



BYOD - Systèmes immuables

Une autre façon d'administrer.

#fiabilité #reproductibilité #traçabilité

Corentin VÉROT VASSAL

Architecte systèmes et logiciels

C'PRAG

« Ça marche sur ma machine »

Un nouvel arrivant.

Un environnement à mettre en place.

Avec les bonnes versions.

*Une demi-journée de **perdue**.*

Une mise à jour.

Régression

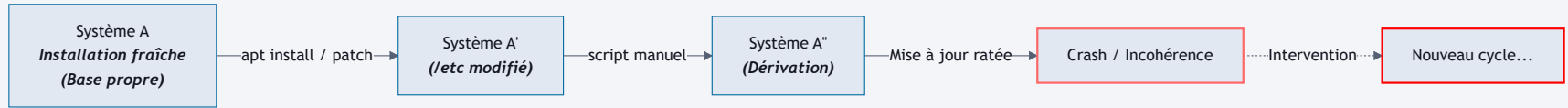
Puis-je revenir en arrière ?

Un état qu'on ne sait pas décrire précisément
ne se rejoue pas.

Pourquoi un système classique casse ?

Mutation

Modification d'un état du système



Préparer et activer

Deux étapes **distinctes**, pourtant traitées **simultanément**.

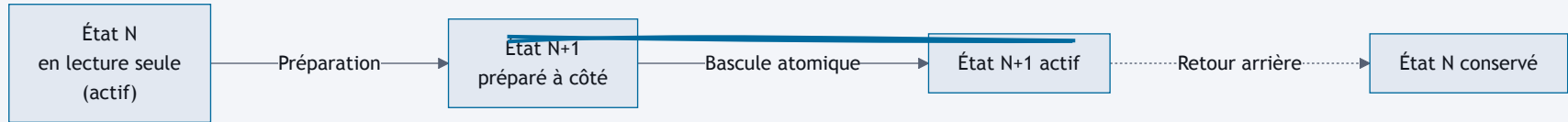
Une solution

Le système immuable

Comme un *commit*

Un état figé, auquel on peut revenir.

On ne modifie rien.
On prépare à côté.
On bascule d'un coup !



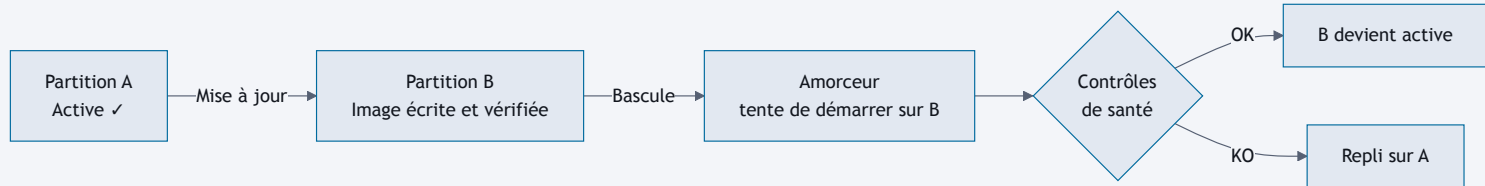
Trois approches

1. Partitionnement A/B
2. OSTree
3. NixOS

Approche 1

Partitionnement A/B

Android (≥ 7.0 | 2016) · ChromeOS (2009) · VanillaOS



Le revers

Occupe deux fois l'espace disque (partitions A/B).

Idéal pour le grand public.

Mais dès qu'on a des besoins particuliers, ça se corse.

Approche 2

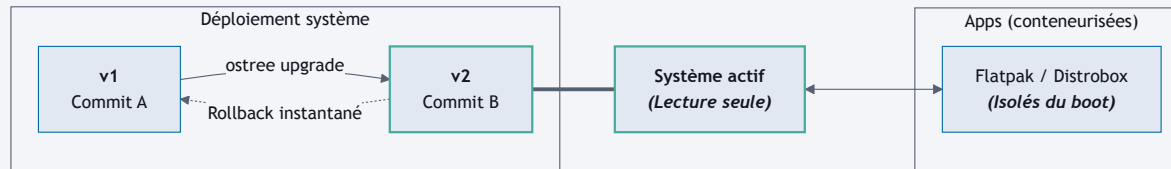
OSTree

2011, première distribution stable (Endless OS) en 2014

Fedora Silverblue · Fedora Atomic Desktop / *Universal Blue* (Aurora, Bazzite)

Le système est un arbre versionné.

Comme Git, mais pour les fichiers système.



Le revers

Les limites du modèle OSTree

Un compromis entre immutabilité et rigidité.

Granularité monolithique

Le pansement `rpm-ostree layering`

Reproductibilité incomplète

Approche 3

Un gestionnaire de paquet : Nix (2003)

Puis un système d'exploitation : NixOS (2008)

Tout est une fonction **pure** et **déterministe**.

Et **vous** écrivez la fonction maître !

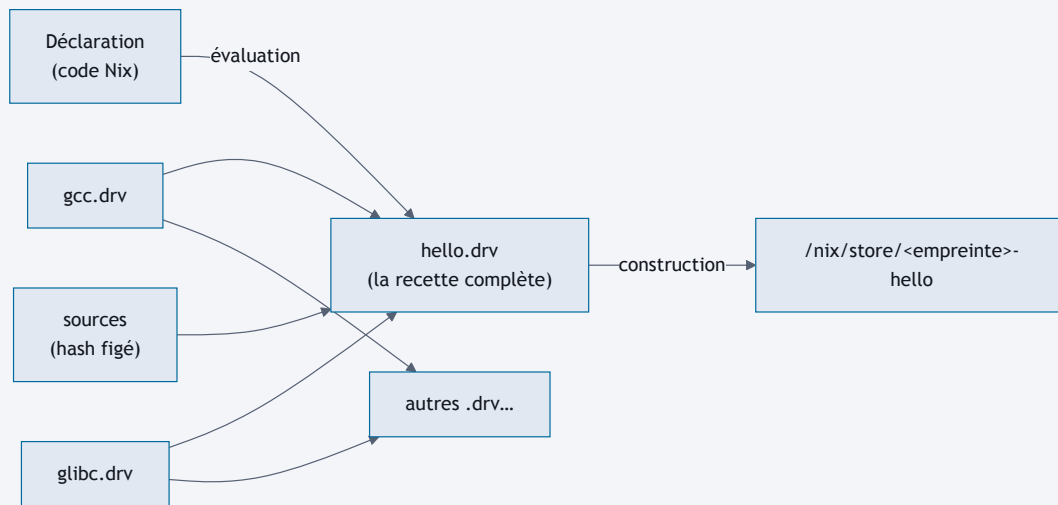
Un paquet se déclare.

```
stdenv.mkDerivation {  
  pname = "hello";  
  version = "2.12.1";  
  src = fetchurl { url = "..."; hash = "sha256-..."; };  
}
```

Une fonction : des entrées, un résultat.

La déclaration devient une recette reproductible.

Nix évalue la fonction et produit une *dérivation* (fichier `.drv`).



Une dérivation dépend d'autres dérivation : même entrée, même résultat.

Pas seulement des paquets.

Tout ce que Nix produit est une dérivation.

Un paquet : `hello`

Une unité systemd : le service qui lance PostgreSQL

Le système entier, qui assemble toutes les précédentes

Un fichier de configuration : la config de PostgreSQL

La liste des utilisateurs et groupes système

Une recette = un chemin haché

L'empreinte est calculée à partir de la dérivation.

```
/nix/store/<empreinte>-hello-2.12.1
```

Autre option, autre empreinte

Compilation, dépendances, versions, flags, patches...

Avec GCC 12 : `/nix/store/a1b2c3d4...-hello-2.12.1`

Avec GCC 13 : `/nix/store/x9y8z7w6...-hello-2.12.1`

Plusieurs versions côte à côte.

Sans conflit.

Une entrée change.

Seul le nécessaire se reconstruit.

Construction isolée.

Rien d'implicite.

Du coup, tout écrire à la main ?

Non : nixpkgs.

Un dépôt de recettes déjà écrites : paquets, et aussi *modules*.

Un module : une option, plusieurs dérivations.

```
services.postgresql = {  
  enable = true;  
  #port = 5432;  
  port = 5434;  
};
```

Le module `postgresql` de `nixpkgs` produit pour vous :

le paquet PostgreSQL

le fichier de configuration

l'unité `systemd`

l'utilisateur et le groupe système

Vous déclarez l'intention, le module écrit les recettes.

nixpkgs

Le plus grand catalogue de paquets.

Distribution	Stable (n° de version)	Rolling / unstable
nixpkgs	~115 000 (26.05)	~119 000 (unstable)
Debian	~38 000 (13)	~42 000 (unstable)
Ubuntu	~40 000 (26.04 LTS)	—
Fedora	~23 000 (44)	~22 000 (Rawhide)
openSUSE	~15 000 (Leap 15.6)	~16 000 (Tumbleweed)
Arch Linux	—	~12 000
Rocky Linux, AlmaLinux (équivalents RHEL)	~2 500 (9)	—

Source : [Repology](#) (octobre 2026)

~23 500 paquets libres

~1 100 non libres, dont l'utilisation est à autoriser.

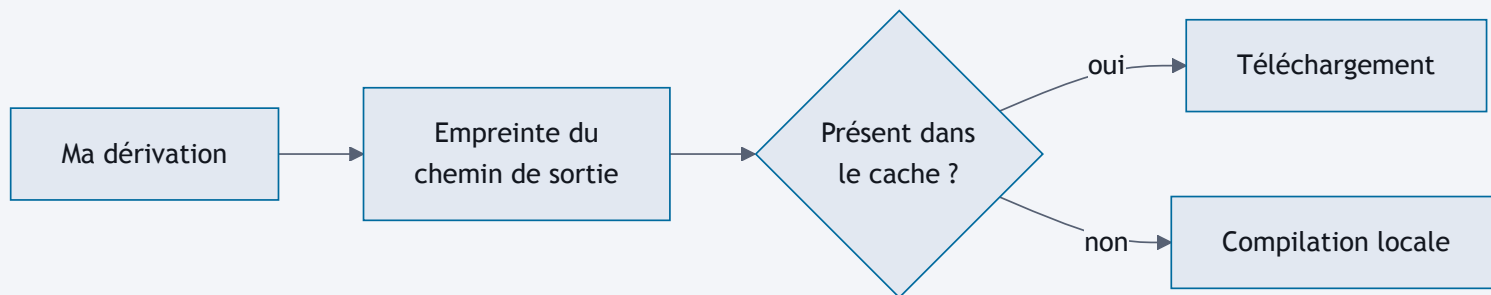
Et bien plus avec les flakes.

On a donc des recettes pour composer des dérivations...

... mais faut-il recompiler systématiquement les binaires ?

Heureusement, non !

Un **cache binaire** : un serveur qui garde des résultats déjà construits, classés par empreinte.

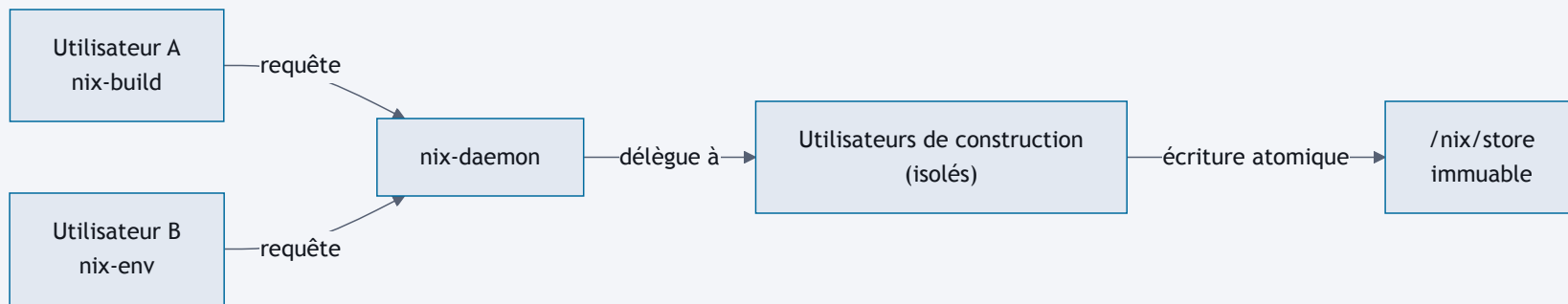


Si disponible, on télécharge : pas de compilation inutile.

Le *store*, ou l'archivage des dérivations

Le *store* (`/nix/store`) : le dossier où atterrit chaque résultat construit. En lecture seule.

Un gardien : le `nix-daemon`



Le prix : NixOS brise le FHS

Filesystem Hierarchy Standard

Pas de `/usr/bin`, ni de `/lib` : tout est dans `/nix/store`.

Un binaire téléchargé ne trouve pas directement ses bibliothèques.

Comment ça tourne sans FHS ?

Les binaires pointent directement vers le store.

```
/nix/store/...-glibc/lib/libc.so
```

Et le \$PATH ?

Des liens symboliques assemblent un profil.

`/run/current-system/sw/bin` → store

Quid des binaires qu'on a besoin de télécharger ?

`nix-ld` · `buildFHSEnv` · `patchelf`

Pour les scripts

```
#!/bin/bash → #!/usr/bin/env bash
```

Configuration déclarative

La configuration devient du code.

```
{ config, pkgs, ... }: {  
  # Configuration réseau  
  networking = {  
    hostName = "demo";  
    interfaces.eth0 = {  
      useDHCP = false;  
      ipv4.addresses = [  
        { address = "10.43.75.2"; prefixLength = 16; }  
      ];  
    };  
    defaultGateway = {  
      address = "10.43.0.1";  
      interface = "eth0";  
    };  
  };  
  time.timeZone = "Europe/Paris";  
}
```

```
{ config, pkgs, ... }: {  
  # Un service  
  services.openssh.enable = true;  
  
  # Un utilisateur  
  users = {  
    users.alice = {  
      uid = "2000";  
      isNormalUser = true;  
      group = "alice";  
      extraGroups = [ "wheel" ];  
    };  
    groups.alice.gid = "2000";  
  };  
  
  # Des paquets disponibles pour tout le monde  
  # Liens symboliques dans /run/current-system/sw/bin  
  environment.systemPackages = with pkgs; [  
    curl  
    htop  
    git  
  ];  
}
```

Le *garbage collector*

Rien n'est jamais écrasé : le `/nix/store` ne fait que grossir.

À configurer en exécution automatique.

```
nix.gc = {  
  automatic = true;  
  dates = "weekly";  
  options = "--delete-older-than 30d";  
};
```

Et les données ?

NixOS ne versionne pas les données.

Mise à jour d'une application

La nouvelle version convertit les fichiers des utilisateurs.

Images, documents : réécrits dans un nouveau format.

L'ancienne version du programme est rétablie.

Celle des données, **non**.

Solution

Un instantané Btrfs.

Btrfs

Les instantanés (*snapshots*)

Instantané immédiat.

Sans copie initiale.

Copie sur écriture

Copy on Write (CoW)

Seules les modifications de fichiers coûtent de l'espace.

Retour arrière

On change d'instantané.

NixOS : le système.

Btrfs : les données.

Flakes

Expérimentaux. Incontournables.

2019-2020

flake.nix

Des entrées. Des sorties.

Les entrées : d'où vient tout

```
inputs = {  
  nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";  
  nixpkgs-unstable.url = "github:NixOS/nixpkgs/nixos-unstable";  
  home-manager = {  
    url = "github:nix-community/home-manager/release-26.05";  
    inputs.nixpkgs.follows = "nixpkgs";  
  };  
};
```

Le canal instable devient `pkgs.unstable`

```
outputs = { self, nixpkgs, home-manager }:@inputs:
  let
    system = "x86_64-linux";
    unstable = final: _: {
      unstable = import inputs.nixpkgs-unstable {
        inherit system;
        config.allowUnfree = true; # Autoriser l'utilisation de paquets non libres
      };
    };
    pkgs = import nixpkgs { inherit system; overlays = [ unstable ]; };
  in {
    # ... les sorties, juste après
  };
```

Une fonction qui fabrique une machine

```
let
  # ...
  mkHost = host: nixpkgs.lib.nixosSystem {
    inherit system;
    modules = [
      {
        nixpkgs = {
          allowUnfree = true;
          overlays = [ unstable ];
        };
      }
      home-manager.nixosModules.home-manager
    ]
    # On charge un fichier dynamiquement en fonction du nom passé en argument
    ./hosts/${host}.nix
  };
in { /* ... */ }
```

Les sorties : un paquet, un shell

```
in {  
  packages.${system}.default = pkgs.unstable.hello;  
  
  devShells.${system}.default =  
    pkgs.mkShell { packages = [ pkgs.git pkgs.unstable.jq ]; };  
}
```

Plusieurs machines, un seul flake

```
nixosConfigurations = {  
  serveur = mkHost "serveur";  
  portable = mkHost "portable";  
  atelier = mkHost "atelier";  
};
```

Une machine = un nom après le

```
# Forme générale
nixos-rebuild <build|switch> --flake <chemin du flake>#<nom de la machine>

# Construire la machine « portable », sans rien changer et pour déployer plus tard
nixos-rebuild build --flake .#portable

# Construire et appliquer la configuration « serveur » sur cette machine
nixos-rebuild switch --flake .#serveur --sudo

# Déployer « atelier » à distance, depuis mon poste
nixos-rebuild switch --flake .#atelier --target-host root@atelier
```

flake.lock

Tout est figé.

package-lock.json · composer.lock

Git ne voit pas ?

Nix en mode flake non plus.

```
git add
```

`nix build` • `nix run` • `nix develop` • `nix shell`

devenv.sh

Des environnements de dev déclaratifs.

Un fichier : `devenv.nix`

Environnements de développement isolés.

Services intégrés : PostgreSQL, Redis, MariaDB (données dans `.devenv`)

Langages : Node.js, PHP, Python, Go, Rust, ...

`devenv.lock` : versions verrouillées (comme `flake.lock`).

Exemple : devenv.nix

```
{ config, pkgs, ... }:  
let  
  dbName = "my_db";  
in {  
  packages = with pkgs; [ gcc git openssl ];  
  
  # Charge le .env (priorité basse : `env` ci-dessous l'emporte)  
  dotenv.enable = true;  
  
  languages.javascript = {  
    enable = true;  
    nodejs.enable = true;  
    npm = {  
      enable = true;  
      install = true; # lance un npm install auto  
    };  
  };  
  
  languages.python.enable = true;  
  services.mailpit.enable = true;  
  
  env = {  
    DATABASE_URL = "postgres://postgres@localhost:${toString config.services.postgres.port}/${dbName}";  
    SMTP_HOST = "127.0.0.1";  
  };  
}
```

```

{ config, pkgs, ... }: {
  services.postgres = {
    enable = true;
    port = 5433;
    extensions = extensions: [
      extensions.pgvector
    ];

    # Auth locale sans mot de passe
    # jamais exposée hors localhost
    hbaConf = ''
      local all all trust
      host  all all 127.0.0.1/32 trust
      host  all all ::1/128      trust
    '';

    initialDatabases = [{
      name = dbName;
    }];

    # autres options ...
  };
}

```

`dotenv.enable`

Le fichier `.env` donne des valeurs par défaut.

L'attribut `env` **les écrase**.

En cas d'activation de `dotenv.enable`, vérifiez l'ordre de chargement des variables d'environnement au sein de votre programme (idéalement variables d'environnement > `.env`) : source d'effets de bord !

Utilisation

Entrée dans l'environnement

Manuellement : `devenv shell`

Avec intégration `direnv` : entrée auto dès cd dans le répertoire

Démarrage des services

`devenv up`

Et Docker dans tout ça ?

NixOS et Docker / Podman se complètent.

NixOS fait tourner les conteneurs

`virtualisation.oci-containers` : conteneurs déclarés dans la configuration.

compose2nix

Projet `docker-compose.yml` → configuration NixOS.

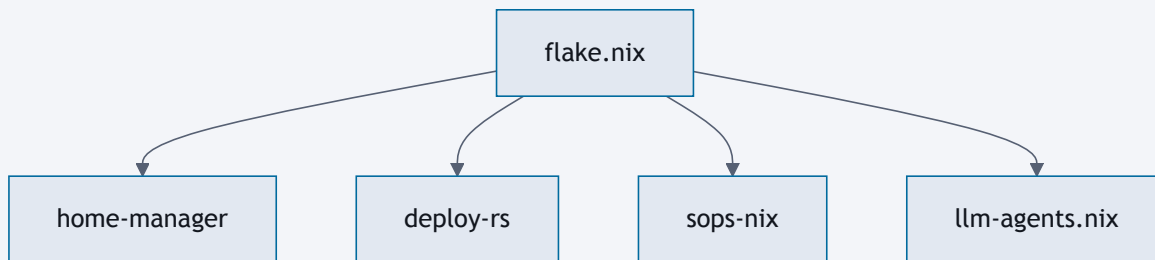
devenv container

L'environnement écrit en nix, mais dans un conteneur.

Une image docker officielle

`docker.io/nixos/nix`

L'écosystème NixOS : quelques flakes sympa



home-manager

Votre environnement utilisateur, versionné.

deploy-rs

Déployer sur des machines distantes.

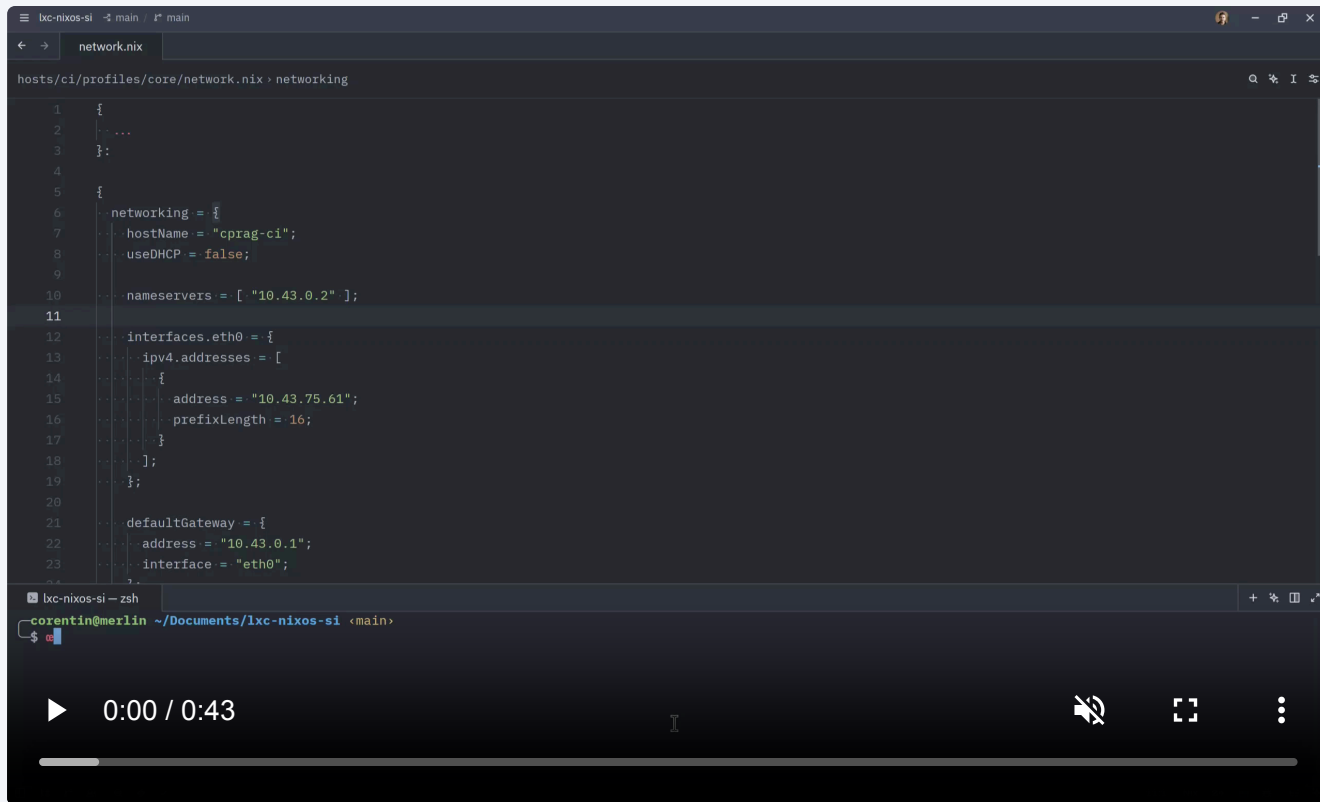
sops-nix

Des secrets chiffrés, compatibles avec Git.

llm-agents.nix

Des agents IA, reproductibles.

Démo deploy-rs : un déploiement raté se défait tout seul



The screenshot shows a video player interface with a dark theme. The main content is a terminal window displaying Nix code. The terminal has a title bar that reads 'lxc-nixos-si - main / # main'. Below the title bar, the current directory is 'network.nix' and the file being edited is 'hosts/ci/profiles/core/network.nix > networking'. The code is a Nix configuration for networking, defining hostnames, DNS servers, and network interfaces. The code is as follows:

```
1 {
2   ...
3 }:
4
5 {
6   --networking = {
7     --hostname = "cprag-ci";
8     --useDHCP = false;
9
10    --nameservers = [ "10.43.0.2" ];
11
12    --interfaces.eth0 = {
13      --ipv4.addresses = [
14        {
15          address = "10.43.75.61";
16          prefixlength = 16;
17        }
18      ];
19    };
20
21    --defaultGateway = {
22      address = "10.43.0.1";
23      interface = "eth0";
24    };
25  };
26}
```

At the bottom of the terminal window, there is a prompt 'corentin@merlin ~/Documents/lxc-nixos-si <main>' and a shell prompt '\$'. The video player controls at the bottom show a play button, a progress bar at 0:00 / 0:43, and icons for volume, full screen, and settings.

Dès demain ?

```
git clone
```

```
devenv shell
```

Le nouvel arrivant peut commencer à travailler.

Fini « ça marche chez moi ».

Mettre à jour sans peur.

Retour arrière en quelques secondes.

Un projet. Un environnement dédié.

Hygiène de vos machines.

Merci pour votre attention !

Place à vos questions 😊