bantime   = 1w
findtime  = 1d
maxretry  = 3
EOF
```

Créer les filtres gitrust-login et gitrust-ssh, puis :

```
ssh gitrust-host 'sudo systemctl restart fail2ban && sudo fail2ban-client status'
```

Phase 8 — Mise à jour vers un nouveau domaine (quand DNS prêt)

8a. Ajouter le SAN au certificat existant

```
ssh gitrust-host 'sudo certbot --nginx --expand \
    -d <your-domain> \
    -d <new-domain> \
    --email <acme-email> \
    --agree-tos --no-eff-email'
```

8b. Mettre à jour la conf Nginx

```
ssh gitrust-host bash <<EOF
sudo sed -i 's/server_name <your-domain>;/server_name <your-domain> <new-domain>;/g' /e
sudo nginx -t && sudo systemctl reload nginx
EOF
```

8c. Mettre à jour `.env.production` et redéployer

```
sed -i 's|SSH_PUBLIC_HOST=.*|SSH_PUBLIC_HOST=<new-domain>|' .env.production
sed -i 's|EMAIL_BASE_URL=.*|EMAIL_BASE_URL=https://<new-domain>|' .env.production
./deployment/deploy.sh gitrust-host
```

Phase 9 — Mises à jour ultérieures

© Gitrust

```
git pull
SKIP_ENV=1 SKIP_DB=1 ./deployment/deploy.sh
```

Dépannage

Symptôme	Diagnostic	Fix
Lockout SSH après reload sshd	—	Console physique / KVM IPMI
Cert Let's Encrypt échoue (`Connection refused`)	`dig +short <your-domain>` ≠ IP publique	Corriger DNS, ou vérifier que :80 traverse bien le NAT
Cert échoue (`unauthorized`)	Nginx `:80` ne répond pas sur `/.well-known/acme-challenge/`	`curl http://<your-domain>/.well-known/acme-challenge/test` doit ne pas rediriger en 301
Login boucle sur `/login` malgré HTTPS	`grep COOKIE_SECURE /opt/gitrust/.env`	Doit être `true` ; vérifier que Nginx forward bien `X-Forwarded-Proto https`
Push SSH `Connection closed`	`journalctl -u gitrust | grep ssh`	Vérifier que gitrust écoute bien `127.0.0.1:2222` et que nginx stream forward
`nginx -t` : `unknown directive "stream"`	Module non chargé	`sudo apt install libnginx-mod-stream` puis `systemctl restart nginx`
Mixed content HTTPS	Templates qui hardcodent `http://`	Vérifier `EMAIL_BASE_URL=https://...` et headers `X-Forwarded-Proto`
Push gros repo timeout	Logs nginx `client intended to send too large body`	Augmenter `client_max_body_size 2G` (déjà fait pour `.git/`)

© Gitrust

Récapitulatif des ports

Port	Process	Exposition	Rôle
22	nginx (stream)	Public	SSH Git → forward vers :2222
80	nginx	Public	ACME challenge + redirect HTTPS
443	nginx	Public	HTTPS → forward vers :4000
2022	sshd système	Public	Admin SSH (à restreindre par firewall si possible)
2222	gitrust	Loopback	Backend SSH russh
4000	gitrust	Loopback	Backend HTTP axum
5432	docker-proxy	Loopback	PostgreSQL

© Gitrust

1.3 Guides pratiques

1.3.1 Sauvegarder et restaurer une instance gitrust

À qui s'adresse cette page

Administrateurs qui souhaitent mettre en place une sauvegarde fiable et tester la restauration. Pour comprendre la stratégie 3-2-1 adaptée à gitrust, consultez Stratégie de sauvegarde.

Ce qu'il faut sauvegarder

Gitrust repose sur trois sources de vérité distinctes :

Source	Chemin / objet	Critique	Fréquence recommandée
Base de données PostgreSQL	database `gitrust`	Oui	Quotidien minimum
Dépôts Git bare	`GIT_REPOS_BASE_PATH` (ex. `/opt/gitrust/data/repos`)	Oui	Quotidien minimum
Clé SSH hôte	`SSH_HOST_KEY_PATH` (ex. `/opt/gitrust/data/ssh_host_ed25519_key`)	Oui	Une seule fois, puis conserver
Fichier `.env`	`/opt/gitrust/.env`	Oui	À chaque modification

> *La clé SSH hôte est immuable après le premier démarrage. Si vous la perdez, tous vos utilisateurs devront supprimer leur entrée `known_hosts` et re-vérifier le fingerprint.*

Sauvegarder

Sauvegarde PostgreSQL (`pg_dump`)

```
BACKUP_DIR="/var/backups/gitrust"
DATE=$(date +%Y%m%d-%H%M%S)
mkdir -p "$BACKUP_DIR"
```


© Gitrust

```
pg_dump \
  -h localhost \
  -U gitrust \
  --format=custom \
  --file="${BACKUP_DIR}/pg-${DATE}.dump" \
  gitrust

echo "Dump : ${BACKUP_DIR}/pg-${DATE}.dump"
# Purger les sauvegardes de plus de 30 jours
find "$BACKUP_DIR" -name "pg-*.dump" -mtime +30 -delete
```

Le format `--format=custom` est compressé et restaurable avec `pg_restore`. Le fichier contient la structure ET les données.

Sauvegarde des dépôts bare (rsync)

```
REPOS_SRC="/opt/gitrust/data/repos"
REPOS_DST="/var/backups/gitrust/repos"

rsync \
  --archive \
  --delete \
  --hard-links \
  --checksum \
  "${REPOS_SRC}/" "${REPOS_DST}/"
```

Les dépôts bare Git sont consistants en lecture même pendant un `git push` actif : Git garantit l'atomicité au niveau du pack.

Sauvegarde de la clé SSH hôte et du `.env`

```
BACKUP_DIR="/var/backups/gitrust"
DATE=$(date +%Y%m%d-%H%M%S)

tar czf "${BACKUP_DIR}/config-${DATE}.tar.gz" \
  /opt/gitrust/data/ssh_host_ed25519_key \
  /opt/gitrust/data/ssh_host_ed25519_key.pub \
  /opt/gitrust/.env

chmod 600 "${BACKUP_DIR}/config-${DATE}.tar.gz"
```


© Gitrust

Script complet `backup.sh`

```
#!/usr/bin/env bash
# /opt/gitrust/scripts/backup.sh
set -euo pipefail

BACKUP_DIR="${BACKUP_DIR:-/var/backups/gitrust}"
DATE=$(date +%Y%m%d-%H%M%S)
PG_USER="gitrust"
PG_DB="gitrust"
REPOS_SRC="/opt/gitrust/data/repos"
SSH_KEY="/opt/gitrust/data/ssh_host_ed25519_key"
ENV_FILE="/opt/gitrust/.env"

mkdir -p "$BACKUP_DIR"

echo "[1/3] Dump PostgreSQL..."
pg_dump -U "$PG_USER" --format=custom \
  --file="${BACKUP_DIR}/pg-${DATE}.dump" "$PG_DB"

echo "[2/3] Rsync dépôts bare..."
mkdir -p "${BACKUP_DIR}/repos"
rsync --archive --delete --checksum \
  "${REPOS_SRC}/" "${BACKUP_DIR}/repos/"

echo "[3/3] Archive config sensible..."
tar czf "${BACKUP_DIR}/config-${DATE}.tar.gz" \
  "$SSH_KEY" "${SSH_KEY}.pub" "$ENV_FILE"
chmod 600 "${BACKUP_DIR}/config-${DATE}.tar.gz"

echo "Sauvegarde terminée dans $BACKUP_DIR"
find "$BACKUP_DIR" -name "pg-*.dump" -mtime +30 -delete
find "$BACKUP_DIR" -name "config-*.tar.gz" -mtime +90 -delete
```

Planifier via cron :

```
# Sauvegarder chaque nuit à 2h00
0 2 * * * /opt/gitrust/scripts/backup.sh >> /var/log/gitrust-backup.log 2>&1
```


© Gitrust

Restaurer

Avant toute restauration

```
# 1. Arrêter gitrust
sudo systemctl stop gitrust   # ou : docker compose stop gitrust

# 2. Vérifier les fichiers de sauvegarde disponibles
ls -lh /var/backups/gitrust/pg-*.dump | tail -5
ls -lh /var/backups/gitrust/config-*.tar.gz | tail -3
du -sh /var/backups/gitrust/repos/
```

Restaurer la base PostgreSQL

```
DUMP="/var/backups/gitrust/pg-20260417-020000.dump"

sudo -u postgres psql -c "DROP DATABASE IF EXISTS gitrust;"
sudo -u postgres psql -c "CREATE DATABASE gitrust OWNER gitrust;"

pg_restore \
  -h localhost \
  -U gitrust \
  --dbname=gitrust \
  --verbose \
  "$DUMP"
```

Vérification :

```
psql "postgres://gitrust:MOT_DE_PASSE@localhost:5432/gitrust" \
  -c "SELECT COUNT(*) FROM users; SELECT COUNT(*) FROM repositories;"
```

Restaurer les dépôts bare

```
REPOS_SRC="/var/backups/gitrust/repos"
REPOS_DST="/opt/gitrust/data/repos"

rsync --archive --delete "${REPOS_SRC}/" "${REPOS_DST}/"
sudo chown -R gitrust:gitrust "${REPOS_DST}"
```


© Gitrust

Restaurer la clé SSH hôte et le `.env`

```
ARCHIVE="/var/backups/gitrust/config-20260417-020000.tar.gz"

sudo tar xzf "$ARCHIVE" -C /
sudo chown gitrust:gitrust /opt/gitrust/data/ssh_host_ed25519_key*
sudo chmod 600 /opt/gitrust/data/ssh_host_ed25519_key
sudo chmod 644 /opt/gitrust/data/ssh_host_ed25519_key.pub
sudo chmod 600 /opt/gitrust/.env
```

Redémarrer et vérifier

```
sudo systemctl start gitrust
sleep 3
curl -s -o /dev/null -w "%{http_code}" http://localhost:4000/
# Attendu : 302
```

Tester la restauration (drill trimestriel)

Une sauvegarde non testée n'est pas une sauvegarde. Testez sur une VM distincte :

```
# Sur une VM de test vierge — restaurer, puis vérifier :
git clone ssh://git@VM_TEST:2222/admin/un-depot-connu.git /tmp/drill-test
ls /tmp/drill-test/
# Les fichiers doivent correspondre à l'état au moment de la sauvegarde
```

Pour aller plus loin

- Stratégie de sauvegarde — comprendre le 3-2-1 adapté à gitrust
- Variables d'environnement : `GIT_REPOS_BASE_PATH`, `SSH_HOST_KEY_PATH`

© Gitrust

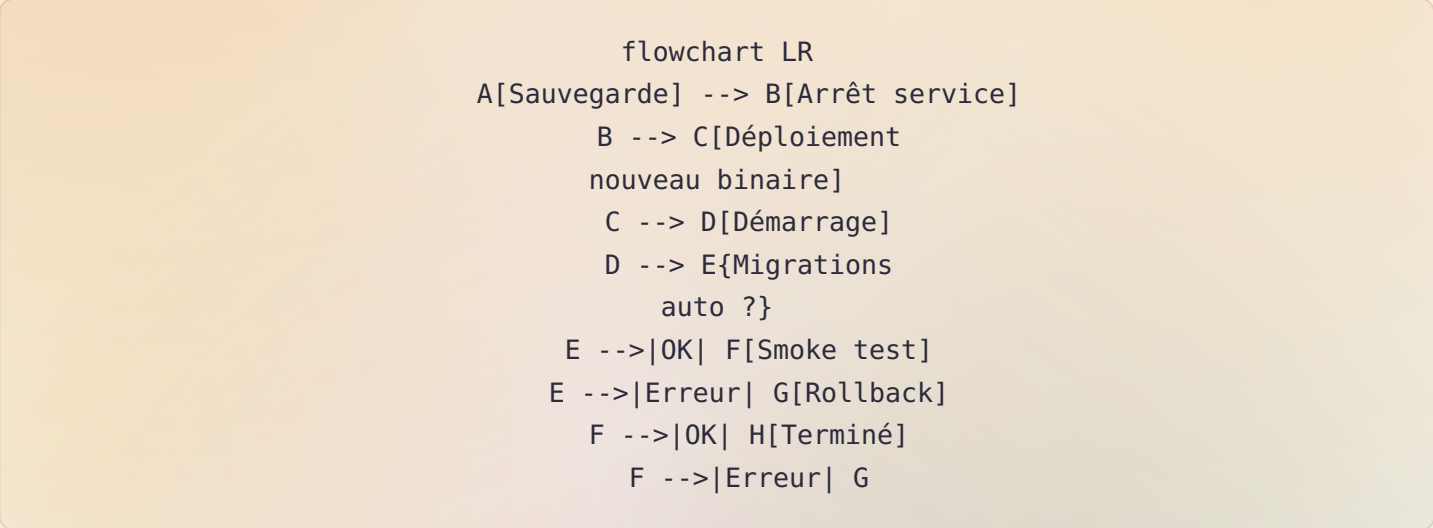
1.3.2 Mettre à jour gitrust

À qui s'adresse cette page

Administrateurs qui souhaitent appliquer une nouvelle version de gitrust sur une instance en production, en limitant les risques.

Principe général

Les migrations de base de données sont appliquées **automatiquement au démarrage** par `AppMigrator`. Vous n'avez jamais à exécuter de commande SQL manuellement. Le binaire contient toujours les migrations cumulatives.



Checklist de mise à jour

1. Sauvegarder avant tout

```
# Sauvegarde complète avant toute modification
/opt/gitrust/scripts/backup.sh
```

Vérifiez que le dump PostgreSQL a été créé :

```
ls -lh /var/backups/gitrust/pg-$(date +%Y%m%d)*.dump
```

Ne passez à l'étape 2 que si la sauvegarde est confirmée.

© Gitrust

2. Arrêter le service

```
# Systemd
sudo systemctl stop gitrust

# Docker Compose
docker compose stop gitrust
```

Attendez la confirmation :

```
sudo systemctl is-active gitrust
# Attendu : inactive
```

3. Déployer le nouveau binaire

Option A — via `deploy.sh` (recommandé si vous compilez localement) :

```
# Sur la machine de développement, depuis la racine du dépôt gitrust
./deployment/deploy.sh user@VOTRE_SERVEUR /opt/gitrust
```

Le script compile en release, synchronise le binaire, les assets statiques et le service systemd, puis redémarre automatiquement.

Option B — manuellement :

```
# Compiler le nouveau binaire
cargo build --release

# Compiler le CSS si les templates ont changé
npx tailwindcss -i static/css/input.css -o static/css/style.css --minify

# Transférer
rsync -avz target/release/gitrust  user@VOTRE_SERVEUR:/opt/gitrust/gitrust
rsync -avz static/                 user@VOTRE_SERVEUR:/opt/gitrust/static/

# Sur le serveur
sudo chown gitrust:gitrust /opt/gitrust/gitrust
sudo chmod 750 /opt/gitrust/gitrust
```


© Gitrust

4. Vérifier les éventuels changements de `.env`

Consultez le `CHANGELOG` de la nouvelle version pour identifier les nouvelles variables d'environnement. Ajoutez-les à `/opt/gitrust/.env` si nécessaire, avec des valeurs appropriées.

```
# Comparer votre .env avec .env.example de la nouvelle version
diff /opt/gitrust/.env .env.example | grep "^>"
```

5. Démarrer le service

```
# Systemd
sudo systemctl start gitrust

# Docker Compose
docker compose up -d gitrust
```

6. Vérifier les migrations automatiques

```
sudo journalctl -u gitrust --since "2 minutes ago" --no-pager | grep -E "migrat|error|p
```

Sortie attendue si de nouvelles migrations ont été appliquées :

```
INFO gitrust_core::migrations: Applied migration m20260501_000014_add_webhooks
INFO gitrust_core::migrations: All migrations applied successfully
```

Sortie attendue si aucune nouvelle migration :

```
INFO gitrust_core::migrations: No pending migrations
```

Si vous voyez `ERROR` ou `panic`, consultez la section Rollback.

7. Smoke test

```
# HTTP
curl -s -o /dev/null -w "%{http_code}" http://localhost:4000/
# Attendu : 302
```


© Gitrust

```
# SSH (fingerprint inchangé — comparer avec la valeur notée)
ssh-keyscan -p 2222 -H localhost 2>/dev/null | ssh-keygen -l -f -
```

Connectez-vous à l'interface `/admin` et vérifiez : - La liste des utilisateurs est correcte - Un dépôt existant est accessible - La page `/admin/settings` s'affiche sans erreur

---

Rollback

Si le démarrage échoue ou si le smoke test révèle une régression :

1. Arrêter le service

```
sudo systemctl stop gitrust
```

2. Restaurer le binaire précédent

Si vous avez conservé l'ancien binaire (bonne pratique : le renommer avant la mise à jour) :

```
sudo cp /opt/gitrust/gitrust.backup /opt/gitrust/gitrust
sudo chown gitrust:gitrust /opt/gitrust/gitrust
```

3. Restaurer la base de données si des migrations ont été appliquées

```
# Identifier la dernière sauvegarde pré-mise-à-jour
DUMP=$(ls -t /var/backups/gitrust/pg-*.dump | head -1)
echo "Restauration depuis : $DUMP"

sudo -u postgres psql -c "DROP DATABASE IF EXISTS gitrust;"
sudo -u postgres psql -c "CREATE DATABASE gitrust OWNER gitrust;"

pg_restore -h localhost -U gitrust --dbname=gitrust "$DUMP"
```

4. Redémarrer avec l'ancienne version

```
sudo systemctl start gitrust
```

> ***Attention** : si des nouvelles migrations ont créé ou modifié des tables, le rollback de la base est indispensable avant de démarrer l'ancienne version. Un binaire ancien sur une base migratée peut produire des erreurs imprévisibles.*

© Gitrust

Conseils

- Conservez toujours le binaire précédent pendant 24 h avant de le supprimer (`mv gitrust gitrust.backup`).
- Planifiez les mises à jour en dehors des heures de pointe (le service est arrêté pendant l'opération).
- Les migrations sont idempotentes : redémarrer deux fois de suite avec la même version est sans risque.

Pour aller plus loin

- Sauvegarder et restaurer
- Variables d'environnement

© Gitrust

1.3.3 Configurer l'envoi d'e-mails (SMTP)

À qui s'adresse cette page

Administrateurs qui souhaitent activer l'envoi d'e-mails (validation de compte, réinitialisation de mot de passe, notifications). Sans configuration SMTP, gitrust démarre normalement mais toutes les fonctionnalités e-mail sont désactivées.

© Gitrust

Variables concernées

Toutes ces variables se placent dans `/opt/gitrust/.env` (ou dans l'`EnvironmentFile` de votre service systemd / `docker-compose.yml`). Un redémarrage du service est nécessaire après modification.

© Gitrust

Variable	Obligatoire	Défaut	Description
`SMTP_HOST`	Oui	—	Nom d'hôte ou IP du serveur SMTP
`SMTP_PORT`	Non	587	Port SMTP (25, 465, 587, 1025)
`SMTP_USER`	Non	—	Identifiant SMTP (souvent = adresse e-mail)
`SMTP_PASSWORD`	Non	—	Mot de passe applicatif SMTP
`SMTP_FROM`	Non	—	Adresse expéditeur visible (doit être valide)
`SMTP_FROM_NAME`	Non	—	Nom affiché dans le champ « De : »
`SMTP_USE_TLS`	Non	`true`	Activer TLS natif (port 465)
`SMTP_USE_STARTTLS`	Non	`true`	Activer STARTTLS (port 587)
`SMTP_ACCEPT_INVALID_CERTS`	Non	`false`	Accepter les certificats invalides (dev seulement)
`SMTP_CA_CERT_PATH`	Non	—	Chemin vers un CA personnalisé (PEM)
`EMAIL_BASE_URL`	Oui (prod)	`http://localhost:3000`	URL de base pour les liens dans les e-mails
`EMAIL_VALIDATION_REQUIRED`	Non	`true`	Exiger la validation e-mail à l'inscription
`EMAIL_QUEUE_BATCH_SIZE`	Non	10	Nombre d'e-mails traités par tick
`EMAIL_QUEUE_RETRY_ATTEMPTS`	Non	5	Nombre de tentatives avant abandon
`EMAIL_QUEUE_RETRY_DELAY_SECONDS`	Non	300	Délai entre deux tentatives (secondes)

© Gitrust

> ***Important** : Ne jamais activer `SMTP_ACCEPT_INVALID_CERTS=true` en production — cela ouvre la voie à des attaques MITM sur tous les e-mails sortants.*

Recettes par prestataire

Gmail (compte personnel ou Google Workspace)

Google exige un **mot de passe applicatif** (pas votre mot de passe Google). Générez-le sur myaccount.google.com/apppasswords en activant au préalable la validation en deux étapes.

```
SMTP_HOST=smtp.gmail.com
SMTP_PORT=587
SMTP_USER=votre.adresse@gmail.com
SMTP_PASSWORD=xxxx_xxxx_xxxx_xxxx   # mot de passe applicatif (16 caractères)
SMTP_FROM=votre.adresse@gmail.com
SMTP_FROM_NAME=Gitrust
SMTP_USE_TLS=false
SMTP_USE_STARTTLS=true
EMAIL_BASE_URL=https://VOTRE_DOMAINE
```

> ***Limite** : Gmail impose un quota de 500 e-mails/jour pour les comptes personnels. Pour les volumes importants, utilisez un service dédié (SendGrid, Brevo, etc.).*

OVH / Infomaniak / Gandi

Ces hébergeurs utilisent STARTTLS sur le port 587 ou SMTPS sur le port 465.

```
# OVH (SSL/TLS natif sur 465)
SMTP_HOST=ssl0.ovh.net
SMTP_PORT=465
SMTP_USER=noreply@VOTRE_DOMAINE.com
SMTP_PASSWORD=MOT_DE_PASSE_EMAIL
SMTP_FROM=noreply@VOTRE_DOMAINE.com
SMTP_FROM_NAME=Gitrust
SMTP_USE_TLS=true
SMTP_USE_STARTTLS=false
EMAIL_BASE_URL=https://VOTRE_DOMAINE
```


© Gitrust

```
# Gandi (STARTTLS sur 587)
SMTP_HOST=mail.gandi.net
SMTP_PORT=587
SMTP_USER=noreply@VOTRE_DOMAINE.com
SMTP_PASSWORD=MOT_DE_PASSE_EMAIL
SMTP_FROM=noreply@VOTRE_DOMAINE.com
SMTP_FROM_NAME=Gitrust
SMTP_USE_TLS=false
SMTP_USE_STARTTLS=true
EMAIL_BASE_URL=https://VOTRE_DOMAINE
```

Postfix local (instance interne sans authentification)

Pour les déploiements sur réseau interne où Postfix est installé sur le même serveur :

```
SMTP_HOST=127.0.0.1
SMTP_PORT=25
# Pas de SMTP_USER ni SMTP_PASSWORD si Postfix accepte sans auth sur loopback
SMTP_FROM=noreply@VOTRE_DOMAINE.com
SMTP_FROM_NAME=Gitrust
SMTP_USE_TLS=false
SMTP_USE_STARTTLS=false
EMAIL_BASE_URL=https://VOTRE_DOMAINE
```

Vérifiez que Postfix accepte les connexions depuis `127.0.0.1` :

```
# Dans /etc/postfix/main.cf
mynetworks = 127.0.0.0/8
```

Mailpit / Mailhog (développement local)

```
SMTP_HOST=localhost
SMTP_PORT=1025
SMTP_FROM=dev@localhost
SMTP_FROM_NAME=Gitrust-Dev
SMTP_USE_TLS=false
SMTP_USE_STARTTLS=false
SMTP_ACCEPT_INVALID_CERTS=false
EMAIL_BASE_URL=http://localhost:4000
EMAIL_VALIDATION_REQUIRED=false
```


© Gitrust

Démarrez Mailpit : `docker run -p 1025:1025 -p 8025:8025 axllent/mailpit`
Interface web : `http://localhost:8025`

Tester la configuration

Après avoir modifié le `.env` et redémarré gitrust, testez via le panel admin :

1. Connectez-vous à `/admin`
2. Accédez à `/admin/email`
3. Saisissez une adresse de test et cliquez **Envoyer un e-mail de test**

En cas d'erreur, consultez les logs :

```
sudo journalctl -u gitrust -n 50 --no-pager | grep -i "smtp\|email\|mail"
```


© Gitrust

Erreurs courantes :

Message dans les logs	Cause	Correction
`Connection refused` sur le port SMTP	`SMTP_HOST` ou `SMTP_PORT` incorrect	Vérifier les paramètres du prestataire
`Authentication failed`	`SMTP_USER` ou `SMTP_PASSWORD` incorrect	Pour Gmail : utiliser un mot de passe applicatif, pas le mot de passe du compte
`Certificate verification failed`	Certificat TLS du serveur SMTP invalide ou CA non reconnu	Spécifier `SMTP_CA_CERT_PATH` ou contacter le prestataire
`EMAIL_BASE_URL` pointe vers `localhost`	Liens dans les e-mails non cliquables pour les destinataires externes	Définir `EMAIL_BASE_URL=https://VOTRE_DOMAINE`

Pour aller plus loin

- Variables d'environnement — section Email
- Auditer l'instance — vérifier les bounces et la file d'envoi

© Gitrust

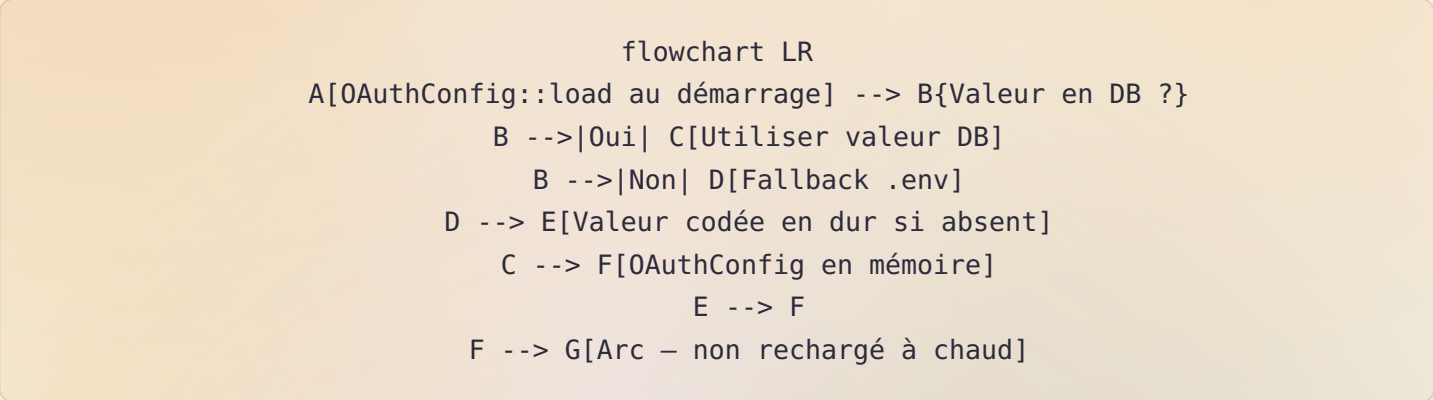
1.3.4 Configurer l'authentification OAuth (Google, GitHub, Discord, Microsoft)

À qui s'adresse cette page

Administrateurs qui souhaitent permettre à leurs utilisateurs de se connecter via un fournisseur OAuth externe plutôt que (ou en plus de) le couple identifiant/mot de passe local.

Principe

OAuth est le **seul sous-système hybride** de gitrust : la configuration peut venir soit de la base de données (via `/admin/settings`), soit du fichier `.env` en fallback. La priorité est toujours à la base de données.



> ***Conséquence** : toute modification de la configuration OAuth (via l'UI ou via `.env`) nécessite un **redémarrage** pour être prise en compte.*

Méthode recommandée : via l'interface `/admin/settings`

C'est la méthode la plus simple. Les secrets sont chiffrés en base (AES-256-GCM).

1. Connectez-vous à `/admin/settings`
2. Activez le toggle **OAuth / SSO activé**
3. Renseignez l'**URL de base des callbacks** (ex. `https://gitrust.votre-domaine.com`)
4. Pour chaque fournisseur souhaité, activez le toggle et renseignez `Client ID` et `Client Secret`
5. Cliquez **Enregistrer**
6. Redémarrez le service : `sudo systemctl restart gitrust`

© Gitrust

Capture a venir

220 admin settings oauth

voir scripts/screenshot-runner/

Méthode alternative : via les variables `.env`

Utile pour le provisionnement automatisé (Ansible, cloud-init) ou pour définir les valeurs au premier démarrage.

```
# Activation globale
OAUTH_ENABLED=true
OAUTH_REDIRECT_BASE_URL=https://gitrust.votre-domaine.com

# Clé de chiffrement des secrets OAuth en base (AES-256-GCM)
# Générer avec : openssl rand -hex 32
OAUTH_ENCRYPTION_KEY=REMPLACER_PAR_32_BYTES_HEX
```

Ces variables ne s'appliquent **qu'en l'absence** de valeur en base de données.

Configurer Google OAuth

1. Créer les identifiants dans Google Cloud Console

1. Ouvrez console.cloud.google.com → **APIs & Services** → **Credentials**
2. Cliquez **Create Credentials** → **OAuth 2.0 Client ID**
3. Type : **Web application**
4. Ajoutez l'URI de redirection autorisée :

   ```
   https://gitrust.votre-domaine.com/api/v1/auth/oauth/google/callback
   ```

5. Notez le **Client ID** et le **Client Secret**

© Gitrust

2. Configurer dans gitrust

Via `.env` :

```
OAUTH_GOOGLE_ENABLED=true
OAUTH_GOOGLE_CLIENT_ID=123456789-xxxxxxxxxxxx.apps.googleusercontent.com
OAUTH_GOOGLE_CLIENT_SECRET=GOCSPX-xxxxxxxxxxxxxxxxxxx
```

Via `/admin/settings` : activez « Google » et renseignez les mêmes valeurs.

Scopes utilisés : `openid`, `email`, `profile` (read-only, pas d'accès au compte Google).

Configurer GitHub OAuth

1. Créer l'application OAuth sur GitHub

1. Ouvrez github.com/settings/developers → **OAuth Apps** → **New OAuth App**
2. **Authorization callback URL** :

   ```
   https://gitrust.votre-domaine.com/api/v1/auth/oauth/github/callback
   ```

3. Notez le **Client ID** et générez un **Client Secret**

2. Configurer dans gitrust

```
OAUTH_GITHUB_ENABLED=true
OAUTH_GITHUB_CLIENT_ID=Iv1.xxxxxxxxxxxxxxxx
OAUTH_GITHUB_CLIENT_SECRET=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Scopes utilisés : `read:user`, `user:email`.

Configurer Discord OAuth

1. Créer l'application sur Discord Developer Portal

1. Ouvrez discord.com/developers/applications → **New Application**
2. Onglet **OAuth2** → **Redirects** → ajoutez :

   ```
   https://gitrust.votre-domaine.com/api/v1/auth/oauth/discord/callback
   ```

3. Notez le **Client ID** et le **Client Secret**

© Gitrust

2. Configurer dans gitrust

```
OAUTH_DISCORD_ENABLED=true
OAUTH_DISCORD_CLIENT_ID=123456789012345678
OAUTH_DISCORD_CLIENT_SECRET=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Configurer Microsoft / Azure AD

1. Créer l'application dans Azure

1. Ouvrez portal.azure.com → **Azure Active Directory** → **App registrations** → **New registration**
2. **Redirect URI** (Web) :

   ```
   https://gitrust.votre-domaine.com/api/v1/auth/oauth/microsoft/callback
   ```

3. Notez le **Application (client) ID** et créez un **Client Secret** dans **Certificates & Secrets**

2. Configurer dans gitrust

```
OAUTH_MICROSOFT_ENABLED=true
OAUTH_MICROSOFT_CLIENT_ID=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
OAUTH_MICROSOFT_CLIENT_SECRET=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# "common" = comptes personnels et professionnels
# Remplacer par le GUID du tenant pour restreindre à votre organisation
OAUTH_MICROSOFT_TENANT=common
```

Comportement lors de la connexion OAuth

Scénario	Comportement par défaut
Premier login OAuth, adresse e-mail inconnue	Compte créé automatiquement (`oauth_auto_register=true`)
Premier login OAuth, adresse e-mail déjà en base	Compte lié au compte existant (`oauth_link_existing_account=true`)
Login OAuth, compte désactivé par l'admin	Connexion refusée

© Gitrust

Ces comportements sont configurables dans `/admin/settings` (paramètres `oauth_auto_register` et `oauth_link_existing_account`).

Vérification après activation

```
# Redémarrer le service
sudo systemctl restart gitrust

# Vérifier les logs de démarrage
sudo journalctl -u gitrust -n 30 --no-pager | grep -i oauth
```

Ouvrez une fenêtre de navigation privée et accédez à `/login` : les boutons OAuth activés doivent apparaître sous le formulaire classique.

Pour aller plus loin

- Variables d'environnement — section OAuth
- Paramètres dynamiques `/admin/settings`

© Gitrust

1.3.5 Configurer le runner CI distant

Guide d'usage du script qui provisionne une machine distante pour exécuter les pipelines CI/CD de gitrust via SSH + rsync + Dagger.

1. Rôle du script

`deployment/setup-remote-ci.sh` (disponible dans le dépôt source gitrust) prépare **un serveur de build distant** utilisé par le worker CI de gitrust. Il automatise 6 étapes :

1. Vérifie la connectivité SSH vers le runner
2. Installe **Docker** (`curl https://get.docker.com | sh`) s'il est absent
3. Installe **Dagger CLI** (`curl https://dl.dagger.io/dagger/install.sh | sh`) s'il est absent
4. Crée le répertoire de travail distant (`CI_REMOTE_PATH`)
5. Synchronise le module `deployment/ci-engine/` (mode CI Easy) via `rsync -az --delete`
6. Smoke test : affiche `docker --version` et `dagger version`

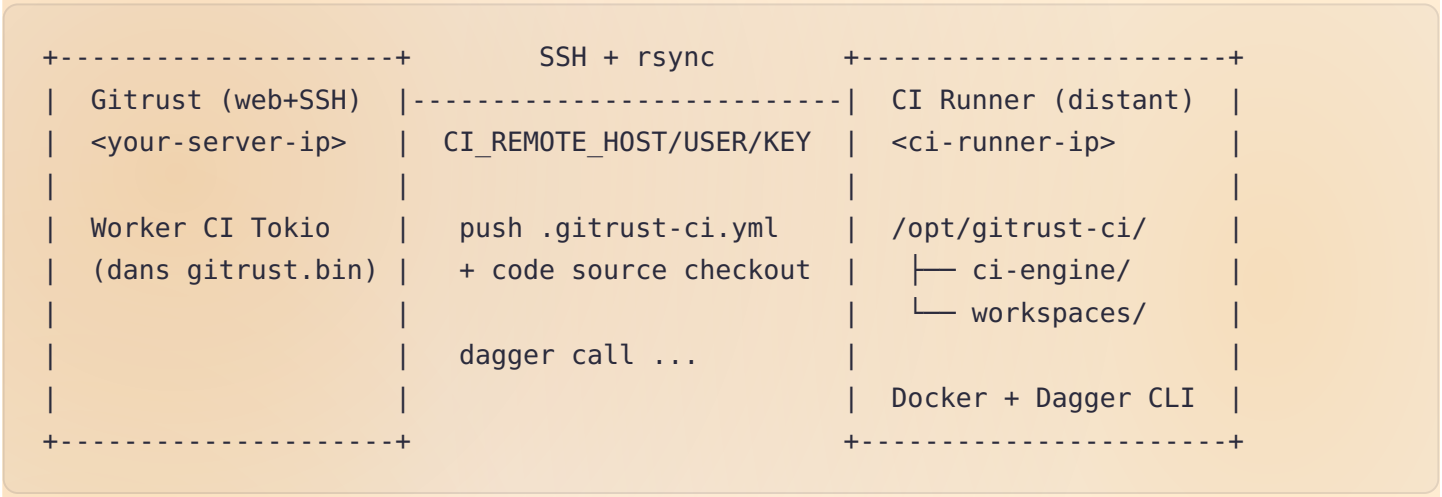
Le script est **idempotent** : rejoue sans dommage, skip ce qui existe déjà.

2. Quand l'utiliser

Scénario	Besoin de `setup-remote-ci.sh` ?
Dev local, CI sur la même machine que gitrust (`CI_REMOTE_HOST=localhost`)	**Non** — Docker et Dagger déjà installés localement, le `ci-engine/` est lu depuis `deployment/ci-engine/` directement
Prod on-premise, un seul serveur (gitrust + CI colocalisés)	**Non** — idem, un simple `curl get.docker.com | sh` suffit sur le serveur gitrust
Prod avec **runner CI dédié** (machine séparée)	**OUI** — c'est le cas cible de ce script
Prod avec plusieurs instances gitrust partageant un runner	**OUI** — le runner est provisionné une fois, chaque instance pointe dessus
CI dans Kubernetes / Nomad / runner managé cloud	**Non** — ce script est pensé pour une VM Debian/ Ubuntu classique

© Gitrust

3. Architecture cible



Le worker CI de gitrust (tâche Tokio dans le binaire principal) ne fait **pas tourner** Docker localement — il délègue au runner via SSH. Cela permet d'isoler les workloads CPU/RAM gourmands (builds Rust, Node, Docker) du process web de gitrust.

---

4. Pré-requis

Sur la machine d'exécution (là où on lance le script)

Outil	Rôle
`bash` ≥ 4	Interpréteur
`ssh`, `rsync`	Transport et synchro
`ssh-agent` chargé **ou** `CI_REMOTE_SSH_KEY` défini	Auth SSH sans mot de passe interactif (`BatchMode=yes`)
Fichier `.env` avec les variables CI_REMOTE_*	Config (voir section 5)

© Gitrust

Sur le runner distant

Pré-requis	Pourquoi
OS Linux récent (Debian 12+, Ubuntu 22.04+)	Docker install script compatible
User avec `sudo` passwordless **ou** droits `docker`	`get.docker.com` fait `sudo` en interne
Clé SSH publique du user local dans `~/.ssh/authorized_keys`	Auth SSH non-interactive
Réseau sortant autorisé vers `get.docker.com` et `dl.dagger.io`	Installation des binaires
Au minimum ~5 Go libres dans `CI_REMOTE_PATH`	Images Docker + workspaces

5. Variables d'environnement attendues (le `.env`)

Le script **source un fichier** `.env` dont le chemin est passé en argument, ou par défaut `../.env` (relatif au script, donc `<racine_projet>/.env`).

Tableau des variables

Variable	Obligatoire	Défaut	Description
`CI_REMOTE_HOST`	**OUI**	—	Hostname ou IP du runner (ex: `ci-runner.internal`)
`CI_REMOTE_USER`	non	`$(whoami)` (user courant)	Compte SSH sur le runner
`CI_REMOTE_SSH_PORT`	non	`22`	Port SSH du runner
`CI_REMOTE_PATH`	non	`/opt/gitrust-ci`	Répertoire de travail distant (créé par le script)
`CI_REMOTE_SSH_KEY`	non	— (ssh-agent)	Chemin d'une clé privée SSH spécifique

Ces variables sont les **mêmes** que celles lues par gitrust au runtime. Centraliser dans `.env` évite les divergences entre le provisionnement et le runtime.

Exemple minimal de `.env` (runner dédié)

© Gitrust

```
# --- CI runner distant ---
CI_REMOTE_HOST=<ci-runner-ip>
CI_REMOTE_USER=ci-runner
CI_REMOTE_SSH_PORT=22
CI_REMOTE_PATH=/opt/gitrust-ci
CI_REMOTE_SSH_KEY=/home/gitrust/.ssh/ci_runner_ed25519
```

Exemple `.env` complet coexistant avec config gitrust

```
DATABASE_URL=postgres://...
JWT_SECRET=...
# ... (voir .env.example pour le reste)

# --- CI/CD (lu par gitrust ET par setup-remote-ci.sh) ---
CI_ENABLED=true
CI_MAX_CONCURRENT=4
CI_REMOTE_HOST=ci-runner.internal
CI_REMOTE_USER=ci-runner
CI_REMOTE_PATH=/opt/gitrust-ci
CI_REMOTE_SSH_KEY=/opt/gitrust/data/ci_runner_key
```

Cas « runner local » (pas besoin du script)

```
CI_REMOTE_HOST=localhost
# Les autres CI_REMOTE_* sont ignorés
```

Dans ce cas, installer Docker et Dagger directement sur la machine gitrust, copier `deployment/ci-engine/` vers `CI_ENGINE_PATH` (défaut `/opt/gitrust/ci-engine`) et passer à la suite. **Ne pas exécuter `setup-remote-ci.sh`** avec `CI_REMOTE_HOST=localhost`.

---

6. Usage

Depuis la racine du projet (par défaut)

```
cd /chemin/vers/gitrust

# .env a la racine (défaut) :
./deployment/setup-remote-ci.sh
```


© Gitrust

Avec un `.env` explicite

```
./deployment/setup-remote-ci.sh /chemin/vers/mon.env
# Utile si vous avez .env.production, .env.staging, etc.
./deployment/setup-remote-ci.sh .env.production
```

Sortie attendue

```
==> Configuration :
    Serveur  : ci-runner@<ci-runner-ip>
    Port SSH : 22
    Chemin   : /opt/gitrust-ci

==> [1/6] Vérification de la connectivité SSH...
SSH OK
    OK
==> [2/6] Vérification de Docker...
    Docker déjà installé
==> [3/6] Vérification de Dagger CLI...
    Installation de Dagger CLI...
    Dagger installé
==> [4/6] Création du répertoire distant...
    /opt/gitrust-ci créé
==> [5/6] Synchronisation du ci-engine...
sending incremental file list
ci-engine/
ci-engine/profiles/
...
    ci-engine synchronisé vers /opt/gitrust-ci/ci-engine/
==> [6/6] Smoke test...
    Versions distantes :
Docker version 28.0.1, build abcd123
dagger v0.19.2 (linux/amd64)

==> Setup terminé. Le serveur ci-runner@<ci-runner-ip> est prêt pour l'exécution CI.
    Pensez à configurer CI_EXECUTION_MODE=remote dans votre .env
```


© Gitrust

7. Ce qui se retrouve sur le runner

Après exécution réussie :

```
/opt/gitrust-ci/                    <- CI_REMOTE_PATH
└── ci-engine/                      <- synchronisé depuis deployment/ci-engine/
    ├── README.md
    └── profiles/                   <- templates par stack (Python, Node, Rust, ...)
```

Plus, installés globalement : - `/usr/bin/docker` (ou équivalent) + socket `/var/run/docker.sock` - `/usr/local/bin/dagger`

Les **workspaces de pipelines** (checkouts temporaires) sont créés par le worker CI de gitrust au moment de l'exécution sous `CI_WORKSPACE_PATH` (défaut `/tmp/gitrust-ci`) — **pas par ce script**.

---

8. Vérifications post-setup

Depuis la machine gitrust

```
# Source le .env
set -a; source .env.production; set +a

# 1. SSH direct (doit passer sans prompt)
ssh -p ${CI_REMOTE_SSH_PORT:-22} \
    ${CI_REMOTE_SSH_KEY:+-i $CI_REMOTE_SSH_KEY} \
    $CI_REMOTE_USER@$CI_REMOTE_HOST 'docker info && dagger version'

# 2. Rsync round-trip (lecture/écriture sur CI_REMOTE_PATH)
echo "test" | ssh $CI_REMOTE_USER@$CI_REMOTE_HOST \
    "cat > $CI_REMOTE_PATH/.gitrust-test && cat $CI_REMOTE_PATH/.gitrust-test && rm $CI
# Attendu : "test"
```

Pipeline de test via gitrust

Pousser un dépôt avec un `.gitrust-ci.yml` minimal :

```
# .gitrust-ci.yml
version: 1
pipeline:
  - name: smoke
```


© Gitrust

```
    image: alpine:3
    run: echo "CI runner OK"
```

Puis dans l'UI gitrust : onglet `Pipelines` → la pipeline doit passer au statut `success`.

---

9. Mise à jour après modification du `ci-engine/`

Le `ci-engine/` synchronisé n'est pas « live-linké » : rejouer le script après modification côté source.

```
# Modifier deployment/ci-engine/profiles/*.py ou similaire
./deployment/setup-remote-ci.sh
# Les étapes 2-3 sont skip (déjà installés), seule l'étape 5 refait le rsync
```

`rsync -az --delete` **supprime** les fichiers côté runner qui n'existent plus localement — garantit la cohérence.

© Gitrust

10. Dépannage

Symptôme	Cause probable	Fix
`ERREUR: fichier .env introuvable`	`.env` absent ou chemin incorrect	`cp .env.example .env && $EDITOR .env` ou passer le chemin : `./setup-remote-ci.sh /path/to/.env`
`ERREUR: CI_REMOTE_HOST non défini`	Variable commentée ou absente	Décommenter `CI_REMOTE_HOST=...` dans le `.env`
`ERREUR: impossible de se connecter`	SSH bloqué, mauvais user, clé non autorisée	Tester manuellement : `ssh -v -p X user@host` ; vérifier `~/.ssh/authorized_keys` sur le runner
`Permission denied (publickey)`	`BatchMode=yes` interdit les prompts, clé non chargée	`ssh-add ~/.ssh/ci_runner_ed25519` ou définir `CI_REMOTE_SSH_KEY=/path/to/key`
`sudo: a password is required` pendant l'install Docker	User sans `NOPASSWD` sudo	Ajouter le user au `sudoers` : `ci-runner ALL=(ALL) NOPASSWD:ALL` (runner uniquement)
`curl: (7) Failed to connect to get.docker.com`	Réseau sortant du runner bloqué	Whitelist `get.docker.com` et `dl.dagger.io`, ou pré-installer Docker + Dagger manuellement
`ATTENTION: ... ci-engine introuvable`	Lancé hors du repo gitrust	`cd` dans la racine du projet avant de lancer
Pipeline reste `queued` indéfiniment	Worker CI ne trouve pas le runner	Vérifier `CI_EXECUTION_MODE=remote` dans `.env` gitrust + logs : `journalctl -u gitrust | grep -i 'ci|dagger'`
`dagger: command not found` au smoke test	`$PATH` du user SSH ne contient pas `/usr/local/bin`	`echo 'export PATH=$PATH:/usr/local/bin' >> ~/.bashrc` sur le runner, ou `CI_DAGGER_BIN=/usr/local/bin/dagger` côté gitrust

11. Sécurité

- Le runner CI exécute du **code arbitraire** venant des dépôts hébergés. **Ne jamais** le colocaliser avec des secrets sensibles (PG prod, clés de prod).

© Gitrust

- Isoler réseau : bloquer l'accès sortant du runner vers le LAN privé (seul l'accès Internet pour `docker pull` est nécessaire).
- Limiter `sudo NOPASSWD` au strict minimum sur le runner (idéalement : juste pour les commandes `docker` et `apt`).
- Rotation régulière de la clé SSH `CI_REMOTE_SSH_KEY`. La révoquer dans `~/.ssh/authorized_keys` côté runner en cas de suspicion.
- Le runner ne doit **pas** pouvoir se connecter en SSH à la machine gitrust (unidirectionnel).

© Gitrust

1.3.6 Intégrer la CI et Dependency-Track

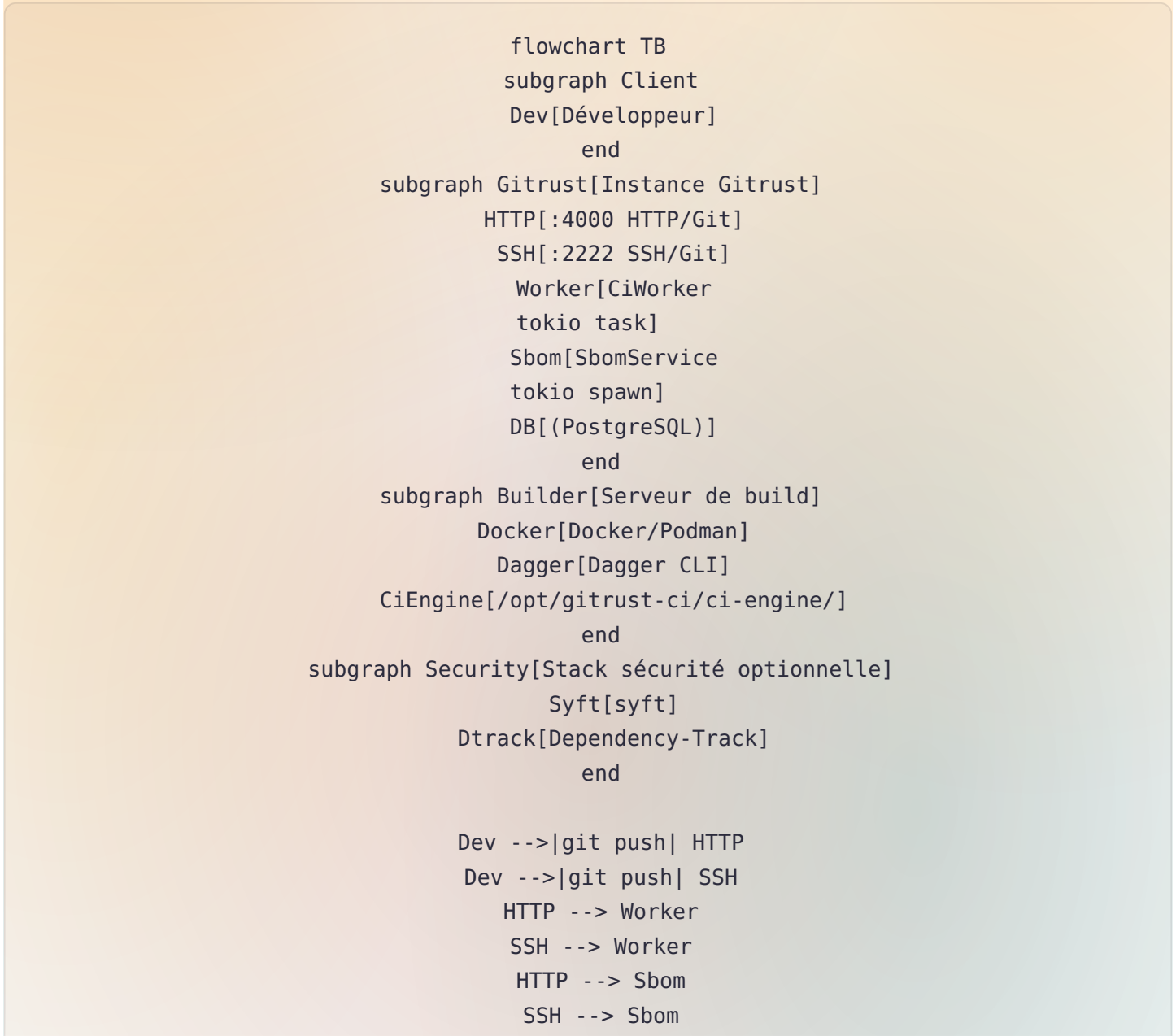
Ce document explique comment activer et configurer les deux fonctionnalités de qualité/sécurité de gitrust :

1. **La CI intégrée** (Dagger) qui exécute les builds/tests/lints à chaque push.
2. **Le dependency tracker** (Syft + Dependency-Track) qui scanne les composants du code à chaque push et détecte les vulnérabilités connues.

Ces deux systèmes sont **indépendants** : on peut activer l'un sans l'autre.

---

1. Architecture générale



© Gitrust

```
Worker -->|SSH + rsync| Builder
Builder -->|logs streamés| Worker
Worker --> DB
Sbom -->|scan workspace| Syft
Sbom -->|PUT BOM| Dtrack
Dtrack -->|findings| Sbom
Sbom --> DB
```

Trois composants fonctionnent en parallèle :

- **CiWorker** : tâche Tokio qui consomme des `CiTask` depuis un channel `mpsc`, limitée par un `Semaphore` à `CI_MAX_CONCURRENT` pipelines simultanés.
- **SbomService** : exécuté dans un `tokio::spawn` après chaque push, totalement découplé du CI.
- **Serveur de build** : machine distante (ou `localhost`) qui exécute Docker + Dagger. Gitrust y envoie le workspace par rsync et lance Dagger par SSH.

---

2. Mettre en place la CI

2.1 Pré-requis serveur de build

Le serveur de build peut être **la même machine que Gitrust** (`CI_REMOTE_HOST=localhost`) ou une machine dédiée. Il doit disposer de :

Outil	Rôle
Docker ou Podman	Containers Dagger
Dagger CLI	Moteur d'exécution
SSH (accès sortant depuis Gitrust)	Transport
rsync	Copie du workspace
git, tar	Extraction du tree du commit

Installation automatique recommandée :

```
# Depuis la machine Gitrust (l'utilisateur doit pouvoir SSH vers le builder)
./deployment/setup-remote-ci.sh .env
```


© Gitrust

Le script exécute :

1. Vérification de la connectivité SSH
2. Installation de Docker si absent
3. Installation de Dagger CLI si absent
4. Création du répertoire de travail distant
5. Synchronisation du `ci-engine`
6. Smoke test (versions)

2.2 Configuration Gitrust (fichier .env)

Ajouter au `.env` de Gitrust :

```
# Activer globalement le CI
CI_ENABLED=true

# Chemin du ci-engine sur le serveur de build
CI_ENGINE_PATH=/opt/gitrust-ci/ci-engine

# Limite de parallélisme
CI_MAX_CONCURRENT=4
CI_DEFAULT_TIMEOUT=3600
CI_WORKSPACE_PATH=/tmp/gitrust-ci

# Serveur de build (mettre localhost pour "même machine")
CI_REMOTE_HOST=builder.example.com
CI_REMOTE_USER=ci-runner
CI_REMOTE_PATH=/opt/gitrust-ci
CI_REMOTE_SSH_PORT=22
# CI_REMOTE_SSH_KEY=/home/gitrust/.ssh/id_ed25519   # si pas ssh-agent
```

Redémarrer Gitrust : `systemctl restart gitrust` (ou `cargo run --release`).

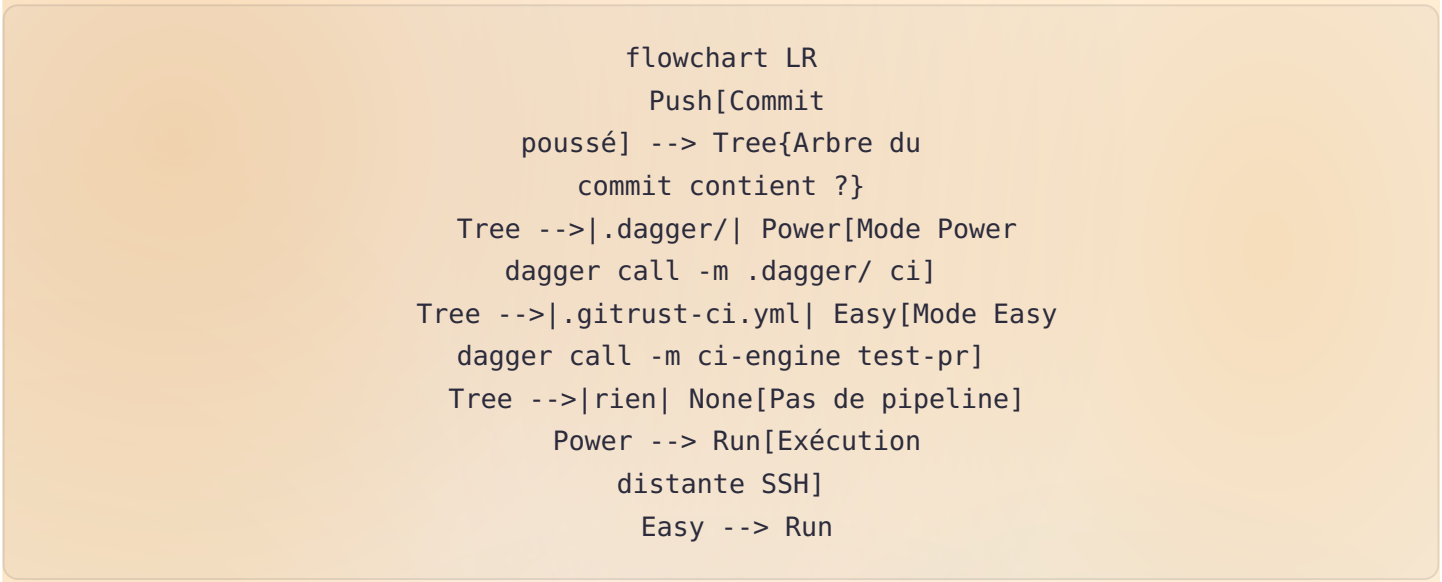
2.3 Activer la CI sur un dépôt

Dans l'interface web : aller sur le dépôt → **Settings** → **CI**.

- Cocher **CI enabled** (requis — par défaut `false`)
- Cocher **Trigger on push** (active l'exécution à chaque push)
- Optionnel : **Auto-cancel** (annule les pipelines en cours quand un nouveau démarre)
- Optionnel : **Allowed branches** (ex. `main,develop` — vide = toutes)

© Gitrust

2.4 Choisir le mode : Easy ou Power



Mode Easy (recommandé pour commencer)

Créer `.gitrust-ci.yml` à la racine du dépôt :

```
# Raccourci : charge un profil pré-configuré (rust | python | node)
language: rust

build:
  command: "cargo build --release"

checks:
  lint: "cargo clippy -- -D warnings"
  format: "cargo fmt -- --check"

tests:
  command: "cargo test --release"
```

Les profils disponibles se trouvent dans `deployment/ci-engine/profiles/` (rust.yaml, python.yaml, node.yaml) dans le dépôt source.

Mode Power

Pour les utilisateurs avancés, créer un module Dagger complet dans `.dagger/` :

```
.dagger/
├── dagger.json
```


© Gitrust

```
├── src/
│   └── main.py         # ou Go/TypeScript
```

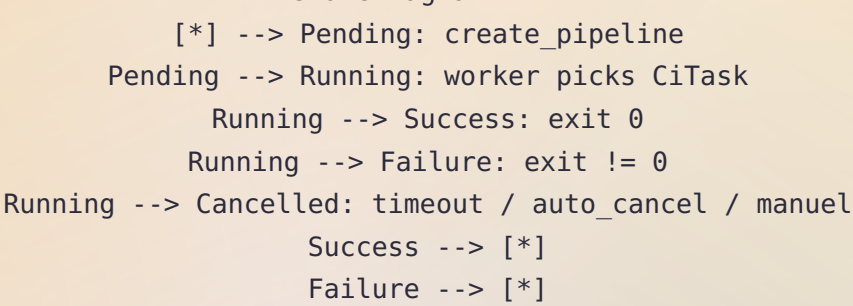
La fonction `ci` du module est appelée directement :

```
dagger call -m .dagger/ ci
```

Avantage : accès au Daggerverse, composition, tests de pipeline. Voir la documentation Dagger.

2.5 Suivi d'un pipeline

Après un `git push`, une entrée apparaît dans l'onglet **CI** du dépôt :



Les logs stdout/stderr sont streamés ligne par ligne dans la table `ci_logs` et visibles en direct dans l'UI. En cas d'échec, une notification est envoyée au propriétaire du dépôt.

---

3. Mettre en place le Dependency Tracker

Cette partie est totalement indépendante de la CI. Elle scanne le code poussé et produit un SBOM CycloneDX, puis (optionnellement) l'envoie à Dependency-Track pour une analyse de vulnérabilités.

3.1 Installer Syft

Sur la machine Gitrust (le scan se fait localement, pas sur le builder) :

```
# Installation officielle
curl -sSfL https://raw.githubusercontent.com/anchore/syft/main/install.sh \
  | sh -s -- -b /usr/local/bin
```


© Gitrust

```
# Vérification
syft --version
```

Dans `.env` :

```
CI_SBOM_ENABLED=true
CI_SYFT_BIN=/usr/local/bin/syft   # optionnel, défaut: syft dans PATH
```

À ce stade : à chaque push, un SBOM CycloneDX est généré et stocké (sans upload externe). Visible dans l'onglet **Security** du dépôt : nombre de composants, sha256 du BOM.

3.2 Déployer Dependency-Track

Dependency-Track est une application Java qui stocke les SBOM et corrèle avec les bases CVE/OSV/NVD. Déploiement Docker recommandé :

```
mkdir -p /opt/dtrack && cd /opt/dtrack
curl -L -o docker-compose.yml \
  https://dependencytrack.org/docker-compose.yml
docker compose up -d
```

L'API est disponible sur `http://localhost:8081` et l'UI sur `http://localhost:8080`. Login initial : `admin` / `admin` (à changer immédiatement).

3.3 Créer une clé API

Dans Dependency-Track → **Administration** → **Access Management** → **Teams** :

1. Créer (ou réutiliser) une équipe `gitrust`.
2. Lui attribuer les permissions :
3. `BOM_UPLOAD`
4. `PROJECT_CREATION_UPLOAD`
5. `VIEW_PORTFOLIO`
6. `VIEW_VULNERABILITY`
7. Générer une API key et la copier.

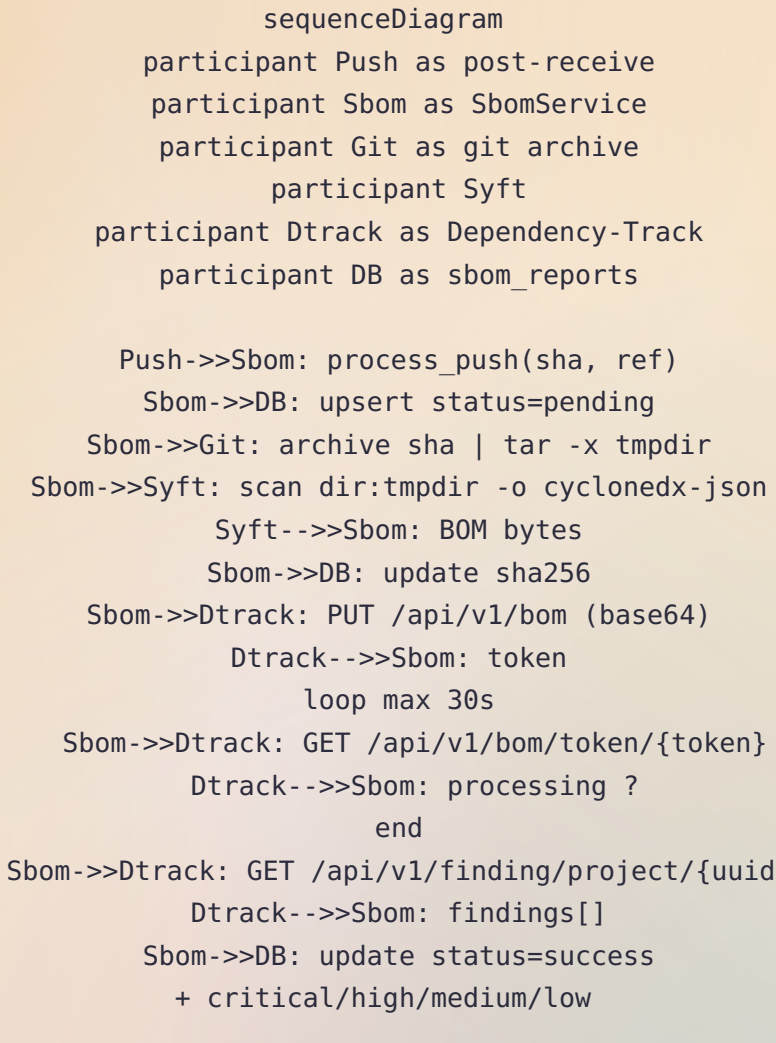
3.4 Configurer Gitrust

Ajouter au `.env` :

© Gitrust

```
CI_DTRACK_ENABLED=true
CI_DTRACK_URL=http://localhost:8081
CI_DTRACK_API_KEY=odt_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Redémarrer Gitrust. À chaque push :



3.5 Lire les résultats

Onglet **Security** du dépôt → encart SBOM :

- Nombre de composants détectés
- Compteurs par sévérité : Critical / High / Medium / Low
- Lien direct vers le projet dans Dependency-Track (si l'UUID est résolu)
- Hash sha256 du BOM (traçabilité)

Si Dependency-Track met plus de 30s à analyser, le status reste `processing` — un sweeper ultérieur ira rechercher les findings.

© Gitrust

4. Débogage

Problèmes CI courants

Symptôme	Cause probable	Vérifier
Pipeline reste `pending`	Worker non démarré, ou channel saturé	Logs `CI worker started`, `CI_MAX_CONCURRENT`
`Échec rsync vers le serveur de build`	SSH bloqué, clé absente	`ssh -p ${CI_REMOTE_SSH_PORT} ${CI_REMOTE_USER}@${CI_REMOTE_HOST}` manuel
`dagger: command not found`	Dagger non installé sur le builder	Relancer `setup-remote-ci.sh`
Status `Cancelled` inattendu	Timeout (`CI_DEFAULT_TIMEOUT`) ou auto-cancel	Augmenter le timeout, vérifier pipelines concurrents

Problèmes SBOM courants

Symptôme	Cause probable	Vérifier
`SBOM generation disabled`	`CI_SBOM_ENABLED=false`	`.env`
`syft spawn failed`	Binaire introuvable	`which syft`, `CI_SYFT_BIN`
`invalid commit sha`	SHA non hex 40 chars	Rare, signale un bug
`Dtrack upload 401`	API key invalide ou permissions manquantes	Régénérer, vérifier les 4 permissions
Status `processing` indéfiniment	Dtrack surchargé, analyse lente	Attendre, ou relancer un push

Logs utiles

```
# Logs Gitrust (systemd)
journalctl -u gitrust -f | grep -E "CI|SBOM|sbom|pipeline"

# Vérifier un pipeline en DB
psql $DATABASE_URL -c "SELECT id, status, commit_sha, created_at FROM ci_pipelines ORDE
```


© Gitrust

```
# Vérifier un SBOM
psql $DATABASE_URL -c "SELECT commit_sha, status, components_count, critical_count, hig
```

---

5. Checklist récapitulative

CI : - [ ] Docker + Dagger installés sur le serveur de build (`setup-remote-ci.sh`) - [ ] `CI_ENABLED=true` et `CI_REMOTE_*` dans `.env` - [ ] Gitrust redémarré - [ ] CI activée dans Settings → CI pour chaque dépôt - [ ] Fichier `.gitrust-ci.yml` (Easy) ou `.dagger/` (Power) commité - [ ] Push → pipeline visible dans l'onglet CI

Dependency Tracker : - [ ] Syft installé sur la machine Gitrust - [ ] `CI_SBOM_ENABLED=true` dans `.env` - [ ] (Optionnel) Dependency-Track déployé + API key créée avec les 4 permissions - [ ] (Optionnel) `CI_DTRACK_ENABLED=true` , `CI_DTRACK_URL` , `CI_DTRACK_API_KEY` dans `.env` - [ ] Gitrust redémarré - [ ] Push → SBOM visible dans l'onglet Security

© Gitrust

1.3.7 Configurer ssh-guard (durcissement SSH)

À qui s'adresse cette page

Administrateurs qui déploient ou ajustent la couche `ssh-guard` de gitrust selon leur topologie réseau (Internet direct, derrière nginx stream, derrière HAProxy, ou réseau privé).

> ***Avant de commencer*** *: si vous voulez d'abord comprendre ce que fait ssh-guard et ses compromis, lisez ssh-guard : détection d'attaques SSH. La référence des variables est dans Variables d'environnement — SSH_GUARD_*.*

1. Choisir le profil de déploiement

Le profil pré-configure les défauts cohérents avec votre topologie. Une variable unique pilote tout le reste : `SSH_GUARD_PROFILE`.

Profil	Topologie	Quand l'utiliser
`direct`	gitrust écoute en `:22` ou `:2222` exposé Internet, sans proxy devant	VPS minimal, démo publique
`nginx`	nginx stream sur `:22` avec `proxy_protocol on;` → gitrust sur `127.0.0.1:2222`	Production avec reverse-proxy nginx (cas le plus courant)
`haproxy`	HAProxy en frontal SSH avec `send-proxy` ou `send-proxy-v2`	Topologies multi-services
`private`	Instance interne (VPN, réseau d'entreprise), pas d'attaque externe attendue	Intranet, lab
`custom`	Aucun preset, vous fixez chaque variable individuellement	Cas exotiques

Le défaut au démarrage si `SSH_GUARD_PROFILE` n'est pas défini est `custom`, qui équivaut au profil `direct` côté détecteurs mais avec PROXY protocol désactivé.

2. Recette : profil `direct` (Internet direct)

Cas typique : VPS, pas de reverse-proxy SSH devant gitrust. Le port `:2222` est ouvert sur Internet et `peer_addr()` est l'IP réelle du client.

© Gitrust

2.1 `.env` minimal

Copiez `template/env/ssh-guard-direct.env` dans votre `/opt/gitrust/.env` ou ajoutez :

```
SSH_GUARD_ENABLED=true
SSH_GUARD_DRY_RUN=false
SSH_GUARD_PROFILE=direct

# Stockage hybride (RAM chaude + write-through PostgreSQL)
SSH_GUARD_STORE_BACKEND=hybrid

# Logs JSON dans journald + fichier dédié pour fail2ban
SSH_GUARD_LOG_FORMAT=json
SSH_GUARD_LOG_TARGET=both
SSH_GUARD_LOG_FILE=/var/log/gitrust-ssh-guard.json
```

Les seuils par défaut s'appliquent : brute-force 5/5min, énumération 10/5min, scan de clés 10/5min, flood 10/s burst 20, ban 1 h.

2.2 Redémarrage et vérification

```
sudo systemctl restart gitrust

# Le module annonce sa configuration au démarrage
sudo journalctl -u gitrust --since "1 min ago" | grep "SSH guard runtime assembly"
# Attendu : profile=direct enabled=true dry_run=false store=Hybrid

# Logrotate sur le fichier dédié
sudo tee /etc/logrotate.d/gitrust-ssh-guard <<'EOF'
/var/log/gitrust-ssh-guard.json {
    daily
    rotate 14
    missingok
    notifempty
    compress
    delaycompress
    copytruncate
}
EOF
```


© Gitrust

3. Recette : profil `nginx` (derrière nginx stream)

Cas typique : nginx termine TLS sur `:443`, fait du SSH stream sur `:22` → `127.0.0.1:2222` avec PROXY protocol v2.

3.1 nginx (rappel — voir aussi Durcir avec Fail2ban)

```
# /etc/nginx/nginx.conf (bloc stream)
stream {
    upstream gitrust_ssh {
        server 127.0.0.1:2222;
    }

    server {
        listen 22;
        proxy_pass         gitrust_ssh;
        proxy_protocol     on;       # IMPORTANT — sinon ssh-guard rejettera
        proxy_timeout      30m;      # git push de gros pack peut être long
    }
}
```

3.2 `.env` côté gitrust

Copiez `template/env/ssh-guard-nginx.env` ou :

```
SSH_GUARD_ENABLED=true
SSH_GUARD_PROFILE=nginx

# Le profil nginx pose déjà :
#   SSH_GUARD_PROXY_PROTOCOL=v2
#   SSH_GUARD_PROXY_PROTOCOL_STRICT=true
#   SSH_GUARD_TRUSTED_PROXIES=127.0.0.1/32,::1/128
# (override individuel possible)

SSH_GUARD_LOG_FORMAT=json
SSH_GUARD_LOG_TARGET=both
SSH_GUARD_LOG_FILE=/var/log/gitrust-ssh-guard.json
```

Côté binding gitrust :

```
SSH_LISTEN_ADDR=127.0.0.1
SSH_PORT=2222
```


© Gitrust

3.3 Vérification que la vraie IP est bien extraite

Faites un `git ls-remote ssh://git@VOTRE_FQDN/owner/repo.git` depuis une IP externe connue, puis :

```
sudo journalctl -u gitrust --since "1 min ago" | grep '"event":"connection_accepted"' |
# Attendu : "ip":"<VOTRE IP EXTERNE>"   et NON "ip":"127.0.0.1"
```

Si vous voyez `127.0.0.1`, c'est que le PROXY protocol n'est pas correctement transmis : vérifiez `proxy_protocol on;` dans nginx.

---

4. Recette : profil `haproxy`

Le profil pose `SSH_GUARD_PROXY_PROTOCOL=any` (auto-détection v1 ou v2) avec `strict=true`, mais **n'a pas** de `trusted_proxies` par défaut (HAProxy est presque toujours sur une IP distincte). À fournir explicitement :

```
SSH_GUARD_ENABLED=true
SSH_GUARD_PROFILE=haproxy
SSH_GUARD_TRUSTED_PROXIES=10.0.0.5/32   # IP de l'haproxy

SSH_GUARD_LOG_FORMAT=json
SSH_GUARD_LOG_TARGET=both
SSH_GUARD_LOG_FILE=/var/log/gitrust-ssh-guard.json
```

Si `SSH_GUARD_TRUSTED_PROXIES` est vide, gitrust **refuse de démarrer** avec l'erreur :

```
SSH_GUARD_TRUSTED_PROXIES required when SSH_GUARD_PROXY_PROTOCOL is enabled
```

C'est volontaire : sans cette protection, n'importe qui pourrait forger un en-tête PROXY pour usurper une IP cliente.

---

5. Recette : profil `private` (réseau interne)

Détecteurs désactivés (seuils `u32::MAX`), événements toujours émis pour audit, store en mémoire (rapide, perdu au restart).

```
SSH_GUARD_ENABLED=true
SSH_GUARD_PROFILE=private

# Tous les détecteurs sont désactivés par le preset.
```


© Gitrust

```
# Les événements connection_accepted / auth_failed / auth_succeeded
# restent émis pour audit.

SSH_GUARD_LOG_FORMAT=json
SSH_GUARD_LOG_TARGET=stderr   # journald uniquement, pas de fichier
```

6. Mode `dry-run` : observer avant de bloquer

Idéal pour valider un nouveau seuil ou laisser fail2ban faire le ban réel :

```
SSH_GUARD_DRY_RUN=true
```

Effets :

- Les détecteurs continuent à corréler.
- Un événement `ip_banned` est émis pour chaque déclenchement (signal pour fail2ban / Loki).
- **Aucun ban n'est persisté** côté ssh-guard. Le `BanManager` reste no-op.

C'est le mode recommandé pendant les premiers jours après un changement de seuil agressif. Surveillez :

```
sudo journalctl -u gitrust --since "24 hours ago" --no-pager \
  | grep '"event":"ip_banned"' \
  | jq -r '.ip + " " + .reason' \
  | sort | uniq -c | sort -rn
```

7. Ajuster les seuils

7.1 Variables disponibles

```
# Brute-force : nombre d'échecs auth depuis une IP dans la fenêtre
SSH_GUARD_BRUTE_FORCE_THRESHOLD=5         # défaut 5
SSH_GUARD_BRUTE_FORCE_WINDOW_SECS=300     # défaut 300 (5 min)

# Énumération : nombre d'usernames distincts essayés depuis une IP
SSH_GUARD_USER_ENUM_THRESHOLD=10
SSH_GUARD_USER_ENUM_WINDOW_SECS=300

# Scan de clés : nombre de fingerprints distincts essayés depuis une IP
```


© Gitrust

```
SSH_GUARD_KEY_SCAN_THRESHOLD=10
SSH_GUARD_KEY_SCAN_WINDOW_SECS=300

# Flood TCP : cap dur de connexions par seconde par IP
SSH_GUARD_CONN_FLOOD_PER_SEC=10
SSH_GUARD_CONN_FLOOD_BURST=20

# Durée d'un ban auto. 0 = permanent.
SSH_GUARD_AUTO_BAN_DURATION_SECS=3600     # défaut 3600 (1 h)
```

7.2 Profils suggérés

Posture	Brute-force	Conn flood	TTL ban
Stricte (instance publique très exposée)	3 / 5 min	5/s burst 10	6 h
Standard (défaut)	5 / 5 min	10/s burst 20	1 h
Tolérante (équipe interne, partenaires CI)	10 / 5 min	20/s burst 50	30 min

Désactiver complètement un détecteur : mettre son seuil à `4294967295` (`u32::MAX`). Le préfixe `private` le fait déjà pour les quatre.

---

8. Allowlist d'un partenaire CI

Une IP allowlistée bypasse les détecteurs et le flood-limit, mais reste auditée. Les ACL se gèrent via la table `ssh_guard_acl` (admin UI ou SQL direct).

```
-- Allow le runner CI de l'équipe build (CIDR /32)
INSERT INTO ssh_guard_acl (id, kind, ip_cidr, reason, created_by, created_at, updated_a
VALUES (
  gen_random_uuid(),
  'allow',
  '198.51.100.42/32',
  'Runner CI équipe build (ticket OPS-1234)',
  '<UUID admin>',
  NOW(), NOW()
);
```


© Gitrust

Vérifier : depuis cette IP, déclencher 50 tentatives ratées en 1 minute, puis vérifier qu'aucun `ip_banned` n'a été émis :

```
sudo journalctl -u gitrust --since "5 min ago" --no-pager \
  | grep '"ip":"198.51.100.42"' \
  | jq -r '.event' | sort | uniq -c
# Attendu : auth_failed: 50, brute_force_detected: 0, ip_banned: 0
```

---

9. Bannir manuellement un CIDR abusif

Pour un AS qui scanne en masse, posez un ban permanent :

```
-- Denylist permanente d'un CIDR
INSERT INTO ssh_guard_bans (id, ip_cidr, reason, banned_at, expires_at, auto_banned)
VALUES (
  gen_random_uuid(),
  '198.51.100.0/24',
  'admin_deny_list',
  NOW(),
  NULL,         -- NULL = permanent
  false
);
```

`expires_at = NULL` rend le ban permanent. Pour un TTL fini : `NOW() + INTERVAL '7 days'`.

---

10. Troubleshooting

10.1 Tous les `ip` valent `127.0.0.1`

Vous êtes en profil `direct` mais nginx fait du stream devant gitrust. Passez en `nginx` et activez `proxy_protocol on;` dans nginx.

10.2 « SSH_GUARD_TRUSTED_PROXIES required »

Vous avez activé le profil `haproxy` (ou `SSH_GUARD_PROXY_PROTOCOL=v1|v2|any`) sans déclarer de proxies de confiance. Ajoutez `SSH_GUARD_TRUSTED_PROXIES=<CIDR>` et redémarrez.

© Gitrust

10.3 Faux positifs sur un développeur avec 8 clés SSH

L'agent SSH du dev présente toutes ses clés successivement. Si la 9e est la bonne, le détecteur de scan de clés pourrait s'activer (seuil 10 par défaut). Solutions, par ordre de préférence :

1. Côté dev : forcer une clé spécifique avec `IdentitiesOnly yes` dans `~/.ssh/config`.
2. Côté admin : monter le seuil à 15 (`SSH_GUARD_KEY_SCAN_THRESHOLD=15`).

10.4 Lever un ban auto sans attendre l'expiration

```
UPDATE ssh_guard_bans
SET unbanned_at = NOW(), unbanned_by = '<UUID admin>'
WHERE id = '<UUID du ban>';
```

ssh-guard émet un `ip_unbanned` au prochain cycle d'`effective_status`.

10.5 Désactiver temporairement ssh-guard

```
# .env
SSH_GUARD_ENABLED=false
sudo systemctl restart gitrust
```

ssh-guard devient un pass-through complet : aucune détection, aucun ban, le listener délègue directement à `russh`. **À éviter sauf urgence**.

---

11. Pour aller plus loin

- Variables d'environnement — SSH_GUARD_*
- Événements ssh-guard (JSON)
- ssh-guard : détection d'attaques SSH
- Durcir avec Fail2ban — chaîner ssh-guard et fail2ban
- Dépanner SSH — diagnostic général SSH
- Régler le rate limiting — distinction rate-limit HTTP vs TCP

© Gitrust

1.3.8 Durcir l'instance avec Fail2ban

> ***Évolution importante** : depuis l'introduction de la crate `gitrust-ssh-guard`, le jail `[gitrust-ssh]` ne lit plus les logs `russh` via `journalctl` mais consomme directement le **flux JSON stable** émis par ssh-guard dans `/var/log/gitrust-ssh-guard.json`. Cela rend le jail bien plus fiable (format garanti stable, pas de regex fragile sur les messages de `russh`). Activez `SSH_GUARD_LOG_TARGET=both` dans `.env` — voir Configurer ssh-guard.*

Configuration complète pour protéger un déploiement gitrust exposant :

- **sshd système** sur `:2022` (admin)
- **Nginx** en reverse proxy HTTPS `:443` + redirect `:80` + `stream :22 → :2222`
- **Gitrust** (web `127.0.0.1:4000`, SSH russh `127.0.0.1:2222`)
- **Dagger** CI runner (local, pas d'exposition réseau directe)
- **Dependency-Track** API `:8081` + UI `:8080` (si exposés)
- **PostgreSQL** `127.0.0.1:5432` (défensif au cas où le bind fuit)

---

1. Installation

```
sudo apt install -y fail2ban
sudo systemctl enable --now fail2ban
fail2ban-client --version
```

Vérifier que `ufw` est actif (fournit `banaction = ufw` dans la config) :

```
sudo ufw status
```

---

2. Fichier principal `/etc/fail2ban/jail.local`

À placer via :

```
sudo tee /etc/fail2ban/jail.local <<'EOF'
[Le contenu ci-dessous]
EOF
```


© Gitrust

```
# ================================================================================
# Défauts globaux
# ================================================================================
[DEFAULT]
backend  = systemd
bantime  = 1h
findtime = 10m
maxretry = 5

# LAN privé et loopback jamais bannis
ignoreip = 127.0.0.1/8 ::1 192.168.1.0/24

# UFW
banaction          = ufw
banaction_allports = ufw

# Notifications (optionnel — nécessite un relay SMTP local)
# destemail = contact@gitrust.eu
# sender    = fail2ban@votre-serveur.example.com
# action    = %(action_mwl)s

# ================================================================================
# 1) sshd admin système — port 2022
# ================================================================================
[sshd]
enabled  = true
port     = 2022
filter   = sshd
backend  = %(sshd_backend)s
logpath  = %(sshd_log)s
maxretry = 3
findtime = 10m
bantime  = 1h

# ================================================================================
# 2) Nginx — 401/403 auth basic (si un jour activé sur /admin, /metrics, etc.)
# ================================================================================
[nginx-http-auth]
enabled  = true
port     = http,https
filter   = nginx-http-auth
logpath  = /var/log/nginx/gitrust.error.log
maxretry = 5
```


© Gitrust

```
# ============================================================================
# 3) Nginx — scanners d'URL (wp-admin, .env, phpmyadmin, config.php, etc.)
# ============================================================================
[nginx-botsearch]
enabled  = true
port     = http,https
filter   = nginx-botsearch
logpath  = /var/log/nginx/gitrust.access.log
maxretry = 2
findtime = 10m
bantime  = 24h

# ============================================================================
# 4) Nginx — bad user-agents (scanners agressifs, masscan, nikto, etc.)
# ============================================================================
[nginx-badbots]
enabled  = true
port     = http,https
filter   = nginx-badbots
logpath  = /var/log/nginx/gitrust.access.log
maxretry = 2
bantime  = 24h

# ============================================================================
# 5) Nginx — dépassement limit_req (voir section 4 — zones HTTPS)
# ============================================================================
[nginx-limit-req]
enabled  = true
port     = http,https
filter   = nginx-limit-req
logpath  = /var/log/nginx/gitrust.error.log
maxretry = 10
findtime = 5m
bantime  = 1h

# ============================================================================
# 6) Gitrust — brute force login (formulaire + API JWT)
# ============================================================================
[gitrust-login]
enabled  = true
port     = http,https
filter   = gitrust-login
```


© Gitrust

```
logpath  = /var/log/nginx/gitrust.access.log
maxretry = 5
findtime = 10m
bantime  = 1h

# ============================================================================
# 7) Gitrust — abus API (tokens personnels leaked, scrapers)
# ============================================================================
[gitrust-api-abuse]
enabled  = true
port     = http,https
filter   = gitrust-api-abuse
logpath  = /var/log/nginx/gitrust.access.log
maxretry = 30
findtime = 1m
bantime  = 2h

# ============================================================================
# 8) Gitrust SSH — consomme les événements JSON stables de ssh-guard
# ============================================================================
# IMPORTANT : ce jail consomme le flux JSON émis par la crate gitrust-ssh-guard
# (champs stables documentés dans ../reference/ssh-guard-evenements.md).
# Le ban est déclenché soit sur le "signal fort" ip_banned (ssh-guard a déjà
# détecté la brute-force, fail2ban relaye au firewall) — un seul ip_banned
# suffit (maxretry=1) — soit sur le brut auth_failed avec le seuil habituel.
#
# Prérequis dans /opt/gitrust/.env :
#   SSH_GUARD_LOG_TARGET=both
#   SSH_GUARD_LOG_FILE=/var/log/gitrust-ssh-guard.json
[gitrust-ssh]
enabled  = true
port     = 22,2222
filter   = gitrust-ssh
logpath  = /var/log/gitrust-ssh-guard.json
maxretry = 1            # ssh-guard a déjà corrélé : 1 ip_banned = 1 ban firewall
findtime = 10m
bantime  = 1h

# ============================================================================
# 9) Gitrust import worker — tokens invalides sur clone de dépôts externes
# ============================================================================
[gitrust-import]
enabled      = true
```


© Gitrust

```
port         = http,https
filter       = gitrust-import
backend      = systemd
journalmatch = _SYSTEMD_UNIT=gitrust.service
maxretry     = 3
findtime     = 5m
bantime      = 30m

# ============================================================================
# 10) Dependency-Track UI — brute force login (port 8080)
# ============================================================================
# Activer UNIQUEMENT si DTrack est exposé publiquement.
# Si DTrack est sur réseau privé/loopback, laisser enabled=false.
[dtrack-login]
enabled  = false
port     = 8080,http,https
filter   = dtrack-login
logpath  = /var/log/nginx/dtrack.access.log
maxretry = 5
findtime = 10m
bantime  = 2h

# ============================================================================
# 11) Dependency-Track API — abus clé API (port 8081)
# ============================================================================
[dtrack-api]
enabled  = false
port     = 8081,http,https
filter   = dtrack-api
logpath  = /var/log/nginx/dtrack.access.log
maxretry = 10
findtime = 5m
bantime  = 1h

# ============================================================================
# 12) PostgreSQL — tentatives de connexion invalides (défensif)
# ============================================================================
# PG doit binder 127.0.0.1 uniquement. Ce jail protège si la config fuit.
[postgresql]
enabled      = true
port         = 5432
filter       = postgresql
backend      = systemd
```


© Gitrust

```
journalmatch = _SYSTEMD_UNIT=postgresql.service + _SYSTEMD_UNIT=docker.service
maxretry     = 5
findtime     = 10m
bantime      = 1h

# ================================================================================
# 13) Récidive — ban long pour IPs bannies 3x en 24h, tous jails confondus
# ================================================================================
[recidive]
enabled   = true
logpath   = /var/log/fail2ban.log
banaction = %(banaction_allports)s
bantime   = 1w
findtime  = 1d
maxretry  = 3
```

3. Filtres personnalisés

À placer dans `/etc/fail2ban/filter.d/` (un fichier par jail custom).

3.1 `gitrust-login.conf`

```
sudo tee /etc/fail2ban/filter.d/gitrust-login.conf <<'EOF'
[Definition]
# POST sur /login ou endpoints JWT avec code 4xx (401/403/422/429)
failregex = ^<HOST> .* "POST /(login|api/v1/auth/login|api/v1/auth/refresh|api/v1/auth/
ignoreregex =
EOF
```

3.2 `gitrust-api-abuse.conf`

```
sudo tee /etc/fail2ban/filter.d/gitrust-api-abuse.conf <<'EOF'
[Definition]
# Abus API : 401/403 répétés sur /api/v1/* (hors auth déjà couvert par gitrust-login)
failregex = ^<HOST> .* "(GET|POST|PUT|DELETE|PATCH) /api/v1/(?!auth/)[^"]*" (401|403) .
ignoreregex =
EOF
```


© Gitrust

3.3 `gitrust-ssh.conf` (consomme le JSON ssh-guard)

Le filtre matche les événements JSON stables produits par `gitrust-ssh-guard`. Voir Événements ssh-guard (JSON) pour le schéma complet.

Le « signal fort » `ip_banned` est privilégié : ssh-guard a déjà corrélé brute-force / énumération / scan de clés, et fail2ban n'a plus qu'à appliquer le ban au niveau firewall (UFW/iptables) pour les autres ports si désiré.

```
sudo tee /etc/fail2ban/filter.d/gitrust-ssh.conf <<'EOF'
[Definition]
# Filtre les événements stables émis par gitrust-ssh-guard dans
# /var/log/gitrust-ssh-guard.json. Format : une ligne JSON par événement.
#
# Capture <HOST> depuis le champ "ip" du JSON pour les variants pertinents.
#
# Tester :
#   sudo fail2ban-regex /var/log/gitrust-ssh-guard.json \
#                       /etc/fail2ban/filter.d/gitrust-ssh.conf

failregex = ^.*"event":"ip_banned".*"ip":"<HOST>".*$
            ^.*"event":"brute_force_detected".*"ip":"<HOST>".*$
            ^.*"event":"user_enumeration_detected".*"ip":"<HOST>".*$
            ^.*"event":"key_scanning_detected".*"ip":"<HOST>".*$
            ^.*"event":"connection_dropped".*"ip":"<HOST>".*"reason":"untrusted_proxy".
            ^.*"event":"connection_dropped".*"ip":"<HOST>".*"reason":"proxy_header_inva

ignoreregex =

# Date au format ISO 8601 UTC produit par ssh-guard
datepattern = "ts":"%%Y-%%m-%%dT%%H:%%M:%%S
EOF
```

> ***Variante « brute uniquement »*** *— si vous préférez que fail2ban corrèle lui-même à partir des* `auth_failed` *sans dépendre du verdict ssh-guard, remplacez le bloc ci-dessus par* `failregex = ^.*"event":"auth_failed".*"ip":"<HOST>".*$` *et passez* `maxretry = 5` *dans le jail. Les deux approches sont valides ; la première est plus rapide à réagir, la seconde est plus indépendante.*

3.4 `gitrust-import.conf`

```
sudo tee /etc/fail2ban/filter.d/gitrust-import.conf <<'EOF'
[Definition]
# Import worker : tokens OAuth/GitHub invalides sur clone de dépôts externes
```


© Gitrust

```
failregex = ^.*import.*authentication failed.*from <HOST>.*$
            ^.*import.*invalid (token|credentials).*from <HOST>.*$
            ^.*import.*clone failed.*401.*from <HOST>.*$
ignoreregex =
EOF
```

3.5 `dtrack-login.conf` (si DTrack exposé)

```
sudo tee /etc/fail2ban/filter.d/dtrack-login.conf <<'EOF'
[Definition]
# Dependency-Track : brute force du POST /api/v1/user/login
failregex = ^<HOST> .* "POST /api/v1/user/login[^"]*" (401|403) .*$
            ^<HOST> .* "POST /api/v1/user/forceChangePassword[^"]*" (401|403) .*$
ignoreregex =
EOF
```

3.6 `dtrack-api.conf` (si DTrack exposé)

```
sudo tee /etc/fail2ban/filter.d/dtrack-api.conf <<'EOF'
[Definition]
# Dependency-Track API : abus de clé API ou requêtes non authentifiées
failregex = ^<HOST> .* "[^"]+ /api/v1/[^"]*" 401 .*$
            ^<HOST> .* "[^"]+ /api/v1/[^"]*" 403 .*$
ignoreregex =
EOF
```

3.7 `postgresql.conf` (créer si absent)

```
if [ ! -f /etc/fail2ban/filter.d/postgresql.conf ]; then
sudo tee /etc/fail2ban/filter.d/postgresql.conf <<'EOF'
[Definition]
failregex = ^.*authentication failed for user.*host=<HOST>.*$
            ^.*FATAL:.*password authentication failed.*<HOST>.*$
            ^.*no pg_hba\.conf entry for host "<HOST>".*$
ignoreregex =
EOF
fi
```

3.8 `nginx-limit-req.conf` (créer si absent sur votre distro)

© Gitrust

```
if [ ! -f /etc/fail2ban/filter.d/nginx-limit-req.conf ]; then
sudo tee /etc/fail2ban/filter.d/nginx-limit-req.conf <<'EOF'
[Definition]
failregex = ^.*limiting requests, excess: .* by zone .*, client: <HOST>.*$
ignoreregex =
EOF
fi
```

3.9 `nginx-badbots.conf` — OBLIGATOIRE (pas fourni par Debian/Ubuntu)

Le paquet `fail2ban` Debian/Ubuntu fournit `apache-badbots.conf` mais PAS `nginx-badbots.conf`. Sans ce filter, `fail2ban-client reload` affiche :

```
Found no accessible config files for 'filter.d/nginx-badbots' under /etc/fail2ban
```

Le filter ci-dessous réutilise la liste de bad user-agents de `apache-badbots` avec un `failregex` adapté au format `combined` de nginx :

```
sudo tee /etc/fail2ban/filter.d/nginx-badbots.conf <<'EOF'
[Definition]
badbotscustom = EmailCollector|WebEMailExtrac|TrackBack/1\.02|sogou music spider
badbots = Atomic_Email_Hunter/4\.0|atSpider/1\.0|autoemailspider|bwh3_user_agent|China

failregex = ^<HOST> -.*"(GET|POST|HEAD).*HTTP.*"(?:%(badbots)s|%(badbotscustom)s)"$
ignoreregex =

datepattern = ^[^\[]*\[({DATE})
              {^LN-BEG}
EOF
```

Alternative : si vous ne voulez pas maintenir la liste, simplement désactiver le jail (`enabled = false` dans le `[nginx-badbots]`) — la combinaison `nginx-botsearch` + `nginx-limit-req` + `gitrust-api-abuse` couvre déjà l'essentiel.

---

4. Rate limiting côté Nginx (requis par `nginx-limit-req`)

4.1 Déclarer les zones globales

```
sudo tee /etc/nginx/conf.d/gitrust-limit.conf <<'EOF'
# Zones de rate limiting gitrust
limit_req_zone $binary_remote_addr zone=gitrust_login:10m rate=1r/s;
```


© Gitrust

```
limit_req_zone $binary_remote_addr zone=gitrust_api:10m   rate=10r/s;
limit_req_zone $binary_remote_addr zone=gitrust_git:10m   rate=30r/s;
EOF
```

4.2 Ajouter les `location` dans `/etc/nginx/sites-available/gitrust`

Dans le bloc `server { listen 443 ssl; ... }`, **avant** `location /` finale :

```
# Protection brute force login (1 req/s par IP, burst 5)
location ~ ^/(login|api/v1/auth/) {
    limit_req zone=gitrust_login burst=5 nodelay;
    proxy_pass http://gitrust_backend;
    proxy_set_header Host              $host;
    proxy_set_header X-Real-IP         $remote_addr;
    proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
}

# Protection API (10 req/s par IP, burst 20)
location ^~ /api/v1/ {
    limit_req zone=gitrust_api burst=20 nodelay;
    proxy_pass http://gitrust_backend;
    proxy_set_header Host              $host;
    proxy_set_header X-Real-IP         $remote_addr;
    proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
}

# Protection Git smart HTTP (30 req/s — clone/push légitime peut être verbeux)
location ~ ^/[^/]+/[^/]+\.git/ {
    limit_req zone=gitrust_git burst=50 nodelay;
    proxy_pass http://gitrust_backend;
    proxy_http_version 1.1;
    proxy_set_header Host              $host;
    proxy_set_header X-Real-IP         $remote_addr;
    proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_buffering off;
    proxy_request_buffering off;
    client_max_body_size 2G;
}
```


© Gitrust

4.3 Reload

```
sudo nginx -t && sudo systemctl reload nginx
```

5. Démarrage et validation

```
sudo systemctl restart fail2ban

# Liste des jails actifs
sudo fail2ban-client status
# Attendu :
#   sshd, nginx-http-auth, nginx-botsearch, nginx-badbots, nginx-limit-req,
#   gitrust-login, gitrust-api-abuse, gitrust-ssh, gitrust-import,
#   postgresql, recidive
#   (+ dtrack-login, dtrack-api si activées)

# Stats par jail
sudo fail2ban-client status gitrust-login
sudo fail2ban-client status sshd
```

Valider chaque filtre custom

```
# 1. gitrust-login contre l'access log nginx
sudo fail2ban-regex /var/log/nginx/gitrust.access.log \
                    /etc/fail2ban/filter.d/gitrust-login.conf

# 2. gitrust-api-abuse
sudo fail2ban-regex /var/log/nginx/gitrust.access.log \
                    /etc/fail2ban/filter.d/gitrust-api-abuse.conf

# 3. gitrust-ssh contre le flux JSON ssh-guard
sudo fail2ban-regex /var/log/gitrust-ssh-guard.json \
                    /etc/fail2ban/filter.d/gitrust-ssh.conf
# Si /var/log/gitrust-ssh-guard.json n'existe pas, vérifiez SSH_GUARD_LOG_TARGET
# et SSH_GUARD_LOG_FILE dans /opt/gitrust/.env (voir how-to/configurer-ssh-guard.md).

# 4. postgresql
journalctl -u docker --no-pager -n 5000 | grep -i postgres > /tmp/pg.log
```


© Gitrust

```
sudo fail2ban-regex /tmp/pg.log \
                    /etc/fail2ban/filter.d/postgresql.conf
```

`0 failregex found` = regex à ajuster après observation des logs réels de chaque service.

Test de bannissement

```
# Depuis un host HORS LAN (sinon ignoreip s'applique)
for i in $(seq 1 10); do
  curl -sk -o /dev/null -w "%{http_code}\n" \
    -X POST https://<votre-domaine>/login \
    -d 'username=admin&password=wrong'
done

# Vérifier le ban
ssh gitrust-host 'sudo fail2ban-client status gitrust-login'
# Currently banned : 1 — Banned IP list : <IP test>

# Débannir manuellement
ssh gitrust-host 'sudo fail2ban-client unban <IP>'
```

---

6. Monitoring

```
# Logs fail2ban temps réel
sudo tail -f /var/log/fail2ban.log

# Toutes les IPs bannies, tous jails confondus
sudo fail2ban-client banned

# Stats détaillées d'un jail
sudo fail2ban-client status gitrust-ssh
#    Currently failed : 2
#    Total failed     : 47
#    Currently banned : 1
#    Total banned     : 12
#    Banned IP list   : 203.0.113.42

sudo grep "Ban " /var/log/fail2ban.log | tail -20
```


© Gitrust

7. Matrice récapitulative

#	Jail	Port(s)	Source logs	Max retry	Ban time	Surface protégée
1	`sshd`	2022	journald sshd	3	1h	SSH admin
2	`nginx-http-auth`	80,443	nginx error	5	1h	Basic auth (admin futur)
3	`nginx-botsearch`	80,443	nginx access	2	24h	Scanners WP/PHP
4	`nginx-badbots`	80,443	nginx access	2	24h	User-agents malveillants
5	`nginx-limit-req`	80,443	nginx error	10	1h	Flood global
6	`gitrust-login`	80,443	nginx access	5	1h	Brute force login UI + API
7	`gitrust-api-abuse`	80,443	nginx access	30	2h	Scrapers API (tokens fuités)
8	`gitrust-ssh`	22,2222	`/var/log/gitrust-ssh-guard.json` (JSON stable ssh-guard)	1	1h	Brute force / scan clés / énumération SSH Git
9	`gitrust-import`	80,443	journald gitrust	3	30m	Brute force PAT/ OAuth import
10	`dtrack-login`	8080	nginx access	5	2h	Brute force UI Dep-Track
11	`dtrack-api`	8081	nginx access	10	1h	Abus clé API Dep-Track
12	`postgresql`	5432	journald	5	1h	Défense en profondeur PG
13	`recidive`	all	fail2ban.log	3	1w	Méta-ban 3×/24h

Ordre de déclenchement typique : brute force → jail spécifique (ban 1h) → si récidive 3x → recidive (ban 1 semaine, tous ports).

© Gitrust

8. Notes spécifiques à la stack gitrust

8.1 Dagger (CI)

Dagger s'exécute **localement** sur la machine gitrust (ou un runner distant — voir `admin/how-to/configurer-ci-runner-remote.md`). **Aucun port réseau ouvert** → pas de jail dédié.

8.2 Dependency-Track

Si DTrack est déployé sur le même serveur et **exposé**, activer les jails 10 et 11 en mettant `enabled = true` et en pointant `logpath` vers les logs nginx du vhost DTrack.

Si DTrack est **interne** (VPN, réseau privé), laisser `enabled = false` — le firewall suffit.

Config DTrack recommandée : - Bind sur `127.0.0.1` via `dtrack.url.base=http://127.0.0.1:8080`

8.3 PostgreSQL

Le jail `postgresql` est **défensif**. PG doit rester bindé sur `127.0.0.1:5432` via Docker. Vérifier :

```
sudo ss -tlnp | grep 5432
# Attendu : 127.0.0.1:5432 — PAS 0.0.0.0:5432
```

8.4 Notifications

Pour recevoir un mail à chaque ban :

1. Installer un relay SMTP local : `sudo apt install msmtp-mta`
2. Configurer `/etc/msmtprc` avec un compte SMTP
3. Décommenter dans `[DEFAULT]` :

```
destemail = contact@gitrust.eu
sender    = fail2ban@votre-serveur.example.com
action    = %(action_mwl)s
```

4. `sudo systemctl restart fail2ban`

9. Limites et évolutions

- **IPv6** : toutes les regex utilisent `<HOST>` qui matche IPv4 ET IPv6. Vérifier que UFW est configuré pour v6 également.
- **CDN/Cloudflare devant** : si un CDN est ajouté, `$remote_addr` côté Nginx sera l'IP du CDN — il faut récupérer la vraie IP via `X-Forwarded-For` et propager au logging Nginx. Changer `failregex` en conséquence. Sinon les jails banniront le CDN.

© Gitrust

- **Docker/Podman** : si gitrust passe en conteneur, les logs de `gitrust.service` deviennent `docker.service` ou `podman.service` → mettre à jour `journalmatch` dans le jail 9 (`gitrust-import`). Le jail 8 (`gitrust-ssh`) n'est pas affecté car il consomme le fichier JSON ssh-guard, à condition que ce fichier soit monté côté hôte.
- **GeoIP** : pour bloquer des pays entiers en amont, ajouter `geo $blocked_country` dans Nginx (module `ngx_http_geoip2_module`) — complémentaire à fail2ban.

© Gitrust

1.3.9 Régler le rate limiting

À qui s'adresse cette page

Administrateurs qui souhaitent ajuster les limites de débit pour protéger l'instance contre les attaques par force brute et les abus d'API, tout en évitant de bloquer les utilisateurs légitimes.

Variables disponibles

Ces variables se définissent dans `/opt/gitrust/.env`. Un redémarrage est nécessaire après modification. `0` désactive la limite (déconseillé en production).

Variable	Défaut	Description
`RATE_LIMIT_LOGIN_PER_MINUTE`	`5`	Tentatives de connexion par IP par minute
`RATE_LIMIT_REFRESH_PER_MINUTE`	`10`	Rafraîchissements de token par IP par minute
`RATE_LIMIT_GENERAL_PER_MINUTE`	`100`	Requêtes générales par IP par minute (tous les autres endpoints)

Les limites s'appliquent **par adresse IP**. Si gitrust est derrière un reverse-proxy, assurez-vous que le vrai IP du client est transmis (en-tête `X-Forwarded-For` ou `X-Real-IP`).

Profils de configuration recommandés

Petite équipe (≤ 5 personnes, réseau de confiance)

Toutes les personnes travaillent depuis un réseau connu. On peut se permettre des limites légèrement plus souples pour éviter les faux positifs lors des scripts d'intégration.

```
RATE_LIMIT_LOGIN_PER_MINUTE=10
RATE_LIMIT_REFRESH_PER_MINUTE=20
RATE_LIMIT_GENERAL_PER_MINUTE=200
```

Équipe moyenne (6-20 personnes, instance interne)

Configuration par défaut, adaptée à la cible principale de gitrust.

© Gitrust

```
RATE_LIMIT_LOGIN_PER_MINUTE=5
RATE_LIMIT_REFRESH_PER_MINUTE=10
RATE_LIMIT_GENERAL_PER_MINUTE=100
```

Instance publique ou exposée sur Internet (50+ utilisateurs)

Limites plus strictes sur le login pour contrer les attaques par dictionnaire. La limite générale peut être augmentée si vos utilisateurs utilisent intensivement l'API.

```
RATE_LIMIT_LOGIN_PER_MINUTE=3
RATE_LIMIT_REFRESH_PER_MINUTE=10
RATE_LIMIT_GENERAL_PER_MINUTE=150
```

CI/CD avec beaucoup d'appels API

Si vos pipelines CI font de nombreuses requêtes API (création de statuts, commentaires, webhooks), augmentez uniquement `RATE_LIMIT_GENERAL_PER_MINUTE` depuis les IPs des runners :

```
RATE_LIMIT_GENERAL_PER_MINUTE=300
```

Ou configurez Nginx pour whitelister les IPs des runners CI avant le rate limiting de gitrust.

Surveiller les réponses 429

Quand une limite est dépassée, gitrust retourne `HTTP 429 Too Many Requests`. Ces réponses apparaissent dans les logs avec le niveau `WARN`.

```
# Surveiller les 429 en temps réel
sudo journalctl -u gitrust -f | grep "429\|rate_limit\|too many"

# Compter les 429 sur les 24 dernières heures
sudo journalctl -u gitrust --since "24 hours ago" --no-pager \
  | grep -c "429\|rate_limit"
```

Si vous utilisez Nginx comme reverse-proxy, vous pouvez aussi consulter les logs Nginx :

```
grep " 429 " /var/log/nginx/gitrust-access.log | awk '{print $1}' | sort | uniq -c | so
```


© Gitrust

Cela vous donne les IPs les plus souvent bloquées, utile pour distinguer une vraie attaque d'un utilisateur légitime avec un script trop agressif.

Gitrust derrière un reverse-proxy

Si gitrust écoute sur `127.0.0.1:4000` derrière Nginx, le rate limiting de gitrust voit l'IP `127.0.0.1` pour toutes les requêtes, ce qui rend la protection inefficace.

Configurez Nginx pour transmettre le vrai IP :

```
# Dans le bloc location de gitrust
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $scheme;
```

Et dans `/opt/gitrust/.env`, indiquez que gitrust est derrière un proxy de confiance :

```
# Ajouter si le reverse-proxy est sur 127.0.0.1
SERVER_HOST=127.0.0.1
```

Gitrust utilise alors `X-Real-IP` pour identifier l'IP cliente réelle dans ses mécanismes de rate limiting.

Rate-limit TCP par IP (ssh-guard)

Les variables `RATE_LIMIT_*` ci-dessus s'appliquent à la **couche HTTP** (login, API, refresh JWT). Pour le **port SSH** (`:2222` par défaut), un mécanisme distinct est fourni par la crate `gitrust-ssh-guard` : un cap dur de connexions TCP/seconde par IP, indépendant de tout traitement applicatif.

Variable	Défaut	Description
`SSH_GUARD_CONN_FLOOD_PER_SEC`	`10`	Cap soutenu de connexions/sec par IP. `0` ou `u32::MAX` = désactivé.
`SSH_GUARD_CONN_FLOOD_BURST`	`20`	Burst momentané autorisé.

© Gitrust

Différences avec le rate-limit HTTP :

- Pas de réponse `429` : la connexion TCP est **droppée immédiatement** au listener (avant tout handshake SSH).
- Pas de ban persistant en DB : un scan de port n'engendre aucune écriture.
- Budgets indépendants par IP via token bucket GCRA (algorithme leaky bucket sans biais).
- Une IP allowlistée dans `ssh_guard_acl` bypasse complètement ce cap.

Pour la recette complète (profils par topologie, dry-run, allowlist CI), voir Configurer ssh-guard. Pour le détail des événements émis (`connection_flood_detected` , `connection_dropped` avec `reason=flood_limit`), voir Événements ssh-guard (JSON).

Fail2ban comme couche complémentaire

Le rate limiting de gitrust agit requête par requête. Fail2ban analyse les logs et peut bannir les IPs pour une durée configurable après plusieurs 429.

Voir Durcir avec Fail2ban pour la configuration complète.

Pour aller plus loin

- Variables d'environnement — section Rate Limiting
- Variables d'environnement — section SSH_GUARD_*
- Configurer ssh-guard
- ssh-guard : détection d'attaques SSH
- Durcir avec Fail2ban
- Conformité ANSSI PA-074 — protection contre les attaques par force brute

© Gitrust

1.3.10 Forcer le 2FA sur toute l'instance

À qui s'adresse cette page

Administrateurs qui souhaitent obliger tous les utilisateurs à configurer l'authentification à deux facteurs (TOTP) avant d'accéder à l'instance.

Activer le 2FA obligatoire

Le paramètre `totp_force_enabled` est un paramètre dynamique stocké en base de données. Il prend effet immédiatement, sans redémarrage du service.

1. Connectez-vous à l'interface d'administration : `/admin`
2. Accédez à `/admin/settings`
3. Trouvez le paramètre **`totp_force_enabled`**
4. Activez le toggle et cliquez **Enregistrer**

© Gitrust

Comportement après activation

Dès que `totp_force_enabled=true` est enregistré :

- Tout utilisateur **sans 2FA configuré** est redirigé vers la page de configuration TOTP à sa prochaine connexion. Il ne peut pas accéder à l'application tant que la configuration n'est pas complète.
- Les utilisateurs **qui ont déjà configuré le 2FA** ne sont pas affectés.
- Les administrateurs sont soumis à la même règle — vérifiez que votre propre compte a le 2FA actif avant d'activer ce paramètre.

> ***Conseil** : prévenez vos utilisateurs avant d'activer ce paramètre. Un e-mail ou une annonce interne évite les appels au support pour des connexions bloquées.*

Vérifier que le 2FA est actif sur votre compte

Avant d'activer `totp_force_enabled`, confirmez que votre compte admin a bien le 2FA configuré :

1. Cliquez sur votre avatar → **Paramètres du profil**
2. Section **Sécurité** → vérifiez que **Authentification à deux facteurs** affiche « Actif »

Si ce n'est pas le cas, configurez le 2FA sur votre compte avant d'activer le paramètre global.

Révoquer le 2FA d'un utilisateur spécifique

Si un utilisateur a perdu accès à son application TOTP (téléphone perdu, application réinstallée) :

1. Accédez à `/admin/users`
2. Ouvrez la fiche de l'utilisateur concerné
3. Cliquez **Révoquer le 2FA**
4. L'utilisateur sera invité à reconfigurer le TOTP à sa prochaine connexion

---

Désactiver le 2FA obligatoire

1. Accédez à `/admin/settings`
2. Désactivez le toggle **`totp_force_enabled`**
3. Cliquez **Enregistrer**

Les utilisateurs peuvent à nouveau se connecter sans 2FA. Les configurations TOTP existantes sont conservées.

© Gitrust

Pour aller plus loin

- Gérer les utilisateurs — révoquer le 2FA individuellement
- Paramètres dynamiques — `totp_force_enabled`
- Conformité ANSSI PA-074 — authentification multi-facteurs

© Gitrust

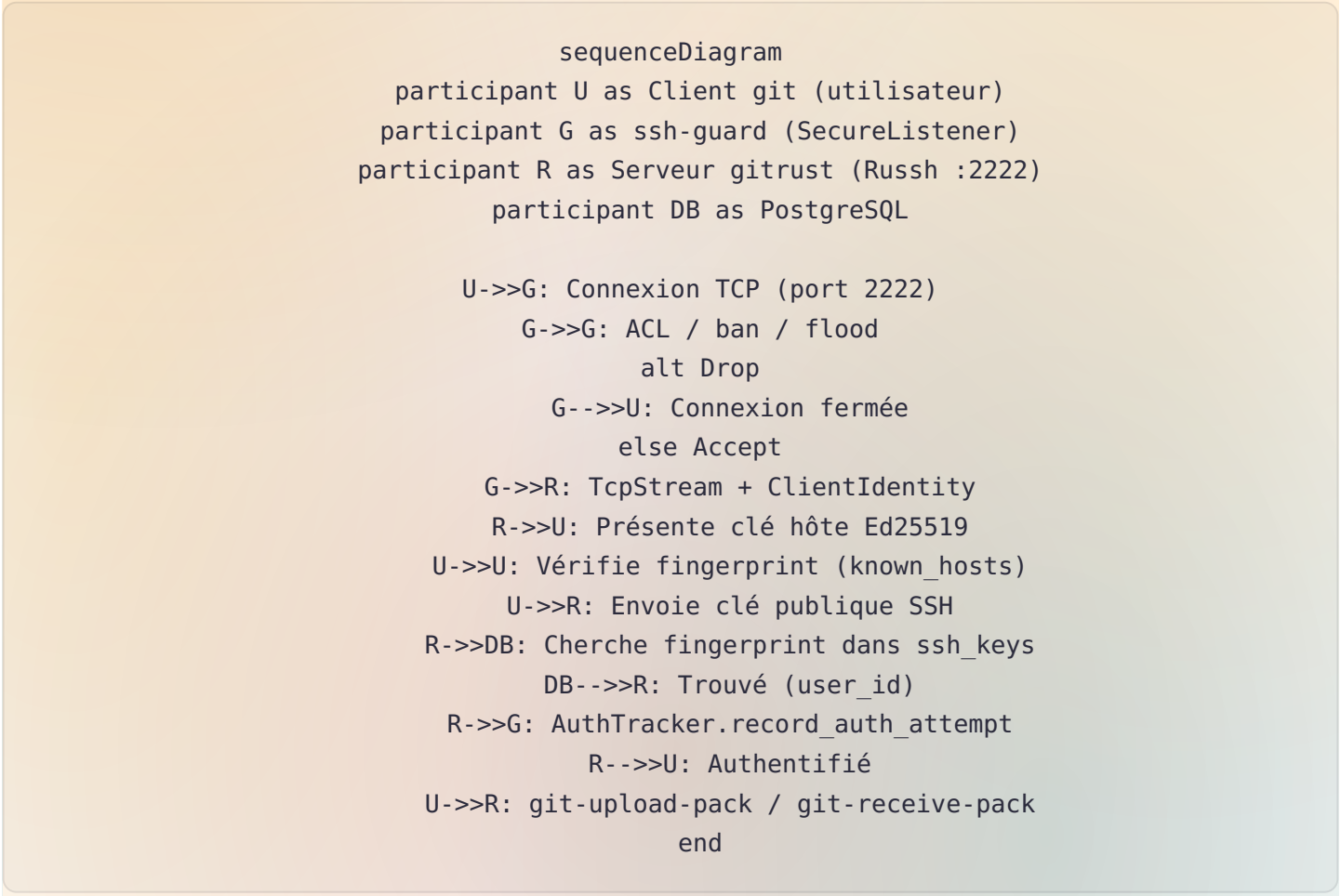
1.3.11 Dépanner les problèmes SSH (clés, ports, fingerprints)

À qui s'adresse cette page

Administrateurs confrontés à des erreurs lors de l'authentification SSH ou des opérations `git clone` / `git push` via SSH sur gitrust.

Architecture SSH de gitrust

Gitrust intègre son propre serveur SSH via la bibliothèque **Russh** — ce n'est pas le démon `sshd` du système. Il écoute par défaut sur le port **2222**. Depuis l'introduction de la crate `gitrust-ssh-guard`, **toute connexion TCP** passe d'abord par un sas (`SecureListener`) qui peut la dropper avant le handshake SSH (ban, flood, en-tête PROXY invalide). Voir Configurer ssh-guard et ssh-guard : détection d'attaques SSH.



Erreur 1 : `Connection refused` sur le port 2222

© Gitrust

```
ssh: connect to host SERVEUR port 2222: Connection refused
```

Diagnostic :

```
# Depuis le serveur : vérifier que gitrust écoute
ss -tlnp | grep 2222
# Attendu : LISTEN 0 ... *:2222 ... gitrust

# Depuis l'extérieur
nc -zv SERVEUR 2222
```

Causes et corrections :

Cause	Correction
gitrust n'est pas démarré	`sudo systemctl start gitrust`
`SSH_PORT` mal configuré dans `.env`	Vérifier `SSH_PORT=2222` et redémarrer
Firewall bloque le port	`sudo ufw allow 2222/tcp` ou règle iptables équivalente
`SSH_LISTEN_ADDR=127.0.0.1`	Le serveur SSH n'est accessible que localement — changer en `0.0.0.0` si accès externe souhaité

Erreur 2 : `WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED`

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING A MITM ATTACK!
```

Cause : la clé SSH hôte du serveur a changé (migration vers un nouveau serveur, régénération de la clé, restauration depuis une sauvegarde différente).

Vérifier le nouveau fingerprint :

© Gitrust

```
# Sur le serveur
sudo -u gitrust ssh-keygen -l -f /opt/gitrust/data/ssh_host_ed25519_key.pub
# Exemple : SHA256:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX gitrust-host-key (ED25519
```

Si ce fingerprint correspond à un changement légitime (migration, etc.), demandez aux utilisateurs de supprimer l'ancienne entrée de leur `known_hosts` :

```
# Sur la machine du client git
ssh-keygen -R "[SERVEUR]:2222"
# Puis re-cloner ou accepter le nouveau fingerprint
```

> ***Si vous ne reconnaissez pas ce changement**, il peut indiquer une compromission. Investiguez avant de continuer.*

Erreur 3 : `Permission denied (publickey)`

```
git@SERVEUR: Permission denied (publickey).
fatal: Could not read from remote repository.
```

Diagnostic pas à pas :

```
# 1. Tester la connexion avec verbosité maximale
ssh -vvv -p 2222 git@SERVEUR

# 2. Vérifier quelle clé est proposée par le client
# Dans la sortie -vvv, chercher :
# debug1: Offering public key: /home/user/.ssh/id_ed25519 ED25519 SHA256:...
# debug1: Authentications that can continue: publickey

# 3. Obtenir le fingerprint de votre clé locale
ssh-keygen -l -f ~/.ssh/id_ed25519.pub
# SHA256:YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY (ED25519)
```

Comparez ce fingerprint avec celui enregistré dans gitrust pour votre compte :

1. Connectez-vous à l'interface web gitrust
2. Accédez à **Paramètres → Clés SSH**
3. Vérifiez que le fingerprint correspond

© Gitrust

Causes courantes :

Cause	Correction
Aucune clé SSH enregistrée dans gitrust	Ajouter la clé publique dans les paramètres du compte
La clé proposée par le client n'est pas celle enregistrée	Vérifier `~/.ssh/config` et l'agent SSH
L'utilisateur est désactivé dans gitrust	L'administrateur doit réactiver le compte via `/admin/users`
La clé enregistrée a été révoquée	Enregistrer une nouvelle clé publique

Vérifier les permissions des fichiers de clés côté client :

```
chmod 700 ~/.ssh
chmod 600 ~/.ssh/id_ed25519
chmod 644 ~/.ssh/id_ed25519.pub
```

Erreur 4 : `Could not resolve hostname`

```
ssh: Could not resolve hostname SERVEUR: Name or service not known
```

Cause : le nom de domaine dans l'URL de clone ne correspond pas à `SSH_PUBLIC_HOST` ou n'est pas résolvable.

Vérifier la configuration :

```
# Dans .env
grep SSH_PUBLIC_HOST /opt/gitrust/.env
# Doit correspondre au FQDN ou à l'IP utilisé dans les URLs de clone

# Tester la résolution DNS
nslookup VOTRE_SSH_PUBLIC_HOST
```


© Gitrust

Erreur 5 : Timeout ou connexion très lente

```
ssh: connect to host SERVEUR port 2222: Operation timed out
```

ou connexion qui s'établit mais met 30+ secondes.

Causes et corrections :

Cause	Correction
Résolution DNS inverse lente	Ajouter `UseDNS no` dans la config SSH hôte (non applicable à Russh — configurable via `SSH_LISTEN_ADDR` côté gitrust)
Firewall qui filtre silencieusement (DROP vs REJECT)	`nc -zv SERVEUR 2222` pour distinguer `refused` (REJECT) de timeout (DROP)
Trop de connexions simultanées	Vérifier les logs gitrust pour des erreurs de pool de threads

Erreur 6 : la connexion est droppée par ssh-guard

Symptôme côté client : la connexion TCP est acceptée puis fermée immédiatement, sans aucun message d'authentification SSH. Ce comportement est typique d'un drop ssh-guard (ban actif, flood, en-tête PROXY refusé).

Diagnostic :

```
# Si SSH_GUARD_LOG_TARGET=both ou file
sudo tail -100 /var/log/gitrust-ssh-guard.json \
  | jq 'select(.event=="connection_dropped")'

# Sinon (target=stderr) — passer par journald
```


© Gitrust

```
sudo journalctl -u gitrust --since "5 min ago" --no-pager \
  | grep '"event":"connection_dropped"' | tail -10
```

`reason` retourné	Cause	Correction
`banned`	IP couverte par un ban actif (auto ou denylist admin)	Lever le ban via la table `ssh_guard_bans` ou attendre l'expiration. Voir Configurer ssh-guard §10.4.
`flood_limit`	Trop de connexions/sec depuis cette IP	Allowlister l'IP si c'est un partenaire CI, ou monter `SSH_GUARD_CONN_FLOOD_PER_SEC`.
`untrusted_proxy`	En-tête PROXY reçu d'une IP non listée	Vérifier `SSH_GUARD_TRUSTED_PROXIES` ; si nginx tourne sur le même hôte, le défaut `127.0.0.1/32,::1/128` du profil `nginx` doit suffire.
`proxy_header_invalid` ou `proxy_header_missing`	nginx/HAProxy n'envoie pas le header attendu	Vérifier `proxy_protocol on;` dans nginx (ou `send-proxy-v2` dans HAProxy). En transition, basculer temporairement `SSH_GUARD_PROXY_PROTOCOL_STRICT=false`.

Diagnostic ssh-guard rapide

Lire le flux d'événements en direct

```
# Dernier événement par IP (ssh-guard log target = stderr/journald)
sudo journalctl -u gitrust -f | grep '"event":"' | jq -c '.'

# Idem si SSH_GUARD_LOG_TARGET=file ou both
sudo tail -f /var/log/gitrust-ssh-guard.json | jq -c '.'
```

Mode dry-run pour reproduire un blocage sans persister

Si vous suspectez qu'un seuil est trop agressif, passez `SSH_GUARD_DRY_RUN=true` dans `.env` puis redémarrez. Les détecteurs continuent à émettre les `ip_banned` (signal pour fail2ban / observabilité) **sans** que ssh-guard pose réellement le ban. Vous pouvez observer le motif sur 24-48 h avant de décider.

Vérifier que la vraie IP est bien extraite (profils nginx/haproxy)

© Gitrust

```
# Faire un git ls-remote depuis une IP externe connue, puis :
sudo journalctl -u gitrust --since "1 min ago" --no-pager \
  | grep '"event":"connection_accepted"' | tail -1 | jq .ip
# Doit retourner l'IP externe, PAS 127.0.0.1
```

Désactiver temporairement ssh-guard

```
# .env
SSH_GUARD_ENABLED=false
sudo systemctl restart gitrust
```

ssh-guard devient un pass-through complet (aucune détection, aucun ban). À utiliser **seulement** pour confirmer qu'un problème vient bien de la couche guard ; remettre `true` immédiatement après.

Pour le détail des événements, voir Événements ssh-guard (JSON).

Consulter les logs SSH

```
# Logs du service gitrust (contient les événements SSH)
sudo journalctl -u gitrust -f | grep -i "ssh\|auth\|key\|fingerprint"

# Filtrer les erreurs d'authentification
sudo journalctl -u gitrust --since "1 hour ago" --no-pager \
  | grep -i "auth.*failed\|permission denied\|invalid key"
```

Pour aller plus loin

- Variables d'environnement — section SSH
- Variables d'environnement — section SSH_GUARD_*
- Configurer ssh-guard
- Événements ssh-guard (JSON)
- Ports et services
- Gérer les utilisateurs : révoquer une clé SSH

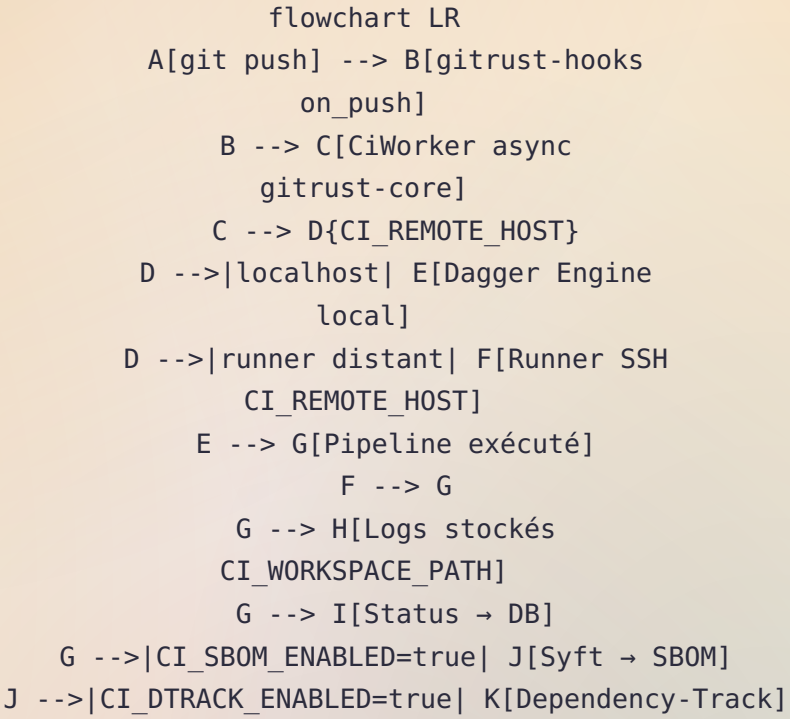
© Gitrust

1.3.12 Dépanner la CI intégrée (Dagger)

À qui s'adresse cette page

Administrateurs confrontés à des pipelines bloqués, des workers CI qui ne répondent plus, ou des erreurs SBOM/Dependency-Track.

Architecture CI de gitrust



Vérifier l'état de la CI

Pipeline bloqué à l'état `pending`

Un pipeline reste en `pending` si le worker ne le prend pas en charge.

```
# Vérifier les logs du worker CI
sudo journalctl -u gitrust -n 100 --no-pager | grep -i "ci\|pipeline\|worker\|dagger"
```


© Gitrust

```
# Vérifier que CI_ENABLED est bien à true
grep CI_ENABLED /opt/gitrust/.env
```

Si `CI_ENABLED=false`, aucun pipeline ne s'exécute. Pour désactiver temporairement la CI sans arrêter gitrust :

```
# Désactiver la CI (les pipelines s'accumulent en pending mais ne s'exécutent pas)
sed -i 's/^CI_ENABLED=.*/CI_ENABLED=false/' /opt/gitrust/.env
sudo systemctl restart gitrust
```

Nombre maximum de pipelines concurrentes atteint

```
grep CI_MAX_CONCURRENT /opt/gitrust/.env
# Défaut : 4 — augmenter si la machine a les ressources
```

Les pipelines supplémentaires attendent en file. Vérifiez qu'aucun pipeline n'est bloqué indéfiniment (timeout `CI_DEFAULT_TIMEOUT`, défaut : 3600 s).

Erreur : `dagger: command not found`

Le binaire `dagger` n'est pas dans le PATH de l'utilisateur `gitrust`.

```
# Vérifier depuis l'utilisateur gitrust
sudo -u gitrust which dagger
sudo -u gitrust dagger version
```

Installation de Dagger :

```
curl -fsSL https://dl.dagger.io/dagger/install.sh | sudo sh
# Le binaire est installé dans /usr/local/bin/dagger
```

Si Dagger est installé dans un chemin non standard, précisez-le dans `.env` :

```
CI_DAGGER_BIN=/usr/local/bin/dagger
```

© Gitrust

Erreur : connexion au runner distant échoue

Si `CI_REMOTE_HOST` pointe vers une machine distante :

```
# Tester la connexion SSH depuis l'utilisateur gitrust
sudo -u gitrust ssh -i /opt/gitrust/data/ci_runner_key \
  -p ${CI_REMOTE_SSH_PORT:-22} \
  ${CI_REMOTE_USER:-$USER}@${CI_REMOTE_HOST} \
  "dagger version"
```

Causes courantes :

Symptôme	Cause	Correction
`Permission denied (publickey)`	Clé CI runner non autorisée sur le runner	Ajouter la clé publique dans `~/.ssh/authorized_keys` du runner
`Connection refused`	Le runner est éteint ou le port SSH est fermé	Vérifier l'état du runner et le firewall
`dagger: command not found` sur le runner	Dagger non installé sur la machine distante	Installer Dagger sur le runner
Timeout rsync	`CI_REMOTE_PATH` inexistant sur le runner	`ssh runner "mkdir -p /opt/gitrust-ci"`

Erreur SBOM : `syft: command not found`

```
# Vérifier depuis l'utilisateur gitrust
sudo -u gitrust which syft

# Installer Syft
curl -sSfL https://raw.githubusercontent.com/anchore/syft/main/install.sh | sudo sh -s

# Ou préciser le chemin dans .env
CI_SYFT_BIN=/usr/local/bin/syft
```

Si vous ne souhaitez pas utiliser SBOM :

© Gitrust

```
CI_SBOM_ENABLED=false
CI_DTRACK_ENABLED=false
```

Erreur Dependency-Track : push SBOM échoue

```
# Vérifier la connectivité vers Dependency-Track
curl -s -H "X-Api-Key: ${CI_DTRACK_API_KEY}" \
  "${CI_DTRACK_URL}/v1/project" | head -100

# Attendu : liste JSON des projets
# Erreur 401 : clé API invalide ou expirée
# Connection refused : CI_DTRACK_URL incorrect ou Dtrack non démarré
```

Vérifiez dans les logs gitrust :

```
sudo journalctl -u gitrust --since "1 hour ago" --no-pager \
  | grep -i "dtrack\|sbom\|dependency"
```

Timeout de pipeline

Si un pipeline dépasse `CI_DEFAULT_TIMEOUT` (défaut : 3600 s = 1 h), il est annulé avec le statut `timeout`.

Pour les projets avec des builds très longs (compilation Rust complète, etc.) :

```
# Augmenter le timeout global
CI_DEFAULT_TIMEOUT=7200   # 2 heures

# Ou surcharger par pipeline dans .gitrust-ci.yml :
# pipeline:
#   timeout: 7200
```


© Gitrust

Nettoyer l'espace disque CI

Les artefacts et workspaces CI s'accumulent dans `CI_WORKSPACE_PATH` (défaut : `/tmp/gitrust-ci`) et les logs dans la base de données.

```
# Espace occupé par les workspaces CI
du -sh /tmp/gitrust-ci/

# Nettoyer manuellement (uniquement si aucun pipeline en cours)
sudo systemctl stop gitrust
sudo rm -rf /tmp/gitrust-ci/*
sudo systemctl start gitrust
```

Les logs en base sont purgés automatiquement selon `CI_LOG_RETENTION_DAYS` (défaut : 30 jours).

Désactiver la CI temporairement

Pour désactiver toute exécution CI sans arrêter gitrust (utile lors d'une maintenance) :

```
# Dans .env
CI_ENABLED=false
sudo systemctl restart gitrust
```

Les pipelines déclenchés pendant cette période restent en `pending` et peuvent être relancés une fois la CI réactivée.

Pour aller plus loin

- Variables d'environnement — section CI/CD
- Comprendre la CI Dagger : Easy Mode vs Power Mode

© Gitrust

1.3.13 Gérer les utilisateurs via le panel d'administration

À qui s'adresse cette page

Administrateurs qui gèrent les comptes utilisateurs de leur instance gitrust depuis l'interface `/admin/users`.

Accéder au panel utilisateurs

Connectez-vous avec un compte administrateur et accédez à :

```
https://VOTRE_INSTANCE/admin/users
```

La liste affiche pour chaque utilisateur : nom d'utilisateur, adresse e-mail, date de création, statut du compte (actif/désactivé), rôle (admin ou utilisateur standard), et si la vérification e-mail a été effectuée.

Créer un utilisateur

Si l'inscription publique est désactivée (`allow_registration=false` dans `/admin/settings`), seuls les administrateurs peuvent créer des comptes.

1. Cliquez **Nouvel utilisateur**
2. Renseignez : nom d'utilisateur, adresse e-mail, mot de passe temporaire
3. Cochez **Forcer le changement de mot de passe à la prochaine connexion** si souhaité
4. Cliquez **Créer**

© Gitrust

L'utilisateur reçoit un e-mail de validation si `EMAIL_VALIDATION_REQUIRED` est activé.

> ***Conseil** : pour les instances privées (équipe interne), désactivez l'inscription publique dans `/admin/settings` et créez les comptes manuellement pour garder le contrôle.*

Désactiver un compte utilisateur

La désactivation empêche la connexion sans supprimer les données (dépôts, issues, commentaires restent intacts).

1. Dans `/admin/users`, cliquez sur l'utilisateur concerné
2. Cliquez **Désactiver le compte**
3. Confirmez

Après désactivation : - L'utilisateur ne peut plus se connecter (HTTP ou SSH) - Ses dépôts restent accessibles aux autres membres des équipes - La désactivation est réversible : cliquez **Réactiver** pour rétablir l'accès

---

Supprimer un compte utilisateur

> *La suppression est **irréversible**. Les dépôts dont l'utilisateur est propriétaire doivent être transférés ou supprimés au préalable.*

1. Désactivez d'abord le compte (étape ci-dessus)
2. Cliquez **Supprimer le compte**
3. Confirmez en saisissant le nom d'utilisateur

Les données supprimées : compte utilisateur, clés SSH, personal access tokens, sessions actives.
Les données conservées : dépôts transférés, issues/commentaires (auteur anonymisé).

Forcer la réinitialisation du mot de passe

Utile si un compte est suspecté d'être compromis ou si l'utilisateur a oublié son mot de passe.

1. Ouvrez la fiche de l'utilisateur dans `/admin/users`
2. Cliquez **Forcer la réinitialisation du mot de passe**
3. L'utilisateur reçoit un e-mail avec un lien de reset (valable `PASSWORD_RESET_EXPIRATION_HOURS` heures, défaut : 1 h)

© Gitrust

Si le compte n'a pas d'adresse e-mail valide ou si le SMTP n'est pas configuré, définissez un nouveau mot de passe directement :

1. Ouvrez la fiche utilisateur
2. Cliquez **Définir un nouveau mot de passe**
3. Saisissez le mot de passe temporaire et cochez **Forcer le changement à la prochaine connexion**

Forcer l'activation du 2FA

Pour exiger que tous les utilisateurs activent l'authentification à deux facteurs (TOTP), configurez le paramètre global dans `/admin/settings` :

1. Accédez à `/admin/settings`
2. Activez **Forcer le 2FA pour tous les utilisateurs** (`totp_force_enabled=true`)
3. Enregistrez

Après activation, tout utilisateur sans 2FA configuré est redirigé vers la page de configuration TOTP à sa prochaine connexion. L'accès à l'application est bloqué jusqu'à la configuration.

Pour forcer le 2FA uniquement pour un utilisateur spécifique :

1. Ouvrez la fiche de l'utilisateur dans `/admin/users`
2. Cliquez **Révoquer le 2FA** (si actif mais compromis) pour forcer une reconfiguration

---

Gérer les permissions administrateur

Accorder les droits administrateur

1. Ouvrez la fiche de l'utilisateur
2. Cliquez **Accorder le rôle administrateur**
3. Confirmez

Un administrateur a accès à `/admin` complet : gestion des utilisateurs, paramètres, équipes, logs d'audit.

Révoquer les droits administrateur

1. Ouvrez la fiche de l'utilisateur
2. Cliquez **Révoquer le rôle administrateur**

> *Vous ne pouvez pas révoquer vos propres droits administrateur. Un autre administrateur doit effectuer cette opération.*

© Gitrust

Droits sur les dépôts (niveau ressource)

Les droits sur les dépôts spécifiques (Reader, Developer, Maintainer) sont gérés au niveau de chaque dépôt ou via les équipes — pas depuis `/admin/users`. Voir le modèle de permissions pour les détails.

Pour aller plus loin

- Auditer l'instance — journal des actions admin
- Paramètres dynamiques — `allow_registration`, `totp_force_enabled`
- Schéma de base de données — table `users`

© Gitrust

1.3.14 Auditer une instance gitrust

À qui s'adresse cette page

Administrateurs qui souhaitent consulter le journal d'audit, filtrer les événements suspects, exporter les logs et les corréler avec les journaux système.

Le journal d'audit dans `/admin/audit`

Gitrust enregistre toutes les actions significatives dans la table `audit_log`. Chaque entrée contient : l'acteur (utilisateur), l'action effectuée, la ressource concernée, l'adresse IP source et l'horodatage.

Les actions auditées par défaut (`audit_log_actions` dans `app_settings`) : - `create` — création d'un dépôt, utilisateur, équipe, etc. - `update` — modification de paramètres, de profil - `delete` — suppression d'une ressource - `reset_password` — réinitialisation de mot de passe

Accédez au journal : `https://VOTRE_INSTANCE/admin/audit`

© Gitrust

Filtrer les événements

L'interface `/admin/audit` propose des filtres combinables :

Filtre	Description
Utilisateur	Filtrer par nom d'utilisateur de l'acteur
Action	Filtrer par type d'action (`create`, `update`, `delete`, `reset_password`)
Période	Sélectionner une plage de dates
Ressource	Filtrer par type de ressource (repository, user, team…)

Exemple de requête typique : « toutes les suppressions de dépôts depuis 30 jours »

1. Action : `delete`
2. Période : 30 derniers jours
3. Cliquez **Filtrer**

Exporter les logs en CSV

1. Appliquez les filtres souhaités
2. Cliquez **Exporter CSV**
3. Le fichier téléchargé contient les colonnes : `timestamp`, `actor_username`, `action`, `resource_type`, `resource_id`, `ip_address`, `details`

Ce fichier peut être importé dans un tableur ou un SIEM pour des analyses avancées.

Interroger la base de données directement

Pour des requêtes avancées non disponibles dans l'UI :

```
-- Connexions de l'administrateur sur les 7 derniers jours
SELECT
    al.created_at,
    u.username,
    al.action,
    al.resource_type,
    al.ip_address
FROM audit_log al
JOIN users u ON al.actor_id = u.id
```


© Gitrust

```
WHERE u.username = 'admin'
  AND al.created_at > NOW() - INTERVAL '7 days'
ORDER BY al.created_at DESC;
```

```
-- Suppressions massives (plus de 5 suppressions par utilisateur sur 1 heure)
SELECT
    u.username,
    COUNT(*) AS nb_suppressions,
    MIN(al.created_at) AS premiere,
    MAX(al.created_at) AS derniere
FROM audit_log al
JOIN users u ON al.actor_id = u.id
WHERE al.action = 'delete'
  AND al.created_at > NOW() - INTERVAL '1 hour'
GROUP BY u.username
HAVING COUNT(*) > 5
ORDER BY nb_suppressions DESC;
```

Configurer le niveau d'audit

Le niveau et les actions auditées sont des paramètres dynamiques modifiables via `/admin/settings` :

Paramètre	Valeur	Description
`audit_log_level`	`INFO`	Niveau de détail (INFO, WARN, ERROR)
`audit_log_actions`	JSON array	Actions enregistrées

Pour ajouter l'action `login` aux événements audités :

1. Accédez à `/admin/settings`
2. Trouvez `audit_log_actions`
3. Ajoutez `"login"` au tableau JSON : `["create","update","delete","reset_password","login"]`
4. Enregistrez (effet immédiat, pas de redémarrage nécessaire)

© Gitrust

Corréler avec `journalctl`

Les logs systemd de gitrust complètent le journal d'audit en capturant les erreurs techniques et les événements bas niveau (connexions SSH refusées, erreurs de migration, etc.).

```
# Logs gitrust des 24 dernières heures
sudo journalctl -u gitrust --since "24 hours ago" --no-pager

# Filtrer les erreurs et avertissements uniquement
sudo journalctl -u gitrust -p err..warning --since "24 hours ago" --no-pager

# Corréler avec une adresse IP spécifique
sudo journalctl -u gitrust --since "24 hours ago" --no-pager \
  | grep "192.168.1.100"

# Suivre les logs en temps réel
sudo journalctl -u gitrust -f
```

Signaux d'alerte à surveiller

```
# Tentatives d'authentification SSH échouées répétées
sudo journalctl -u gitrust --since "1 hour ago" --no-pager \
  | grep -i "auth.*fail\|invalid.*key\|permission denied" \
  | wc -l

# Réponses 429 (rate limiting déclenché)
sudo journalctl -u gitrust --since "1 hour ago" --no-pager \
  | grep "429" | awk '{print $NF}' | sort | uniq -c | sort -rn | head
```

Conservation des logs

La rétention des logs d'audit en base de données n'a pas de limite automatique dans la version actuelle — les entrées s'accumulent indéfiniment. Pour une instance à long terme, planifiez une purge périodique :

```
-- Supprimer les entrées d'audit de plus de 1 an (hors suppressions et reset mdp)
DELETE FROM audit_log
```


© Gitrust

```
WHERE created_at < NOW() - INTERVAL '1 year'
  AND action NOT IN ('delete', 'reset_password');
```

> *Vérifiez les exigences de conservation de votre organisation avant de purger (RGPD, ANSSI PA-074 recommande 12 mois minimum pour les événements de sécurité).*

Pour aller plus loin

- Conformité ANSSI PA-074 — journalisation
- Schéma de base de données — table `audit_log`
- Paramètres dynamiques — `audit_log_level`, `audit_log_actions`

© Gitrust

1.4 Référence

1.4.1 Variables d'environnement

Référence exhaustive de toutes les variables d'environnement de gitrust. Ces variables se définissent dans le fichier `.env` (ou `.env.production`) et sont chargées **une seule fois au démarrage**. Modifier le fichier sans redémarrer le service n'a aucun effet.

> ***Principe d'autonomie** : ce tableau contient intégralement le contenu documenté de `.env.example`. La documentation ne référence pas le fichier source par lien — tout est ici.*

Les variables marquées **SEC** ont des contraintes de sécurité vérifiées au démarrage : gitrust refuse de démarrer si elles ne sont pas conformes.

1. Base de données

Variable	Défaut	Obligatoire	Description
`DATABASE_URL`	—	**Oui**	URL de connexion PostgreSQL. Format : `postgres://USER:PASSWORD@HOST:PORT/DB`. Le mot de passe doit être fort en production. PostgreSQL ne doit jamais binder sur `0.0.0.0` en production.

Exemples :

```
# Développement local (Docker Compose)
DATABASE_URL=postgres://gitrust:gitrust@localhost:5432/gitrust

# Production (réseau privé)
DATABASE_URL=postgres://gitrust:MOT_DE_PASSE_FORT@db.internal:5432/gitrust
```


© Gitrust

2. Serveur HTTP

Variable	Défaut	Description
`SERVER_HOST`	`0.0.0.0`	Adresse d'écoute HTTP. En production derrière un reverse-proxy : `127.0.0.1`. En développement : `0.0.0.0`.
`SERVER_PORT`	`3000`	Port HTTP. Convention gitrust : `4000` en production.

3. Logging

Variable	Défaut	Description
`RUST_LOG`	`debug`	Niveau de log (syntaxe `tracing_subscriber::EnvFilter`). Valeurs : `error`, `warn`, `info`, `debug`, `trace`. Syntaxe avancée : `"info,gitrust_core=debug,sea_orm=warn"`.

4. Serveur SSH Git

Variable	Défaut	Description
`SSH_HOST_KEY_PATH`	`./data/ssh_host_ed25519_key`	Chemin de la clé hôte SSH Ed25519. Générée automatiquement au premier démarrage si absente. **SEC** : permissions `600`, propriétaire = utilisateur systemd. En production : chemin absolu.
`SSH_PORT`	`2222`	Port d'écoute du serveur SSH Russh (intégré, distinct de `sshd`).
`SSH_LISTEN_ADDR`	`0.0.0.0`	Adresse d'écoute SSH. `127.0.0.1` si derrière un proxy nginx stream.
`SSH_PUBLIC_HOST`	`localhost`	Nom d'hôte affiché dans les URLs de clone SSH dans l'UI. En production : FQDN public.
`MAX_SSH_KEY_SIZE`	`16384`	Taille maximale d'une clé SSH publique acceptée (octets).

© Gitrust

5. JWT et authentification

Variable	Défaut	Description
`JWT_SECRET`	—	**SEC OBLIGATOIRE** — Secret de signature JWT. Minimum 32 bytes. Générer avec `openssl rand -hex 64`. Les valeurs exemples (`change-me-in-production`, etc.) sont explicitement rejetées au démarrage. Changer cette valeur invalide toutes les sessions actives.
`JWT_EXPIRATION_MINUTES`	`15`	Durée de vie d'un JWT d'accès (minutes).
`JWT_ISSUER`	`mon-app`	Valeur du claim `iss` dans le JWT. En production : FQDN public.
`REFRESH_TOKEN_EXPIRATION_DAYS`	`7`	Durée du refresh token (jours). Rotation à chaque usage.
`REMEMBER_ME_EXPIRATION_DAYS`	`30`	Durée étendue quand « Se souvenir de moi » est coché. **SEC** : doit être ≥ `REFRESH_TOKEN_EXPIRATION_DAYS` et ≤ 90 jours. Au-delà de 60 jours : warning dans les logs.

6. Sessions

Variable	Défaut	Description
`SESSION_TIMEOUT_MINUTES`	`30`	Timeout d'inactivité d'une session (minutes).
`SESSION_BACKEND`	`seaorm`	Backend de stockage : `memory` (volatil, dev uniquement), `seaorm` (PostgreSQL, recommandé), `redis` (multi-instances, nécessite `REDIS_URL`).
`REDIS_URL`	—	URL Redis (obligatoire si `SESSION_BACKEND=redis`). Exemple : `redis://localhost:6379`.

© Gitrust

7. Rate limiting

Variable	Défaut	Description
`RATE_LIMIT_LOGIN_PER_MINUTE`	`5`	Tentatives de connexion par IP par minute. Protection contre le brute force.
`RATE_LIMIT_REFRESH_PER_MINUTE`	`10`	Rafraîchissements de token par IP par minute.
`RATE_LIMIT_GENERAL_PER_MINUTE`	`100`	Limite globale sur tous les autres endpoints. `0` = désactivé (déconseillé).

8. Cookies et debug

Variable	Défaut	Description
`APP_DEBUG`	`false`	**SEC** : `true` = logs verbeux, cookies non sécurisés (HTTP local). `false` = production, enforce `COOKIE_SECURE=true` même si défini à `false`. Ne jamais mettre `true` en production.
`COOKIE_SECURE`	`false`	Cookies envoyés uniquement via HTTPS. **SEC** : forcé à `true` si `APP_DEBUG=false`.
`COOKIE_SAME_SITE`	`Lax`	Politique SameSite. `Strict` (le plus sûr, casse les redirections OAuth cross-site), `Lax` (compatible OAuth), `None` (nécessite `Secure=true`).

© Gitrust

9. Bootstrap admin initial

Variable	Défaut	Description
`ADMIN_USERNAME`	`admin`	Nom d'utilisateur du compte admin créé au premier démarrage (si la table `users` est vide). 3-64 caractères alphanumériques + `_-`.
`ADMIN_EMAIL`	`admin@example.com`	Adresse e-mail du compte admin initial. Doit contenir `@` + domaine avec `.`.
`ADMIN_PASSWORD`	—	**SEC OBLIGATOIRE** — Mot de passe du compte admin initial. Minimum 8 caractères. À remplacer immédiatement après le premier démarrage.
`CREATE_DEFAULT_ADMIN`	`true`	Mettre à `false` une fois d'autres admins créés pour désactiver la création automatique.

© Gitrust

10. Email SMTP

© Gitrust

Variable	Défaut	Description
`SMTP_HOST`	—	Serveur SMTP. Obligatoire pour activer les e-mails. Sans cette variable, gitrust démarre mais les fonctionnalités e-mail sont désactivées.
`SMTP_PORT`	`587`	Port SMTP : `25` (non chiffré, interdit en prod), `465` (TLS natif), `587` (STARTTLS, recommandé), `1025` (dev/Mailpit).
`SMTP_USER`	—	Identifiant SMTP. Souvent = adresse e-mail expéditrice.
`SMTP_PASSWORD`	—	Mot de passe applicatif SMTP (pas le mot de passe du compte e-mail).
`SMTP_FROM`	—	Adresse expéditeur. Doit être valide (validée par `is_valid_email`). En prod : valider DKIM/SPF sur le domaine.
`SMTP_FROM_NAME`	—	Nom affiché dans le champ « De : » des e-mails.
`SMTP_USE_TLS`	`true`	Activer TLS natif (port 465).
`SMTP_USE_STARTTLS`	`true`	Activer STARTTLS (port 587). **SEC** : ne jamais désactiver les deux en production.
`SMTP_ACCEPT_INVALID_CERTS`	`false`	**SEC** : accepter les certificats SMTP invalides. Warning au démarrage si `true`. Uniquement pour les serveurs de mail internes en développement.
`SMTP_CA_CERT_PATH`	—	Chemin vers un CA personnalisé (PEM) pour valider le certificat SMTP.
`EMAIL_BASE_URL`	`http://localhost:3000`	URL de base pour les liens dans les e-mails. **Obligatoire en prod** sinon les liens pointent vers localhost. Doit commencer par `http://` ou `https://`.
`EMAIL_QUEUE_BATCH_SIZE`	`10`	Nombre d'e-mails traités par tick du worker.
`EMAIL_QUEUE_RETRY_ATTEMPTS`	`5`	Tentatives d'envoi avant abandon.

© Gitrust

Variable	Défaut	Description
`EMAIL_QUEUE_RETRY_DELAY_SECONDS`	`300`	Délai entre deux tentatives (secondes).
`EMAIL_QUEUE_INTERVAL_SECS`	`30`	Intervalle du tick du worker e-mail (secondes).
`EMAIL_VALIDATION_REQUIRED`	`true`	Exiger la validation e-mail à l'inscription. Note : cette variable est documentée dans `.env.example` mais la valeur effective est contrôlée par le paramètre dynamique `validation_email_required` en base de données.

11. Inscription

Variable	Défaut	Description
`ALLOW_REGISTRATION`	`false`	Autoriser l'inscription publique. Note : valeur effective contrôlée par le paramètre dynamique `allow_registration` en base de données — voir Paramètres dynamiques.

12. Application (branding)

Variable	Défaut	Description
`APP_NAME`	`Mon Application`	Nom affiché dans l'UI, les e-mails et la balise `<title>`.
`APP_THEME`	`light`	Thème visuel. Valeurs : `light`, `dark`.
`DEFAULT_LOCALE`	`fr`	Langue par défaut. Valeurs : `fr`, `en`, `de`, `es`, `pt`, `it`.

© Gitrust

13. IMAP — traitement des bounces (optionnel)

Si `IMAP_HOST` est vide, la fonctionnalité est désactivée sans erreur.

Variable	Défaut	Description
`IMAP_HOST`	—	Serveur IMAP. Obligatoire pour activer le traitement des bounces.
`IMAP_PORT`	`993`	Port IMAP : `993` (TLS direct, recommandé), `143` (STARTTLS).
`IMAP_USER`	= `SMTP_USER`	Identifiant IMAP.
`IMAP_PASSWORD`	= `SMTP_PASSWORD`	Mot de passe IMAP.
`IMAP_USE_TLS`	`true`	TLS direct (port 993).
`IMAP_USE_STARTTLS`	`false`	STARTTLS après connexion (port 143).
`IMAP_ACCEPT_INVALID_CERTS`	`false`	Accepter les certificats invalides (warning si `true`).
`IMAP_MAILBOX`	`INBOX`	Boîte à scanner pour les bounces.
`IMAP_POLL_INTERVAL_SECONDS`	`300`	Intervalle de poll (secondes).
`IMAP_SOCKET_TIMEOUT_SECS`	`30`	Timeout socket read/write.

© Gitrust

14. Tâches de fond — intervalles et expirations

Ces variables ont des valeurs par défaut raisonnables. Ne les ajustez que si vous avez un besoin spécifique.

Variable	Défaut	Description
`PASSWORD_CHANGE_RETRY_INTERVAL_SECS`	`600`	Retry e-mail changement de mot de passe (10 min).
`PASSWORD_CHANGE_RETRY_GAP_MINUTES`	`20`	Espacement minimum entre deux retries.
`PASSWORD_CHANGE_MAX_RETRIES`	`3`	Abandon après N échecs.
`PASSWORD_CHANGE_CLEANUP_INTERVAL_SECS`	`86400`	Nettoyage quotidien des demandes expirées.
`PASSWORD_CHANGE_EXPIRATION_HOURS`	`24`	Validité d'une demande de changement de mot de passe.
`PASSWORD_CHANGE_RATE_LIMIT_MAX`	`3`	Demandes maximum par fenêtre.
`PASSWORD_CHANGE_RATE_LIMIT_WINDOW_SECS`	`3600`	Fenêtre de rate limit (1 heure).
`PASSWORD_RESET_EXPIRATION_HOURS`	`1`	Validité du lien de réinitialisation de mot de passe.
`EMAIL_VALIDATION_EXPIRATION_HOURS`	`24`	Validité du lien de validation e-mail.
`JWT_CLEANUP_INTERVAL_SECS`	`3600`	Purge de la blacklist JWT expirés.
`COOKIE_PREFERENCE_MAX_AGE_SECS`	`31536000`	Durée de vie du cookie de préférences (locale/thème) — 1 an.
`PAGINATION_DEFAULT_PER_PAGE`	`50`	Taille par défaut des pages de liste.
`PAGINATION_MAX_PER_PAGE`	`100`	Taille maximale des pages de liste.

© Gitrust

15. Dépôts Git

Variable	Défaut	Description
`GIT_REPOS_BASE_PATH`	`./data/repos`	Chemin des dépôts bare. Structure : `{base}/{owner}/{repo}.git`. En production : chemin absolu (ex. `/opt/gitrust/data/repos`). **SEC** : path traversal bloqué au niveau service.

16. Fichiers statiques

Variable	Défaut	Description
`STATIC_FILES_PATH`	`./static`	Chemin du dossier des assets statiques (CSS, JS, images). En production avec systemd : chemin absolu (ex. `/opt/gitrust/static`).

17. CI/CD Dagger

Variable	Défaut	Description
`CI_ENABLED`	`true`	Activation globale de la CI. `false` = plus aucune exécution de pipeline, mais l'UI reste accessible.
`CI_DAGGER_BIN`	`dagger`	Chemin du binaire Dagger. En production : chemin absolu recommandé.
`CI_ENGINE_PATH`	`./deployment/ci-engine`	Chemin du module CI (Easy Mode — interprète `.gitrust-ci.yml`).
`CI_MAX_CONCURRENT`	`4`	Pipelines exécutés simultanément. Au-delà : file d'attente.
`CI_DEFAULT_TIMEOUT`	`3600`	Timeout par défaut d'un pipeline (secondes). Surchargeable par pipeline.
`CI_WORKSPACE_PATH`	`/tmp/gitrust-ci`	Dossier de travail temporaire CI (checkouts, artefacts). En production : `/var/lib/gitrust-ci`.
`CI_LOG_RETENTION_DAYS`	`30`	Rétention des logs de pipeline (jours).

© Gitrust

18. Runner CI distant (SSH + rsync)

Variable	Défaut	Description
`CI_REMOTE_HOST`	`localhost`	Host du runner CI. `localhost` = CI sur la même machine.
`CI_REMOTE_USER`	`$USER`	Utilisateur SSH sur le runner.
`CI_REMOTE_PATH`	`/opt/gitrust-ci`	Chemin de travail sur le runner.
`CI_REMOTE_SSH_KEY`	—	Chemin de la clé privée SSH pour le runner. Optionnel si ssh-agent est chargé.
`CI_REMOTE_SSH_PORT`	`22`	Port SSH du runner.

19. Import de dépôts externes

Variable	Défaut	Description
`IMPORT_MAX_CONCURRENT`	`2`	Nombre maximum de clones concurrents.
`IMPORT_TIMEOUT_SECS`	`1800`	Timeout par import (secondes). Augmenter pour les gros monorepos.
`DB_WORKER_POOL_SIZE`	`4`	Pool de connexions DB dédié au worker d'import. Évite la saturation du pool HTTP principal.

© Gitrust

20. SBOM / Dependency-Track

Variable	Défaut	Description
`CI_SBOM_ENABLED`	`false`	Générer un SBOM (Software Bill of Materials) à chaque pipeline via Syft. Nécessite `syft` installé.
`CI_SYFT_BIN`	`syft`	Chemin du binaire Syft.
`CI_DTRACK_ENABLED`	`false`	Pousser le SBOM vers Dependency-Track (nécessite `CI_SBOM_ENABLED=true`).
`CI_DTRACK_URL`	—	URL API de Dependency-Track (ex. `https://dtrack.internal/api`).
`CI_DTRACK_API_KEY`	—	Clé API Dependency-Track (générer dans l'UI Dtrack : Administration > API keys).
`CI_DTRACK_FRONTEND_URL`	—	URL UI Dependency-Track. Si absent : déduite de `CI_DTRACK_URL`.

© Gitrust

21. OAuth / SSO

Variable	Défaut	Description
`OAUTH_ENCRYPTION_KEY`	—	Clé de chiffrement AES-256-GCM des secrets OAuth en base. Générer avec `openssl rand -hex 32`.
`OAUTH_ENABLED`	`false`	Activation globale OAuth. Fallback si pas de valeur en base.
`OAUTH_REDIRECT_BASE_URL`	`http://localhost:3000`	URL de base pour les callbacks OAuth : `{URL}/api/v1/auth/oauth/{provider}/callback`.
`OAUTH_GOOGLE_ENABLED`	`false`	Activer Google OAuth.
`OAUTH_GOOGLE_CLIENT_ID`	—	Client ID Google.
`OAUTH_GOOGLE_CLIENT_SECRET`	—	Client Secret Google.
`OAUTH_GITHUB_ENABLED`	`false`	Activer GitHub OAuth.
`OAUTH_GITHUB_CLIENT_ID`	—	Client ID GitHub.
`OAUTH_GITHUB_CLIENT_SECRET`	—	Client Secret GitHub.
`OAUTH_DISCORD_ENABLED`	`false`	Activer Discord OAuth.
`OAUTH_DISCORD_CLIENT_ID`	—	Client ID Discord.
`OAUTH_DISCORD_CLIENT_SECRET`	—	Client Secret Discord.
`OAUTH_MICROSOFT_ENABLED`	`false`	Activer Microsoft / Azure AD OAuth.
`OAUTH_MICROSOFT_CLIENT_ID`	—	Client ID Azure (Application ID).
`OAUTH_MICROSOFT_CLIENT_SECRET`	—	Client Secret Azure.
`OAUTH_MICROSOFT_TENANT`	`common`	Tenant Azure AD : `"common"` (comptes perso + pro), `"organizations"` (pro uniquement), ou GUID du tenant.

© Gitrust

22. Durcissement SSH (`SSH_GUARD_*`)

Variables consommées par la crate `gitrust-ssh-guard`. Toutes sont préfixées `SSH_GUARD_`. Pour le détail des comportements, voir Configurer ssh-guard et ssh-guard : détection d'attaques SSH.

> ***Modèle de configuration :*** *`SSH_GUARD_PROFILE` sélectionne un preset cohérent (`direct` / `nginx` / `haproxy` / `private` / `custom`). Chaque autre variable override individuellement le défaut du preset.*

22.1 Kill switch et profil

Variable	Défaut	Description
`SSH_GUARD_ENABLED`	`true`	`false` = pass-through complet, ssh-guard ne fait rien. **À éviter sauf urgence**.
`SSH_GUARD_DRY_RUN`	`false`	`true` = les détecteurs tournent et émettent les événements `ip_banned`, mais aucun ban n'est persisté. Idéal pour valider un nouveau seuil.
`SSH_GUARD_PROFILE`	`custom`	Preset : `direct` (Internet direct), `nginx` (derrière nginx stream + PROXY v2), `haproxy` (derrière HAProxy), `private` (réseau interne, détecteurs OFF), `custom` (aucun preset).

© Gitrust

22.2 PROXY protocol

Variable	Défaut	Description
`SSH_GUARD_PROXY_PROTOCOL`	`disabled`	Mode parsing PROXY : `disabled`, `v1` (texte HAProxy legacy), `v2` (binaire nginx stream / HAProxy moderne), `any` (auto-détection v1/v2).
`SSH_GUARD_PROXY_PROTOCOL_STRICT`	`true`	`true` = drop si en-tête PROXY absent ou invalide quand le mode est activé. `false` = fallback sur `peer_addr` avec warn (utile en transition).
`SSH_GUARD_PROXY_PROTOCOL_TIMEOUT_MS`	`3000`	Timeout (ms) pour lire l'en-tête PROXY après accept TCP.
`SSH_GUARD_TRUSTED_PROXIES`	—	**Obligatoire si PROXY activé**. Liste CIDR (séparée par virgules) des proxies autorisés à émettre un en-tête PROXY. Sans cette protection, n'importe qui pourrait forger une IP cliente. Exemple : `127.0.0.1/32,::1/128`.

© Gitrust

22.3 Détecteurs (seuils + fenêtres)

Variable	Défaut	Description
`SSH_GUARD_BRUTE_FORCE_THRESHOLD`	`5`	Nombre d'échecs d'auth depuis une IP avant ban auto. `4294967295` (`u32::MAX`) = détecteur désactivé.
`SSH_GUARD_BRUTE_FORCE_WINDOW_SECS`	`300`	Fenêtre glissante (secondes).
`SSH_GUARD_USER_ENUM_THRESHOLD`	`10`	Nombre d'usernames distincts essayés depuis une IP.
`SSH_GUARD_USER_ENUM_WINDOW_SECS`	`300`	
`SSH_GUARD_KEY_SCAN_THRESHOLD`	`10`	Nombre de fingerprints de clé distincts essayés depuis une IP.
`SSH_GUARD_KEY_SCAN_WINDOW_SECS`	`300`	
`SSH_GUARD_CONN_FLOOD_PER_SEC`	`10`	Cap dur de nouvelles connexions TCP par IP par seconde. `0` ou `u32::MAX` = désactivé.
`SSH_GUARD_CONN_FLOOD_BURST`	`20`	Burst autorisé au-dessus du cap soutenu.
`SSH_GUARD_MAX_CONCURRENT_PER_IP`	`10`	Limite de sessions concurrentes par IP (placeholder, non encore appliqué).

22.4 Ban

Variable	Défaut	Description
`SSH_GUARD_AUTO_BAN_DURATION_SECS`	`3600`	TTL des bans posés par les détecteurs. `0` = ban permanent.

© Gitrust

22.5 Stockage

Variable	Défaut	Description
`SSH_GUARD_STORE_BACKEND`	`hybrid`	`memory` (DashMap, perdu au restart), `postgres` (chaque write en DB), `hybrid` (RAM chaude + write-through DB + rehydrate au boot — défaut recommandé).
`SSH_GUARD_STORE_FLUSH_INTERVAL_MS`	`1000`	Intervalle entre deux flushs RAM → Postgres pour le mode `hybrid`.
`SSH_GUARD_EVENTS_RETENTION_DAYS`	`90`	Rétention de la table `ssh_guard_events` (utilisée par les détecteurs). Au-delà, les lignes sont purgées.

22.6 Observabilité

Variable	Défaut	Description
`SSH_GUARD_LOG_FORMAT`	`json`	`json` (stable, fail2ban-friendly) ou `text` (debug uniquement).
`SSH_GUARD_LOG_TARGET`	`stderr`	`stderr` (via `tracing` → journald), `file` (fichier dédié uniquement), `both` (les deux).
`SSH_GUARD_LOG_FILE`	`/var/log/gitrust-ssh-guard.json`	Chemin du fichier dédié si `LOG_TARGET=file` ou `both`. **Logrotate quasi obligatoire** (sinon le fichier grossit sans limite).
`SSH_GUARD_METRICS_ENABLED`	`true`	Expose les métriques Prometheus (placeholder, exposition future).

22.7 Validations bloquantes au démarrage

gitrust **refuse de démarrer** si :

- `SSH_GUARD_PROXY_PROTOCOL` est activé sans `SSH_GUARD_TRUSTED_PROXIES` (forge d'IP triviale sinon).
- `SSH_GUARD_LOG_TARGET=file` ou `both` sans `SSH_GUARD_LOG_FILE` valide.

Le message d'erreur identifie la variable fautive.

© Gitrust

Récapitulatif local vs production

Dimension	Local (dev)	Production
`DATABASE_URL` host	`localhost`	Réseau privé
`SERVER_HOST`	`0.0.0.0`	`127.0.0.1` (derrière nginx)
`SERVER_PORT`	`3000` ou `4000`	`4000` (nginx devant)
`SSH_LISTEN_ADDR`	`0.0.0.0`	`127.0.0.1` (nginx stream)
`SSH_PUBLIC_HOST`	`localhost`	FQDN public
`COOKIE_SECURE`	`false` (HTTP)	`true` (HTTPS, forcé)
`APP_DEBUG`	`true`	`false` (force sécurité)
`EMAIL_BASE_URL`	`http://localhost:...`	`https://FQDN`
`JWT_SECRET`	Valeur courte OK	`openssl rand -hex 64`
`ADMIN_PASSWORD`	Faible accepté	Fort (≥ 16 caractères)
`ALLOW_REGISTRATION`	`true`	`false` (instance fermée)
`RUST_LOG`	`debug`	`info` ou `warn`
`GIT_REPOS_BASE_PATH`	Relatif (`./data`)	Absolu (`/opt/gitrust`)

Pour aller plus loin

- Paramètres dynamiques `/admin/settings` — variables modifiables à chaud sans redémarrage
- Tutoriel 02 — Installation systemd — exemple de `.env` minimal de production
- Configurer ssh-guard — recettes par profil de déploiement
- Événements ssh-guard (JSON) — schéma des événements émis

© Gitrust

1.4.2 Événements ssh-guard (JSON)

Référence stable du flux d'événements émis par la couche `ssh-guard` du serveur SSH gitrust. Ce format est **garanti stable** pour les consommateurs externes (fail2ban, Loki, Vector, SIEM). Tout changement incompatible passe par un nouveau nom d'événement.

> ***Où trouver ces événements*** *: selon* `SSH_GUARD_LOG_TARGET`*, dans le journald du service (* `stderr`*), dans* `/var/log/gitrust-ssh-guard.json` *(* `file`*), ou les deux (* `both`*). Voir Variables d'environnement — SSH_GUARD_*.*

1. Forme générale

Tous les événements partagent trois champs :

Champ	Type	Description
`event`	string	Nom snake_case du type d'événement (clé de filtrage).
`ts`	string ISO 8601 UTC	Horodatage UTC de l'événement (ex. `2026-04-19T14:32:11.482Z`).
`ip`	string	Adresse IPv4 ou IPv6 réelle du client (après extraction PROXY si applicable).

Champs additionnels selon le type d'événement (voir tables ci-dessous).

© Gitrust

2. Catalogue des événements

`event`	Catégorie	Émis par	Cas d'usage admin
`connection_accepted`	Trafic	`SecureListener`	Audit volume
`connection_dropped`	Décision	`SecureListener` ou `ConnectionFloodDetector`	Volumétrie des refus
`auth_failed`	Authentification	`AuthTracker`	Source principale fail2ban
`auth_succeeded`	Authentification	`AuthTracker`	Audit accès légitimes
`brute_force_detected`	Détection	`BruteForceDetector`	Signal fort fail2ban
`user_enumeration_detected`	Détection	`UserEnumerationDetector`	Signal fort fail2ban
`key_scanning_detected`	Détection	`KeyScanningDetector`	Signal fort fail2ban
`connection_flood_detected`	Détection	`ConnectionFloodDetector`	Signal fort fail2ban
`ip_banned`	Action	`BanManager`	Ban à appliquer côté firewall (fail2ban)
`ip_unbanned`	Action	`BanManager`	Levée de ban (manuelle ou TTL)

© Gitrust

3. Référence par événement

3.1 `connection_accepted`

Une connexion TCP a passé tous les contrôles ssh-guard. Le `russh` handshake va démarrer.

Champ	Type	Description
`event`	`"connection_accepted"`	
`ts`	string	Horodatage UTC
`session_id`	string UUID v4	Identifiant unique de session, présent ensuite dans tous les événements liés (`auth_failed` , `auth_succeeded`)
`ip`	string	IP cliente réelle

```
{"event":"connection_accepted","ts":"2026-04-19T14:32:11.482Z","session_id":"a4c2b8e1-9
```

3.2 `connection_dropped`

ssh-guard a refusé la connexion **avant** le handshake SSH.

Champ	Type	Description
`event`	`"connection_dropped"`	
`ts`	string	Horodatage UTC
`ip`	string	IP source (réelle si PROXY parsé, sinon `peer_addr`)
`reason`	enum string	Voir tableau des raisons ci-dessous

© Gitrust

Valeurs de `reason` :

`reason`	Signification
`banned`	IP couverte par un ban actif (auto ou denylist admin)
`flood_limit`	Cap de connexions/seconde par IP atteint
`proxy_header_missing`	PROXY protocol obligatoire mais en-tête absent (timeout)
`untrusted_proxy`	En-tête PROXY reçu d'un socket pas dans `trusted_proxies`
`proxy_header_invalid`	En-tête PROXY malformé ou version non autorisée
`concurrent_limit`	Limite de sessions concurrentes par IP atteinte (placeholder, non actif)


```
{"event":"connection_dropped","ts":"2026-04-19T14:33:02.117Z","ip":"203.0.113.42","reas
```


3.3 `auth_failed`

Tentative d'authentification SSH refusée. **Source principale pour fail2ban.**

Champ	Type	Description
`event`	`"auth_failed"`	
`ts`	string	Horodatage UTC
`session_id`	string UUID	Lien avec le `connection_accepted`
`ip`	string	IP cliente réelle
`user`	string ou `null`	Nom d'utilisateur tenté (si fourni par le client)
`method`	enum string	Méthode SSH : `none`, `password`, `public_key`, `keyboard_interactive`, `host_based`
`fingerprint`	string ou `null`	Fingerprint SHA256 de la clé publique tentée (forme `SHA256:...`) ou `null` si non `public_key`


```
{"event":"auth_failed","ts":"2026-04-19T14:32:13.221Z","session_id":"a4c2b8e1-9f3d-4d7e
```


© Gitrust

3.4 `auth_succeeded`

Tentative d'authentification SSH validée. Audit des accès légitimes.

Champ	Type	Description
`event`	`"auth_succeeded"`	
`ts`	string	Horodatage UTC
`session_id`	string UUID	
`ip`	string	
`user`	string	Nom d'utilisateur authentifié
`fingerprint`	string ou `null`	Fingerprint de la clé utilisée (peut être `null` si méthode sans clé)


```
{"event":"auth_succeeded","ts":"2026-04-19T14:32:14.005Z","session_id":"a4c2b8e1-9f3d-4
```


3.5 `brute_force_detected`

Le seuil `SSH_GUARD_BRUTE_FORCE_THRESHOLD` a été atteint. Un événement `ip_banned` suit immédiatement.

Champ	Type	Description
`event`	`"brute_force_detected"`	
`ts`	string	Horodatage UTC
`ip`	string	IP fautive
`count`	number	Nombre d'`auth_failed` comptés dans la fenêtre
`window_secs`	number	Largeur de la fenêtre (en secondes)


```
{"event":"brute_force_detected","ts":"2026-04-19T14:36:42.998Z","ip":"203.0.113.42","co
```


© Gitrust

3.6 `user_enumeration_detected`

Le seuil `SSH_GUARD_USER_ENUM_THRESHOLD` (nombre d'usernames distincts essayés depuis la même IP) a été atteint.

Champ	Type	Description
`event`	`"user_enumeration_detected"`	
`ts`	string	Horodatage UTC
`ip`	string	
`distinct_users`	number	Nombre d'usernames distincts dans la fenêtre
`window_secs`	number	


```
{"event":"user_enumeration_detected","ts":"2026-04-19T14:38:15.402Z","ip":"203.0.113.42
```


3.7 `key_scanning_detected`

Le seuil `SSH_GUARD_KEY_SCAN_THRESHOLD` (nombre de fingerprints distincts essayés depuis la même IP) a été atteint.

Champ	Type	Description
`event`	`"key_scanning_detected"`	
`ts`	string	Horodatage UTC
`ip`	string	
`distinct_keys`	number	Nombre de fingerprints distincts dans la fenêtre
`window_secs`	number	


```
{"event":"key_scanning_detected","ts":"2026-04-19T14:39:08.117Z","ip":"203.0.113.42","d
```


© Gitrust

3.8 `connection_flood_detected`

Le cap `SSH_GUARD_CONN_FLOOD_PER_SEC` a été dépassé pour cette IP. Un `connection_dropped` avec `reason="flood_limit"` suit dans le même millième de seconde.

Champ	Type	Description
`event`	`"connection_flood_detected"`	
`ts`	string	Horodatage UTC
`ip`	string	
`rate_per_sec`	number	Cap nominal (valeur de `SSH_GUARD_CONN_FLOOD_PER_SEC`)

```
{"event":"connection_flood_detected","ts":"2026-04-19T14:40:01.555Z","ip":"203.0.113.42
```

3.9 `ip_banned`

Un ban a été appliqué (auto par un détecteur, manuel par un admin, ou simulé en `dry_run`).

Champ	Type	Description
`event`	`"ip_banned"`	
`ts`	string	Horodatage UTC
`ip`	string	IP bannie (pour un CIDR > /32, l'adresse réseau)
`reason`	enum string	Voir tableau ci-dessous
`expires_at`	string ou `null`	Horodatage UTC d'expiration. `null` = ban permanent

© Gitrust

Valeurs de `reason` :

`reason`	Origine
`brute_force`	`BruteForceDetector`
`user_enumeration`	`UserEnumerationDetector`
`key_scanning`	`KeyScanningDetector`
`connection_flood`	`ConnectionFloodDetector` (rare : le flood drop ne pose pas de ban persistant par défaut)
`admin_deny_list`	Ajout admin dans l'ACL deny
`manual`	Ban manuel via UI/API admin


```
{"event":"ip_banned","ts":"2026-04-19T14:36:42.999Z","ip":"203.0.113.42","reason":"brut
```


> ***Mode*** `dry_run` *: un* `ip_banned` *est émis pour signal externe (fail2ban)* ***sans*** *que ssh-guard inscrive le ban dans son store. Utile pour observer un nouveau seuil sans risque.*

3.10 `ip_unbanned`

Un ban a été levé (TTL expiré ou action admin).

Champ	Type	Description
`event`	`"ip_unbanned"`	
`ts`	string	Horodatage UTC
`ip`	string	
`manual`	boolean	`true` = action admin, `false` = expiration auto


```
{"event":"ip_unbanned","ts":"2026-04-19T15:36:43.001Z","ip":"203.0.113.42","manual":fal
```


© Gitrust

4. Filtrage côté outils externes

4.1 Fail2ban (filtre par regex sur le fichier JSON)

```
[Definition]
# Brute-force détecté → ban dur côté firewall
failregex = ^.*"event":"auth_failed".*"ip":"<HOST>".*$
ignoreregex =
```

Voir le template clé en main : `template/security/fail2ban-gitrust-ssh-guard.conf` et le how-to Durcir avec Fail2ban.

4.2 Loki (LogQL)

```
# Toutes les détections fortes (signal pour alerting Grafana)
{job="gitrust-ssh-guard"}
  | json
  | event=~"brute_force_detected|user_enumeration_detected|key_scanning_detected|connec
```

4.3 jq (extraction shell)

```
# 10 IP les plus actives en échec d'auth sur la dernière heure
sudo journalctl -u gitrust --since "1 hour ago" --no-pager \
  | jq -r 'select(.event=="auth_failed") | .ip' \
  | sort | uniq -c | sort -rn | head -10
```

---

5. Pour aller plus loin

- Variables d'environnement — SSH_GUARD_*
- Configurer ssh-guard
- ssh-guard : détection d'attaques SSH
- Durcir avec Fail2ban
- Crate gitrust-ssh-guard (pour les contributeurs)

© Gitrust

1.4.3 Paramètres dynamiques

Ce document explique comment fonctionne la configuration de la plateforme, les deux mécanismes en jeu, leur précédence, et les pièges à connaître.

Vue d'ensemble : deux systèmes distincts

Gitrust utilise **deux systèmes de configuration** indépendants :

Système	Source	Modifiable à chaud	Visible dans l'UI admin
Configuration statique	Fichier `.env` + variables d'environnement	Non (restart requis)	Non
Configuration dynamique	Table `app_settings` (PostgreSQL)	Oui (effet immédiat)	Oui (`/admin/settings`)

Il existe un **cas hybride** pour OAuth (DB prioritaire, `.env` en fallback).

1. Configuration statique (`.env`)

Principe

Les variables d'environnement sont chargées **une seule fois au démarrage** par `dotenvy::dotenv()` et stockées dans des structs Rust immuables. Elles ne sont jamais relues après le boot.

Fichiers de référence

Fichier	Rôle
`.env.example`	Template documenté avec toutes les variables
`.env`	Configuration locale (gitignore)
`.env.production`	Configuration de déploiement
`.env.test`	Configuration de tests

© Gitrust

Structs de chargement

Struct	Variables concernées
`GitrustConfig`	`GIT_REPOS_BASE_PATH`, `SSH_PORT`, `SSH_LISTEN_ADDR`, `SSH_HOST_KEY_PATH`, `CI_*`, `IMPORT_*`
`AppConfig`	`APP_NAME`, `APP_THEME`, `APP_DEBUG`
`AuthConfig`	`JWT_SECRET`, `JWT_EXPIRATION_MINUTES`, `SESSION_*`, `RATE_LIMIT_*`, `COOKIE_*`
`EmailConfig`	`SMTP_*`, `EMAIL_*`, `IMAP_*`

Comportement

- Chargées dans `RustwardenApp::build()` au démarrage
- Stockées dans des `Arc<Config>` partagés via l'état axum
- **Modifier le `.env` sans redémarrer n'a aucun effet**
- **Ces variables ne sont PAS affichées dans l'interface admin**

Liste des variables statiques (non exhaustive)

```
DATABASE_URL, SERVER_HOST, SERVER_PORT, RUST_LOG,
SSH_HOST_KEY_PATH, SSH_PORT, SSH_LISTEN_ADDR, SSH_PUBLIC_HOST,
JWT_SECRET, JWT_EXPIRATION_MINUTES, JWT_ISSUER,
REFRESH_TOKEN_EXPIRATION_DAYS, REMEMBER_ME_EXPIRATION_DAYS,
SESSION_TIMEOUT_MINUTES, SESSION_BACKEND,
RATE_LIMIT_LOGIN_PER_MINUTE, RATE_LIMIT_REFRESH_PER_MINUTE,
RATE_LIMIT_GENERAL_PER_MINUTE,
APP_DEBUG, COOKIE_SECURE, COOKIE_SAME_SITE,
ADMIN_USERNAME, ADMIN_EMAIL, ADMIN_PASSWORD,
SMTP_HOST, SMTP_PORT, SMTP_USER, SMTP_PASSWORD, SMTP_FROM,
EMAIL_BASE_URL, EMAIL_QUEUE_*,
APP_NAME, APP_THEME, DEFAULT_LOCALE,
GIT_REPOS_BASE_PATH,
CI_ENABLED, CI_MAX_CONCURRENT, CI_REMOTE_HOST, CI_WORKSPACE_PATH
```

2. Configuration dynamique (table `app_settings`)

Principe

Les réglages dynamiques sont stockés en base PostgreSQL dans la table `app_settings` et lus **à chaque requête** via `AppSettingsService`. Ils sont modifiables à chaud par un administrateur depuis l'interface `/admin/settings`.

© Gitrust

Schéma de la table

```
CREATE TABLE app_settings (
    id          UUID PRIMARY KEY,
    key         VARCHAR(100) UNIQUE NOT NULL,
    value       TEXT NOT NULL,
    description TEXT,
    updated_by  UUID REFERENCES users(id) ON DELETE SET NULL,
    updated_at  TIMESTAMPTZ NOT NULL
);
```

Initialisation au démarrage

Au boot, `AppSettingsService::initialize_default_settings()` crée les réglages par défaut **uniquement s'ils n'existent pas encore en base** :

```
let existing = Self::get_setting(db, key).await?;
if existing.is_none() {
    // ... insert valeur par défaut
}
```

Conséquence critique : une fois qu'un réglage existe en base (créé au premier démarrage ou modifié via l'UI), il n'est **jamais écrasé** par un redémarrage ultérieur.

© Gitrust

Liste des réglages dynamiques et leurs valeurs par défaut

© Gitrust

Clé	Défaut	Descriptio
`app_domain`	`env::var("APP_DOMAIN")` ou `"localhost"`	Domaine de l'applicatio
`allow_registration`	`false`	Autoriser l'inscription publique
`validation_email_required`	`true`	Exiger la validation email
`audit_log_level`	`INFO`	Niveau de log d'audit
`audit_log_actions`	`["create","update","delete","reset_password"]`	Actions auditées
`password_min_length`	`8`	Longueur minimum mot de pass
`password_require_uppercase`	`false`	Exiger des majuscules
`password_require_lowercase`	`false`	Exiger des minuscules
`password_require_digits`	`false`	Exiger des chiffres
`password_require_special`	`false`	Exiger des caractères spéciaux
`password_change_require_email`	`true`	Confirmatio email pour changemen mdp
`password_expiration_enabled`	`false`	Activer l'expiration

© Gitrust

Clé	Défaut	Description
		des mots de passe
`password_expiration_days`	`0`	Durée d'expiration (jours)
`password_expiration_alert_enabled`	`false`	Alerte email avant expiration
`password_expiration_alert_days_before`	`7`	Jours avant expiration pour alerte
`oauth_enabled`	`false`	Activer OAuth/SSO
`oauth_google_enabled`	`false`	Activer Google OAuth
`oauth_github_enabled`	`false`	Activer GitHub OAuth
`oauth_discord_enabled`	`false`	Activer Discord OAuth
`oauth_microsoft_enabled`	`false`	Activer Microsoft OAuth
`oauth_redirect_base_url`	`""`	URL de base pour les callbacks OAuth
`oauth_auto_register`	`true`	Créer un compte aut

© Gitrust

Clé	Défaut	Description
		au premier login OAuth
`oauth_link_existing_account`	`true`	Lier un compte OAuth à un utilisateur existant
`oauth_{provider}_client_id`	`""`	Client ID du provider OAuth
`oauth_{provider}_client_secret`	`""`	Client secret (chiffré AES-256-GCM en base)
`oauth_microsoft_tenant`	`""`	Tenant Azure AD

3. Cas hybride : OAuth

OAuth est le **seul sous-système** qui combine les deux sources. La logique dans `OAuthConfig::load()` :

```
Pour chaque réglage OAuth :
  1. Chercher dans app_settings (DB)
     → si trouvé et non vide : utiliser cette valeur
  2. Sinon : fallback sur la variable d'environnement
     → si absente : utiliser la valeur par défaut codée en dur
```

Important : `OAuthConfig::load()` est appelé au démarrage. Le résultat est stocké dans un `Arc` et n'est pas relu dynamiquement. Donc :

- Modifier un réglage OAuth **via l'UI admin** nécessite un **redémarrage** pour que `OAuthConfig` soit rechargé
- Le `.env` n'est qu'un fallback si la DB n'a pas de valeur

© Gitrust

4. Précédence et comportement au redémarrage

Réglages statiques (`.env` uniquement)

Action	Effet immédiat	Après restart
Modifier `.env`	Non	Oui

Réglages dynamiques (DB uniquement)

Action	Effet immédiat	Après restart
Modifier via UI admin	Oui	Oui (valeur en DB persiste)
Modifier le `.env`	Aucun effet	Aucun effet

Réglages hybrides (OAuth)

Action	Effet immédiat	Après restart
Modifier via UI admin	Non (OAuthConfig est en Arc)	Oui (DB prime sur .env)
Modifier `.env`	Non	Seulement si aucune valeur en DB

5. Piège principal : les variables `.env` fantômes

Certaines variables présentes dans `.env.example` donnent l'**illusion** de configurer des réglages qui sont en réalité gérés par la base de données :

Variable `.env`	Réglage DB correspondant	La variable `.env` est-elle lue ?
`ALLOW_REGISTRATION=true`	`allow_registration`	**Non** — valeur par défaut codée en dur (`"false"`)
`EMAIL_VALIDATION_REQUIRED=true`	`validation_email_required`	**Non** — valeur par défaut codée en dur (`"true"`)
`OAUTH_ENABLED=true`	`oauth_enabled`	Oui, mais **seulement en fallback** si la DB n'a pas de valeur

© Gitrust

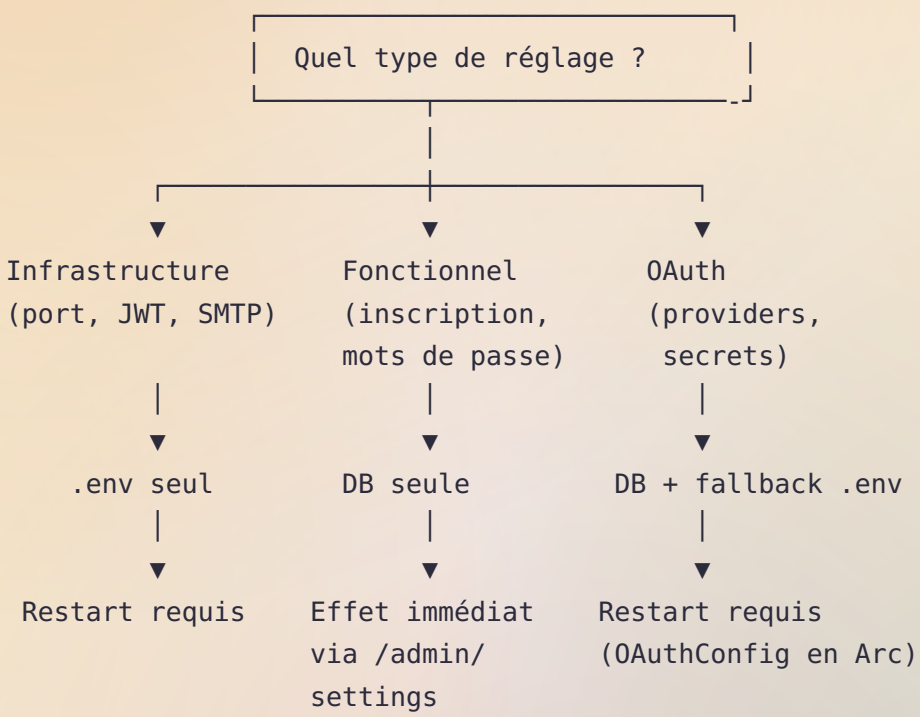
Pour `ALLOW_REGISTRATION` et `EMAIL_VALIDATION_REQUIRED` :

- La variable `.env` est documentée dans `.env.example` mais **n'est jamais consultée** par `initialize_default_settings()`
- La valeur initiale est **codée en dur** dans le service
- Seule la valeur en base de données (modifiable via `/admin/settings`) fait foi

Pour `app_domain`, c'est le seul réglage dynamique qui lit le `.env` comme seed initial.

---

6. Diagramme de décision



---

7. Guide pour les administrateurs

Modifier un réglage d'infrastructure

Éditer `.env` (ou `.env.production`) puis redémarrer le service :

```
# Éditer
vim .env

# Redémarrer
```


© Gitrust

```
systemctl restart gitrust    # production
# ou Ctrl+C + cargo run      # dev
```

Modifier un réglage fonctionnel

Se connecter en tant qu'administrateur, aller dans `/admin/settings`, modifier la valeur, et cliquer "Save". L'effet est **immédiat**, aucun redémarrage nécessaire.

Modifier un réglage OAuth

Via `/admin/settings`, modifier les valeurs OAuth puis **redémarrer le service** (la config OAuth est chargée en mémoire au boot et n'est pas relue dynamiquement).

Vérifier la valeur effective d'un réglage

Pour les réglages dynamiques, interroger la base directement :

```
SELECT key, value, updated_at, updated_by
FROM app_settings
WHERE key = 'allow_registration';
```

Pour les réglages statiques, vérifier les logs au démarrage (`RUST_LOG=debug`).

---

8. Guide pour les développeurs

Ajouter un nouveau réglage dynamique

1. Ajouter le tuple `(clé, valeur_défaut, description)` dans `AppSettingsService::initialize_default_settings()` :

```
("ma_nouvelle_cle", "valeur_par_defaut", Some("Description pour l'UI admin")),
```

2. Lire la valeur dans le code via le service :

```
// Boolean avec défaut
let enabled = AppSettingsService::get_bool(&db, "ma_nouvelle_cle", false).await?;

// String
let val = AppSettingsService::get_setting(&db, "ma_nouvelle_cle").await?;
```

3. La valeur apparaîtra automatiquement dans `/admin/settings`.

© Gitrust

Ajouter un nouveau réglage statique

1. Ajouter la variable dans `.env.example` avec documentation complète
2. Lire la variable dans le struct `Config` correspondant via `env::var()`
3. La valeur ne sera **pas** visible dans l'interface admin

Ajouter un réglage hybride (DB + fallback .env)

Suivre le pattern OAuth dans `crates/rustwarden-core/src/config/oauth.rs` :

```
let ma_valeur = match db_val("ma_cle").await {
    Some(v) => v,
    None => env::var("MA_CLE").unwrap_or_else(|_| "defaut".to_string()),
};
```


© Gitrust

1.4.4 Schéma de la base de données

Référence des tables PostgreSQL de gitrust, avec colonnes, index, contraintes et relations. Cette page concerne les tables visibles par un administrateur — pour les détails internes du code (migrations SeaORM, patterns de contribution), voir le manuel développeur.

Diagramme entité-relation

```
erDiagram
    users {
        uuid id PK
        string username UK
        string email UK
        string password_hash
        boolean must_change_password
        timestamptz last_login_at
        boolean email_verified
        timestamptz created_at
        timestamptz updated_at
    }

    resources {
        uuid id PK
        string resource_type
        uuid resource_id
        uuid owner_id FK
        boolean is_public
        timestamptz created_at
    }

    resource_shares {
        uuid id PK
        uuid resource_id FK
        uuid shared_with_user_id FK
        string permission_level
        uuid shared_by_user_id FK
        timestamptz created_at
    }

    repositories {
```


© Gitrust

uuid id PK
uuid owner_id FK
string slug
string description
string disk_path
string default_branch
boolean is_empty
timestamptz created_at
timestamptz updated_at
}
ssh_keys {
uuid id PK
uuid user_id FK
string title
string fingerprint UK
string key_type
text key_data
timestamptz last_used_at
timestamptz created_at
}
teams {
uuid id PK
uuid owner_id FK
string slug
string description
timestamptz created_at
timestamptz updated_at
}
team_members {
uuid id PK
uuid team_id FK
uuid user_id FK
string role
timestamptz created_at
}
team_repository_access {
uuid id PK
uuid team_id FK
uuid repository_id FK

© Gitrust

```
        string permission
      timestamptz created_at
              }

personal_access_tokens {
              uuid id PK
           uuid user_id FK
            string name
       string token_hash UK
           string scopes
      timestamptz expires_at
    timestamptz last_used_at
      timestamptz created_at
              }

           issues {
            uuid id PK
       uuid repository_id FK
          integer number
         uuid author_id FK
           string title
             text body
           string state
        uuid closed_by FK
      timestamptz created_at
      timestamptz updated_at
              }

       pull_requests {
            uuid id PK
       uuid repository_id FK
          integer number
         uuid author_id FK
           string title
             text body
       string source_branch
       string target_branch
           string state
      timestamptz merged_at
        uuid merged_by FK
    string merge_commit_sha
      timestamptz created_at
      timestamptz updated_at
```


© Gitrust

```
    }

    labels {
        uuid id PK
        uuid owner_id FK
        uuid repository_id FK
        string name
        string color
        string description
        string label_type
        timestamptz created_at
    }

    app_settings {
        uuid id PK
        string key UK
        text value
        text description
        uuid updated_by FK
        timestamptz updated_at
    }

    audit_log {
        uuid id PK
        uuid actor_id FK
        string action
        string resource_type
        uuid resource_id
        string ip_address
        text details
        timestamptz created_at
    }

    users ||--o{ repositories : "possède"
    users ||--o{ ssh_keys : "enregistre"
    users ||--o{ teams : "crée"
    users ||--o{ resources : "possède"
    users ||--o{ resource_shares : "partage avec"
    users ||--o{ personal_access_tokens : "crée"
    users ||--o{ audit_log : "acteur"
    repositories ||--o| resources : "enregistré dans"
    resources ||--o{ resource_shares : "partagé"
    teams ||--o{ team_members : "contient"
```


© Gitrust

```
                    users ||--o{ team_members : "membre de"
                teams ||--o{ team_repository_access : "accède à"
            repositories ||--o{ team_repository_access : "accessible par"
                    repositories ||--o{ issues : "contient"
                repositories ||--o{ pull_requests : "contient"
```

Tables rustwarden-core (framework)

Ces tables sont gérées par le framework. Ne les modifiez pas directement.

Table	Rôle
`users`	Comptes utilisateurs (authentification, profil)
`roles` / `permissions` / `role_permissions` / `user_roles`	RBAC global (admin, user)
`resources`	Registre générique de ressources (ownership, visibilité publique/privée)
`resource_shares`	Partage de ressources avec niveaux (read/write/admin)
`refresh_tokens`	Refresh tokens JWT
`jwt_blacklist`	Tokens JWT révoqués (invalidation après déconnexion)
`app_settings`	Configuration dynamique de l'application
`audit_log`	Journal d'audit de toutes les actions significatives
`oauth_accounts`	Comptes OAuth liés

© Gitrust

Tables gitrust

`repositories`

Dépôts Git hébergés sur la plateforme.

Colonne	Type	Contraintes	Description
`id`	UUID	PK	Identifiant unique
`owner_id`	UUID	FK → `users.id` CASCADE, NOT NULL	Propriétaire du dépôt
`slug`	VARCHAR(64)	NOT NULL	Nom URL du dépôt (ex. `mon-projet`)
`description`	VARCHAR(500)	NULL	Description affichée dans l'UI
`disk_path`	VARCHAR(512)	NOT NULL	Chemin absolu du bare repo sur disque
`default_branch`	VARCHAR(255)	NOT NULL, DEFAULT `'main'`	Branche par défaut
`is_empty`	BOOLEAN	NOT NULL, DEFAULT `true`	Vrai si aucun commit poussé
`created_at`	TIMESTAMPTZ	NOT NULL	Date de création
`updated_at`	TIMESTAMPTZ	NOT NULL	Date de dernière modification

Index : `UNIQUE(owner_id, slug)`, `INDEX(owner_id)`

Chaque dépôt est aussi enregistré dans `resources` avec `resource_type = "repository"` pour gérer la visibilité (public/privé) et le partage via `resource_shares`.

© Gitrust

`ssh_keys`

Clés publiques SSH des utilisateurs pour l'authentification Git.

Colonne	Type	Contraintes	Description
`id`	UUID	PK	Identifiant unique
`user_id`	UUID	FK → `users.id` CASCADE, NOT NULL	Propriétaire de la clé
`title`	VARCHAR(255)	NOT NULL	Libellé (ex. `MacBook Pro`)
`fingerprint`	VARCHAR(128)	UNIQUE, NOT NULL	Fingerprint SHA256 calculé à l'insertion
`key_type`	VARCHAR(32)	NOT NULL	Type : `ssh-ed25519`, `ssh-rsa`, `ecdsa-*`
`key_data`	TEXT	NOT NULL	Clé au format `authorized_keys`
`last_used_at`	TIMESTAMPTZ	NULL	Dernière utilisation SSH
`created_at`	TIMESTAMPTZ	NOT NULL	Date d'ajout

Validation : types autorisés — Ed25519, RSA ≥ 4096 bits, ECDSA P-256/P-384.

`teams`

Groupes d'utilisateurs pour le partage d'accès aux dépôts.

Colonne	Type	Contraintes	Description
`id`	UUID	PK	Identifiant unique
`owner_id`	UUID	FK → `users.id` CASCADE, NOT NULL	Créateur / administrateur de l'équipe
`slug`	VARCHAR(64)	NOT NULL	Nom URL de l'équipe
`description`	VARCHAR(500)	NULL	Description
`created_at`	TIMESTAMPTZ	NOT NULL	Date de création
`updated_at`	TIMESTAMPTZ	NOT NULL	Date de dernière modification

© Gitrust

Index : `UNIQUE(owner_id, slug)`

`team_members`

Appartenance d'un utilisateur à une équipe avec un rôle.

Colonne	Type	Contraintes	Description
`id`	UUID	PK	Identifiant unique
`team_id`	UUID	FK → `teams.id` CASCADE, NOT NULL	Équipe
`user_id`	UUID	FK → `users.id` CASCADE, NOT NULL	Membre
`role`	VARCHAR(20)	NOT NULL, DEFAULT `'read'`	Rôle dans l'équipe
`created_at`	TIMESTAMPTZ	NOT NULL	Date d'ajout

Index : `UNIQUE(team_id, user_id)`, `INDEX(user_id)`

`team_repository_access`

Permissions d'accès d'une équipe sur un dépôt.

Colonne	Type	Contraintes	Description
`id`	UUID	PK	Identifiant unique
`team_id`	UUID	FK → `teams.id` CASCADE, NOT NULL	Équipe
`repository_id`	UUID	FK → `repositories.id` CASCADE, NOT NULL	Dépôt
`permission`	VARCHAR(20)	NOT NULL, DEFAULT `'read'`	Niveau : `read`, `write`, `admin`
`created_at`	TIMESTAMPTZ	NOT NULL	Date d'attribution

Index : `UNIQUE(team_id, repository_id)`, `INDEX(repository_id)`

© Gitrust

`personal_access_tokens`

Jetons d'accès personnels pour l'authentification API et Git HTTPS.

Colonne	Type	Contraintes	Description
`id`	UUID	PK	Identifiant unique
`user_id`	UUID	FK → `users.id` CASCADE, NOT NULL	Propriétaire du jeton
`name`	VARCHAR(255)	NOT NULL	Nom descriptif (ex. `CI pipeline`)
`token_hash`	VARCHAR(128)	UNIQUE, NOT NULL	Hash SHA-256 du jeton (le jeton brut n'est jamais stocké)
`scopes`	VARCHAR(255)	NOT NULL	Portées autorisées (séparées par espace)
`expires_at`	TIMESTAMPTZ	NULL	Expiration (NULL = sans expiration)
`last_used_at`	TIMESTAMPTZ	NULL	Dernière utilisation
`created_at`	TIMESTAMPTZ	NOT NULL	Date de création

© Gitrust

`issues`

Tickets de suivi (bugs, tâches, demandes de fonctionnalité).

Colonne	Type	Contraintes	Description
`id`	UUID	PK	Identifiant unique
`repository_id`	UUID	FK → `repositories.id` CASCADE	Dépôt propriétaire
`number`	INTEGER	NOT NULL (auto-incrémenté par dépôt)	Numéro affiché (`#1`, `#2`…)
`author_id`	UUID	FK → `users.id` CASCADE	Auteur
`title`	VARCHAR(255)	NOT NULL	Titre
`body`	TEXT	NOT NULL DEFAULT `''`	Corps (Markdown)
`state`	VARCHAR(20)	NOT NULL DEFAULT `'open'`	État : `open`, `closed`
`closed_by`	UUID	FK → `users.id` NULL	Utilisateur qui a fermé l'issue
`created_at`	TIMESTAMPTZ	NOT NULL	Date de création
`updated_at`	TIMESTAMPTZ	NOT NULL	Date de dernière modification

Index : `UNIQUE(repository_id, number)`, `INDEX(repository_id, state)`

© Gitrust

`pull_requests`

Propositions de merge d'une branche source vers une branche cible.

Colonne	Type	Contraintes	Description
`id`	UUID	PK	Identifiant unique
`repository_id`	UUID	FK → `repositories.id` CASCADE	Dépôt
`number`	INTEGER	NOT NULL (auto-incrémenté par dépôt)	Numéro affiché
`author_id`	UUID	FK → `users.id` CASCADE	Auteur
`title`	VARCHAR(255)	NOT NULL	Titre
`body`	TEXT	NOT NULL DEFAULT `''`	Description
`source_branch`	VARCHAR(255)	NOT NULL	Branche source
`target_branch`	VARCHAR(255)	NOT NULL	Branche cible (ex. `main`)
`state`	VARCHAR(20)	NOT NULL DEFAULT `'open'`	État : `open`, `merged`, `closed`
`merged_at`	TIMESTAMPTZ	NULL	Date de merge
`merged_by`	UUID	FK → `users.id` NULL	Utilisateur qui a mergé
`merge_commit_sha`	VARCHAR(40)	NULL	SHA du commit de merge
`created_at`	TIMESTAMPTZ	NOT NULL	Date de création
`updated_at`	TIMESTAMPTZ	NOT NULL	Date de dernière modification

Index : `UNIQUE(repository_id, number)`, `INDEX(repository_id, state)`

© Gitrust

`labels`

Étiquettes à deux niveaux : `classification` (scope owner) et `subject` (scope dépôt).

Colonne	Type	Contraintes	Description
`id`	UUID	PK	Identifiant unique
`owner_id`	UUID	FK → `users.id` NULL	Propriétaire (labels de classification)
`repository_id`	UUID	FK → `repositories.id` NULL	Dépôt (labels de sujet)
`name`	VARCHAR(50)	NOT NULL	Nom du label
`color`	VARCHAR(7)	NOT NULL	Couleur hexadécimale (ex. `#e11d48`)
`description`	VARCHAR(255)	NULL	Description
`label_type`	VARCHAR(20)	NOT NULL	`'classification'` ou `'subject'`
`created_at`	TIMESTAMPTZ	NOT NULL	Date de création

Index : `UNIQUE(owner_id, repository_id, name, label_type)`

© Gitrust

app_settings

Configuration dynamique de l'application (modifiable à chaud via `/admin/settings`).

Colonne	Type	Contraintes	Description
`id`	UUID	PK	Identifiant unique
`key`	VARCHAR(100)	UNIQUE, NOT NULL	Clé de configuration (ex. `allow_registration`)
`value`	TEXT	NOT NULL	Valeur sous forme de texte
`description`	TEXT	NULL	Description affichée dans l'UI admin
`updated_by`	UUID	FK → `users.id` ON DELETE SET NULL	Dernier administrateur ayant modifié
`updated_at`	TIMESTAMPTZ	NOT NULL	Date de dernière modification

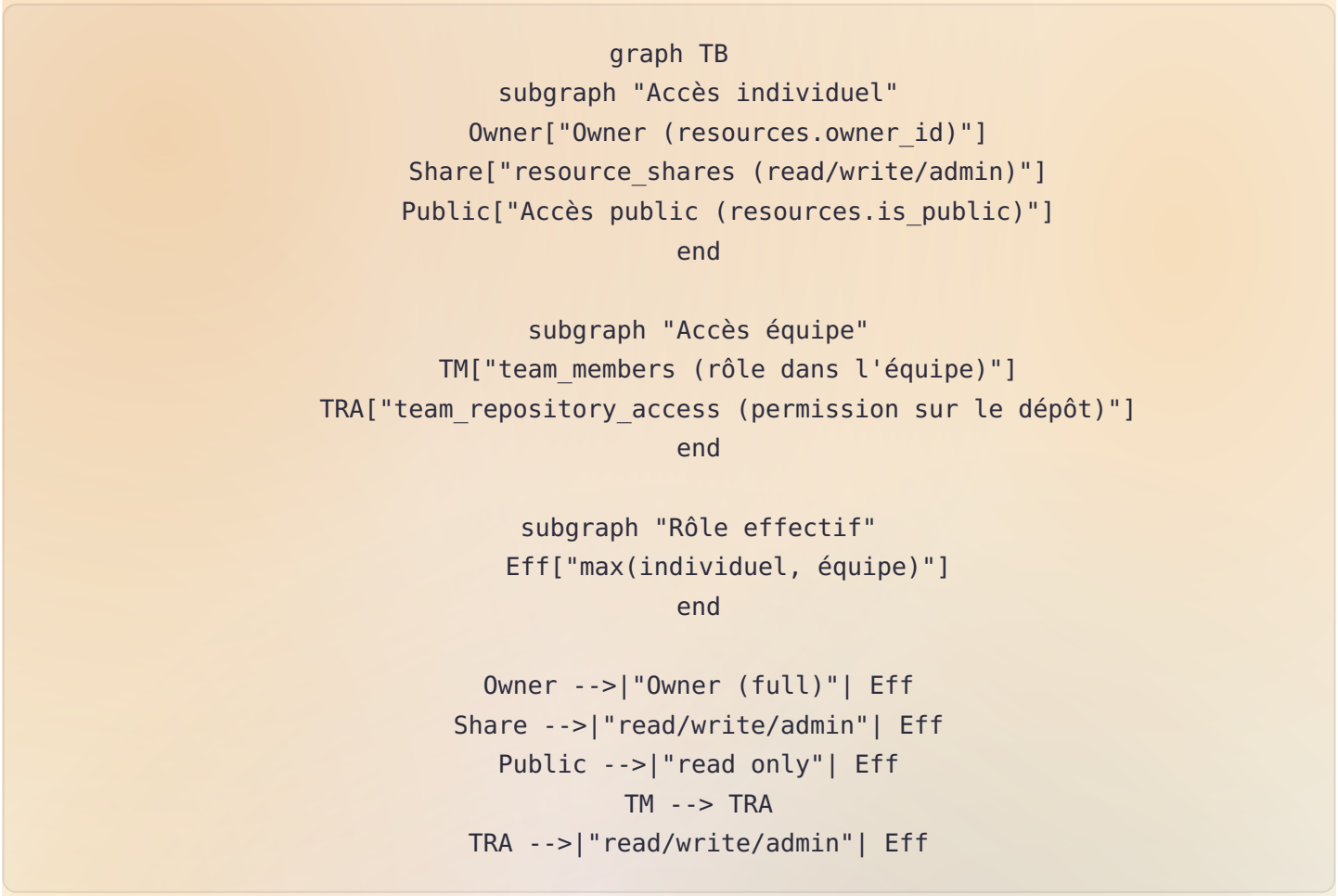
audit_log

Journal d'audit de toutes les actions significatives.

Colonne	Type	Contraintes	Description
`id`	UUID	PK	Identifiant unique
`actor_id`	UUID	FK → `users.id` NULL	Utilisateur auteur de l'action
`action`	VARCHAR(50)	NOT NULL	Type d'action (`create`, `update`, `delete`, `reset_password`…)
`resource_type`	VARCHAR(50)	NULL	Type de ressource concernée
`resource_id`	UUID	NULL	Identifiant de la ressource
`ip_address`	VARCHAR(45)	NULL	Adresse IP source
`details`	TEXT	NULL	Détails supplémentaires (JSON)
`created_at`	TIMESTAMPTZ	NOT NULL	Horodatage

© Gitrust

Modèle de permissions



Rôle	Permissions
Reader	Clone, navigation (lecture seule)
Developer	Reader + push
Maintainer	Developer + paramètres, collaborateurs, protections de branche
Owner	Maintainer + suppression, transfert

© Gitrust

Ordre des migrations

#	Migration	Tables créées
1	`m20260305_000001_initial_schema`	users, refresh_tokens, jwt_blacklist… (core)
2	`m20260306_000002_create_app_settings_table`	app_settings (core)
3	`m20260309_000003_create_permissions_tables`	roles, permissions, role_permissions, user_roles (core)
4	`m20260309_000004_create_resources_tables`	resources, resource_shares (core)
5	`m20260310_000005_create_oauth_accounts_table`	oauth_accounts (core)
6	`m20260325_000001_create_repositories`	**repositories**
7	`m20260325_000002_create_ssh_keys`	**ssh_keys**
8	`m20260325_000003_create_teams`	**teams**
9	`m20260325_000004_create_team_members`	**team_members**
10	`m20260325_000005_create_team_repository_access`	**team_repository_access**
11	`m20260327_000006_create_personal_access_tokens`	**personal_access_tokens**
12	`m20260327_000007_create_issues`	**issues**
13	`m20260327_000008_create_issue_comments`	**issue_comments**
14	`m20260327_000009_create_labels`	**labels**, **issue_labels**
15	`m20260327_000010_add_label_type`	alter labels
16	`m20260327_000011_labels_owner_scope`	alter labels
17	`m20260327_000012_create_pull_requests`	**pull_requests**
18	`m20260327_000013_create_pr_comments`	**pr_comments**

Toutes les migrations sont appliquées automatiquement au démarrage par `AppMigrator`. Elles sont idempotentes.

© Gitrust

Pour aller plus loin

- Auditer l'instance — requêtes SQL sur `audit_log`
- Paramètres dynamiques — table `app_settings`
- Sauvegarder et restaurer — `pg_dump` / `pg_restore`

© Gitrust

1.4.5 Ports et services

Référence des ports réseau utilisés par gitrust et ses dépendances.

Tableau des ports

Port	Protocole	Service	Processus	Binding par défaut	Configurable
4000	TCP/HTTP	Interface web + API REST	`gitrust` (axum)	`SERVER_HOST:SERVER_PORT`	Oui
2222	TCP/SSH	Serveur Git SSH	`gitrust` (Russh)	`SSH_LISTEN_ADDR:SSH_PORT`	Oui
5432	TCP	PostgreSQL	`postgres`	`127.0.0.1:5432`	Via PG config
6379	TCP	Redis (sessions)	`redis-server`	`127.0.0.1:6379`	Via Redis config
25	TCP/SMTP	Envoi e-mail sortant (non chiffré)	Client SMTP gitrust	Outbound seulement	Via `SMTP_PORT`
465	TCP/SMTPS	Envoi e-mail sortant (TLS natif)	Client SMTP gitrust	Outbound seulement	Via `SMTP_PORT`
587	TCP/ SMTP+STARTTLS	Envoi e-mail sortant (recommandé)	Client SMTP gitrust	Outbound seulement	Via `SMTP_PORT`

Détail par service

Port 4000 — Interface web et API REST (HTTP)

gitrust expose son interface web HTMX et son API REST v1 sur ce port via le framework axum.

Configuration :

© Gitrust

```
SERVER_HOST=127.0.0.1   # derrière un reverse-proxy (recommandé en prod)
SERVER_PORT=4000
```

En production : gitrust écoute sur `127.0.0.1:4000` et un reverse-proxy (Nginx, Caddy, Traefik) termine TLS et proxifie vers ce port. Les clients n'accèdent jamais directement au port 4000.

Règle firewall (si reverse-proxy sur la même machine) :

```
# Aucune ouverture nécessaire depuis l'extérieur — le proxy écoute sur 443
# Bloquer le port 4000 depuis l'extérieur :
sudo ufw deny 4000/tcp
```

En développement : gitrust peut écouter sur `0.0.0.0:4000` pour un accès depuis le LAN.

Port 2222 — Serveur Git SSH (Russh)

gitrust intègre son propre serveur SSH via la bibliothèque **Russh** — ce n'est **pas** le démon `sshd` du système d'exploitation. Les deux peuvent coexister sans conflit.

Configuration :

```
SSH_PORT=2222
SSH_LISTEN_ADDR=0.0.0.0    # accès direct
# ou
SSH_LISTEN_ADDR=127.0.0.1  # derrière nginx stream proxy
```

Accès direct (recommandé pour la simplicité) : ouvrir le port 2222 dans le firewall.

```
sudo ufw allow 2222/tcp
```

URL de clone : `ssh://git@VOTRE_DOMAINE:2222/owner/repo.git`

Via nginx stream proxy (port 22) : si vous souhaitez que les utilisateurs clonent sur le port standard 22, configurez nginx comme proxy de flux TCP :

```
# Dans nginx.conf, section stream (hors section http)
stream {
    upstream gitrust_ssh {
        server 127.0.0.1:2222;
```


© Gitrust

```
    }
    server {
        listen 22;
        proxy_pass gitrust_ssh;
    }
}
```

Dans ce cas, `SSH_LISTEN_ADDR=127.0.0.1` et `SSH_PUBLIC_HOST=VOTRE_DOMAINE` (sans port dans les URLs de clone).

> ***Attention** : si le port 22 est déjà utilisé par `sshd` pour l'administration du serveur, déplacez `sshd` sur un autre port avant de configurer le proxy nginx.*

Règle firewall :

```
sudo ufw allow 2222/tcp   # accès direct
# ou
sudo ufw allow 22/tcp     # si proxy nginx stream
```

Port 5432 — PostgreSQL

PostgreSQL doit **toujours** écouter sur `127.0.0.1` (ou un réseau privé), jamais sur `0.0.0.0` en production.

Vérification :

```
sudo -u postgres psql -c "SHOW listen_addresses;"
# Attendu : localhost ou 127.0.0.1
```

Règle firewall : aucune ouverture vers l'extérieur.

```
# Bloquer explicitement depuis l'extérieur (si ufw)
sudo ufw deny 5432/tcp
```

Pour une architecture multi-serveurs (gitrust et PostgreSQL sur des machines distinctes), utilisez un réseau privé ou un tunnel SSH — jamais une exposition directe sur Internet.

© Gitrust

Port 6379 — Redis (sessions multi-instances)

Redis n'est nécessaire que si `SESSION_BACKEND=redis` dans `.env`. Par défaut, gitrust utilise PostgreSQL pour les sessions.

Configuration Redis (dans `/etc/redis/redis.conf`) :

```
bind 127.0.0.1
requirepass VOTRE_MOT_DE_PASSE_REDIS
```

Règle firewall : aucune ouverture vers l'extérieur.

Ports SMTP sortants (25, 465, 587)

Ces ports sont utilisés en **sortant** uniquement : gitrust se connecte au serveur SMTP configuré dans `SMTP_HOST`. Gitrust n'héberge pas de serveur SMTP.

Port	Usage	Recommandation
25	SMTP non chiffré	Interdit en production, souvent bloqué par les FAI et hébergeurs
465	SMTPS (TLS natif)	Recommandé avec `SMTP_USE_TLS=true`
587	SMTP + STARTTLS	Recommandé avec `SMTP_USE_STARTTLS=true`

Aucune règle firewall entrante nécessaire pour SMTP.

Architecture réseau par topologie

Développement local

```
Navigateur → http://localhost:4000  (direct, pas de proxy)
Git client  → ssh://localhost:2222
```

Tous les ports en `0.0.0.0`, `COOKIE_SECURE=false`, `APP_DEBUG=true`.

Production mono-machine (recommandé)

© Gitrust

```
Internet → :443 (HTTPS) → nginx → 127.0.0.1:4000 (gitrust HTTP)
Internet → :2222 (SSH)  → gitrust directement
Internet ← SMTP sortant → smtp.prestataire.com:587
```

```
SERVER_HOST=127.0.0.1
SERVER_PORT=4000
SSH_LISTEN_ADDR=0.0.0.0
SSH_PORT=2222
```

Règles firewall :

```
sudo ufw allow 443/tcp   # HTTPS via nginx
sudo ufw allow 2222/tcp  # Git SSH
sudo ufw deny 4000/tcp   # Masqué derrière nginx
sudo ufw deny 5432/tcp   # PostgreSQL interne uniquement
```

Production avec proxy SSH sur port 22

```
Internet → :22 (SSH) → nginx stream → 127.0.0.1:2222 (gitrust SSH)
```

```
SSH_LISTEN_ADDR=127.0.0.1
SSH_PORT=2222
SSH_PUBLIC_HOST=gitrust.votre-domaine.com
```

URL de clone : `git@gitrust.votre-domaine.com:owner/repo.git`

Pour aller plus loin

- Architecture globale
- Variables d'environnement
- Tutoriel 02 — Installation systemd

© Gitrust

1.4.6 Conformité ANSSI PA-074

Checklist de conformité au guide PA-074 de l'ANSSI (Recommandations de sécurité pour les applications web) appliquée à une instance gitrust. Pour chaque point : statut dans gitrust et action requise de l'administrateur.

> ***Base :*** `docs/QA_RULES.md` *du dépôt gitrust source, enrichi des contrôles applicables à l'administration d'une instance.*

© Gitrust

1. Authentification et gestion des sessions

Critère ANSSI	Statut gitrust	Action administrateur requise
Mot de passe haché avec algorithme fort (bcrypt/Argon2)	Implémenté (bcrypt)	Aucune — automatique
Longueur minimale de mot de passe configurable	Implémenté (`password_min_length` dans `app_settings`)	Configurer `password_min_length ≥ 12` via `/admin/settings`
Complexité du mot de passe configurable	Implémenté (majuscules, minuscules, chiffres, spéciaux)	Activer les règles souhaitées via `/admin/settings`
Expiration des mots de passe	Implémenté (`password_expiration_enabled`)	Activer si requis par votre politique SSI
Tokens comparés en temps constant	Implémenté (`subtle::ConstantTimeEq`)	Aucune
Durée de vie des sessions limitée	Implémenté (`SESSION_TIMEOUT_MINUTES`, défaut 30 min)	Ajuster selon votre politique (recommandé : ≤ 30 min)
Invalidation de session à la déconnexion	Implémenté (JWT blacklist + refresh token révoqué)	Aucune
Refresh token rotation à chaque usage	Implémenté	Aucune
Durée « Se souvenir de moi » bornée	Implémenté (≤ 90 jours, warning > 60 jours)	Vérifier `REMEMBER_ME_EXPIRATION_DAYS ≤ 30`
Authentification multi-facteurs (2FA TOTP)	Implémenté	Activer `totp_force_enabled=true` via `/admin/settings` pour forcer le 2FA sur tous les comptes
Protection contre le brute force (rate limiting)	Implémenté (`RATE_LIMIT_LOGIN_PER_MINUTE`)	Vérifier `RATE_LIMIT_LOGIN_PER_MINUTE ≤ 5` ; ajouter Fail2ban

© Gitrust

2. Chiffrement des communications

Critère ANSSI	Statut gitrust	Action administrateur requise
HTTPS obligatoire en production	Non géré par gitrust (délégué au reverse-proxy)	Configurer TLS sur Nginx/Caddy avec certificat valide (Let's Encrypt ou CA interne)
HSTS activé	Non géré par gitrust	Ajouter `Strict-Transport-Security: max-age=63072000; includeSubDomains` dans Nginx
TLS ≥ 1.2 uniquement	Non géré par gitrust	Configurer `ssl_protocols TLSv1.2 TLSv1.3` dans Nginx
Cookies `Secure` sur HTTPS	Implémenté (`COOKIE_SECURE=true` si `APP_DEBUG=false`)	Vérifier `APP_DEBUG=false` en production
Cookies `HttpOnly`	Implémenté (défaut framework)	Aucune
Cookies `SameSite=Lax` minimum	Implémenté (`COOKIE_SAME_SITE=Lax` par défaut)	Aucune
SSH via Ed25519 (algorithme moderne)	Implémenté (Russh, clé hôte Ed25519)	Aucune
TLS pour SMTP sortant	Implémenté (`SMTP_USE_TLS` / `SMTP_USE_STARTTLS`)	Vérifier `SMTP_USE_STARTTLS=true` ou `SMTP_USE_TLS=true` ; jamais les deux à `false`

© Gitrust

3. Protection contre les injections et les failles web

Critère ANSSI	Statut gitrust	Action administrateur requise
Validation des entrées utilisateur (formulaires, params)	Implémenté (newtypes `RepoSlug`, `Fingerprint`, etc.)	Aucune
Protection CSRF sur toutes les actions mutantes	Implémenté (token CSRF sur POST/PUT/DELETE)	Aucune
Pas d'injection SQL (ORM paramétré)	Implémenté (SeaORM, requêtes préparées)	Aucune
Pas d'accès inter-utilisateurs (anti-IDOR)	Implémenté (vérification `user_id` côté serveur)	Aucune
Path traversal bloqué (dépôts Git)	Implémenté (`RepoSlug` validé, pas de `..` ou `/`)	Aucune
Content Security Policy	Partiellement implémenté (CSP bloque les CDN externes)	Vérifier les en-têtes CSP dans Nginx si souhaité
Pas de ressources CDN externes	Implémenté (zéro CDN, tout servi depuis `static/`)	Aucune
`unsafe_code` interdit (Rust)	Implémenté (`#![forbid(unsafe_code)]` dans core/web/hooks)	Aucune

© Gitrust

4. Gestion des secrets et des clés

Critère ANSSI	Statut gitrust	Action administrateur requise
`JWT_SECRET` minimum 32 bytes	Implémenté (refus au démarrage si trop court)	Générer avec `openssl rand -hex 64`
Secrets OAuth chiffrés en base (AES-256-GCM)	Implémenté	Définir `OAUTH_ENCRYPTION_KEY` avec `openssl rand -hex 32`
Personal access tokens hachés (SHA-256)	Implémenté	Aucune
Secrets jamais loggués	Implémenté (pas de `Display` / `Debug` sur les secrets)	Vérifier `RUST_LOG=info` ou `warn` en production
Clé SSH hôte avec permissions restrictives	Implémenté (600, owner = user systemd)	Vérifier après restauration depuis sauvegarde
Rotation de la clé JWT possible	Possible (modifier `JWT_SECRET` + redémarrage)	Planifier une rotation annuelle ou en cas de compromission
Zeroize des types sensibles	Implémenté (`Zeroize` / `ZeroizeOnDrop`)	Aucune

© Gitrust

5. Journalisation et audit

Critère ANSSI	Statut gitrust	Action administrateur requise
Journal d'audit des actions significatives	Implémenté (table `audit_log`)	Vérifier que `audit_log_actions` couvre vos besoins via `/admin/settings`
Horodatage des événements	Implémenté (`TIMESTAMPTZ` avec fuseau)	Vérifier le fuseau horaire du serveur (`timedatectl`)
IP source enregistrée	Implémenté (`ip_address` dans `audit_log`)	Si derrière proxy : vérifier que `X-Real-IP` est transmis par Nginx
Non-répudiation (acteur identifié)	Implémenté (`actor_id` → `users.id`)	Aucune
Conservation des logs ≥ 12 mois	Non automatique (pas de purge auto actuellement)	Planifier une purge manuelle annuelle en conservant les événements de sécurité
Logs système via journald	Implémenté (tous les logs gitrust dans journald)	Configurer `journald.conf` avec `SystemMaxUse` et `MaxRetentionSec` appropriés
Export des logs d'audit	Implémenté (CSV depuis `/admin/audit`)	Archiver régulièrement les exports CSV

© Gitrust

6. Gestion des accès et des droits

Critère ANSSI	Statut gitrust	Action administrateur requise
Principe du moindre privilège	Implémenté (4 rôles : Reader/Developer/Maintainer/Owner)	Vérifier que les utilisateurs ont le niveau minimal nécessaire
Séparation des rôles (utilisateur / admin)	Implémenté (rôle admin global distinct des rôles dépôt)	Limiter le nombre de comptes admin
Révocation immédiate des accès	Implémenté (désactivation compte = effet immédiat)	Désactiver les comptes des personnes ayant quitté l'équipe
Inscription publique désactivée par défaut	Implémenté (`allow_registration=false` par défaut)	Garder `false` pour les instances privées
Validation e-mail obligatoire	Implémenté (`validation_email_required=true` par défaut)	Maintenir activé en production
Audit des créations/suppressions de comptes	Implémenté (enregistré dans `audit_log`)	Consulter régulièrement `/admin/audit`

© Gitrust

7. Durcissement système (responsabilité de l'administrateur)

Ces points sont hors du périmètre de gitrust mais nécessaires pour la conformité globale.

Critère ANSSI	Action requise
Service lancé sous utilisateur non-root	Créer l'utilisateur `gitrust` (voir tutoriel 02)
Durcissement systemd	Utiliser les directives `NoNewPrivileges`, `ProtectSystem=strict`, `PrivateTmp` (incluses dans `gitrust.service`)
Mises à jour de sécurité OS	`sudo apt-get update && apt-get upgrade` planifié (cron/unattended-upgrades)
Fail2ban contre le brute force SSH/HTTP	Voir Durcir avec Fail2ban
Sauvegarde chiffrée hors site	Chiffrer les archives de sauvegarde avec GPG ou age ; stocker sur un système distinct
Inventaire des dépendances (SBOM)	Activer `CI_SBOM_ENABLED=true` et `CI_DTRACK_ENABLED=true` pour le suivi CVE

Résumé des actions prioritaires

Pour une instance conforme PA-074, effectuez ces actions dans l'ordre :

1. **TLS obligatoire** — configurer Nginx avec certificat valide, HSTS, TLS 1.2+ uniquement
2. **JWT_SECRET fort** — `openssl rand -hex 64` dans `.env`
3. **APP_DEBUG=false** — force `COOKIE_SECURE=true` automatiquement
4. **Forcer le 2FA** — `totp_force_enabled=true` dans `/admin/settings`
5. **Politique de mots de passe** — `password_min_length=12` et complexité dans `/admin/settings`
6. **Inscription fermée** — `allow_registration=false` dans `/admin/settings`
7. **Rate limiting** — `RATE_LIMIT_LOGIN_PER_MINUTE=3` + Fail2ban
8. **Sauvegarde chiffrée** — script `backup.sh` + chiffrement GPG + stockage hors site
9. **Conservation des logs** — configurer journald et archivage CSV trimestriel
10. **SBOM** — `CI_SBOM_ENABLED=true` + Dependency-Track pour le suivi CVE

© Gitrust

Pour aller plus loin

- Régler le rate limiting
- Durcir avec Fail2ban
- Auditer l'instance
- Sauvegarder et restaurer

© Gitrust

1.5 Explications

1.5.1 Comprendre l'architecture globale de gitrust

Ce que vous allez comprendre

- Identifier les processus qui s'exécutent au sein d'une instance gitrust et leur rôle respectif
- Analyser le chemin d'une requête HTTP et d'un push SSH du client jusqu'à la réponse
- Évaluer les implications opérationnelles de cette architecture pour les décisions d'administration (sauvegarde, mise à l'échelle, durcissement)

Le problème concret

Vous administrez gitrust et vous devez décider : « Sur quel processus dois-je redémarrer quand je modifie la configuration SMTP ? Pourquoi la clé SSH hôte ne doit-elle jamais être régénérée ? Qu'est-ce qui se passe si PostgreSQL est indisponible pendant 2 minutes ? »

Ces questions n'ont de réponse claire que si vous comprenez l'architecture interne.

L'analogie

Pensez à gitrust comme à un immeuble de bureaux avec un seul accueil (le binaire `gitrust`) qui abrite plusieurs services distincts :

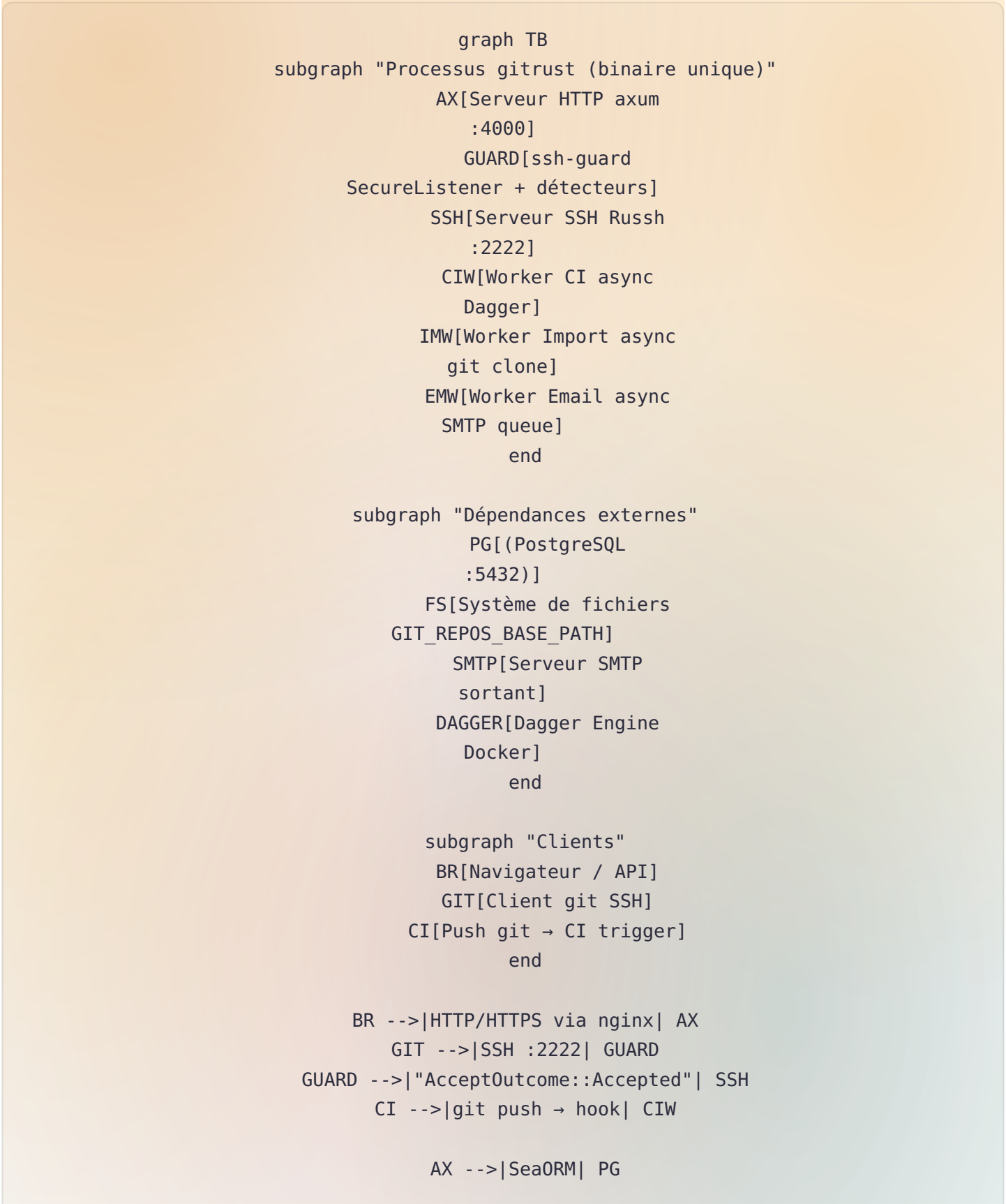
- Le **guichet HTTP** reçoit les navigateurs et les appels API
- Le **guichet SSH** reçoit les clients `git`
- La **salle des archives** (PostgreSQL) conserve toutes les données
- Les **ateliers** (workers CI, import) font du travail en arrière-plan
- Le **vestiaire** (système de fichiers) garde les manteaux (les dépôts bare)

Un seul processus, mais plusieurs « couloirs » internes. Si l'immeuble ferme (redémarrage), tous les guichets ferment simultanément.

© Gitrust

Le modèle

Composants au runtime



© Gitrust

```
AX -.->|"admin ACL/ban"| GUARD
GUARD -->|Bans, ACL, events| PG
SSH -->|Lit authorized_keys| PG
SSH -->|git-receive-pack| FS
AX -->|Lit/écrit dépôts| FS
CIW -->|dagger call| DAGGER
IMW -->|git clone| FS
EMW -->|SMTP| SMTP
```

Flux d'une requête HTTP (ex. : afficher un dépôt)

```
sequenceDiagram
participant B as Navigateur
participant N as Nginx (TLS)
participant AX as axum handler
participant DB as PostgreSQL
participant FS as Système de fichiers

B->>N: GET https://gitrust.domain/alice/mon-depot
N->>AX: proxy_pass http://127.0.0.1:4000/alice/mon-depot
AX->>DB: SELECT * FROM repositories WHERE owner=alice AND slug=mon-depot
DB-->>AX: Repository { id, disk_path, ... }
AX->>FS: git log --oneline HEAD (libgit2)
FS-->>AX: Liste des commits
AX-->>N: HTML (template Askama rendu)
N-->>B: HTTP 200 + HTML
```

Flux d'un push SSH (ex. : `git push origin main`)

```
sequenceDiagram
participant G as git client
participant SSH as Russh :2222
participant DB as PostgreSQL
participant FS as Dépôt bare (.git)
participant H as gitrust-hooks

G->>SSH: Connexion SSH, propose clé publique
SSH->>DB: SELECT id FROM ssh_keys WHERE fingerprint=SHA256:...
DB-->>SSH: user_id = alice
SSH-->>G: Authentifié
```


© Gitrust

```
G->>SSH: git-receive-pack /alice/mon-depot.git
SSH->>DB: Vérifie permission write sur le dépôt
DB-->>SSH: OK (Developer ou Owner)
SSH->>FS: Reçoit les objets Git, met à jour les refs
FS-->>SSH: Succès
SSH->>H: Déclenche post-receive hook
H->>DB: Enregistre l'événement push
H-->>SSH: Fin
SSH-->>G: remote: OK
```

Implications opérationnelles

Un seul processus, une seule unité d'arrêt

gitrust est un binaire unique. Redémarrer le service (`systemctl restart gitrust`) interrompt simultanément : - Le serveur HTTP (quelques secondes d'indisponibilité web) - Le serveur SSH (les push en cours sont coupés) - Les workers CI/Import (les tâches en cours peuvent être interrompues et reprises au prochain démarrage)

Conséquence : planifiez les redémarrages (mise à jour, changement de `.env`) en dehors des heures de pointe.

PostgreSQL est le point central

Toutes les décisions d'authentification, d'autorisation et de persistance passent par PostgreSQL. Si PostgreSQL est indisponible : - Les connexions HTTP retournent `503` - Les connexions SSH sont refusées (impossible de valider la clé publique) - Les workers mettent leurs tâches en file d'attente locale

Conséquence : la sauvegarde et la haute disponibilité de PostgreSQL sont prioritaires sur tout le reste.

Le système de fichiers et la base de données doivent être cohérents

Un dépôt existe à deux endroits : en base (`repositories` table) et sur disque (`disk_path`). Une restauration partielle (base seule, ou disque seul) laisse l'instance dans un état incohérent.

Conséquence : toujours sauvegarder et restaurer la base ET les dépôts ensembles (voir Sauvegarder et restaurer).

ssh-guard intercale un sas devant le serveur SSH

Toutes les connexions sur `:2222` passent d'abord par `SecureListener` (ssh-guard). Cette couche extrait l'IP réelle (PROXY protocol si nginx stream est devant), consulte ACL et bans actifs, applique un cap TCP par IP, puis ne remet le `TcpStream` à `russh` que si tout est en règle. Un événement JSON stable est émis pour chaque décision (`connection_accepted` , `connection_dropped` , `auth_failed` , `ip_banned` , ...) — pratique pour fail2ban et SIEM.

Conséquence : modifier `SSH_GUARD_*` dans `.env` change immédiatement le comportement défensif après redémarrage. Les ACL admin (allow/deny par CIDR) sont en revanche modifiables à chaud via l'UI : la table `ssh_guard_acl` est partagée par référence entre le routeur HTTP et le listener SSH. Voir Configurer ssh-guard et ssh-guard : détection d'attaques SSH.

© Gitrust

La clé SSH hôte identifie l'instance

La clé Ed25519 dans `SSH_HOST_KEY_PATH` est l'identité SSH de votre serveur. Tous vos utilisateurs ont ce fingerprint dans leur `~/.ssh/known_hosts`. La régénérer provoque une alerte `REMOTE HOST IDENTIFICATION HAS CHANGED` sur toutes les machines de vos utilisateurs.

Conséquence : sauvegardez la clé SSH hôte et ne la régénérez jamais sauf en cas de compromission avérée.

Alternatives et compromis

Pourquoi un seul binaire plutôt que des microservices ?

gitrust a choisi l'architecture monolithe modulaire (crates Rust séparées, un binaire) pour : - **Simplicité opérationnelle** : un seul processus à surveiller, une seule unité de déploiement - **Performances** : pas de sérialisation/désérialisation inter-services sur le chemin critique - **Cohérence transactionnelle** : les transactions PostgreSQL couvrent plusieurs domaines (auth + repository + audit) sans coordinateur distribué

Le compromis : mise à l'échelle horizontale plus contrainte (voir Modèle de déploiement).

Pourquoi Russh plutôt que sshd ?

L'intégration du serveur SSH dans le processus gitrust permet : - Authentification via la base de données (pas de `authorized_keys` fichier) - Autorisation au niveau dépôt (pas juste au niveau système) - Logs d'audit unifiés

Le compromis : gitrust gère sa propre implémentation SSH — les mises à jour de sécurité SSH dépendent des releases gitrust.

Vérifier votre compréhension

1. Un utilisateur ne peut plus faire `git push` mais peut se connecter à l'interface web. Quel composant est en cause ? Quelles informations cherchez-vous en premier dans les logs ?

2. Vous devez augmenter la mémoire allouée aux workers CI sans affecter le serveur HTTP. Est-ce possible avec l'architecture actuelle ? Que faudrait-il changer ?

Pour aller plus loin

- Modèle de déploiement — topologies mono-machine et HA
- Ports et services — tableau des ports et bindings
- ssh-guard : détection d'attaques SSH
- Configurer ssh-guard
- Sauvegarder et restaurer

© Gitrust

1.5.2 Comprendre les modèles de déploiement de gitrust

Ce que vous allez comprendre

- Comparer les trois topologies de déploiement (dev local, mono-machine prod, haute disponibilité) et leurs compromis
- Identifier les limites de la mise à l'échelle horizontale de gitrust dans sa version actuelle
- Évaluer quelle topologie correspond à votre contexte (taille d'équipe, contraintes de disponibilité, budget)

Le problème concret

Vous hébergez gitrust pour une équipe de 8 développeurs. Tout fonctionne bien, mais vous vous posez la question : faut-il un deuxième serveur pour la résilience ? Peut-on mettre gitrust derrière un load balancer ? Que se passe-t-il si le disque dur tombe ?

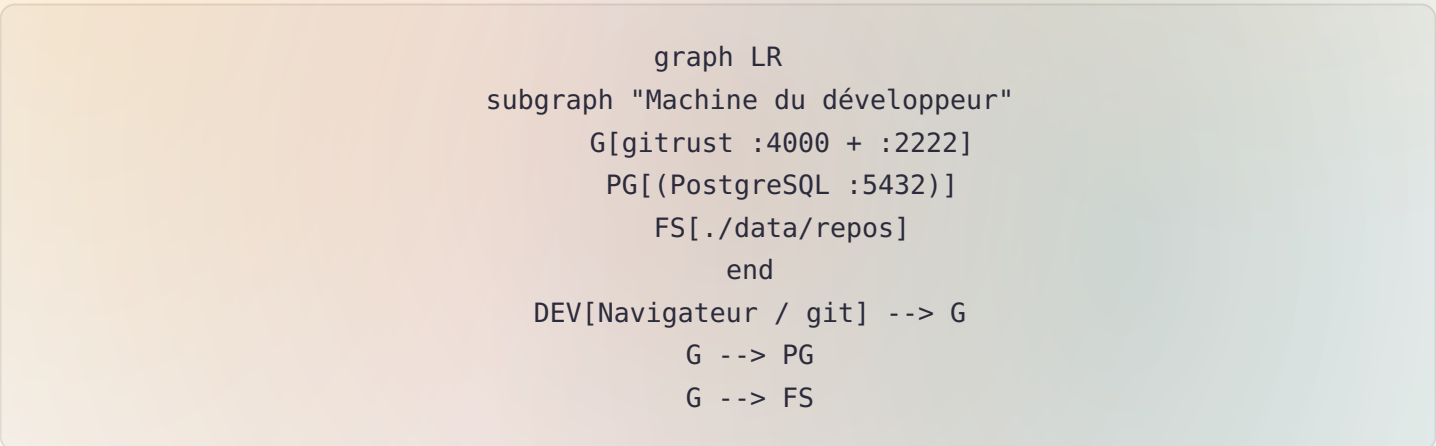
L'analogie

Les trois topologies ressemblent aux trois tailles de cuisine dans un restaurant : - **Cuisine de maison** (dev local) : tout sur le même plan de travail, pratique, pas question d'accueillir 50 couverts - **Cuisine de restaurant** (mono-machine prod) : séparation des postes, robuste, suffisant pour 20-50 couverts - **Cuisine industrielle** (HA) : plusieurs cuisines qui travaillent en parallèle, mais la coordination est un métier à part entière

La plupart des équipes de 3-20 personnes n'ont besoin que de la cuisine de restaurant.

Le modèle

Topologie 1 — Développement local

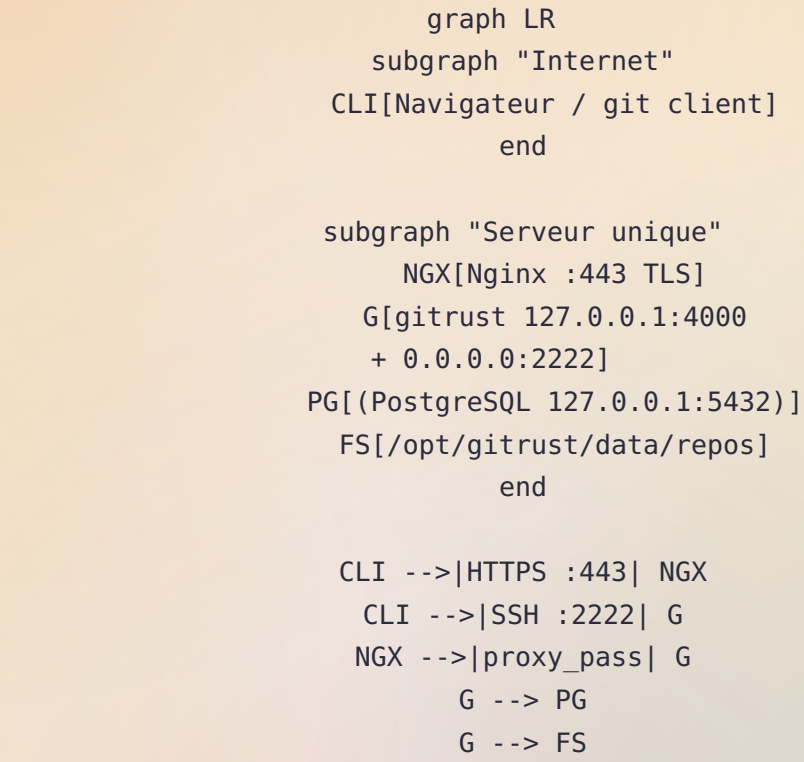


© Gitrust

Caractéristiques : - Tout sur la même machine, tout en `localhost` - `APP_DEBUG=true`, `COOKIE_SECURE=false`, `RUST_LOG=debug` - PostgreSQL lancé via `docker compose up -d` dans `database/` - Pas de reverse-proxy, pas de TLS - `SESSION_BACKEND=memory` acceptable (données perdues au redémarrage)

Usage : test de fonctionnalités, développement de plugins, démonstrations offline.

Topologie 2 — Mono-machine production (recommandée)



Caractéristiques : - Nginx termine TLS et proxifie vers gitrust sur `127.0.0.1:4000` - PostgreSQL accessible uniquement en local - Dépôts bare sur le disque local (ou volume attaché) - Un seul processus gitrust, un seul PostgreSQL - Sauvegarde avec `pg_dump` + `rsync` nocturne vers stockage externe

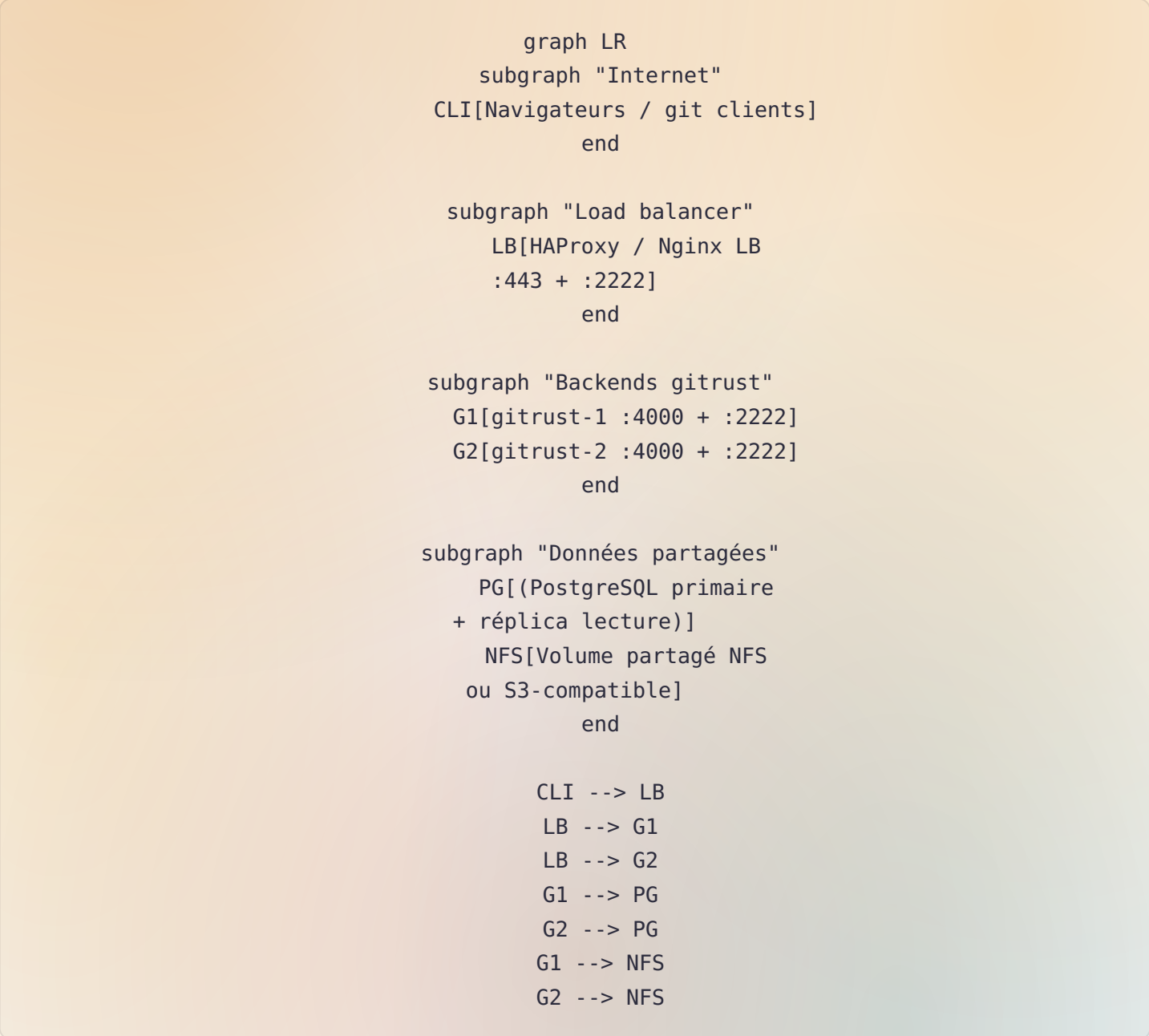
Dimensionnement indicatif :

Équipe	vCPU	RAM	Disque
1-5 personnes	1-2	1 Go	20 Go SSD
6-20 personnes	2-4	2-4 Go	50-100 Go SSD
20-50 personnes	4-8	8 Go	200+ Go SSD

© Gitrust

Limites : point de défaillance unique — si le serveur tombe, tout tombe. Acceptable si RTO < 1 h (restauration depuis sauvegarde sur nouvelle VM).

Topologie 3 — Haute disponibilité (HA)



Caractéristiques : - Plusieurs instances gitrust derrière un load balancer - PostgreSQL en mode primary/replica (streaming replication) - `SESSION_BACKEND=redis` obligatoire (sessions partagées entre instances) - Dépôts bare sur volume partagé (NFS, CEPH, ou objet S3 via FUSE)

© Gitrust

Limites de la mise à l'échelle horizontale

Workers stateful

Les workers CI et Import sont des goroutines async **dans le processus gitrust**. Leur état (pipelines en cours, file d'attente) n'est pas partagé entre instances. En HA, il faut s'assurer qu'un seul nœud exécute les workers CI, ou accepter que des pipelines soient redémarrés si le nœud tombe.

Contournement actuel : désigner un nœud « primary » pour les workers (`CI_ENABLED=true` sur un seul nœud, `CI_ENABLED=false` sur les autres).

Dépôts bare sur volume partagé

Les dépôts Git bare doivent être accessibles depuis tous les nœuds gitrust. Options : - **NFS** : simple, suffisant pour la plupart des cas, mais latence réseau visible sur les gros push - **CEPH / GlusterFS** : haute performance distribuée, complexité opérationnelle élevée - **S3-compatible (s3fs/mountpoint)** : économique pour l'archivage, trop lent pour les accès Git fréquents - **Réplication rsync en quasi-temps-réel** : simple mais non-atomique (risque de dépôt corrompu pendant une réplication)

Recommandation : pour la grande majorité des équipes (< 50 personnes), la topologie mono-machine avec une bonne stratégie de sauvegarde offre un rapport complexité/disponibilité bien meilleur que la HA.

PostgreSQL en réplication

La réplication streaming PostgreSQL (primary/replica) est bien documentée et fiable. Le failover automatique nécessite un outil supplémentaire (Patroni, repmgr). Sans failover automatique, la HA PostgreSQL n'apporte qu'un RPO bas (perte de données minimale), pas un RTO bas (indisponibilité réduite).

Alternatives et compromis

Critère	Mono-machine	HA (2 nœuds)
Complexité opérationnelle	Faible	Élevée
Coût infra	Faible	2-3x
RTO (temps de restauration)	15-60 min (depuis sauvegarde)	< 5 min (failover)
RPO (perte de données max)	Depuis la dernière sauvegarde	Quelques secondes (WAL)
Maintenance sans coupure	Non (redémarrage = coupure)	Oui (rolling update)
Adapté pour	Équipes 1-50 pers.	Équipes > 50 pers. ou SLA strict

© Gitrust

Pour la cible principale de gitrust (équipes de 3-20 personnes) : la topologie mono-machine avec sauvegarde quotidienne offre un RTO acceptable. Investissez plutôt dans une bonne stratégie de sauvegarde que dans une architecture HA complexe.

Vérifier votre compréhension

1. Votre équipe passe de 10 à 40 personnes. Quels indicateurs surveilleriez-vous sur votre serveur mono-machine pour décider si une migration vers la HA est nécessaire ?

2. En topologie HA avec deux nœuds gitrust et un volume NFS partagé, quel est le scénario de défaillance qui ne peut pas être résolu par la HA seule ? Que faut-il prévoir en plus ?

Pour aller plus loin

- Architecture globale — composants et flux
- Stratégie de sauvegarde — 3-2-1 adapté à gitrust
- Ports et services
- Variables d'environnement — `SESSION_BACKEND`, `REDIS_URL`

© Gitrust

1.5.3 Comprendre la CI Dagger : Easy Mode et Power Mode

Ce que vous allez comprendre

- Expliquer pourquoi gitrust utilise Dagger plutôt qu'un système CI YAML classique
- Distinguer le « Easy Mode » (`.gitrust-ci.yml`) du « Power Mode » (module `.dagger/`)
- Évaluer quand conseiller à vos utilisateurs de passer de l'un à l'autre

---

Le problème concret

Un utilisateur vous signale que sa CI « ne fait pas ce qu'il veut » avec `.gitrust-ci.yml`. Il a besoin de logique conditionnelle, de dépendances entre jobs, et de construire plusieurs images Docker en parallèle. Devez-vous modifier le moteur CI ou lui dire qu'il doit changer d'approche ?

L'analogie

Easy Mode, c'est la **recette de cuisine imprimée sur la boîte de gâteau** : elle marche pour 95 % des cas, est accessible aux débutants, mais on ne peut pas y ajouter de la fleur d'oranger sans sortir du cadre.

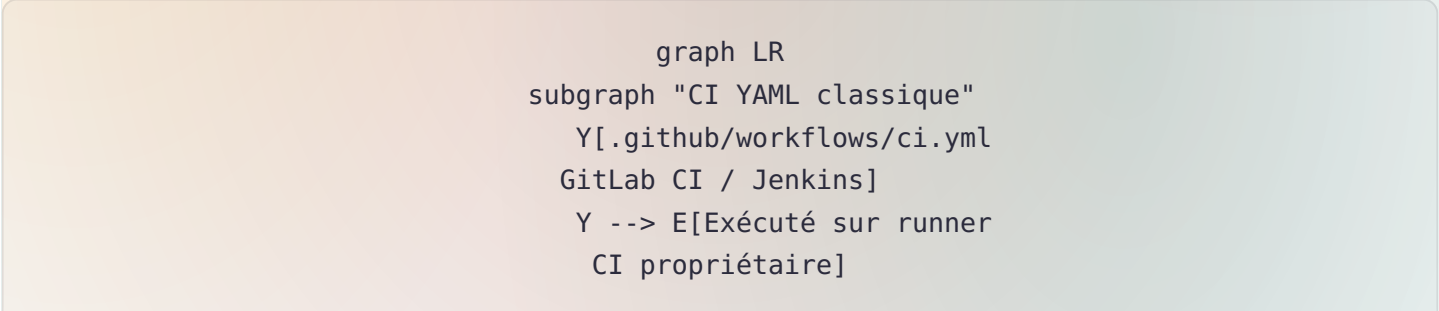
Power Mode, c'est le **livre de pâtisserie professionnel** : vous écrivez vos propres recettes en code (Go, Python, TypeScript), vous pouvez tout faire, mais il faut savoir pâtisser.

Dagger est l'**atelier** qui garantit que les deux approches s'exécutent dans exactement le même environnement, reproductible sur n'importe quelle machine.

Le modèle

Pourquoi Dagger ?

Les systèmes CI YAML classiques (GitHub Actions, GitLab CI) ont un problème fondamental : la logique de pipeline est encodée dans du YAML qui devient vite illisible, non-testable, et non-composable. Dagger a été créé par l'équipe fondatrice de Docker pour résoudre ce problème.



© Gitrust

```
E --> R1[Résultat dépend
   du runner]
        end

subgraph "CI Dagger"
   D[Pipeline en code
Go / Python / TS]
 D --> DE[Dagger Engine
Docker containers]
 DE --> R2[Reproductible
local = CI = prod]
        end
```

Avantages de Dagger pour gitrust : - Reproductibilité : le même pipeline s'exécute identiquement en local et sur le serveur - Isolation : chaque job dans son propre conteneur Docker - Cache automatique : les layers Docker non modifiés ne sont pas reconstruits - Pas de dépendance à un service CI externe (GitHub Actions, etc.)

Easy Mode : `.gitrust-ci.yml`

Le fichier `.gitrust-ci.yml` est un profil YAML simplifié interprété par le **moteur CI générique** de gitrust (un module Dagger Python interne à la plateforme).

```
flowchart LR
A[.gitrust-ci.yml
dans le dépôt] -->|lecture| B[Moteur CI gitrust
module Dagger Python]
B -->|dagger call| C[Dagger Engine]
C --> D[Container isolé
build + test]
D --> E[Résultat → DB + UI]
```

Structure type :

```
# .gitrust-ci.yml
pipeline:
  name: CI principale
  image: rust:1.77-slim
  steps:
    - name: Vérifier le formatage
      run: cargo fmt --check
    - name: Linting
      run: cargo clippy -- -D warnings
```


© Gitrust

```
    - name: Tests
      run: cargo test --workspace
  cache:
    - path: ~/.cargo/registry
    - path: target/
```

Ce que Easy Mode fait bien : - Build + test + lint pour des projets simples à intermédiaires - Configuration en quelques lignes - Accessible aux utilisateurs sans expertise CI/CD

Limites du Easy Mode : - Pas de logique conditionnelle complexe (si branche = main alors déployer) - Pas de fan-out/fan-in (jobs parallèles avec synchronisation) - Pas d'accès au Daggerverse (bibliothèque de modules Dagger communautaires) - La logique reste dans le YAML de la plateforme, pas dans le code du projet

Power Mode : module `.dagger/`

Quand un dépôt contient un dossier `.dagger/`, gitrust détecte sa présence et exécute directement le module Dagger natif. La plateforme ne l'interprète pas : elle lance `dagger call` sur le module du projet.

```
flowchart LR
   A[.dagger/
main.go ou main.py] -->|détecté| B[gitrust CI Worker]
   B -->|dagger call -m .dagger/ ci| C[Dagger Engine]
   C --> D[Pipeline défini
    par l'utilisateur]
   D --> E[Résultat → DB + UI]
```

Ce que Power Mode apporte : - La logique pipeline est du code versionné avec le projet - Tests unitaires des pipelines possibles - Composition via le Daggerverse - Accès à toutes les fonctionnalités Dagger (secrets, services, multi-platform builds) - Aucune dépendance au moteur CI de gitrust pour la logique

Exemple minimal en Go :

```
// .dagger/main.go
package main

import (
    "context"
    "dagger.io/dagger"
)

func main() {
    ctx := context.Background()
```


© Gitrust

```
    client, _ := dagger.Connect(ctx)
    defer client.Close()

    rust := client.Container().
        From("rust:1.77-slim").
        WithMountedCache("/usr/local/cargo/registry", client.CacheVolume("cargo-registr
        WithDirectory("/src", client.Host().Directory(".")).
        WithWorkdir("/src")

    rust.WithExec([]string{"cargo", "fmt", "--check"}).
        WithExec([]string{"cargo", "clippy", "--", "-D", "warnings"}).
        WithExec([]string{"cargo", "test", "--workspace"}).
        Sync(ctx)
}
```

Alternatives et compromis

Critère	Easy Mode (`.gitrust-ci.yml`)	Power Mode (`.dagger/`)
Accessibilité	Excellent (quelques lignes YAML)	Courbe d'apprentissage
Puissance	Limitée aux cas simples	Illimitée
Logique conditionnelle	Non	Oui
Parallélisme avancé	Non	Oui
Testabilité du pipeline	Nulle	Totale
Composition (Daggerverse)	Non	Oui
Maintenance	Moteur géré par la plateforme	Chaque dépôt est autonome
Alignement philosophie Dagger	Partiel (YAML encode de la logique)	Total

Recommandation pour vos utilisateurs : - Commencer en Easy Mode pour les projets simples - Migrer en Power Mode quand : besoin de logique conditionnelle, de fan-out/fan-in, d'intégration avec des services externes, ou quand le `.gitrust-ci.yml` dépasse 50 lignes

© Gitrust

Implications pour l'administration

En tant qu'administrateur :

- `CI_ENABLED=false` désactive toute exécution CI (Easy et Power Mode), utile lors d'une maintenance
- `CI_MAX_CONCURRENT` s'applique aux deux modes
- Le moteur Easy Mode (`CI_ENGINE_PATH`) doit être synchronisé lors des mises à jour de gitrust
- Les modules Power Mode sont autonomes — leur mise à jour est la responsabilité de l'utilisateur

---

Vérifier votre compréhension

1. Un utilisateur vous demande d'ajouter une étape « déploiement vers staging si la branche est `main` » dans son `.gitrust-ci.yml`. Quelle est la réponse correcte ? Quelle alternative proposez-vous ?

2. En quoi le fait que Dagger garantisse la reproductibilité facilite-t-il le débogage d'un pipeline qui échoue uniquement sur le serveur et pas en local ?

Pour aller plus loin

- Dépanner la CI
- Variables d'environnement — section CI/CD
- Architecture globale — workers CI

© Gitrust

1.5.4 Comprendre la stratégie de sauvegarde adaptée à gitrust

Ce que vous allez comprendre

- Appliquer la règle 3-2-1 aux trois sources de vérité de gitrust (base de données, dépôts bare, clé SSH hôte)
- Analyser les RPO et RTO réalistes pour une instance gitrust mono-machine
- Évaluer les compromis entre `pg_dump` classique et réplication WAL en continu

---

Le problème concret

Votre serveur gitrust est hors service suite à une défaillance disque. Vous avez une sauvegarde de la base de données, mais pas des dépôts bare. Ou l'inverse. Combien de temps pour remettre l'instance en service ? Combien de données perdez-vous ?

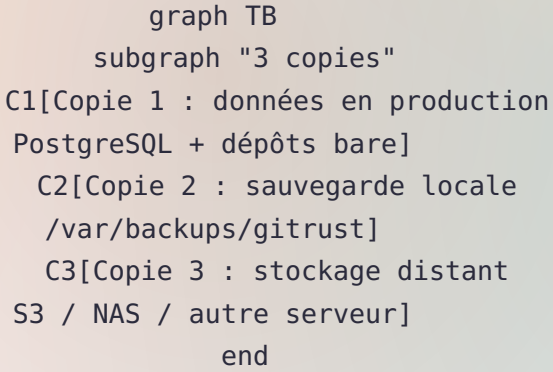
L'analogie

Imaginez gitrust comme un cabinet d'architecte : - La **base de données** est le registre des contrats et des clients (qui possède quoi, les métadonnées) - Les **dépôts bare** sont les tiroirs de plans (les fichiers réels) - La **clé SSH hôte** est la plaque officielle à l'entrée (l'identité du cabinet)

Si vous avez le registre mais pas les tiroirs, vous savez que le projet « Maison Martin » existe mais vous n'avez plus les plans. Si vous avez les tiroirs mais pas le registre, vous avez des dossiers anonymes sans savoir à qui ils appartiennent. Les deux doivent être sauvegardés ensemble.

Le modèle

La règle 3-2-1 appliquée à gitrust



© Gitrust

```
    subgraph "2 supports différents"
      S1[Disque SSD serveur principal]
       S2[Disque NAS ou volume objet]
                  end

      subgraph "1 copie hors site"
          O1[Datacenter différent
      ou cloud object storage]
                  end

                C1 --> S1
                C2 --> S1
                C3 --> S2
                S2 --> O1
```

Les 3 copies : 1. Production (données en vie sur le serveur) 2. Sauvegarde locale (`/var/backups/gitrust` sur le même serveur ou volume attaché) 3. Sauvegarde distante (rsync vers NAS, `rclone` vers S3, etc.)

Les 2 supports : le disque du serveur principal + un support physiquement distinct.

1 hors site : en cas d'incendie, d'inondation ou de compromission du datacenter, une copie doit être inaccessible depuis le réseau principal.

Les trois sources de vérité de gitrust

Source	Taille typique	Consistance	Fréquence recommandée
PostgreSQL (`pg_dump`)	Quelques Mo à plusieurs Go	Cohérente à un instant T (`--format=custom`)	Quotidien minimum, idéalement 2x/jour
Dépôts bare (rsync)	Dizaines de Mo à plusieurs To	Cohérente par dépôt (Git garantit l'atomicité des packs)	Quotidien minimum
Clé SSH hôte (`ssh_host_ed25519_key`)	< 1 Ko	Immuable après le premier démarrage	Une seule fois, puis conserver en lieu sûr

© Gitrust

RPO et RTO réalistes

RPO (Recovery Point Objective) : quantité maximale de données perdues.

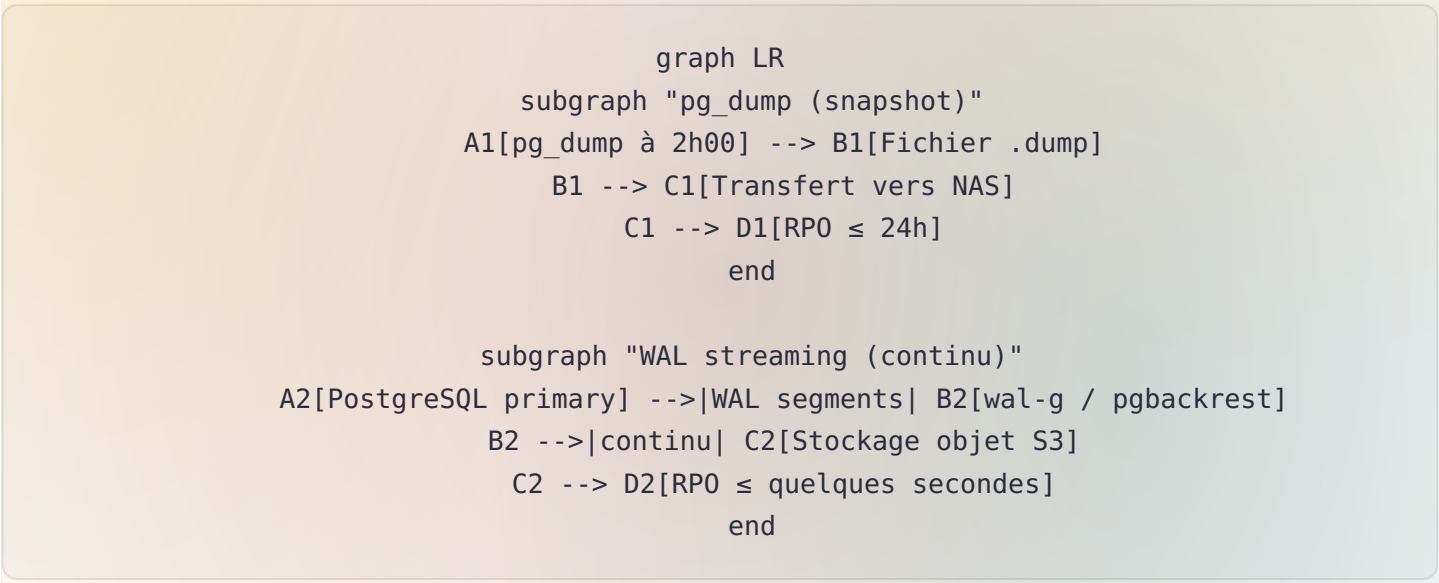
Stratégie	RPO
`pg_dump` quotidien à 2h	≤ 24 h de données perdues
`pg_dump` toutes les 6 h	≤ 6 h de données perdues
WAL streaming (wal-g)	≤ quelques secondes

Pour la cible principale de gitrust (équipes 3-20 personnes), un RPO de 24 h est généralement acceptable. Si votre équipe code intensément la nuit, envisagez 2 sauvegardes quotidiennes.

RTO (Recovery Time Objective) : temps pour remettre l'instance en service après incident.

Scénario	RTO estimé
Restauration sur la même machine (disque remplacé)	30-60 min
Restauration sur une nouvelle VM (même datacenter)	15-30 min
Restauration sur une nouvelle VM (nouveau datacenter)	60-90 min (transfert des sauvegardes)

`pg_dump` vs réplication WAL



© Gitrust

`pg_dump` — recommandé pour commencer : - Simple à mettre en place et à tester - Fichier standalone restaurable avec `pg_restore` - RPO = interval entre deux dumps - Fonctionne même si PostgreSQL est sous charge (cohérence transactionnelle garantie par PostgreSQL)

WAL streaming (`wal-g` , `pgbackrest`) — pour les cas avancés : - RPO quasi nul (Point-in-Time Recovery) - Nécessite un stockage objet (S3, MinIO) - Complexité opérationnelle significativement plus élevée - Indispensable si votre activité impose un RPO < 1 h

Recommandation : commencez par `pg_dump` quotidien + rsync. Migrez vers WAL streaming uniquement si votre SLA l'exige.

Cohérence entre base de données et dépôts bare

La sauvegarde la plus dangereuse est celle qui est **partiellement réussie** : base sauvegardée, dépôts non sauvegardés (ou l'inverse).

Un dépôt existe à deux endroits : - En base : `repositories.disk_path` enregistre le chemin - Sur disque : le dossier `.git` bare contient les objets Git

Pour éviter l'incohérence, sauvegardez toujours les deux dans la **même fenêtre temporelle**. Le script `backup.sh` recommandé effectue les deux en séquence :

```
# Ordre correct : d'abord la DB, puis les dépôts
pg_dump → fichier .dump
rsync repos → destination
```

En pratique, le décalage entre les deux (quelques secondes/minutes) est acceptable : un dépôt créé entre les deux sauvegardes sera absent du rsync mais présent en base — une re-synchronisation après restauration suffit.

Tester la restauration (le drill)

Une sauvegarde non testée n'est pas une sauvegarde — c'est une espérance.

Planifiez un **drill trimestriel** sur une VM éphémère : 1. Copier les fichiers de sauvegarde sur la VM 2. Installer gitrust et PostgreSQL 3. Restaurer avec les procédures documentées dans Sauvegarder et restaurer 4. Vérifier : connexion admin, liste des dépôts, `git clone` d'un dépôt connu 5. Mesurer le temps effectif → comparer avec votre RTO cible 6. Documenter les étapes imprévues → améliorer le script

Le drill révèle les problèmes invisibles : fichier de sauvegarde corrompu, script qui suppose un chemin qui n'existe pas, mot de passe PostgreSQL non documenté.

© Gitrust

Alternatives et compromis

Approche	RPO	Complexité	Coût
`pg_dump` quotidien + rsync	≤ 24 h	Faible	Faible
`pg_dump` 2x/jour + rsync	≤ 12 h	Faible	Faible
WAL streaming + rsync	≤ 1 min	Élevée	Moyen (stockage objet)
Réplication PostgreSQL (HA) + rsync	≤ quelques s	Très élevée	Élevé

Pour les équipes de 3-20 personnes avec gitrust, la première ligne est le bon point de départ.

Vérifier votre compréhension

1. Vous utilisez `pg_dump` quotidien à 2h00. Un développeur pousse 3 jours de travail le mercredi soir à 23h45. Le serveur tombe à minuit. Que perdez-vous, et pourquoi ?

2. Quelle est la différence entre RPO et RTO ? Lequel est le plus facile à améliorer avec une architecture mono-machine ? Lequel nécessite une refonte architecturale ?

Pour aller plus loin

- Sauvegarder et restaurer — procédures pas-à-pas
- Modèle de déploiement — HA et ses implications sur la sauvegarde
- Architecture globale — les trois sources de vérité

© Gitrust

1.5.5 ssh-guard : détection d'attaques SSH

Ce que vous allez comprendre

- Identifier les quatre motifs d'attaque détectés automatiquement et les seuils par défaut.
- Analyser comment ssh-guard décide de bannir une IP, dans quel ordre les règles s'appliquent et pourquoi.
- Évaluer les compromis (fail-open, dry-run, allowlist bypass) pour ajuster la posture de sécurité de votre instance.

> ***Public** : administrateurs gitrust qui veulent comprendre **avant** de toucher aux variables `SSH_GUARD_*`. Pour la recette pas à pas, voir Configurer ssh-guard.*

---

1. Le problème concret

Vous administrez une instance gitrust exposée sur Internet. Les logs `russh` remontent des dizaines de tentatives d'authentification ratées chaque jour, sans corrélation. Vous savez intuitivement qu'il y a une attaque par dictionnaire en cours, mais :

- vous ne savez pas combien d'IP distinctes la mènent ;
- vous ne pouvez pas distinguer un développeur qui a 3 clés mal configurées d'un scanner qui essaie 50 clés en série ;
- votre fail2ban actuel se base sur des regex sur les messages d'erreur de `russh`, qui changent à chaque mise à jour ;
- vous craignez de bannir un partenaire CI qui ouvre 200 connexions SSH par minute pour ses pipelines.

ssh-guard a été conçu exactement pour ce périmètre : observer, classer, corréler, bannir — sans casser les utilisations légitimes intensives.

---

2. L'analogie

Imaginez le hall d'entrée d'un immeuble. Quatre comportements doivent déclencher un signal pour le gardien :

1. **Quelqu'un sonne 5 fois en 5 minutes au même appartement** → c'est une tentative de force brute. *(brute-force)*
2. **Quelqu'un sonne aux 10 appartements différents en 5 minutes** → il cherche qui est chez lui pour cibler ensuite. *(énumération d'utilisateurs)*
3. **Quelqu'un présente 10 cartes magnétiques différentes en 5 minutes** → il essaie un trousseau volé. *(scanning de clés)*
4. **Quelqu'un ouvre la porte 30 fois en 1 seconde** → il sature le système, peu importe ses intentions. *(flood TCP)*

ssh-guard distingue les quatre, parce qu'ils appellent des réponses différentes. Le flood est un drop instantané (sinon le système se noie). Les trois autres déclenchent un ban temporaire après un nombre d'essais avéré.

© Gitrust

3. Les quatre détecteurs

3.1 Vue synthétique

Détecteur	Mesure	Seuil défaut	Fenêtre	Action
Brute-force	Échecs d'auth depuis une IP	5	5 min	Ban 1 h
Énumération d'utilisateurs	Usernames distincts essayés depuis une IP	10	5 min	Ban 1 h
Scan de clés	Fingerprints distincts essayés depuis une IP	10	5 min	Ban 1 h
Flood TCP	Nouvelles connexions TCP par IP par seconde	10 (burst 20)	1 s glissante	Drop immédiat (pas de ban persistant)

Tous les seuils sont ajustables via `SSH_GUARD_*` — voir Variables d'environnement.

3.2 Brute-force

Compte les événements `auth_failed` par IP dans la fenêtre glissante. Au seuil, émet `brute_force_detected` puis pose un ban auto avec TTL `SSH_GUARD_AUTO_BAN_DURATION_SECS`. Ne **compte pas** les `auth_succeeded` (c'est l'objectif : ignorer le bruit légitime).

3.3 Énumération d'utilisateurs

Compte les **usernames distincts** essayés par IP. Un attaquant qui sonde `root`, `admin`, `git`, `deploy`, `ci`, `bot`, … pour trouver un compte existant déclenche ce détecteur même si chacune de ses tentatives n'est faite qu'une fois.

Cas typiques de faux positif : un script CI qui clone plusieurs dépôts d'orgs différentes avec différents `git@host:org/...` — mais le `user` SSH reste `git`, donc le détecteur ne s'active pas. Un développeur qui se trompe trois fois de username en jouant avec son `~/.ssh/config` reste largement sous le seuil.

3.4 Scan de clés

Compte les **fingerprints SHA256 distincts** présentés par IP. Aligné sur les recommandations CrowdSec : un humain n'a jamais 10 clés SSH distinctes en 5 minutes. Un trousseau volé qui défile en automatique, oui.

Cas typique d'observation : `ssh-add -L` côté client liste 8 clés, l'agent SSH les présente toutes successivement. Le seuil par défaut (10) absorbe ce cas.

© Gitrust

3.5 Flood TCP

Implémenté avec un token bucket GCRA (algorithme de Generic Cell Rate, équivalent leaky bucket sans biais), keyé par IP. Différent des trois autres :

- **Pas de ban persistant** : sinon un scan de port créerait des milliers de lignes inutiles dans la DB.
- **Drop au niveau du listener TCP** : la connexion ne consomme aucune ressource au-delà du `accept()`.
- **Budgets indépendants par IP** : une IP rate-limitée n'affecte pas les autres.

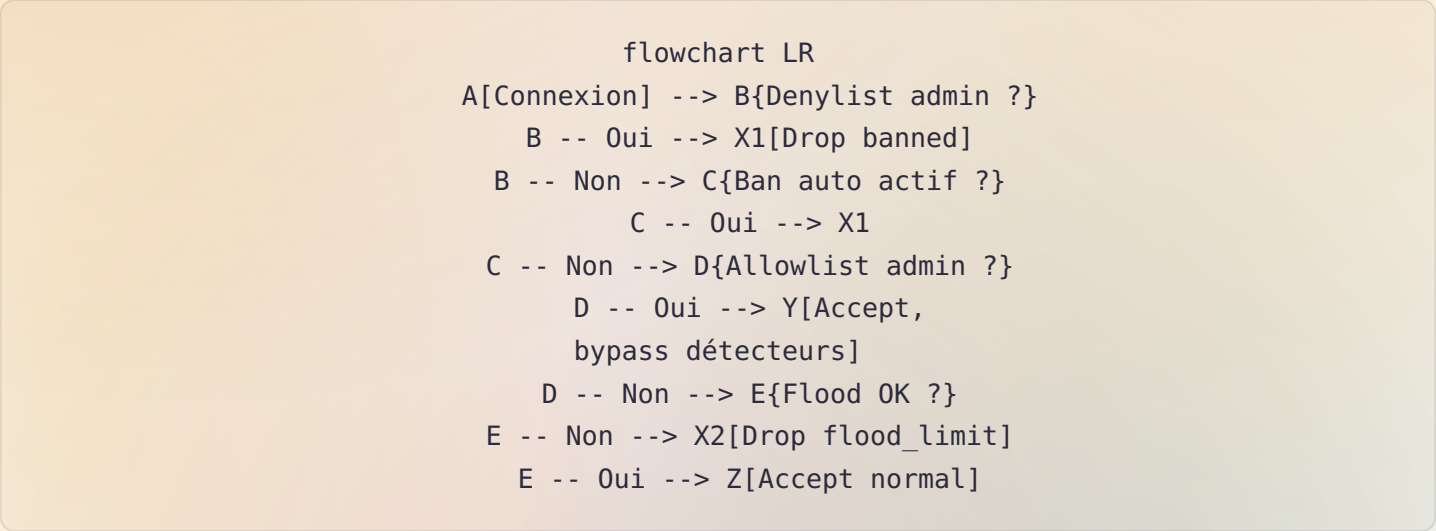
C'est la première ligne de défense, peu coûteuse, qui absorbe les scans avant que les détecteurs comportementaux ne soient sollicités.

---

4. Le modèle de décision

4.1 Priorité des règles d'ACL

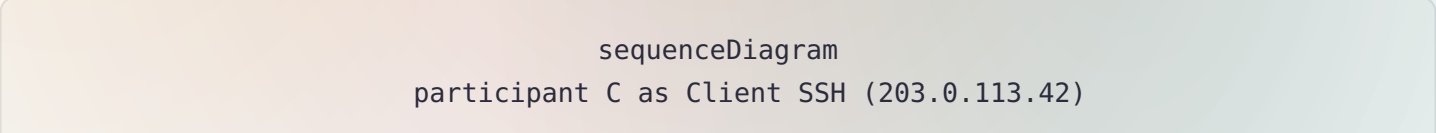
Quand une connexion arrive, ssh-guard consulte dans cet ordre :



Conséquences pratiques :

- `deny` **admin** > **tout** : un opérateur peut bannir une IP même si elle a été allowlistée par erreur ailleurs.
- `auto_ban` > `allow` : un ban posé par un détecteur ne saute pas en allowlistant l'IP après coup. Pour libérer l'IP, il faut explicitement la débannir. C'est volontaire — un `allow` ne doit pas effacer la trace d'un comportement passé.
- `allow` > `default` : une IP allowlistée bypasse les détecteurs et le flood-limit, mais ses événements `auth_succeeded` / `auth_failed` restent loggés pour audit.

4.2 Cycle de vie d'un ban brute-force



© Gitrust

```
participant L as SecureListener
participant T as AuthTracker
participant D as BruteForceDetector
participant B as BanManager
participant S as Sink JSON
participant F as Fail2ban

Note over C,F: Tentatives 1 à 4 (sous le seuil)
C->>L: TCP accept
L->>S: connection_accepted
C->>T: auth fail (russh)
T->>S: auth_failed
T->>D: on_event(auth_failed)
D->>D: count = 4 < seuil 5
Note over D: Aucun ban

Note over C,F: Tentative 5 (atteint le seuil)
C->>T: auth fail (russh)
T->>S: auth_failed
T->>D: on_event(auth_failed)
D->>D: count = 5 >= seuil
D->>S: brute_force_detected
D->>B: auto_ban(BruteForce)
B->>S: ip_banned (expires_at = now + 1h)
F->>F: Lit ip_banned, applique iptables/UFW

Note over C,F: Tentative 6 (bannie)
C->>L: TCP accept
L->>B: effective_status(203.0.113.42)
B-->>L: BannedAuto
L->>S: connection_dropped (banned)
L-->>C: connexion fermée
```

4.3 Idempotence

Un détecteur ne **rebannit jamais** une IP déjà bannie. Le `BanManager.auto_ban` retourne `Ok(None)` dans trois cas :

- `dry_run` actif (un événement `ip_banned` est tout de même émis pour fail2ban) ;
- l'IP est allowlistée ;
- un ban auto actif couvre déjà l'IP.

Sans cette idempotence, chaque nouvelle tentative ratée après le ban produirait un nouvel `ip_banned` et noierait le sink.

© Gitrust

5. Les leviers admin

5.1 Modes opérationnels

Variable	Effet
`SSH_GUARD_ENABLED=false`	Coupe totale. ssh-guard devient un pass-through. **À éviter sauf urgence.**
`SSH_GUARD_DRY_RUN=true`	Les détecteurs tournent et émettent les événements `ip_banned`, mais aucun ban n'est persisté. Mode parfait pour valider un nouveau seuil ou laisser fail2ban faire le ban réel.
`SSH_GUARD_PROFILE=private`	Profil réseau interne : tous les détecteurs désactivés (seuil `u32::MAX`), les événements continuent d'être émis pour audit.

5.2 Allowlist d'un partenaire CI ou d'un VPN

Une IP allowlistée bypasse les quatre détecteurs et le flood-limit. C'est le levier propre pour autoriser :

- un runner CI qui ouvre des dizaines de connexions SSH par minute ;
- un bastion VPN qui multiplexe plusieurs développeurs vers une seule IP source ;
- un job de monitoring qui teste la disponibilité SSH.

Configuré via l'API admin (table `ssh_guard_acl`, kind = `Allow`, CIDR du runner). Voir Configurer ssh-guard.

5.3 Denylist d'un AS abusif

Un opérateur peut bannir un CIDR entier (ex. `198.51.100.0/24`) en mode permanent. C'est le pendant inverse de l'allowlist, prioritaire sur tout (y compris les bans auto qui n'auraient pas encore été posés).

6. Pourquoi ces compromis

6.1 Fail-open quand PostgreSQL est indisponible

Si la lecture du store échoue (DB down, latence anormale), ssh-guard **laisse passer** la connexion et log un warn. Raison : éviter de transformer une panne DB en panne SSH généralisée. Conséquence acceptée : pendant la fenêtre de panne, un attaquant peut atteindre le handshake `russh` (mais doit toujours présenter une clé valide pour s'authentifier).

Pour basculer en fail-closed, il faut modifier le code (voir page développeur). C'est rare dans la vraie vie ; la plupart des admins préfèrent la disponibilité.

© Gitrust

6.2 Bans temporaires plutôt que permanents

Le défaut `SSH_GUARD_AUTO_BAN_DURATION_SECS=3600` (1 heure) est un compromis :

- assez long pour ralentir significativement un attaquant patient ;
- assez court pour qu'un faux positif (un script mal écrit chez un utilisateur légitime) se résolve sans intervention admin.

Mettre `0` rend les bans permanents — à n'utiliser qu'avec un workflow admin pour purger régulièrement la table.

6.3 Fenêtre glissante de 5 minutes

Suffisamment large pour qu'un attaquant ne puisse pas « réinitialiser » le compteur en attendant 30 secondes entre tentatives. Suffisamment courte pour qu'un développeur qui se trompe 4 fois de mot de passe à 8h, puis revient à 14h, ne déclenche pas de ban.

6.4 Allowlist qui bypasse les détecteurs

Une IP allowlistée n'est plus surveillée par les détecteurs. Donc une compromission depuis une IP « de confiance » ne sera pas détectée par ssh-guard. Le compromis est explicite : l'allowlist est un signal de confiance opérationnelle, pas une garantie de sécurité. Si un partenaire CI est compromis, l'attaque sera vue par les logs `auth_succeeded` (qui sont toujours émis), pas par les détecteurs.

---

7. Quand ssh-guard ne suffit pas

ssh-guard est une couche de défense. Elle ne dispense pas de :

- un firewall en amont (UFW, iptables) qui ferme les ports inutiles ;
- des clés SSH fortes (ed25519, ou RSA 4096+) côté utilisateur ;
- une politique de rotation des PAT pour l'API HTTP (gitrust ne sert pas que SSH) ;
- un fail2ban pour l'HTTP (login web, API), couvert par d'autres jails — voir Durcir avec Fail2ban.

Les quatre motifs d'attaque détectés sont les plus fréquents au niveau SSH, mais ils ne couvrent **pas** : attaques par chronométrie, exfiltration via tunnel, abus de privilèges après auth réussie. Ces vecteurs nécessitent d'autres outils (audit syscalls, EDR, revue de RBAC).

---

8. Vérifier votre compréhension

1. Une IP allowlistée présente 50 fingerprints différents en 30 secondes. Combien d'événements `key_scanning_detected` sont émis ? L'IP est-elle bannie ?
2. Vous activez `SSH_GUARD_DRY_RUN=true` puis vous voyez 12 événements `ip_banned` dans la dernière heure. Combien d'IP sont effectivement bloquées par ssh-guard ? Que faut-il pour qu'elles soient bloquées ?
3. PostgreSQL est inaccessible pendant 10 minutes. Une attaque brute-force depuis une IP nouvelle démarre à la 5e minute. (a) Le détecteur va-t-il déclencher un ban ? (b) Les événements `auth_failed` apparaissent-ils dans les logs ?

© Gitrust

9. Pour aller plus loin

- Configurer ssh-guard — recettes par profil de déploiement
- Événements ssh-guard (JSON) — schéma stable du flux d'événements
- Variables d'environnement — SSH_GUARD_*
- Durcir avec Fail2ban — chaîner ssh-guard et fail2ban
- Architecture globale — où ssh-guard se place dans le runtime gitrust
- Conformité ANSSI PA-074

© Gitrust