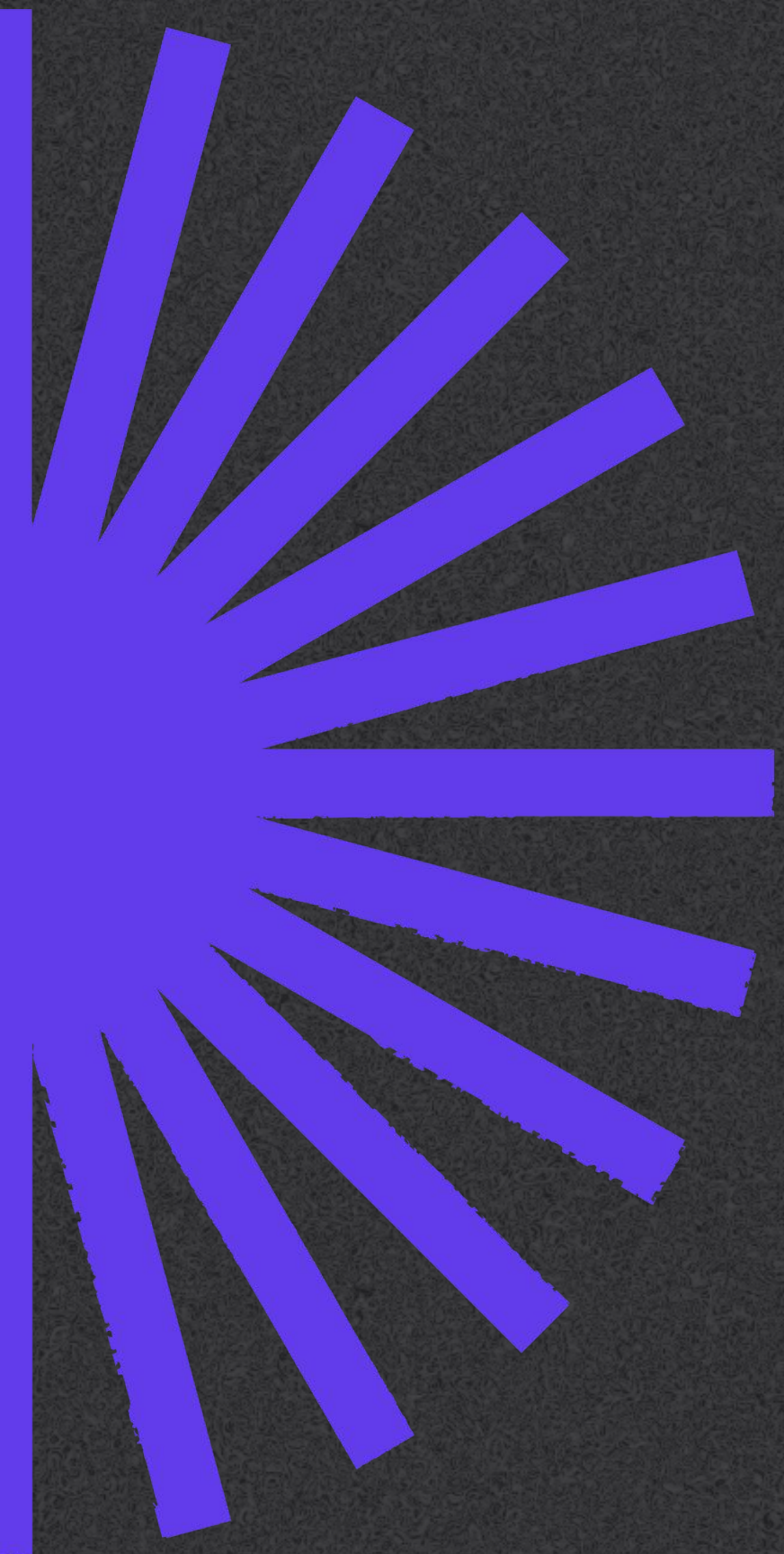
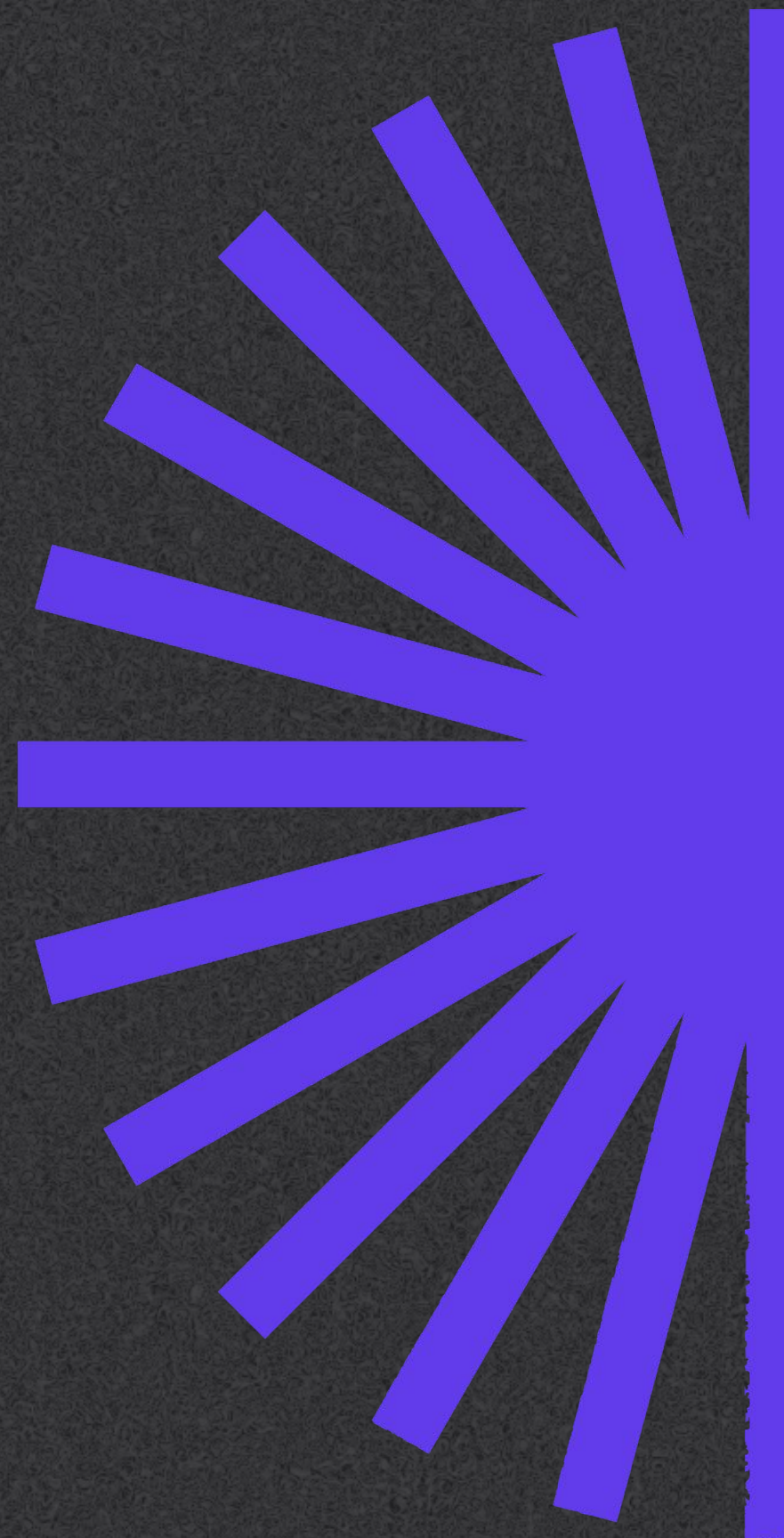


Intégration continue Déploiement continu

Skolae ESGI Tours niveau Master 1 dev
Nicolas Hersant – © 2026

Séance 1



Au programme

Le problème et la culture DevOps

Pourquoi la CI/CD existe.

CI, livraison et déploiement CD

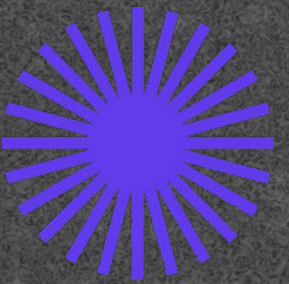
Trois notions à ne pas confondre.

Mesurer la valeur : DORA

Les 5 métriques de référence.

Branches et mise en place

Stratégie Git, projet GitLab, runner.





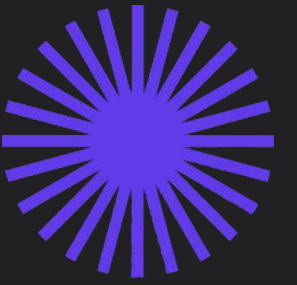
Objectifs de la séance

- Pourquoi la CI/CD existe et la valeur qu'elle apporte.
- Distinguer intégration continue, livraison continue et déploiement continu — trois notions souvent confondues.
- Lire les cinq métriques DORA.
- Choisir une stratégie de branche adaptée.
- Mettre en place concrètement votre fork GitLab dans le groupe du cours.

Mettre en place le projet fil rouge

App de fiches de films, codée séance après séance.
Trois conteneurs orchestrés par Docker Compose :
MongoDB stocke, une API Node expose, un front en
JavaScript affiche.

**Cette application sera le support de toutes vos pipelines
— et c'est elle qui constituera votre projet noté.**



Les problèmes à résoudre : dev au quotidien

« Ça marche chez moi »

Des différences d'environnement font tomber en production ce qui marchait en local.

Intégration tardive

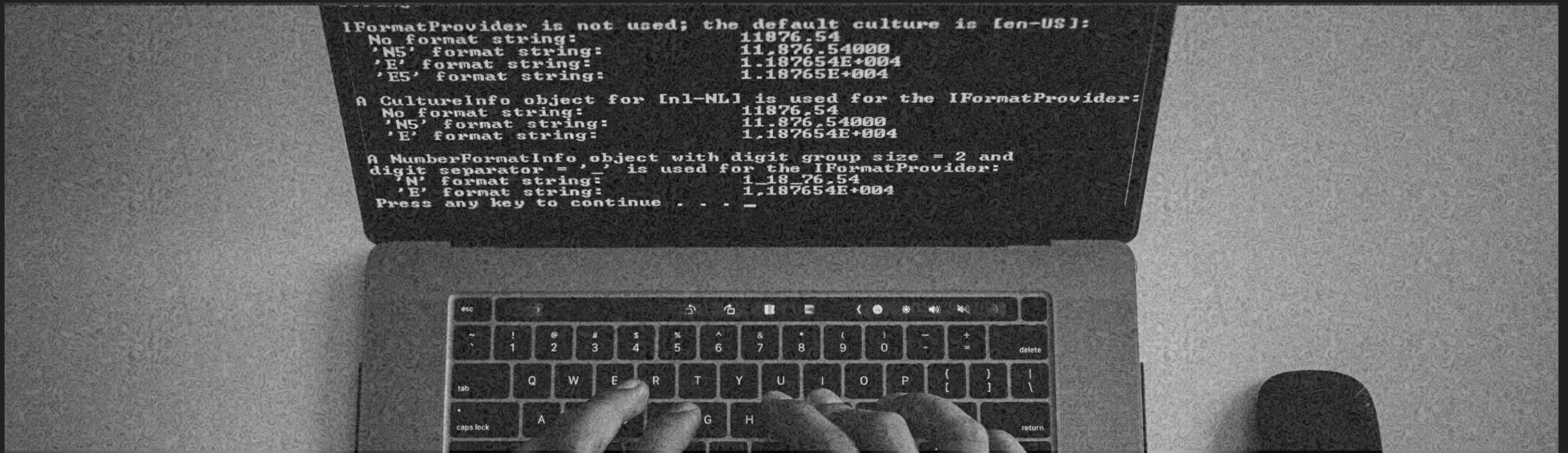
On fusionne tout en fin de sprint : conflits à répétition, le « merge hell ».

Déploiements manuels

Rares, stressants et sources d'erreurs — souvent lancés un vendredi.

Détection trop tardive

Les bugs remontent loin du commit qui les a introduits, donc plus durs à corriger.



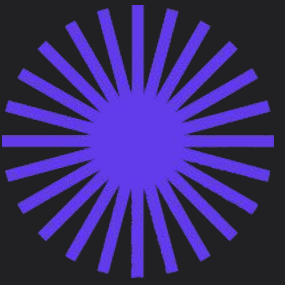
DevOps : une culture

Casser le mur entre ceux qui font évoluer (Dev) et ceux qui stabilisent (Ops).

Responsabilité partagée : « you build it, you run it ».

On automatise ce qui est répétitif et risqué, et on s'améliore par petites boucles rapides.

Les outils comme GitLab ne font que matérialiser cette culture : ils viennent après l'état d'esprit, jamais avant.



CI, livraison, déploiement

Intégration continue (CI)

Chaque commit est fusionné souvent et validé automatiquement : build, lint et tests en quelques minutes.

Livraison continue

Toute version qui passe la CI est prête à déployer ; le déclenchement vers la prod reste humain (un bouton).

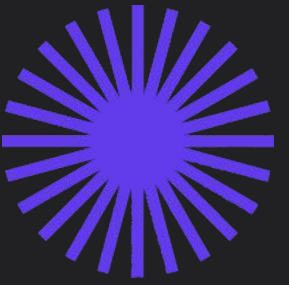
Déploiement continu

Chaque version validée part automatiquement en production, sans aucune intervention manuelle.

La nuance à retenir

« Prêt à déployer » n'est pas « déployé » : beaucoup d'équipes s'arrêtent à la livraison continue.

La chaîne commit → prod



1 · Tests

Build, lint et tests unitaires à chaque commit.

2 · Qualité et sécurité

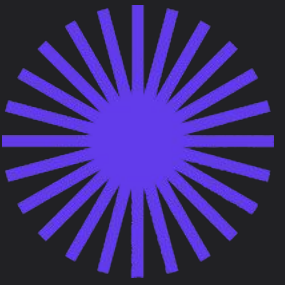
Couverture, quality gate et scan des vulnérabilités.

3 · Build et image

Un conteneur versionné, poussé sur le registre.

4 · Déploiement

Vers staging, puis production avec validation.



Les 5 métriques **DORA** (DevOps Research and Assessment)

Fréquence de déploiement

À quelle fréquence on livre réellement en production.

Délai de livraison

Le temps écoulé entre un commit et sa mise en production (lead time).

Taux d'échec des changements

La part des déploiements qui provoquent un incident en production.

Temps de rétablissement

Le temps pour rétablir le service après un déploiement en échec (anciennement MTTR).

Taux de reprise

La part des déploiements non planifiés, faits pour corriger un incident (rework rate).

Rapides ET stables

jamais l'un sans l'autre.

Stratégies de branche

Le choix qui conditionne toute la CI

Plus une branche vit longtemps, plus la fusion fait mal : **la CI n'a de sens que si on intègre souvent.**

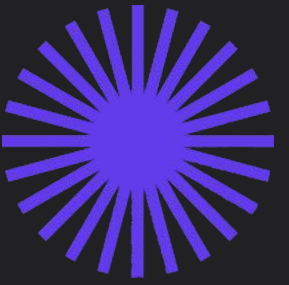
Trunk-based : des branches courtes fusionnées vite sur la branche principale
— la tendance 2026, idéale pour la CI/CD.

Gitflow : plus structuré
(develop, release, feature, hotfix),
utile pour des produits versionnés mais plus lourd.

Notre choix : branches courtes + Merge Requests,
avec la branche principale protégée
— impossible d'y fusionner du rouge.



Démo : on met en place l'atelier



1 · Projet GitLab

Rejoindre le groupe du cours, puis forker le projet de base.

2 · Lancer en local

Vérifier que l'appli tourne avec Docker Compose.

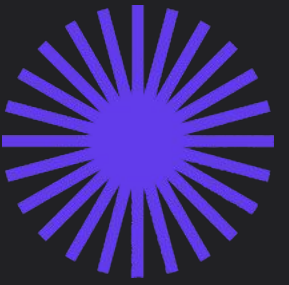
3 · Runner de groupe

Un runner déjà installé sur le VPS exécute vos jobs. Son installation est détaillée en séance 2.

4 · Premier vert

Un job de test s'exécute et passe : l'atelier est prêt.

Un fork par élève, dans le groupe



1 · Projet de référence

Dans le groupe du cours : le setup de base.

2 · Votre fork

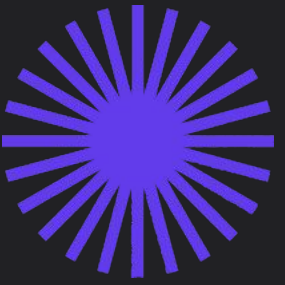
Votre copie : vos variables, votre main protégée.

3 · Branche et MR

Vous travaillez sur une branche ; la MR vise votre main.

4 · Runner de groupe

Le même runner exécute les pipelines de tous les forks.



Pourquoi un fork ?

Variables isolées

Votre clé SSH n'est visible que dans votre projet.

Runner partagé

Un runner de groupe sert tous les forks.

main protégée

Vous la protégez vous-même : MR obligatoire, pipeline verte.

Le piège

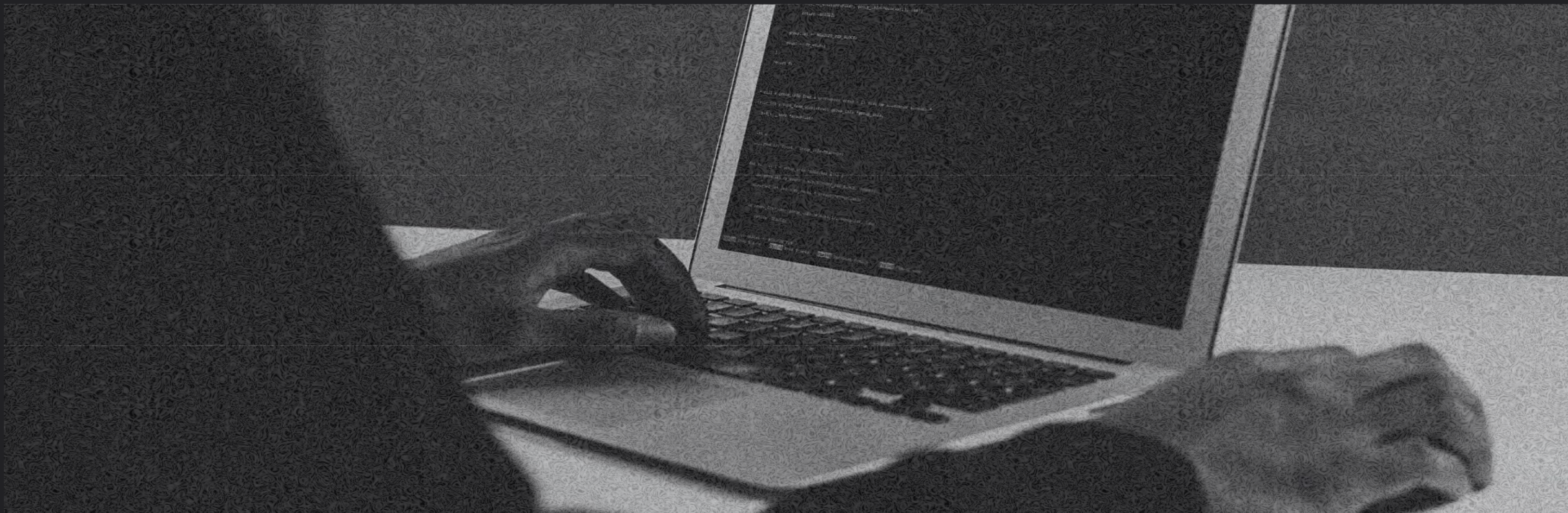
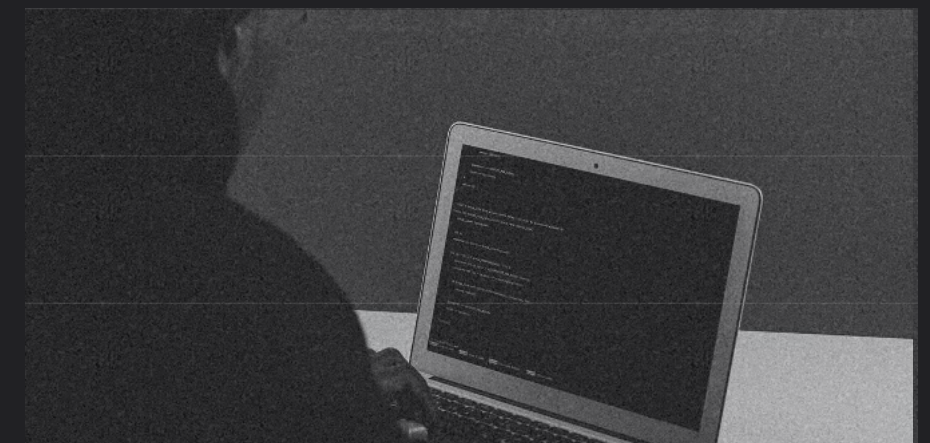
Un secret commité reste dans l'historique, même dans un fork privé.

Récap séance 1

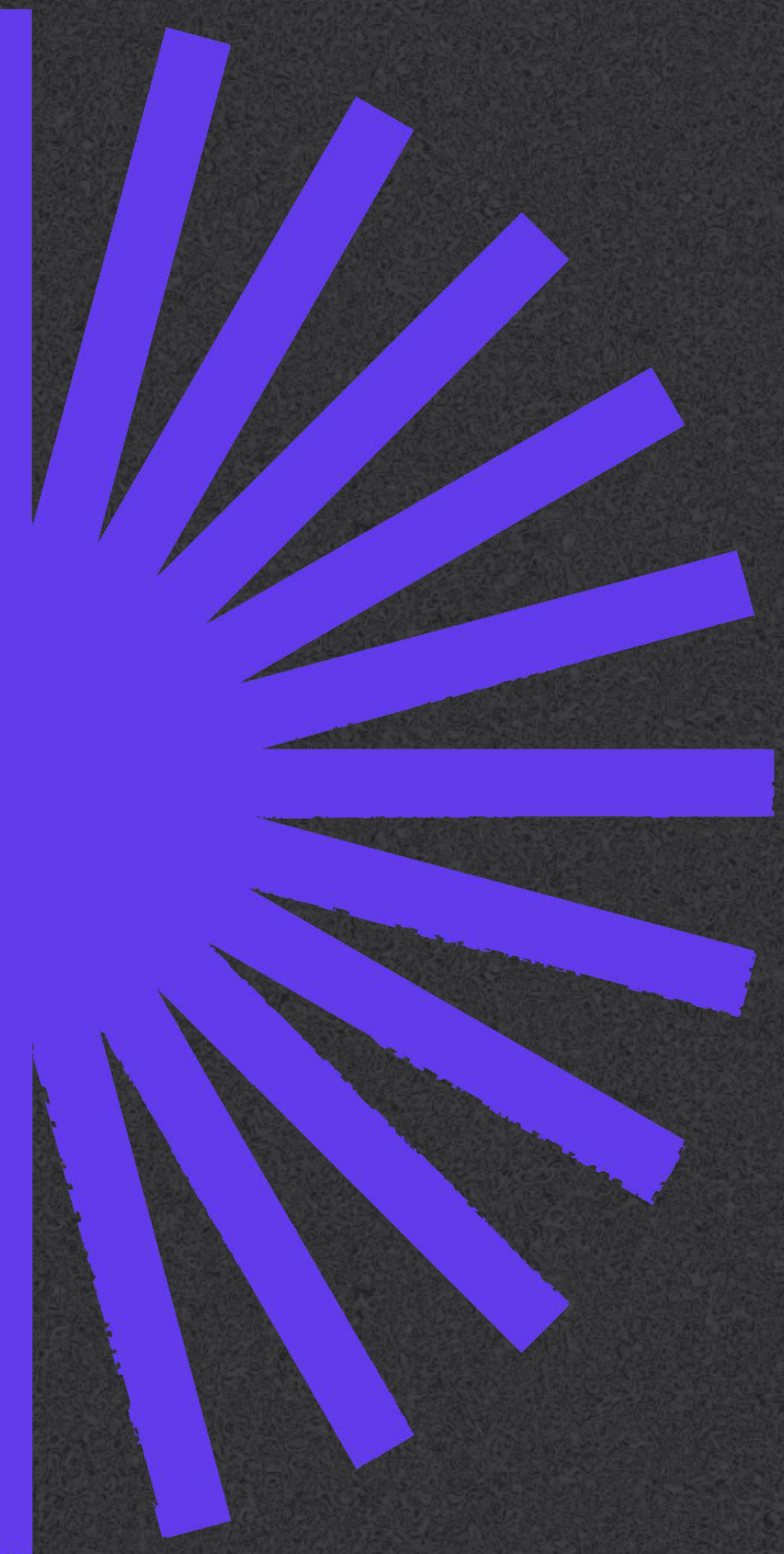
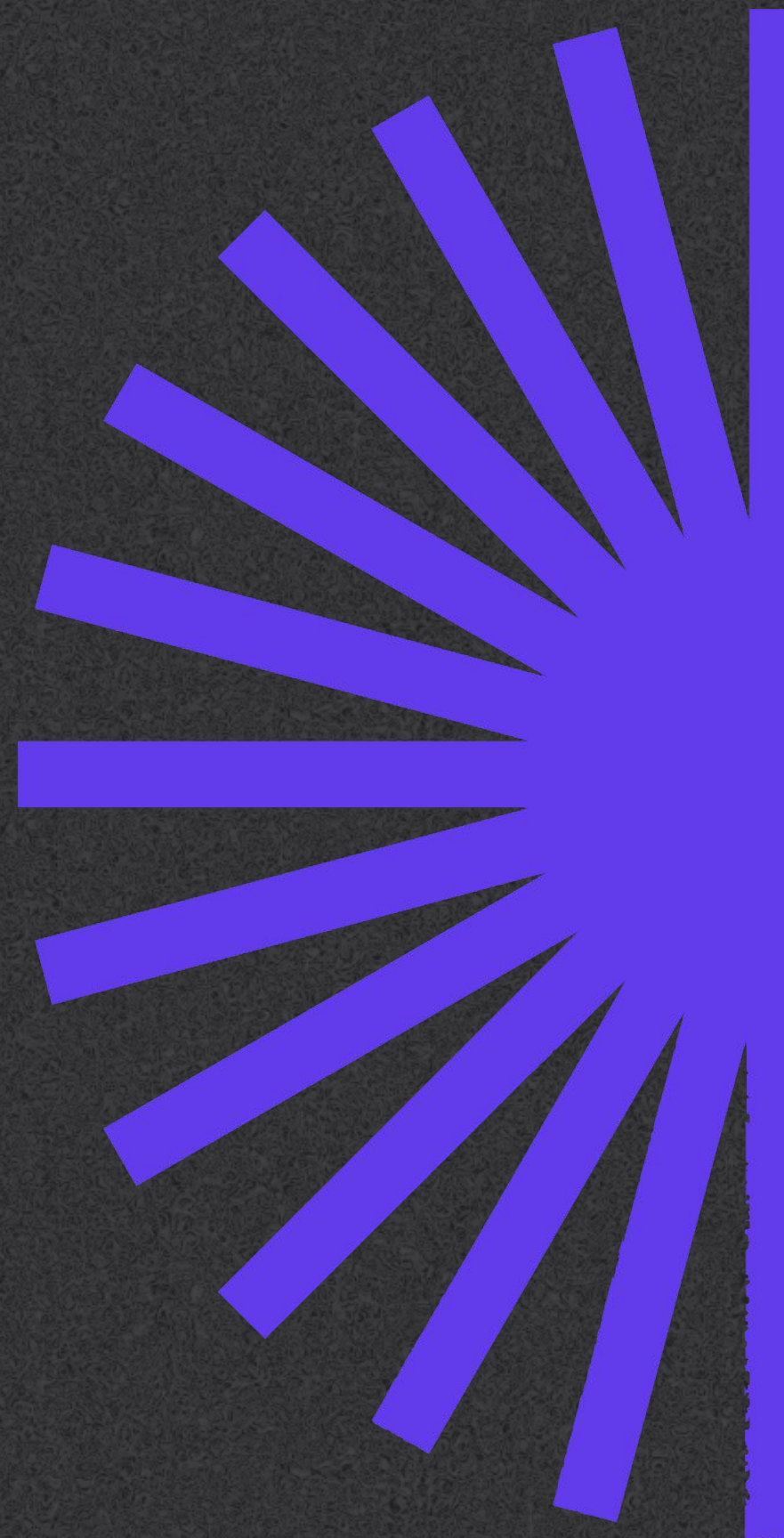
Ce qui doit être acquis :

pourquoi la CI/CD,
la différence entre intégration, livraison et déploiement continus,
les cinq métriques DORA ,
les stratégies de branche.

Prêt pour la suite : votre fork GitLab dans le groupe du cours.



Séance 2



Au programme

Anatomie d'une pipeline

Pipeline, stages, jobs, runner.

Premier job

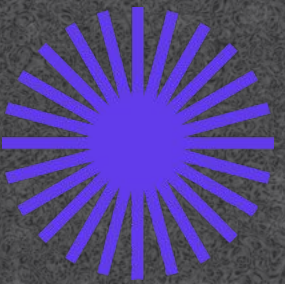
Build + lint.

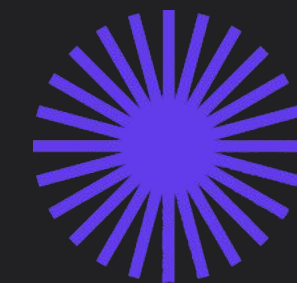
Pipelines de Merge Request

Tester avant de fusionner.

Cache, artefacts & atelier

Optimiser et livrer.





Anatomie d'une pipeline

Pipeline

Tout ce qui se déclenche sur un commit.

Stage

Une étape (build, test...) ; les stages s'enchaînent.

Job

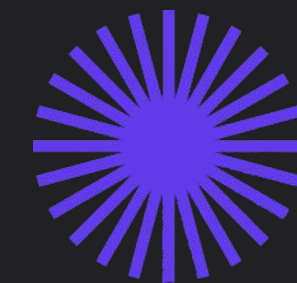
Une tâche ; les jobs d'un stage tournent en parallèle.

Runner

La machine qui exécute les jobs.

```
 .gitlab-ci.yml
1  stages:
2    - test
3    - build
4    - deploy
5
```

 Passed ⌚ 00:01:44 📅 4 days ago	#2640586807 try base64 🔗 main ↻ 7d63976d  branch		
 Failed ⌚ 00:01:37 📅 4 days ago	#2640580574 test 🔗 main ↻ 4a712028  branch		



Le runner en bref

Son rôle

Il interroge GitLab : « un job pour moi ? »

La portée

Un runner de groupe sert tous les forks du groupe.

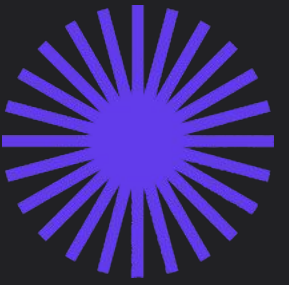
L'executor

Docker : chaque job tourne dans une image jetable.

Le piège

Un runner exécute du code : sa sécurité est celle du serveur.

Installer le runner sur le VPS



1 · Créer le runner

Dans le groupe, menu Runners : créer un runner de groupe et copier son jeton.

2 · Installer

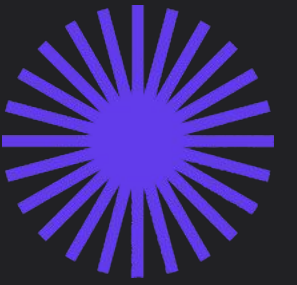
Le paquet gitlab-runner, depuis le dépôt officiel.

3 · Enregistrer

gitlab-runner register : URL, jeton, executor docker.

4 · Vérifier

Le runner passe « en ligne » dans GitLab, un job s'exécute.



Un runner exécute du code

Isolation

Docker rootless dédié au runner, séparé du reste du serveur.

Secrets

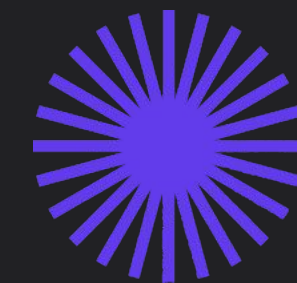
Variables masquées ; ne jamais les afficher dans un job.

Concurrence

4 jobs à la fois (4 vCPU) : les autres patientent.

Entretien

Purger régulièrement images et caches.



Cache vs artefacts

Cache

Accélère (ex. `node_modules`) — non garanti.

La règle

Cache = optimisation ; artefacts = livrables.

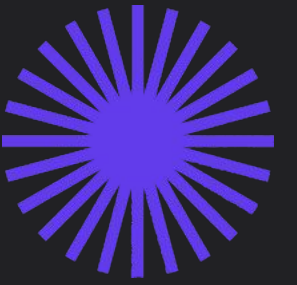
Artefacts

Résultats conservés ou passés au job suivant.

Le piège

Mauvais usage = pipelines lentes ou cassées.

Un **job**, de quoi est-il fait ?



stage

À quelle étape il appartient.

image

L'environnement d'exécution (ex. node:24).

script

Les commandes à lancer.

rules

Quand déclencher le stage.

```
deploy:production:
  stage: deploy
  image: debian:bookworm-slim
  before_script:
    - apt-get update -qq && apt-get inst
    - mkdir -p ~/.ssh && chmod 700 ~/.ss
    - echo "$SSH_PRIVATE_KEY" | base64 -
    - chmod 600 ~/.ssh/deploy_key
    - echo "$SSH_KNOWN_HOSTS" > ~/.ssh/k
    - chmod 644 ~/.ssh/known_hosts
  script:
    - rsync -avz --delete -e "ssh -i ~/.
      var/www/html/cd/moviesjs
  environment:
    name: production
    url: http://cd.kptn.fr
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
```

Le fichier .gitlab-ci.yml

Au cœur du projet

Il vit à la **racine** du repo et décrit **stages** et **jobs** en YAML.

La **pipeline** est du code : revue, historique, retour arrière possible.

```
> api
> coverage
> dist
> docker
> node_modules
> public
> src
> test-results
> tests
🐳 .dockerignore
⚙️ .env
🔗 .gitignore
🐱 .gitlab-ci.yml
⚙️ apache.conf
🐳 docker-compose.yml
JS fetchMovies.js
<> index.html
{} package-lock.json
{} package.json
JS playwright.config.js
📖 README.md
JS vitest.config.js
```

```
🐱 .gitlab-ci.yml
1  stages:
2    - test
3    - build
4    - deploy
5
6  # — Stage 1 : Tests unitaires —
7  test:
8    stage: test
9    image: node:20
10 > cache: ...
14 > before_script: ...
16 > script: ...
18
19 # — Stage 2 : Build Vite —
20 build:
21   stage: build
22   image: node:20-alpine
23 > cache: ...
27 > before_script: ...
29 > script: ...
31 > artifacts: ...
35
36 # — Stage 3 : Déploiement sur le VPS —
37 deploy:production:
38   stage: deploy
39   image: debian:bookworm-slim
40 > before_script: ...
47 > script: ...
49 > environment: ...
52 > rules: ...
54
```

Premier **job** : le **lint**

Le style et les erreurs ●

Évaluation **locale** avant d'autoriser un push sur le dépôt distant.

ESLint sur l'API et le front, à chaque push. ●

Démo : pousser une faute volontaire, voir le rouge, corriger, repasser au vert.

Stages & ordre d'exécution

1 · build

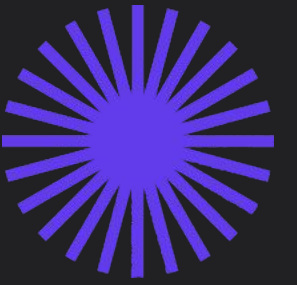
Install et compilation.

2 · test qualité

Lint maintenant puis les tests classiques en séance 3.

Tester avant de fusionner





Pipelines de Merge Request

Pipeline de branche

Lancé par un push, même sans MR ouverte.

Pipeline de MR

Lancé par la MR : c'est lui qui conditionne la fusion.

Un seul pipeline par push

```
workflow:
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    - if: $CI_COMMIT_BRANCH && $CI_OPEN_MERGE_REQUESTS
      when: never
    - if: $CI_COMMIT_BRANCH
```

Atelier : votre premier `.gitlab-ci.yml`

1 · Créer le fichier

À la racine du repo.

2 · Stage build

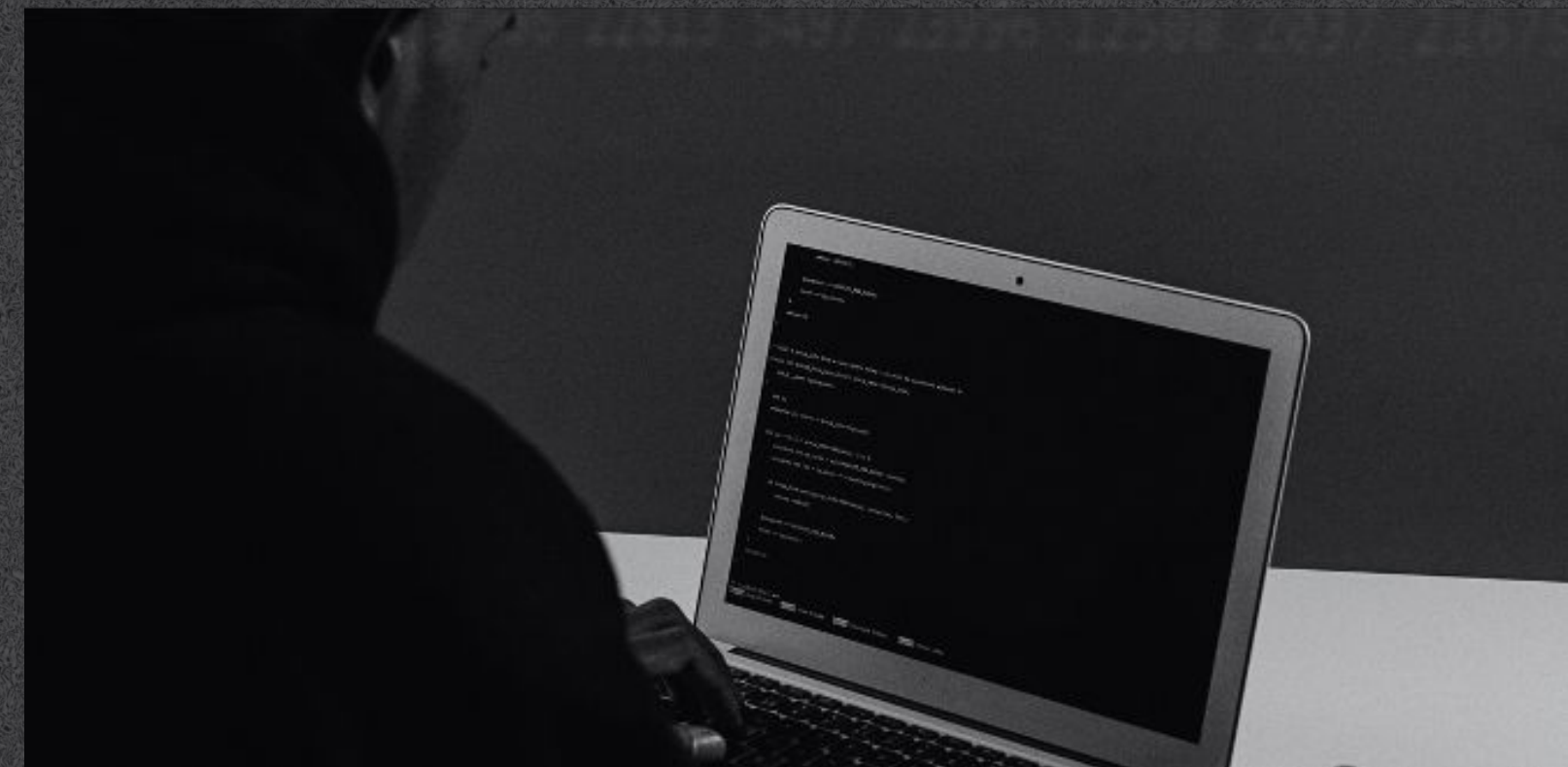
npm ci.

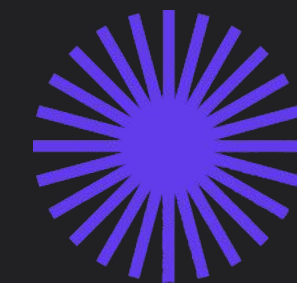
3 · Stage test

Le lint.

4 · Ouvrir une MR

Pousser et observer la pipeline.





Pièges fréquents

Indentation YAML

La cause n°1 d'échec.

Cache inutile

Mauvaise clé de cache.

Job qui ne part pas

rules mal posées, ou mélangées avec only.

Tout dans un job

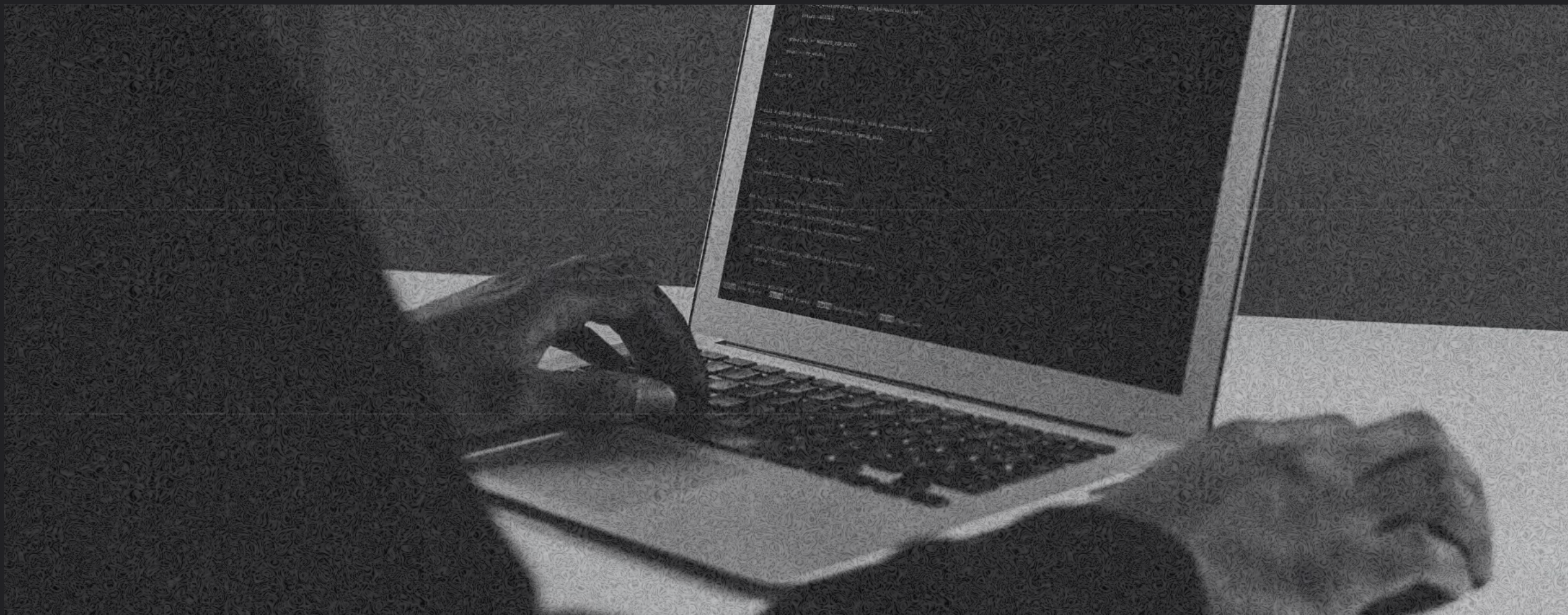
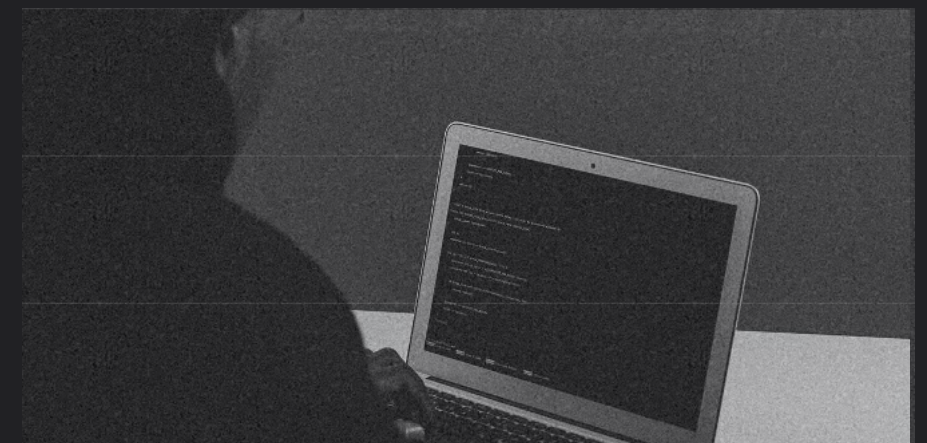
Illisible et non parallélisable.

Récap & séance 3

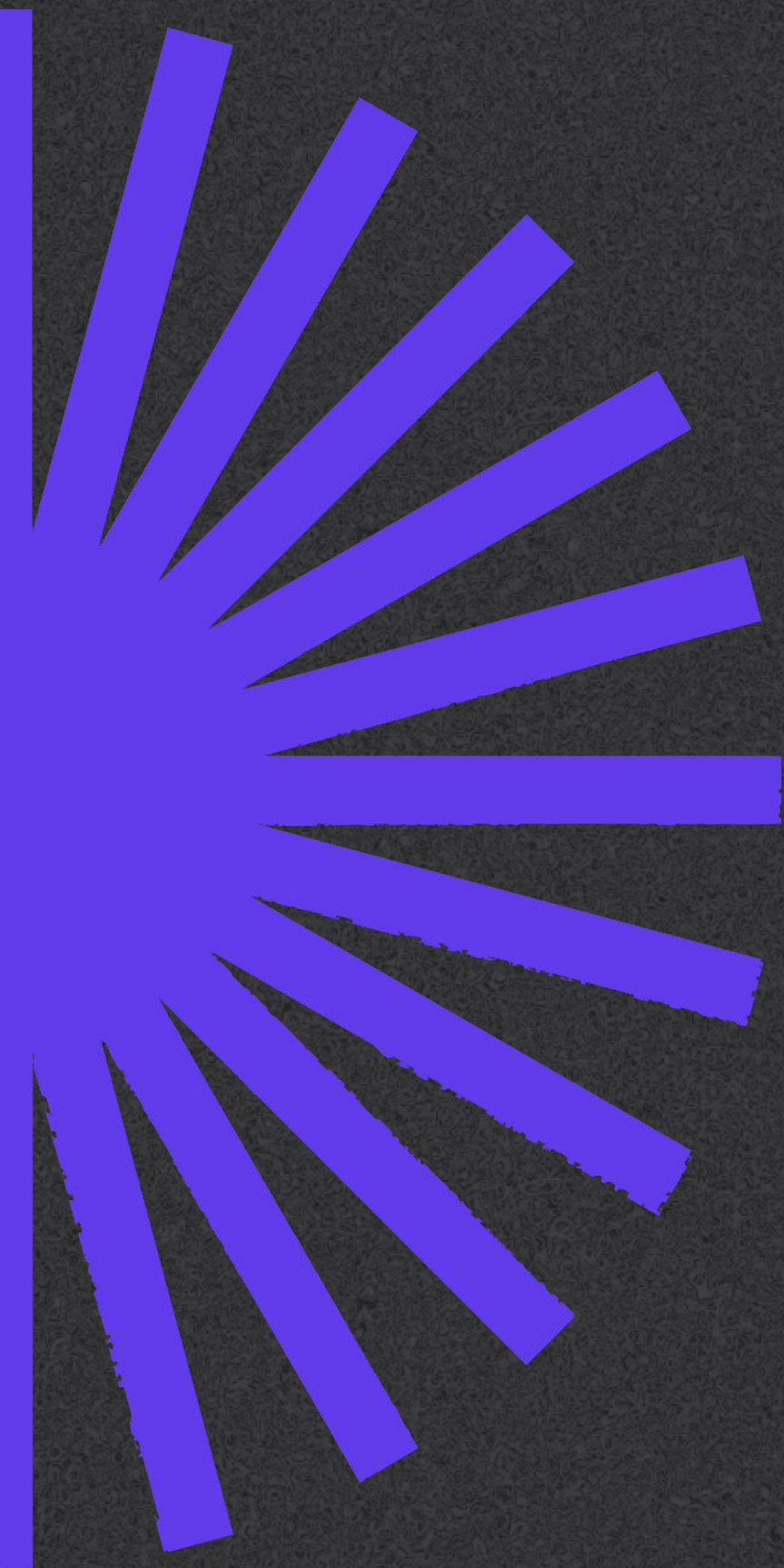
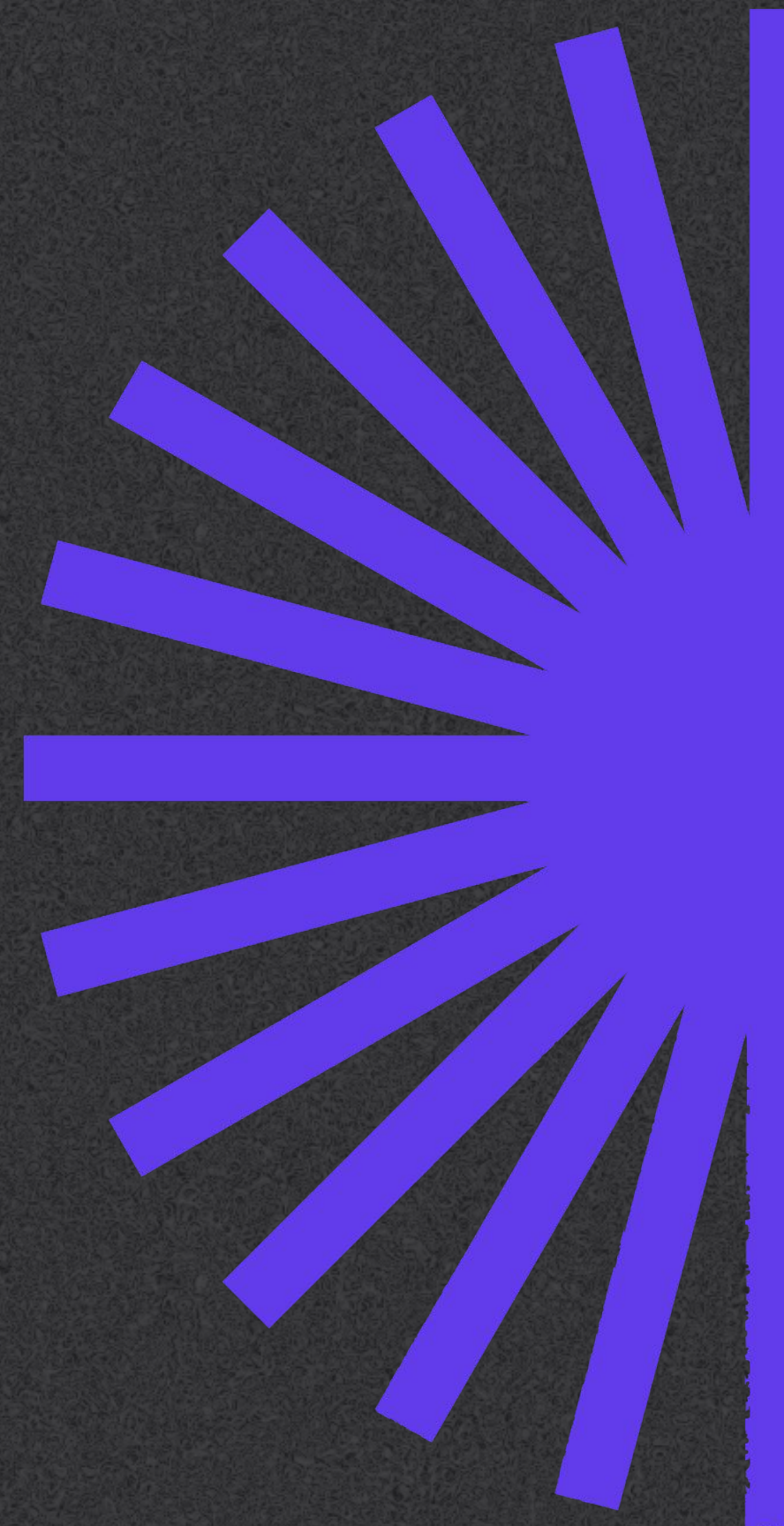
Ce qu'on tient comme acquis :

anatomie d'une pipeline, premiers jobs, pipelines de Merge Request.

Devoir : faire passer votre pipeline au vert sur votre projet.



Séance 3



Au programme

La pyramide de tests

Unitaires, intégration, e2e.

Tests en CI

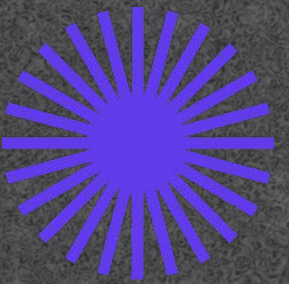
Sur l'API et la base.

Couverture & quality gate

Mesurer et bloquer.

Branches protégées

La qualité non négociable.



La pyramide de tests



1 · Unitaires et Fonctionnels

Nombreux et rapides.

Orienté fonctionnalités métier : MVP.

2 · Intégration

Vérifient les connexions (API ↔ Mongo).

Validation de l'archi multi services.

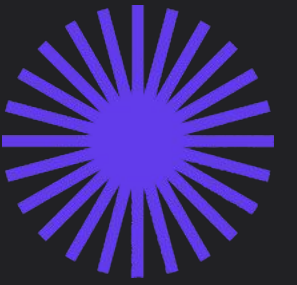
3 · End-to-end

Peu nombreux, lents, proches de l'utilisateur.

Validation de la disponibilité, du parcours UX

Coût

Croît à mesure qu'on monte en charge.



Tests **unitaires** de l'API

Tester la logique métier sans dépendances, avec node:test ou Vitest.
Valider une fonctionnalité indispensable

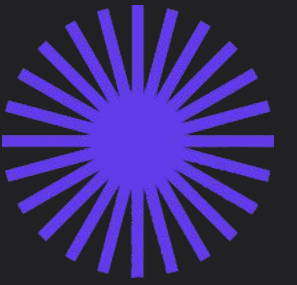
Démo : un test simple sur un endpoint, du rouge au vert.
validation d'une contrainte, vérification des erreurs.

Tests **d'intégration** : API ↔ Mongo

On vérifie les vraies connexions

Lancer un service mongo dans le job CI et vérifier que l'API lit et écrit vraiment.

Base jetable, données de seed minimales.



La **couverture** de code, Pourquoi **tester en CI** ?

Un test qui ne tourne pas automatiquement finit par ne plus tourner du tout.

La CI transforme les tests en garde-fou permanent, pas en corvée optionnelle qu'on oublie sous la pression.

Ce que ça mesure

Les lignes exécutées par les tests.

Ce que ça ne dit pas

Si les tests sont pertinents.

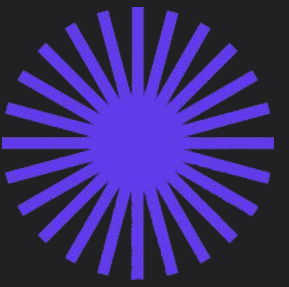
Utile

Repérer le code jamais testé.

Piège

Viser 100 % à tout prix.

Pièges fréquents



Tests flaky

Passent ou échouent au hasard

- mal conçus
 - dépendant d'une date ou de données
 - sensibles à la performance ou trop lents
 - dépendants de paramètres extérieurs
 - utilisation en parallèle du même env
-

BDD non isolée

Les tests se polluent.

Couverture trompeuse

Code couvert mais rien d'asserté.

Code couvert via une fonction annexe.

Tests trop lents

La pipeline devient pénible.

La capacité du runner peut saturer

Branches protégées

Verrouiller main : Merge Request obligatoire et pipeline verte requise.

Atelier : ajouter les tests

1 · Unitaires

Job de tests unitaires.

2 · Intégration

Avec un service mongo.

3 · Couverture

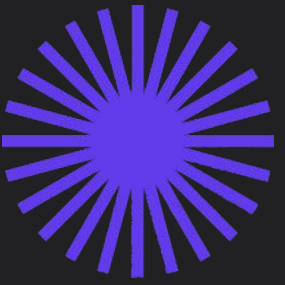
Seuil bloquant.

4 · Protéger main

Tester sur une MR.



résumé des actions à réaliser



Réorganiser les tests en tests/unit, tests/integration, tests/e2e

Refactorer db.test.js pour utiliser un vrai Mongo via MONGO_URL au lieu de mongodb-memory-server

Exposer le port Mongo dans docker-compose.yml pour les tests locaux

Ajouter la couverture de code (vitest coverage) avec seuil bloquant

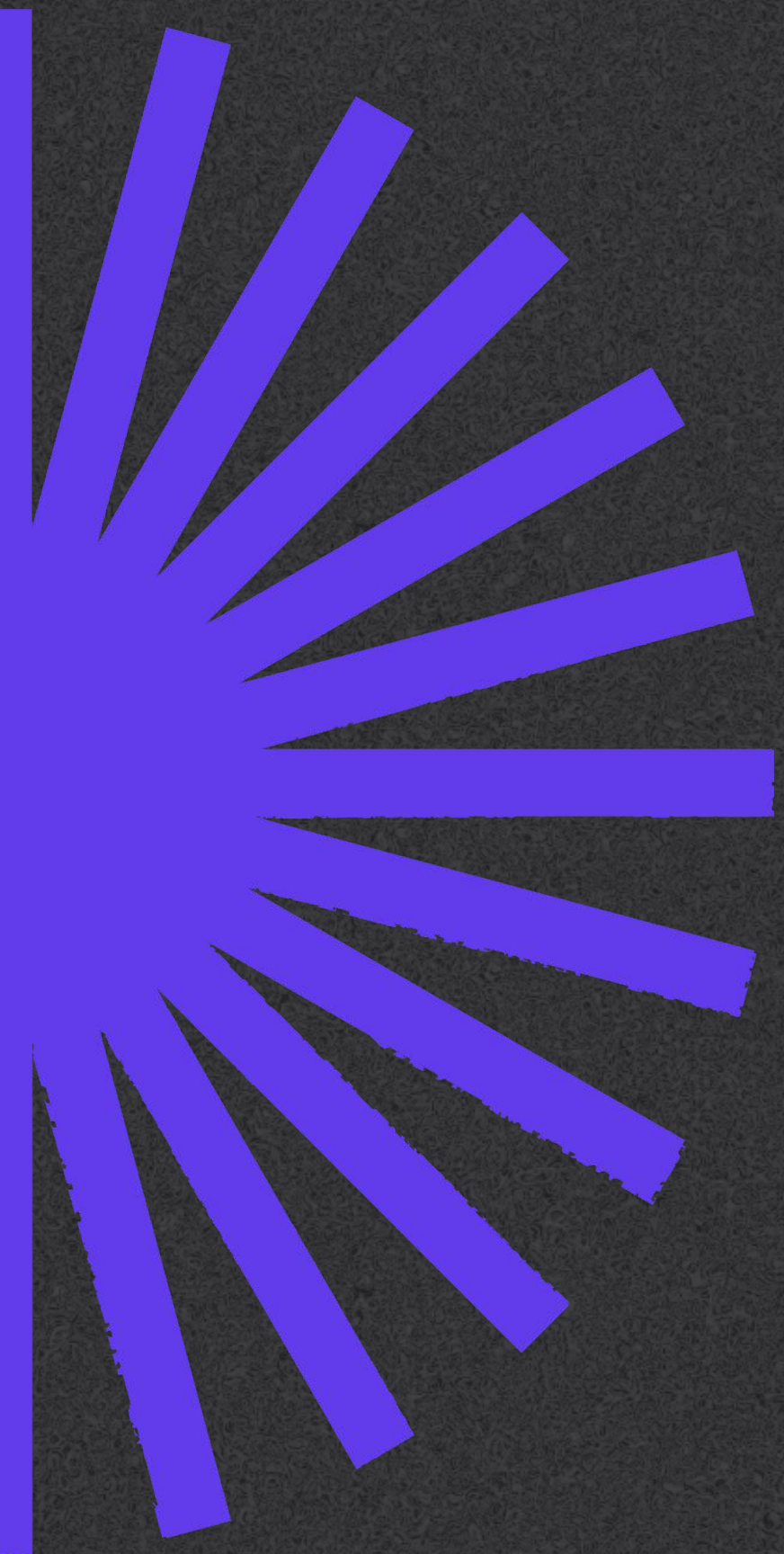
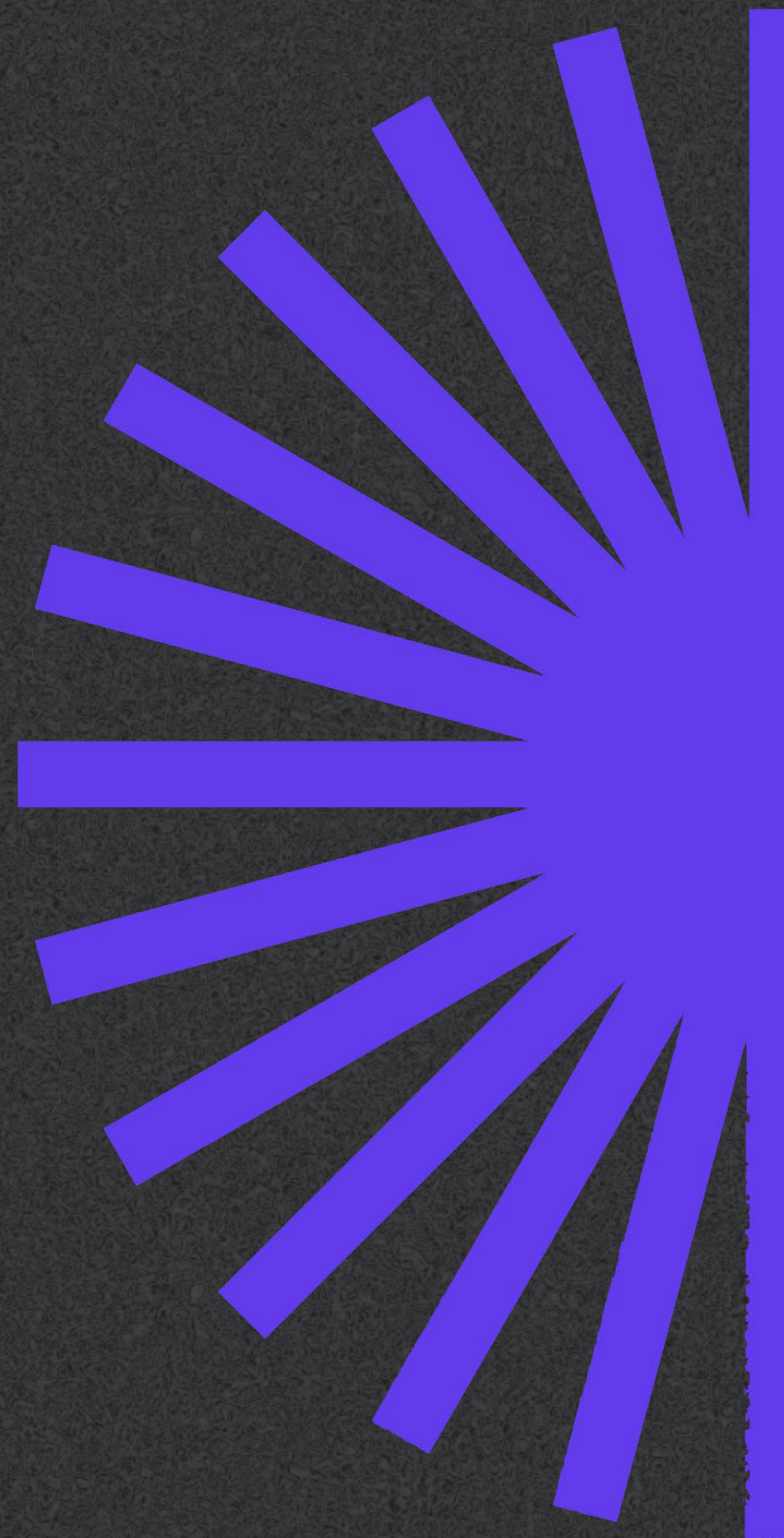
Ajouter les scripts npm test:unit / test:integration / test:coverage

Ajouter les jobs CI unit-tests / integration-tests (service mongo:7) / coverage

Vérifier que tout fonctionne (local + simulation image CI)

Signaler la protection de branche main (réglage GitLab, hors repo)

Séance 4



Au programme

Builder l'image en CI

Automatiser ce que vous savez faire.

Le Container Registry

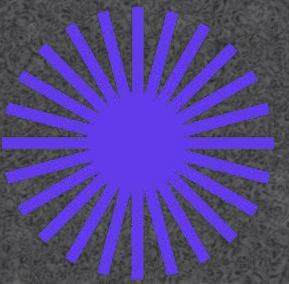
Intégré à GitLab.

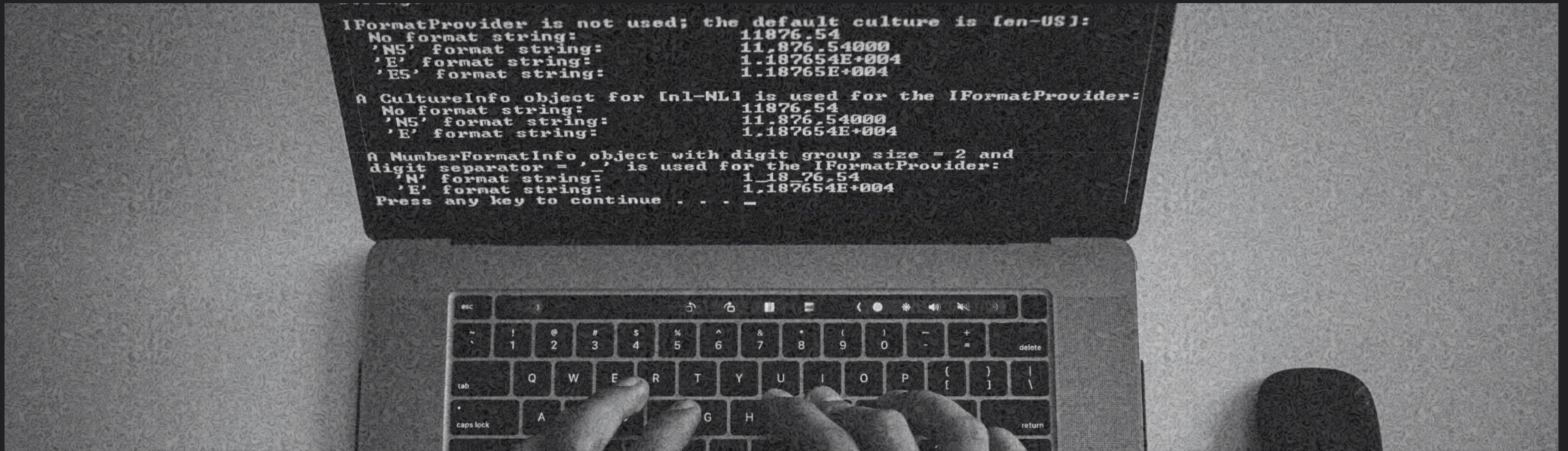
Stratégie de tags

SHA, SemVer, latest.

Atelier

Build + push.





Du code à l'image

Vous savez déjà faire une image Docker — ici, on l'automatise et on la versionne.

Un job build qui construit l'image, pour l'API et pour le front.

L'image construite est stockée dans le [registry GitLab](#)

Le GitLab **Container Registry**

Tout intégré dans un seul

Un registre intégré au projet, gratuit, avec ses variables [CI_REGISTRY](#) fournies.

Démo : pousser une image et la retrouver dans le registre.

Stratégie de tags

SHA court

Traçabilité exacte du commit.

SemVer

Versions lisibles (v1.2.0).

latest

Pratique... mais ambigu en production.

La règle

On déploie un tag immuable, jamais latest.



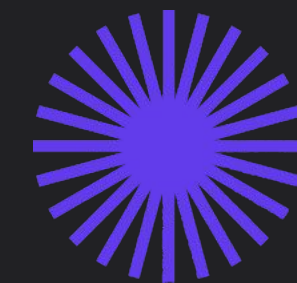


Image mince & reproductible

Multi-stage

Build lourd, runtime léger.

.dockerignore

Ne pas embarquer le superflu.

Base épinglée

Version d'image de base figée.

Moins = mieux

Surface d'attaque réduite, builds rapides.

S'authentifier au registre

Sans secret en dur

Connexion via les **variables CI du projet**, avec un jeton à **courte durée** de vie.

Aucun identifiant écrit dans le repo.



Atelier : build + push

1 · API

Job de build de l'image API.

2 · Front

Job de build du front.

3 · Tags

SHA court + SemVer.

4 · Push

Publier et vérifier.

Pièges fréquents

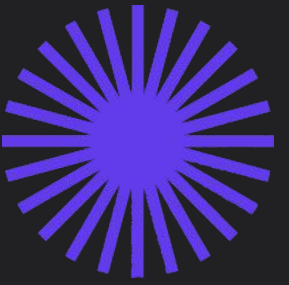


Image énorme

Pas de multi-stage.

Secret dans l'image

Clé copiée par erreur.

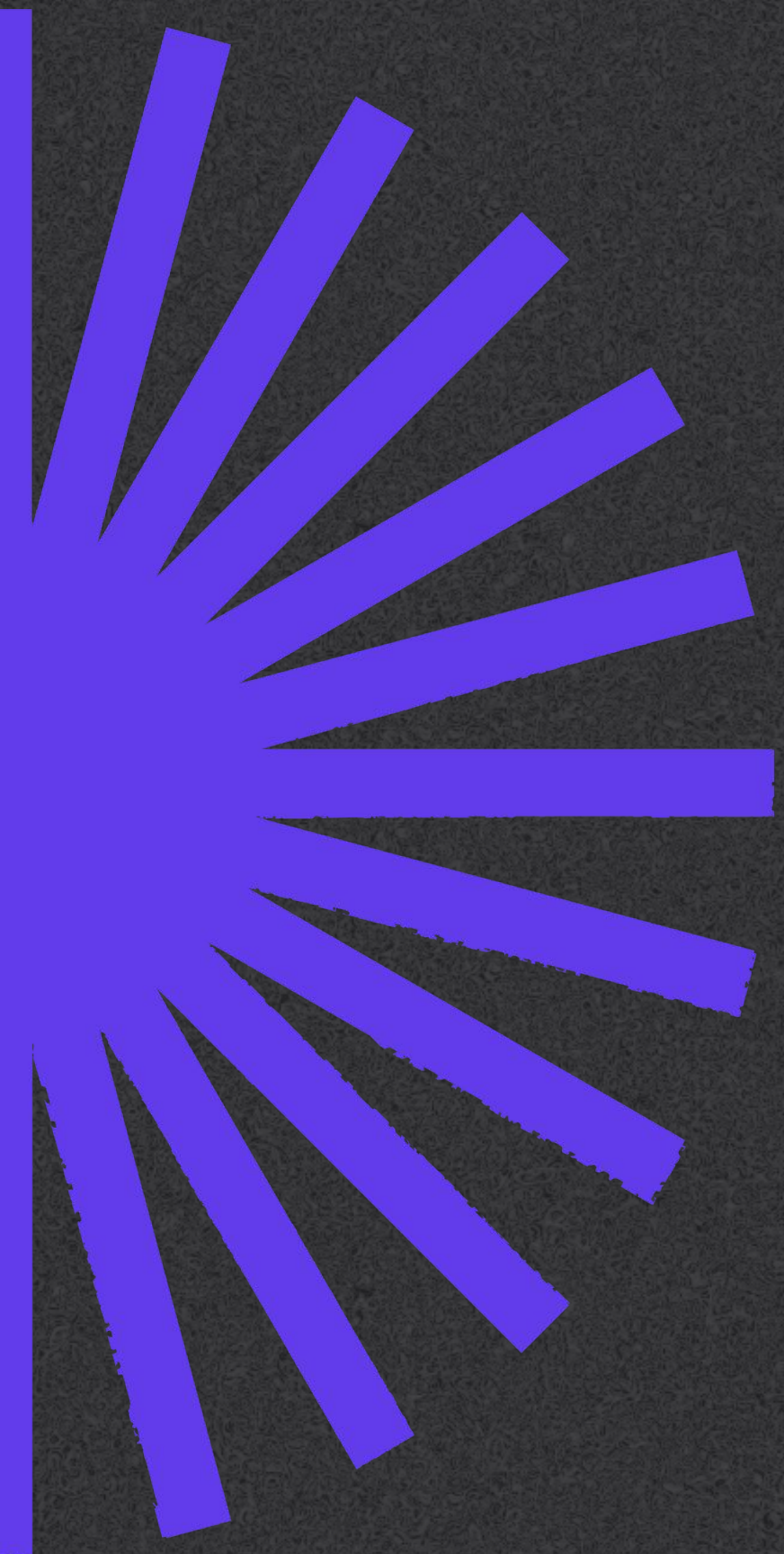
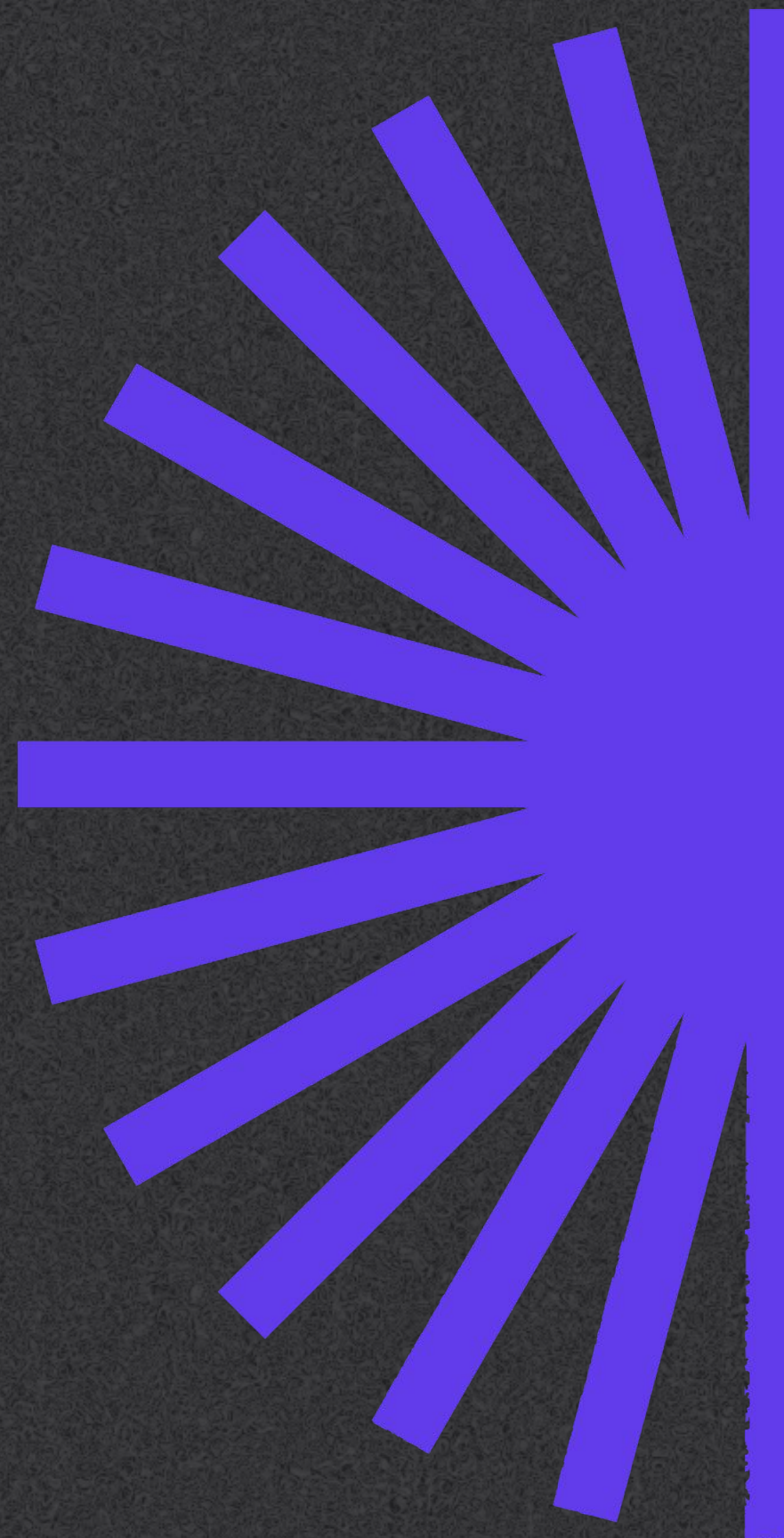
latest en prod

Impossible de savoir ce qui tourne.

Pas de cache

Pipelines lentes.

Séance 5



Au programme

Le cas TMDB

Une clé exposée.

Gérer les secrets

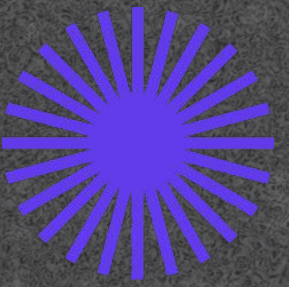
Masquage, OIDC.

Les scans

Code, dépendances, image, secrets.

SBOM & signature

Chaîne d'approvisionnement.



Le front appelle TMDB en direct :
la clé est visible dans le navigateur.

Le client n'est jamais sûr
Tout ce qui part au navigateur est public :
une clé côté client est une clé fuitée.

Démo : ouvrir les DevTools et voir la clé.

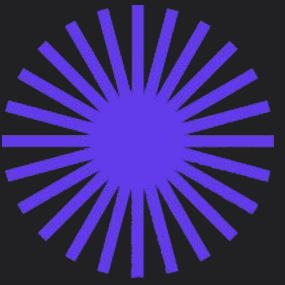
Le correctif : un **proxy**

La clé reste **côté serveur**

Le front appelle notre API, qui appelle TMDB et
garde la clé côté serveur.

La clé devient une **variable d'environnement**
injectée par la CI, jamais committée.





Gérer les **secrets** en CI

Variables masquées

Cachées dans les logs.

Jamais committées

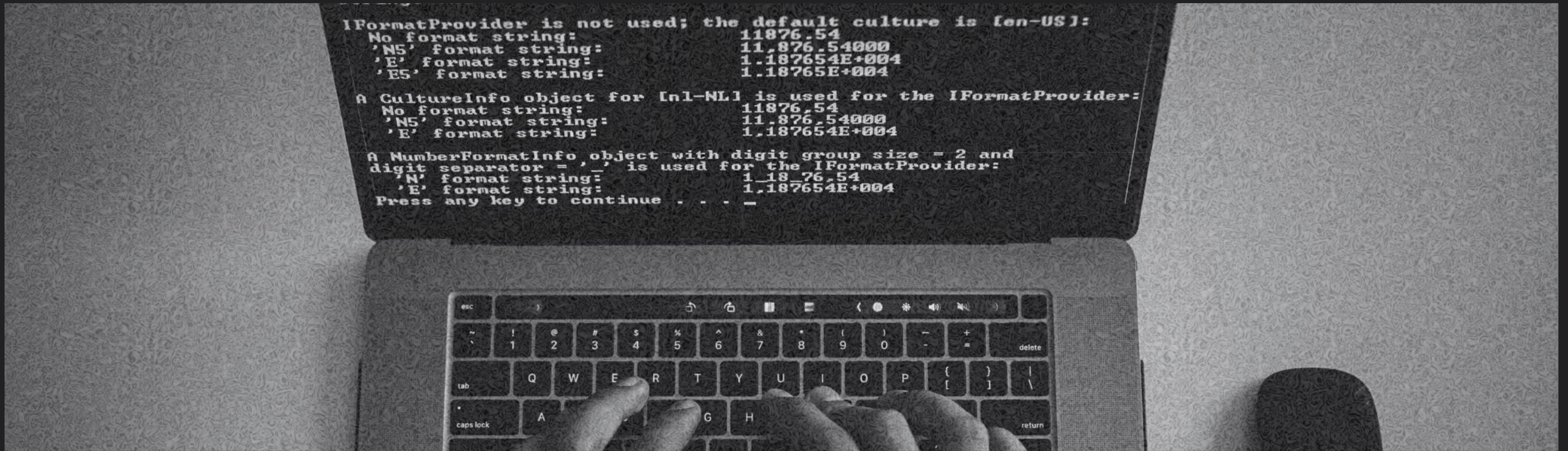
Ni en clair, ni en base64.

Variables protégées

Réservées aux branches/env protégés.

Rotation & Maintenance

Révoquer sans toucher au code.

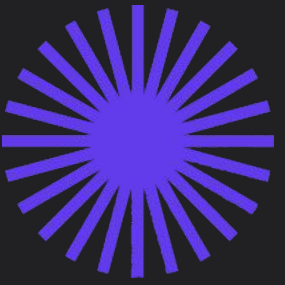


OIDC OpenID Connect : des identifiants éphémères

Le meilleur secret est celui qu'on ne stocke pas : OIDC fournit un jeton temporaire au moment du job.

Notion clé en 2026 : sert à signer (cosign) et à accéder au registre ou au cloud sans secret long terme.

<https://openid.net/developers/how-connect-works/>



Les types de scan

SAST / Static Application Security Testing

Analyse le code source.

Secrets

Détecte les secrets committés.

SCA / Software Composition Analysis

analyse de dépendances

Faibles des bibliothèques.

Image

Vulnérabilités du conteneur.

SAST & secrets

Code et historique

[SemGrep](#) repère les motifs dangereux dans le code JS.

[gitleaks](#) détecte les secrets dans l'historique git.

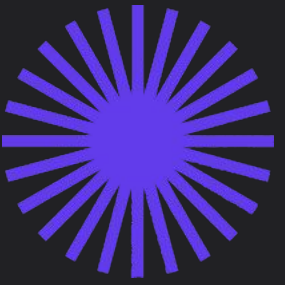
Scan des dépendances & de l'image

[Trivy](#) à la manœuvre

Trivy scanne à la fois les dépendances et l'image conteneur.

[Jobs bloquants](#) au-delà d'un seuil de gravité.





Chaîne d'approvisionnement

SBOM (syft)

La liste des ingrédients de l'image.

Signature (cosign)

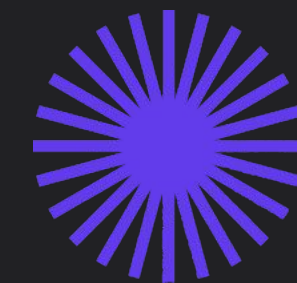
Prouver l'origine de l'image.

Provenance

Qui, quoi, quand a produit l'artefact.

Pourquoi

Se prémunir des attaques de chaîne logicielle.



Pièges fréquents

Secret en clair

Dans un fichier de conf.

Faux positifs

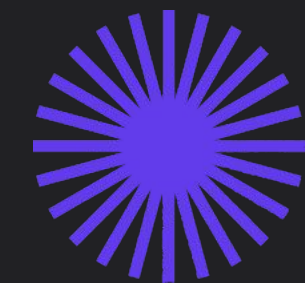
Tout désactiver par dépit.

Scan non bloquant

Les alertes sont ignorées.

Base obsolète

Failles connues non corrigées.



Atelier : sécuriser la pipeline

1 · Proxy TMDb

Déplacer la clé côté API.

2 · Secrets

gitleaks + Semgrep.

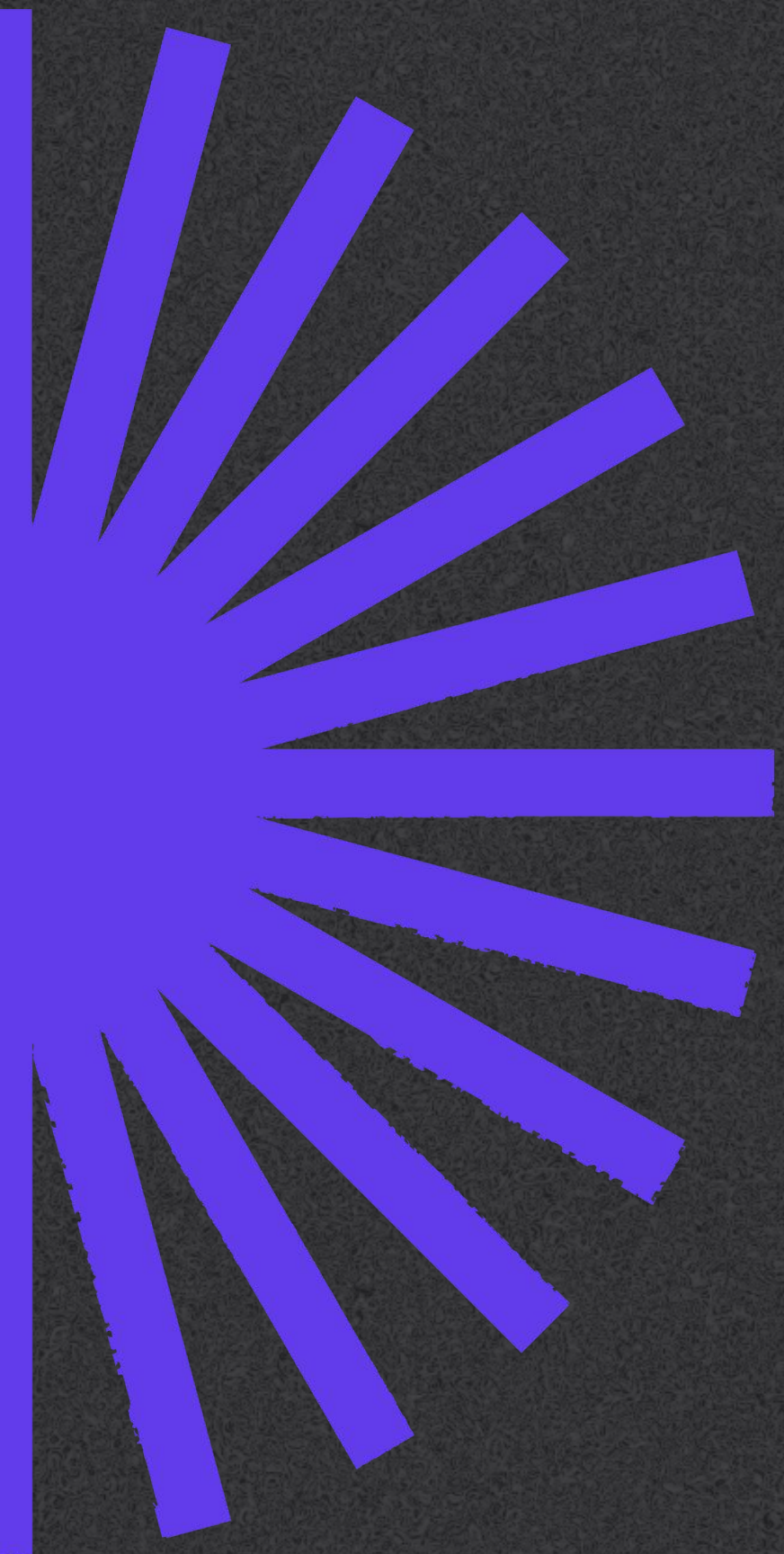
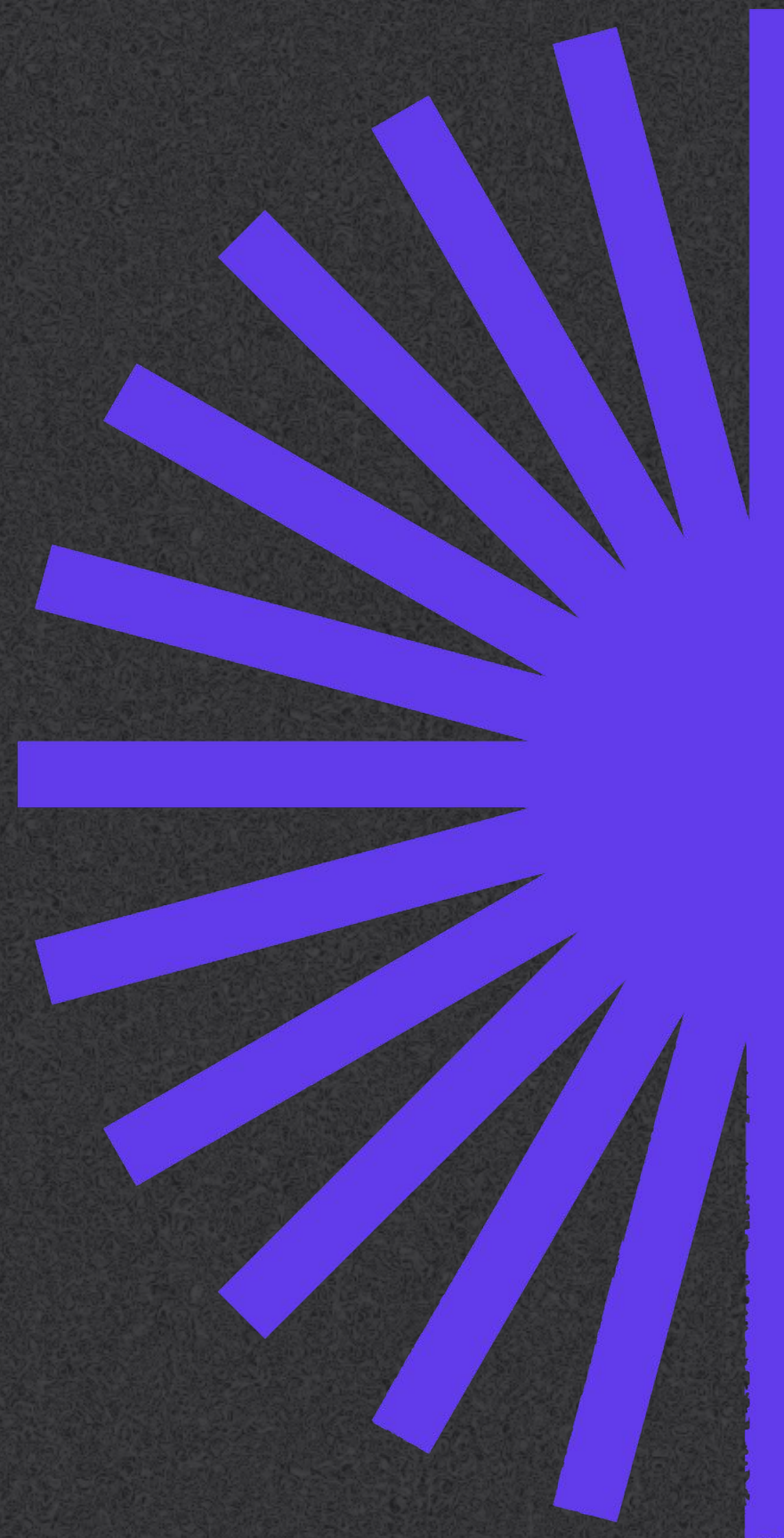
3 · Trivy

Dépendances et image, bloquant.

4 · Bonus

SBOM + signature.

Séance 6



Au programme

Ce qui est transférable

Les concepts d'abord.

Vocabulaire & syntaxe

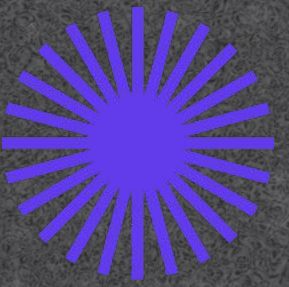
Comparés côte à côte.

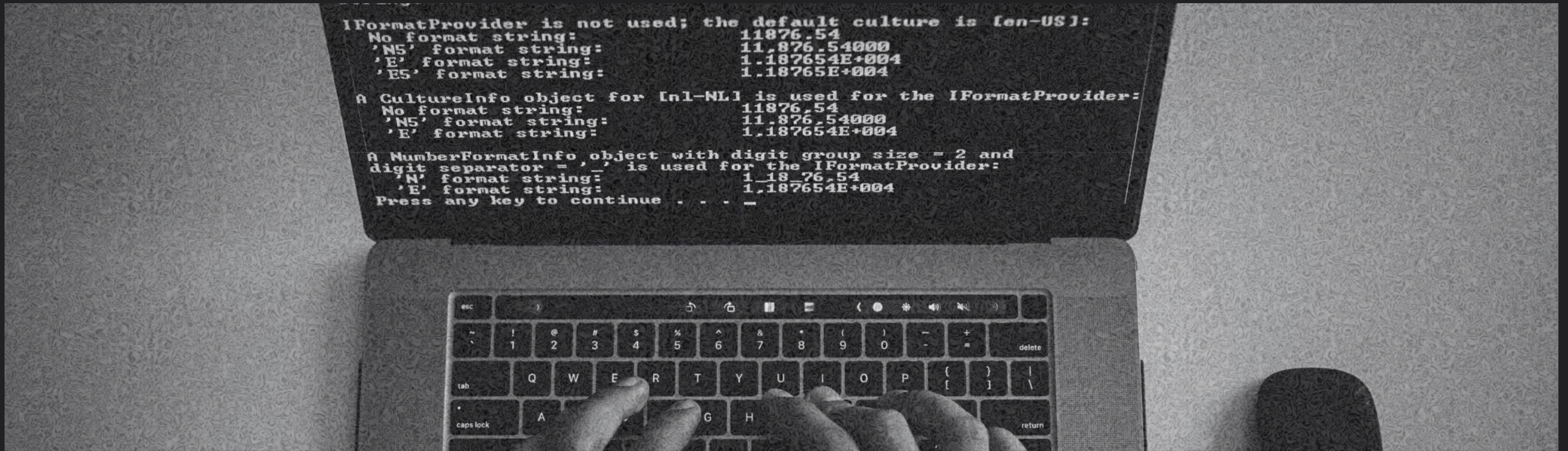
Runners, secrets, écosystème

Les différences qui comptent.

Quand choisir quoi

Un choix à argumenter.





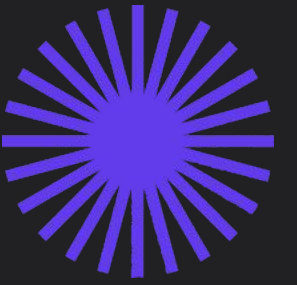
Les concepts sont transférables

Pipeline, stages, jobs, runners, secrets : ce vocabulaire existe partout. Seule la syntaxe change.

Apprendre un outil de CI/CD, c'est apprendre les concepts, pas un fichier YAML particulier.

Dans le même esprit, la démarche DevOps doit être généralisable à tout projet informatique.

Vocabulaire comparé Gitlab et GitHub



Deux syntaxes, une idée

Le même job de lint, le même job écrit en `.gitlab-ci.yml` et en workflow GitHub, côte à côte.

Les différences sautent aux yeux ; la logique, elle, est identique.

GitLab

stages, puis jobs.

Déclencheurs

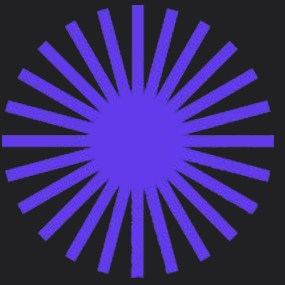
rules contre on:.

GitHub

workflows, jobs, puis steps.

Idée commune

Décrire des étapes automatiques.



Runners

GitLab

Partagés ou auto-hébergés (notre VPS).

Quota

Minutes gratuites des deux côtés.

GitHub

Hébergés ou self-hosted.

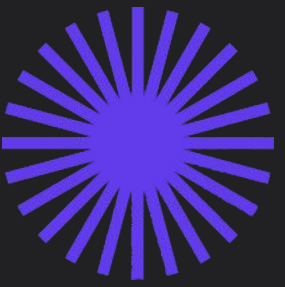
Réalité pro

Tôt ou tard auto-hébergé.

Secrets & OIDC

- Mêmes **bonnes pratiques**
- Variables et secrets **chiffrés** des deux côtés.
- **OIDC** disponible partout (cf. séance 5).

Écosystème



GitLab

Templates et CI/CD components intégrés.

GitHub

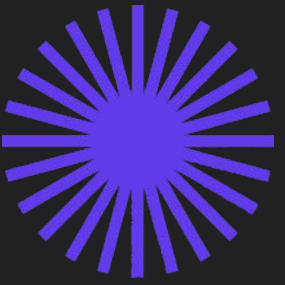
Actions Marketplace très riche.

Intégré vs composable

Deux philosophies.

Attention

Une Action tierce = du code tiers à auditer.



Quand choisir quoi

Où vit le code

Souvent décisif.

Coût / quotas

À comparer.

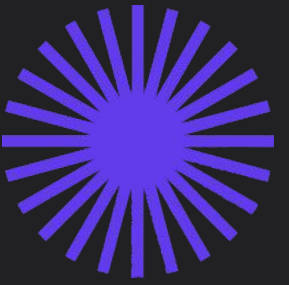
Intégré vs marketplace

Simplicité ou flexibilité.

Compétences équipe

Ce que l'équipe maîtrise.

Atelier : porter un job GitHub



1 · Reprendre

Les jobs de lint + tests.

2 · Réécrire

En GitHub Actions.

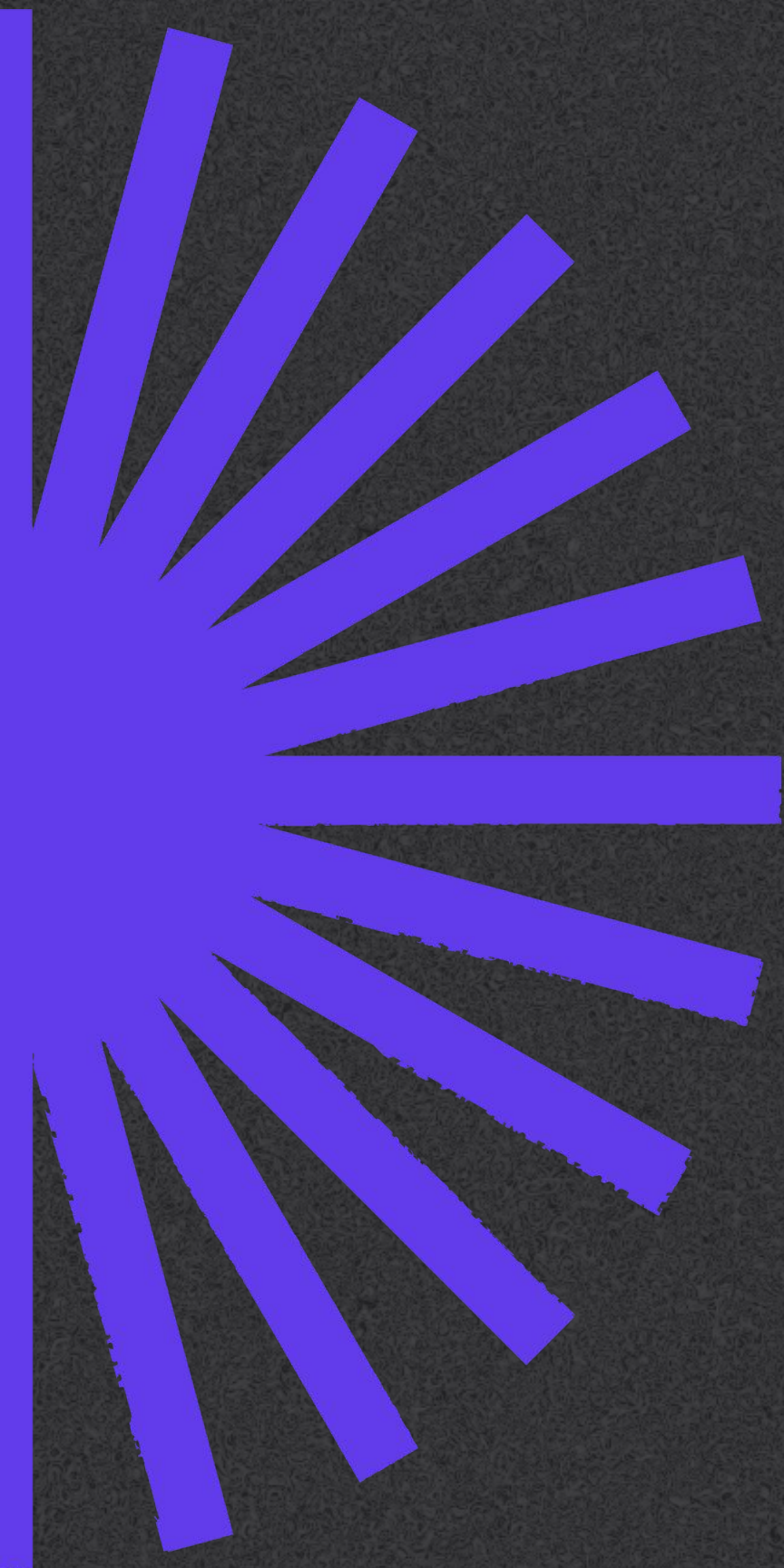
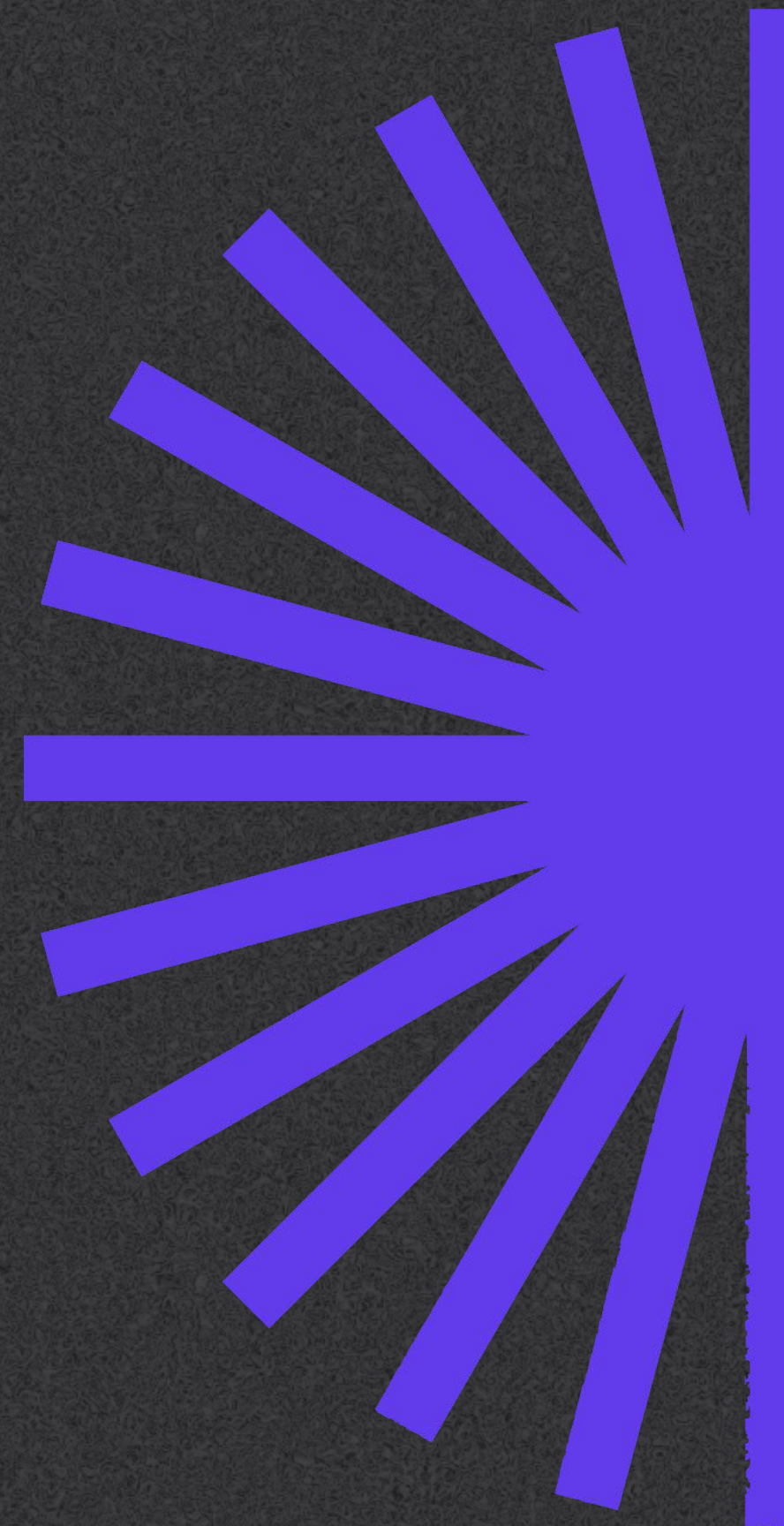
3 · Comparer

Déclencheurs, secrets, runner.

Reco

Savoir argumenter un choix.

Séance 7



Au programme

Delivery vs Deployment

Rappel.

Environnements & promotion

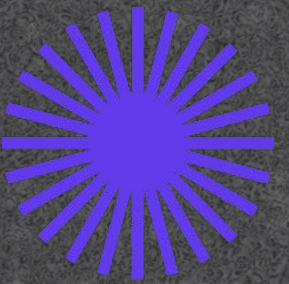
staging, prod.

Stratégies de déploiement

Rolling, blue-green...

Rollback & atelier

Le filet de sécurité.





Delivery vs Deployment

Livraison continue : prêt à déployer en un clic.

Déploiement continu : ça part tout seul.

On met en place les deux :

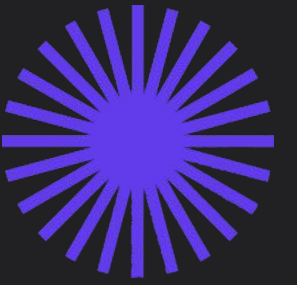
- un staging automatique, pour valider le recettage
- une prod déclenchée sur décision.

On garde la main avec une **Gate Manuelle** :

when: manual

et environnement protégé : seules certaines personnes lancent la prod.

Démo : un bouton « play » sur le job de production.



Stratégies de déploiement

Recreate

On coupe, on remplace (downtime).

Blue-green

Deux versions, on bascule.

Rolling

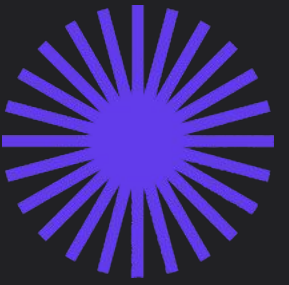
On remplace par lots.

Canary

Exposer à une fraction d'utilisateurs.

Déployer la version précédent doit être aussi simple que déployer la nouvelle.

Atelier : staging auto + prod manuelle



1 · Staging

Déploiement automatique.

2 · Prod

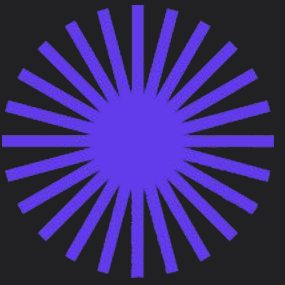
Job when: manual.

3 · Protéger

Environnement prod protégé.

4 · Rollback

Simuler un retour arrière.



Pièges fréquents

Prod auto

Sans garde-fou.

Env non protégé

N'importe qui déploie.

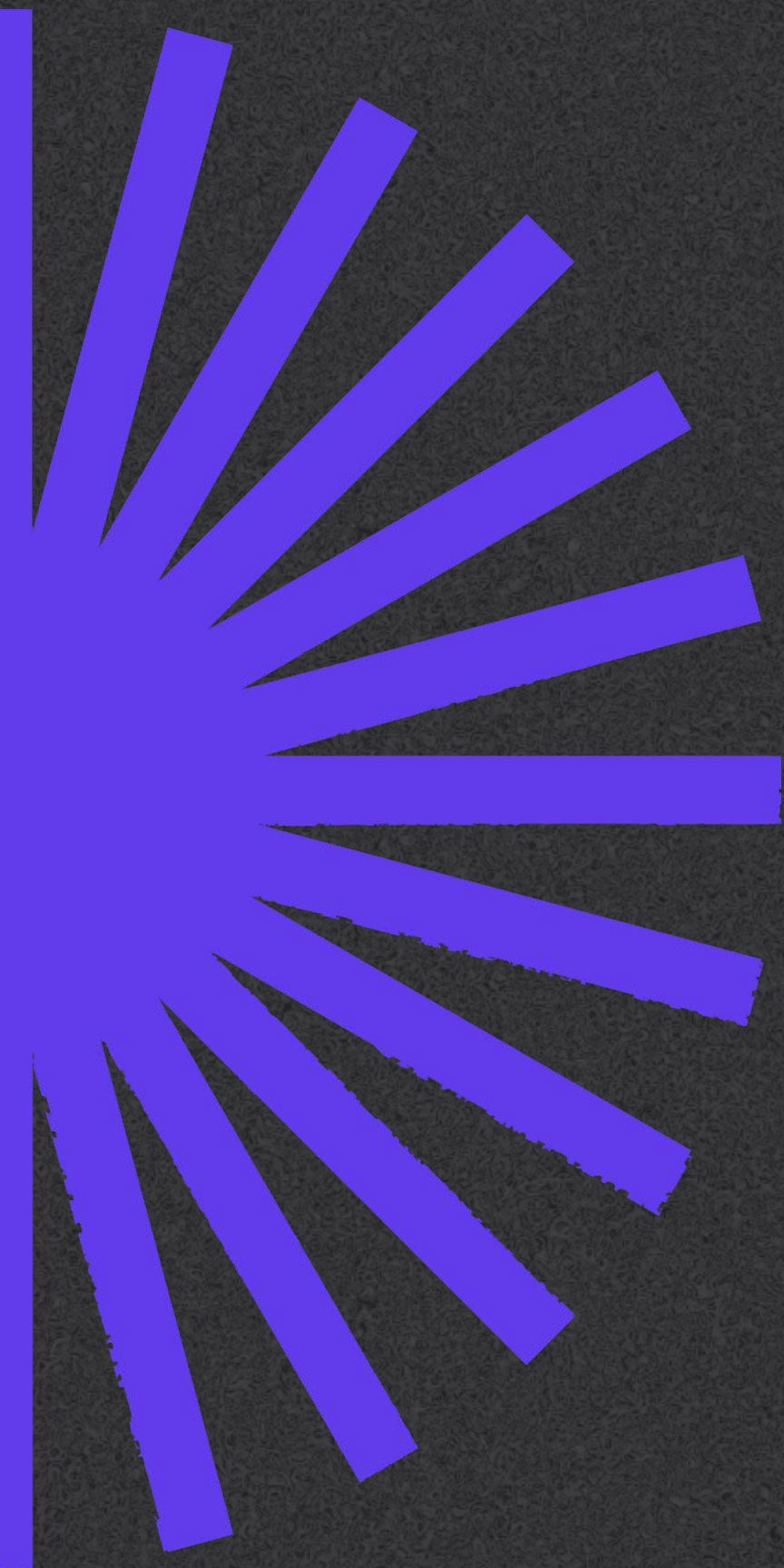
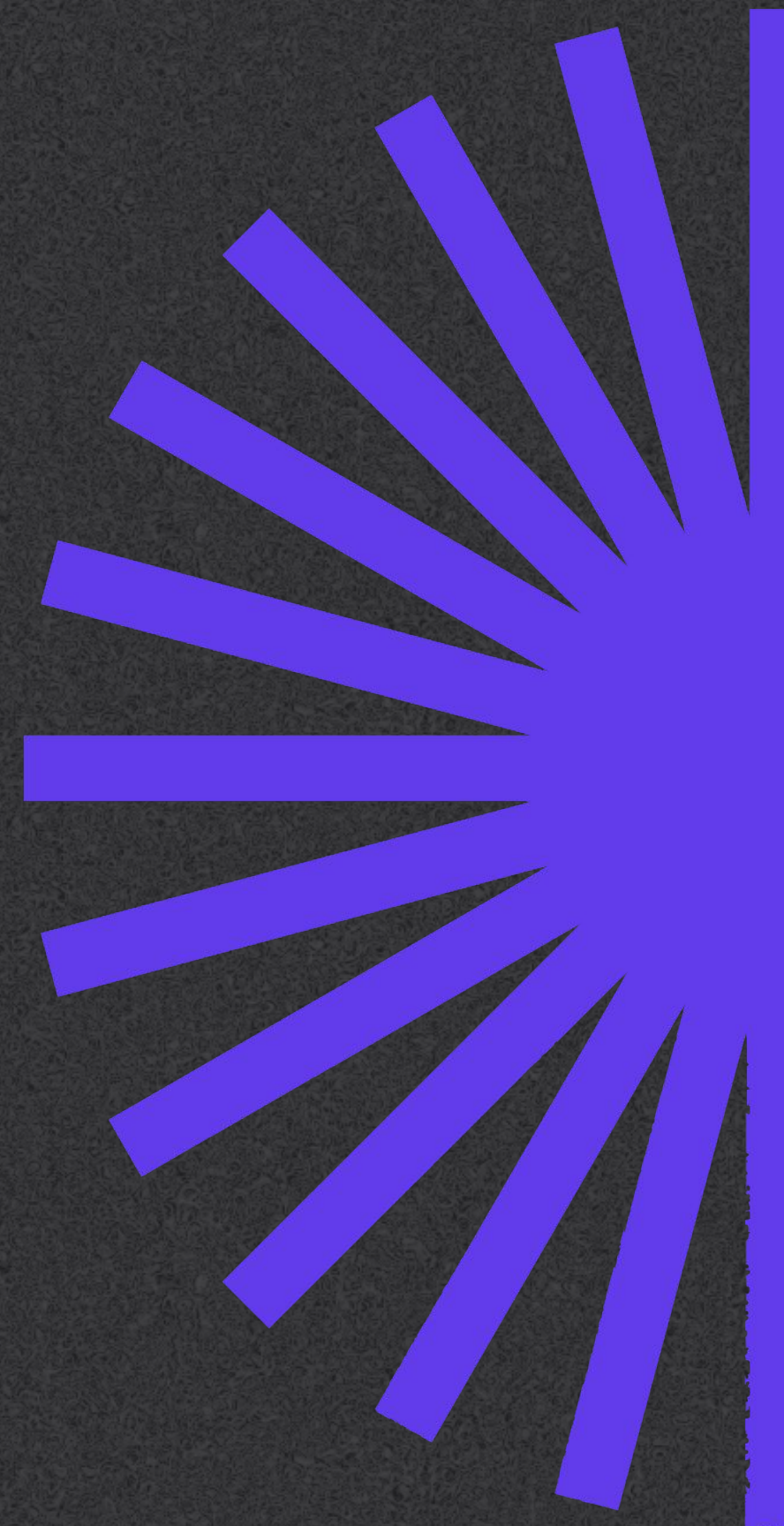
Pas de rollback

Aucun filet prévu.

Image reconstruite

≠ l'artefact testé.

Séance 8



Au programme

La CI parle au VPS

Connexion SSH.

Pull + Docker Compose

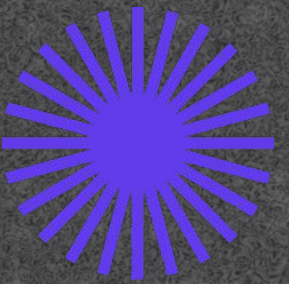
Lancer l'appli.

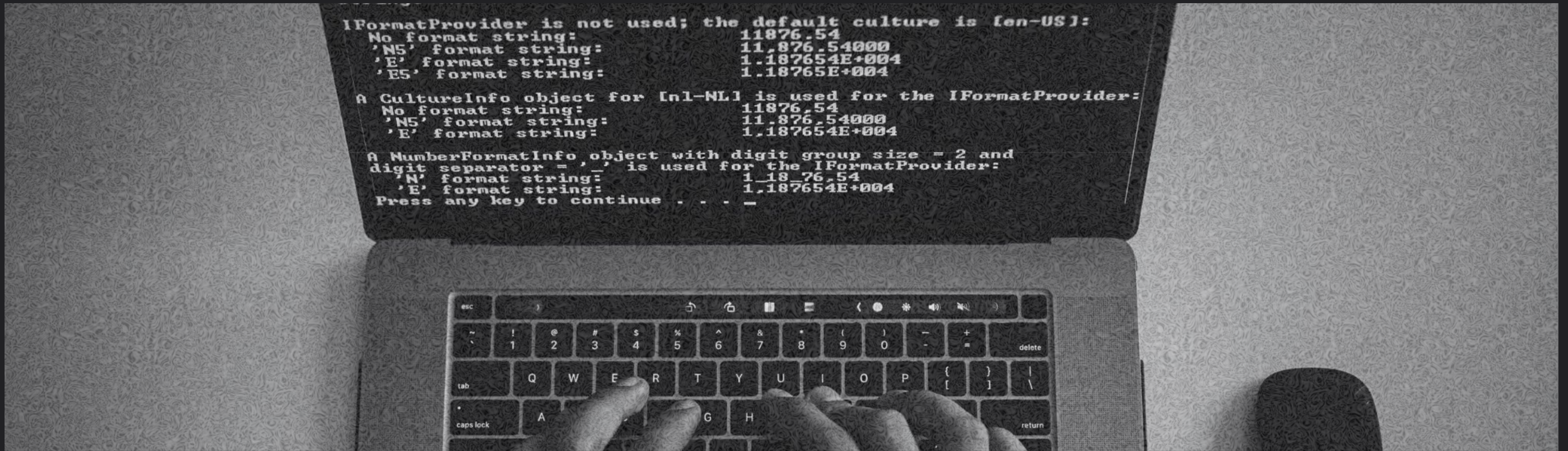
Reverse proxy

Un point d'entrée, du TLS.

Staging → prod

En live.





Notre cible : **VPS + Docker**

Pas de Kubernetes : un [serveur](#), [Docker](#), [Compose](#).
Simple, réaliste, et c'est ce que font beaucoup d'entreprises.

Maîtriser ce socle vaut mieux que survoler une orchestration plus lourde.

La CI parle au VPS

Connexion SSH depuis le job, avec une **clé privée** en variable **CI protégée**.

La clé n'est jamais dans le repo ;
l'hôte est connu (known_hosts).

Sur le VPS : pull + up

1 · Login

Au registre docker.

2 · Pull

docker compose pull (l'image testée).

3 · Up

docker compose up -d.

4 · Healthcheck

On vérifie que c'est sain.



Le **reverse proxy**

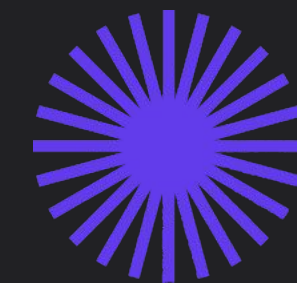
Traefik ou Caddy

Un proxy devant web et api : **HTTPS** et routage par domaine.

Un seul point d'entrée,
TLS géré au même endroit.

Démarrer le nouveau conteneur, le vérifier, puis basculer : **aucune coupure**.





Staging puis prod

Deux stacks

Compose séparés (ou deux hôtes).

Prod

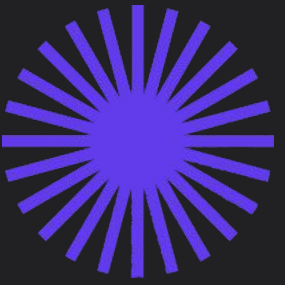
Derrière la gate manuelle (S7).

Mêmes images

Configs différentes par variables.

Suivi

Logs et healthchecks par environnement.



Pièges fréquents

Clé SSH

En clair ou trop permissive.

Pas de healthcheck

On déploie du cassé.

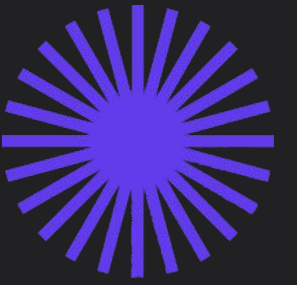
Ports

Exposés inutilement.

Compose

Non idempotent : états incohérents.

Atelier : déployer en live



1 · Clé SSH

Variable CI protégée.

2 · Staging

SSH → pull → up.

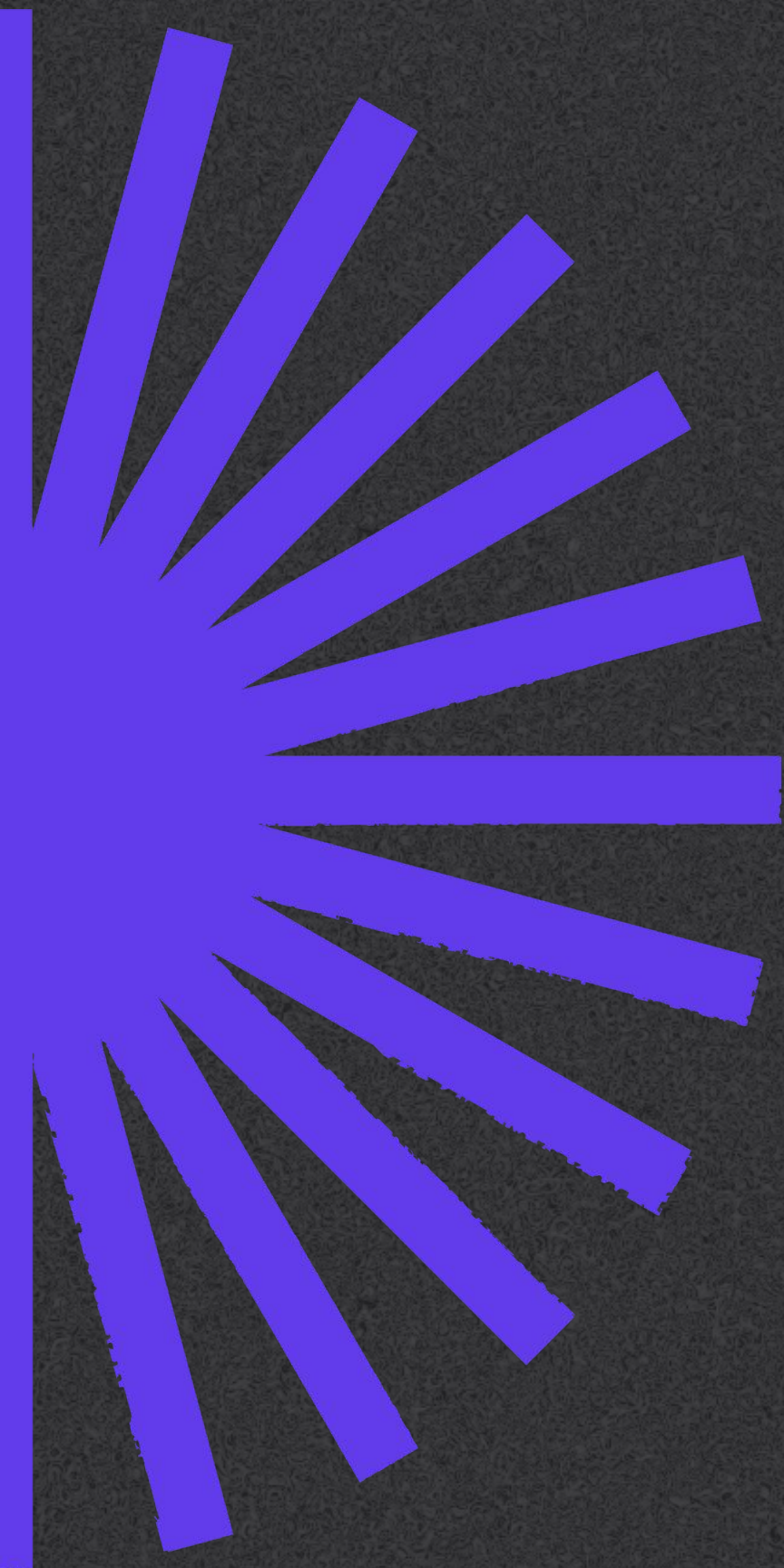
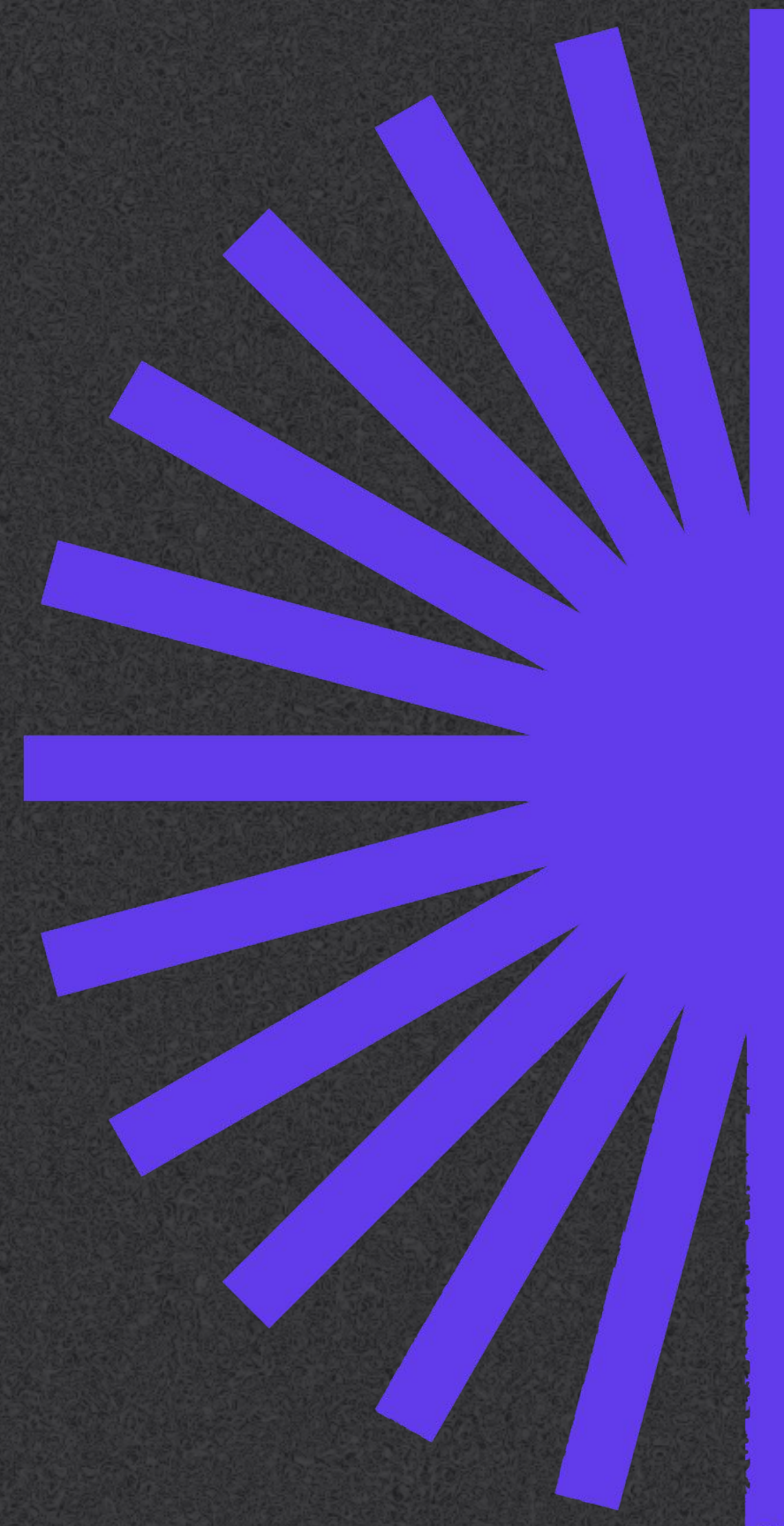
3 · Vérifier

L'appli en staging.

4 · Prod

Promouvoir manuellement.

Séance 9



Au programme

Observabilité

Logs, métriques, healthchecks.

Mesurer son pipeline

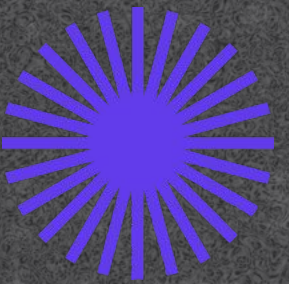
DORA en pratique.

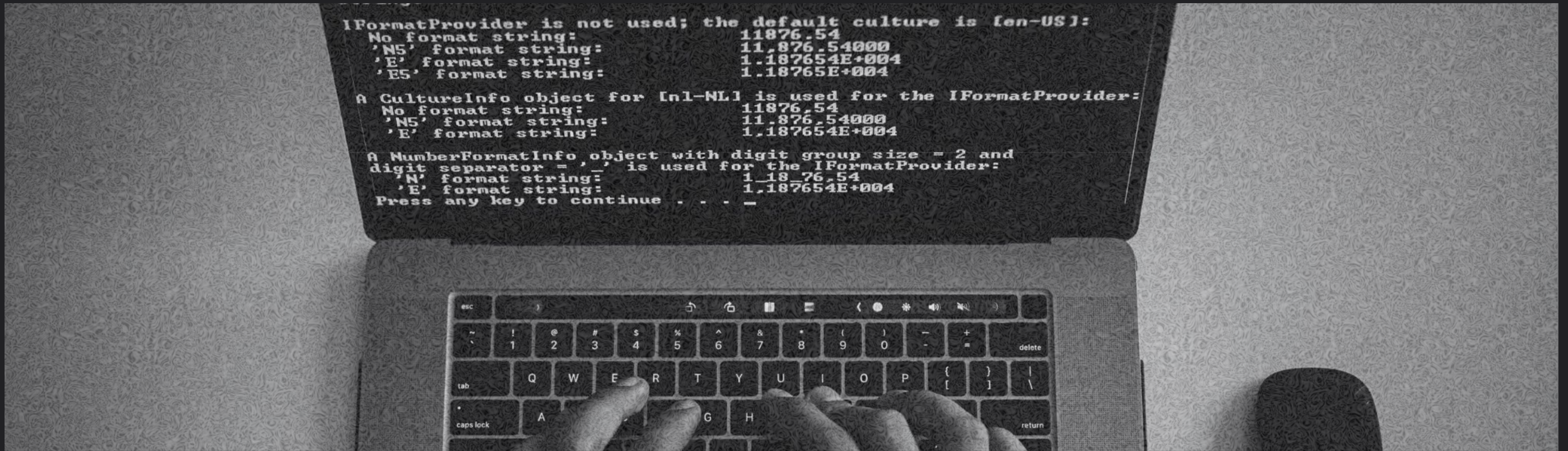
Documenter

README, runbook, ADR.

Sauvegarde & rollback

Protéger la donnée.

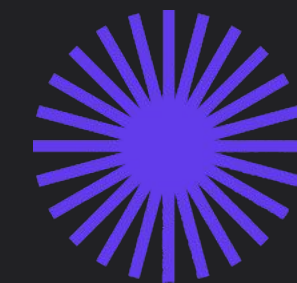




Déployer n'est pas **maintenir**

Mettre en prod, c'est le début. **Garder le service sain** et compréhensible, c'est le métier.

La valeur d'un ingénieur se mesure aussi dans la durée, pas seulement le jour du déploiement.



Observabilité

Logs

Que s'est-il passé ?

Healthchecks

Le conteneur est-il vivant ?

Métriques

Comment se porte le système ?

Alertes

Être prévenu avant les utilisateurs.

Surveiller le déploiement

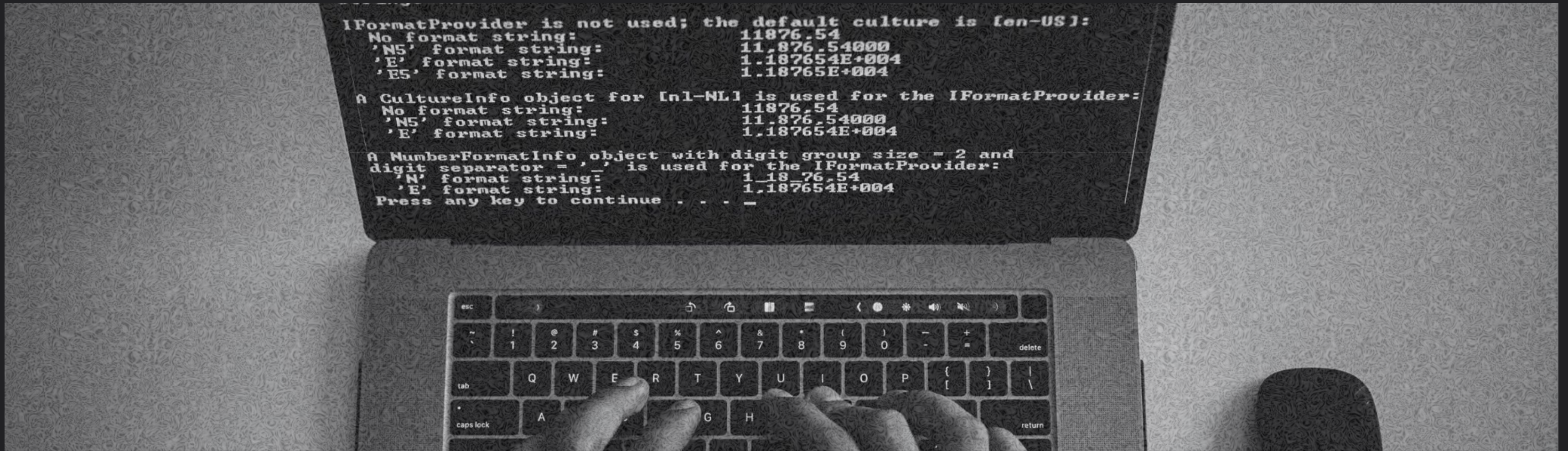
Sain ou pas ?

Un **healthcheck** dans Compose, la pipeline vérifie la santé après déploiement.

Un déploiement « réussi » mais un service KO reste un échec.

Vous avez maintenant un pipeline :
mesurez votre lead time et votre fréquence de déploiement.

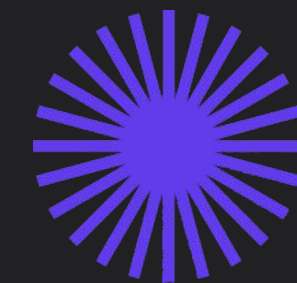




Documenter pour les autres, et pour soi

La meilleure doc s'adresse au collègue qui reprend le projet — ou à vous-même dans six mois.

La documentation fait partie de la compétence, ce n'est pas un bonus optionnel.



README, runbook, ADR

README

Démarrer le projet.

Runbook

Quoi faire pour déployer ou en cas d'incident.

ADR

Pourquoi telle décision a été prise.

Schéma d'archi

La vue d'ensemble.

Sauvegarde & rollback

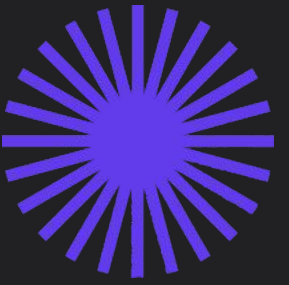
La donnée à part

Sauvegarder les données (Mongo) et documenter le rollback pas à pas.

La donnée ne se rollback pas comme le code.



Atelier : healthcheck + doc



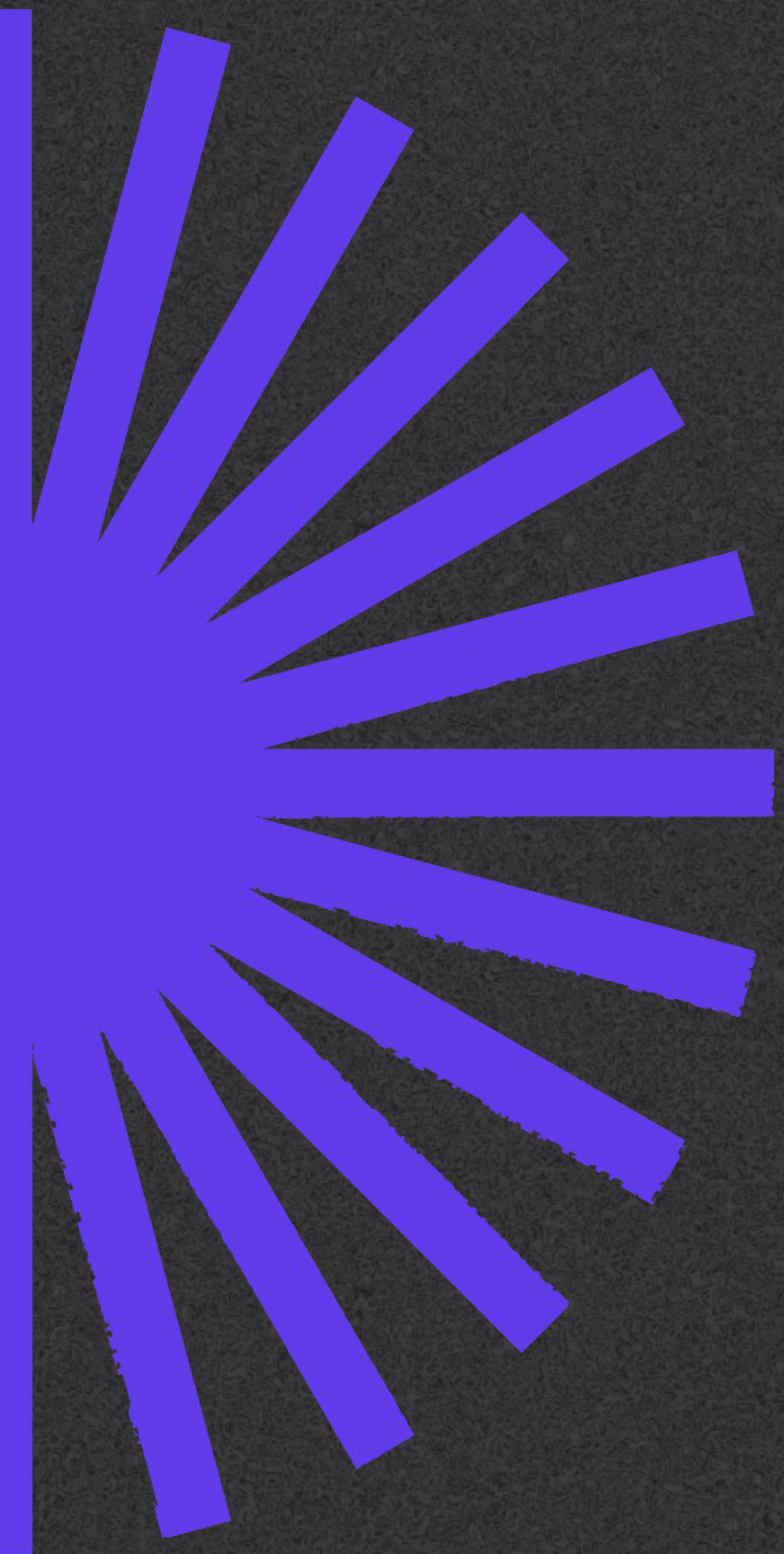
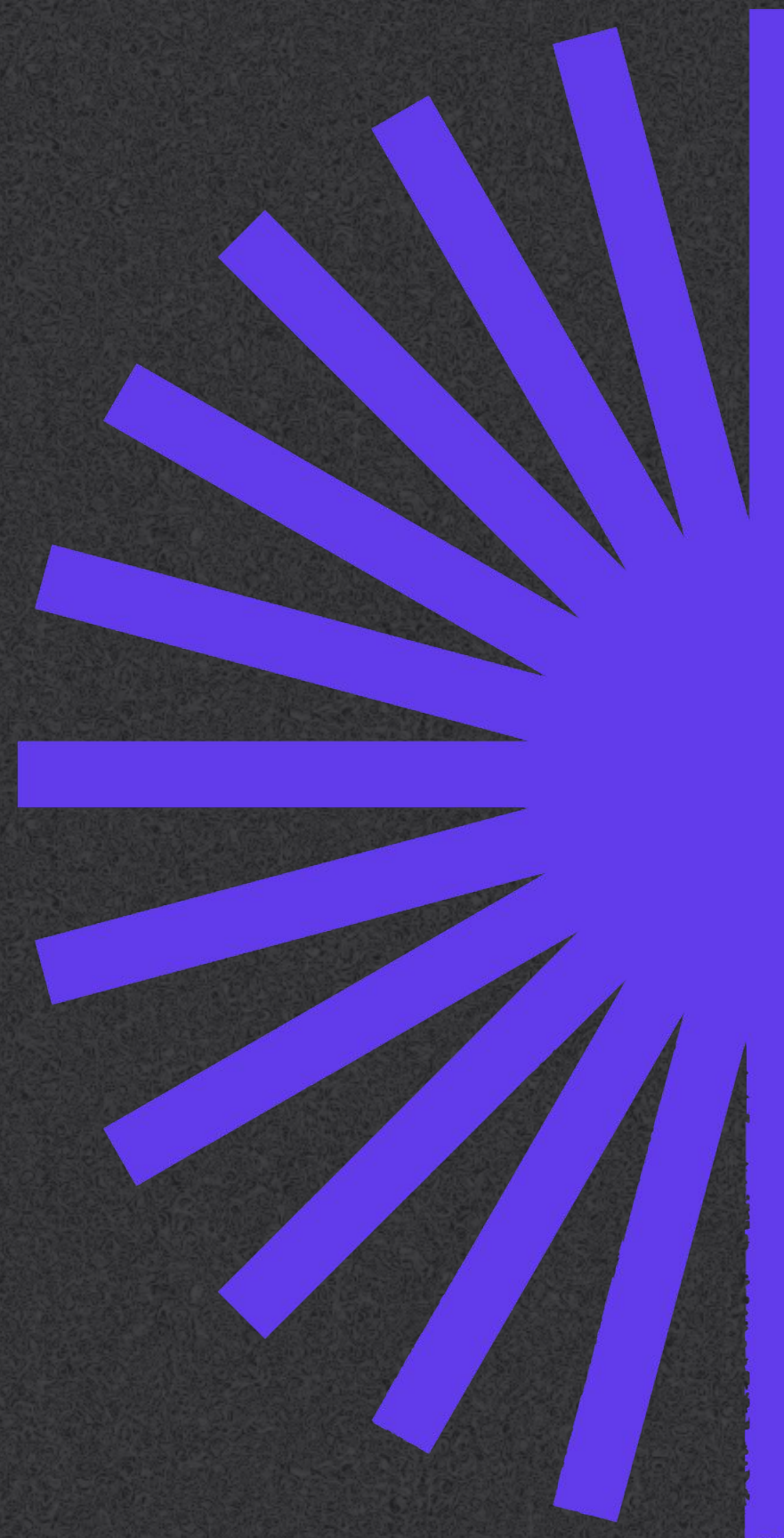
1 · Healthcheck
Ajouter une sonde de santé.

2 · Runbook
Déploiement et incident.

3 · ADR
Ex. pourquoi GitLab.

Bilan
Une archi reprenable.

Séance 10



Au programme

Factoriser & optimiser

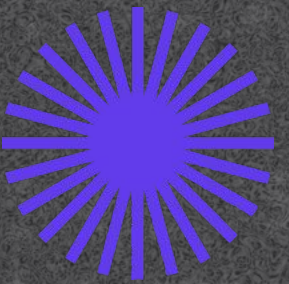
Une pipeline propre.

Préparer les évaluations

QCM & soutenance.

Clinique projet

Dépannage libre.



Factoriser la pipeline

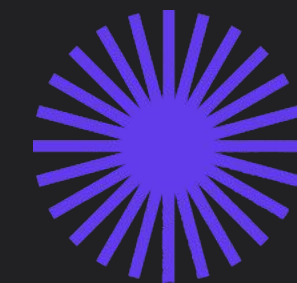
Du code lisible

Jobs réutilisables, extends, et CI/CD components pour mutualiser.

Un .gitlab-ci.yml propre se lit et se maintient ;
montrer un avant/après.

déporter certains éléments en script serveur.





Optimiser

Cache

Bien réglé.

Règles

Ne lancer que le nécessaire.

Parallélisme

Des jobs indépendants.

Images

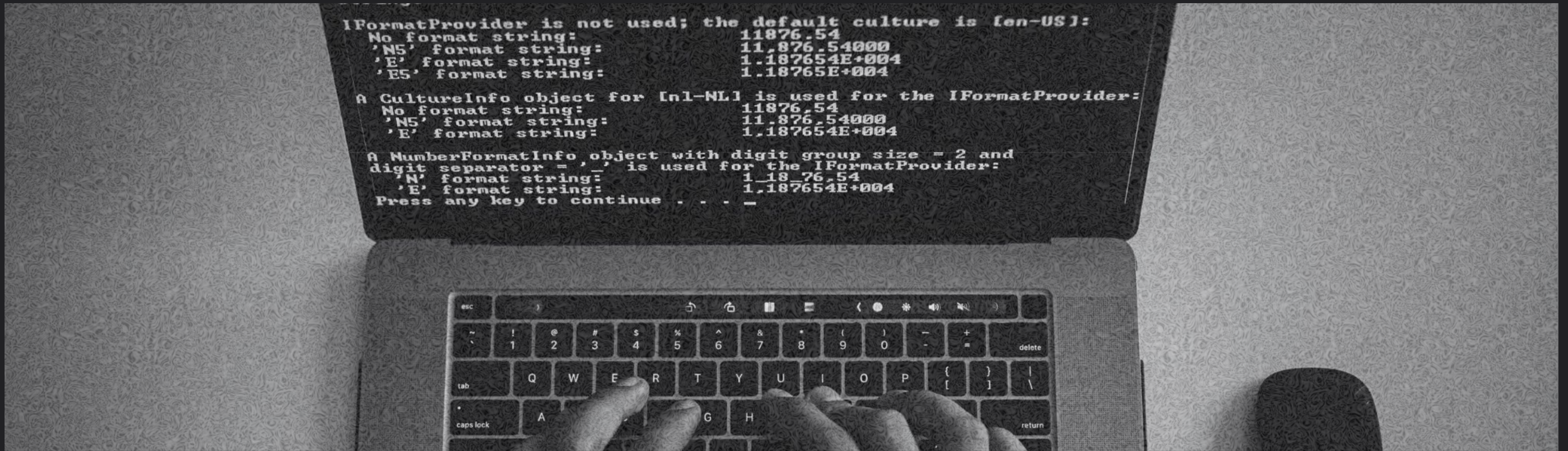
Légères.



CI /CD la démarche DevOps

De « ça marche chez moi »

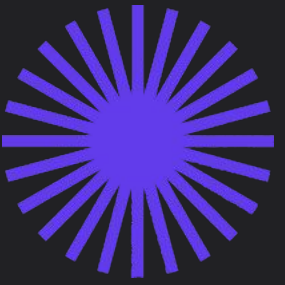
à une appli **livrée, testée,
sécurisée et documentée.**



Démo bonus : GitOps en deux mots

Et si l'état désiré du système vivait, lui aussi, dans un dépôt Git ?

C'est l'idée du GitOps (ArgoCD, Flux) — hors scope de ce cours, mais bon à connaître pour la culture.



Préparer le QCM 4 thèmes

Concepts

CI vs delivery vs deployment, DORA.

GitLab

Stages, jobs, runners, artefacts, environnements.

Sécurité

Secrets, types de scan, SBOM/signature.

Maintien & doc

Healthcheck, runbook, ADR.

Préparer la soutenance

Montrer et argumenter

Montrer votre **pipeline** qui tourne et argumenter vos choix.

Critères :

CI, tests, sécurité, déploiement, documentation.



Questions ?

You can find me at :

nicolas.hersant@gmail.com

@NicoHersant // [github](#) || [gitlab](#) || [linkedIn](#)

onimaru6172 // [discord](#)