Master of Science HES-SO in Engineering

AI Job scheduling and GPU allocation. Which system for which usage?

Friday, 4 July 2025

Professors: Sébastien Rumley, Giorgios Michelogiannakis

Olivier D'Ancona

Contents

Abstract

This is absolutely Awesome thesis HAHA

1 Introduction

1.1 Motivation

Scheduler

1.2 Objectives

- Automatiser, répéter et comparer des clusters virtuels HPC, chacun avec des topologies et schedulers différents, sur une machine physique unique. - Changer rapidement de scheduler (SLURM, K8s+Volcano, Flux), de workload, de taille de cluster, etc. - Collecter des métriques fiables pour des analyses scientifiques (Pareto, reproductibilité...) - Itérer vite (provision/teardown), changer de config, et garantir que chaque expérience est isolée et identique à la précédente.

Automated provisioning

Holistic Configuration

Declarative vs Imperative 1. Reproducibility: Nix excels at creating reproducible environments, which is critical for benchmark validity. Each configuration will produce identical results given the same inputs, eliminating environment-related variables from your measurements.

2. Declarative Infrastructure: Nix allows you to specify your entire virtual environment as code, aligning perfectly with your formal approach of defining hardware topologies

3. Isolation: Each package and configuration exists in isolation, preventing dependency conflicts between different scheduler installations or benchmark tools.

4. Fine-grained Hardware Profiles: Nix can precisely control resource allocations for your VMs, matching the hardware profiles you specified (1,2,4,8 CPUs, 1,2,4,8 GPUs, various RAM configurations).

5. Composability: Nix configurations can be easily composed, which aligns with your need to mix and match different hardware topologies, schedulers, and workloads.

Its guarantees of reproducibility are invaluable for benchmark validity.

J'ai choisi une stack basée sur NixOS pour la génération des images, une orchestration Python (Libvirt) pour le provisionnement, et une configuration déclarative typée en Python, afin d'obtenir une reproductibilité totale, une automatisation fine, et une simplicité d'audit et d'évolution.

Ta stack (NixOS + Python/libvirt + config déclarative) est la seule qui garantit: - Reproductibilité scientifique stricte (indispensable pour des benchmarks HPC). - Rapidité et fiabilité du cycle d'expérimentation (provision/teardown). - Flexibilité et évolutivité (topologies, schedulers, workloads, ressources). - Programmabilité avancée (orchestration, collecte de métriques, monitoring). - Nettoyage et isolation parfaits entre expériences. - Simplicité de la chaîne d'outils: tout en Python et Nix, pas de glue fragile entre Terraform, Ansible, shell, cloud-init, etc.

2 State Of The Art

There is a plethor of technologies and alternative involved in the provisionning of a virtual cluster automatically. This section introduces some of the most relevant ones.

2.1 Technologies

2.1.1 Terraform

Terraform is an open-source infrastructure as code (IaC) tool developed by HashiCorp. It allows users to define and provision infrastructure resources using a declarative configuration language called HashiCorp Configuration Language (HCL). Terraform enables the automation of infrastructure management across various cloud providers, including AWS, Azure, Google Cloud, and others.

2.1.2 OpenTofu

OpenTofu is a fork of Terraform that aims to provide an open-source alternative to the original Terraform project. It retains the core functionality of Terraform while focusing on community-driven development and governance. OpenTofu allows users to define and manage infrastructure resources using the same declarative configuration language (HCL) as Terraform, making it compatible with existing Terraform configurations.

2.1.3 Ansible

Ansible is an open-source automation tool that simplifies the management and configuration of systems. It uses a declarative language to define the desired state of infrastructure and applications, allowing users to automate tasks such as provisioning, configuration management, and application deployment. Ansible operates in an agentless manner, using SSH or WinRM to communicate with target systems, making it easy to integrate into existing environments.

2.1.4 Nix

Nix is a powerful package manager and build system that provides a declarative approach to managing software and system configurations. It allows users to define their entire system environment, including packages, services, and configurations, in a single file called the Nix expression. Nix ensures reproducibility by isolating dependencies and providing a consistent environment across different machines.

2.1.5 Kubernetes

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It provides a robust framework for managing clusters of containers across multiple hosts, enabling users to define the desired state of their applications and automatically maintain that state. Kubernetes supports various container runtimes and integrates with cloud providers, making it a popular choice for modern application deployment.

2.1.6 Helm

Helm is a package manager for Kubernetes that simplifies the deployment and management of applications on Kubernetes clusters. It uses a templating system to define reusable application configurations, called charts, which can be easily shared and versioned. Helm allows users to manage complex applications with multiple components, making it easier to deploy, upgrade, and roll back applications in Kubernetes environments.

2.1.7 Kustomize

Kustomize is a tool for customizing Kubernetes resource configurations. It allows users to define a base set of resources and apply overlays to modify them for different environments or use cases. Kustomize enables users to manage Kubernetes manifests without the need for templating, making it easier to maintain and version control configurations. It is integrated into kubectl, the Kubernetes command-line tool, allowing users to apply customizations directly from the command line.

2.1.8 Pulumi

Pulumi is an open-source infrastructure as code (IaC) tool that allows users to define and manage cloud infrastructure using general-purpose programming languages such as JavaScript, TypeScript, Python, Go, and C#. Pulumi enables developers to leverage their existing programming skills to create reusable components and automate infrastructure provisioning across various cloud providers. It supports both declarative and imperative approaches to infrastructure management, providing flexibility in how users define their infrastructure.

2.1.9 Libvirt

Libvirt is an open-source API and management tool for virtualization technologies. It provides a unified interface for managing different hypervisors, such as KVM, QEMU, and Xen, allowing users to create, manage, and monitor virtual machines (VMs) across various platforms. Libvirt supports multiple programming languages and provides a rich set of features for managing VM lifecycles, storage, networking, and more.

2.1.10 Cloud-init

Cloud-init is an open-source tool used for initializing cloud instances during boot time. It allows users to configure and customize cloud instances by executing scripts, applying configurations, and installing packages based on metadata provided by the cloud provider. Cloud-init is widely used in cloud environments to automate the setup of virtual machines and ensure they are ready for use immediately after provisioning.

2.1.11 Avahi

Avahi is an open-source implementation of the Zeroconf protocol, which enables automatic discovery of network services and devices on a local network. It allows applications to discover and communicate with services without the need for manual configuration. Avahi is commonly used in local area networks (LANs) to facilitate service discovery and enable seamless communication between devices.

2.1.12 NixOps

NixOps is a deployment tool for Nix-based systems that allows users to manage and deploy NixOS configurations across multiple machines or cloud providers. It provides a declarative approach to infrastructure management, enabling users to define their entire system configuration in Nix expressions. NixOps supports various deployment targets, including virtual machines, cloud instances, and physical servers, making it a versatile tool for managing Nix-based environments.

3 Analysis

3.1 NixConfig vs NixFlakes

3.2 Base Image: NixOS vs Ubuntu

Avantages - Reproductibilité totale : - Avec NixOS, tout (paquets, configuration système, users, SSH, firewall, services...) est versionné et reconstruit à l'identique sur n'importe quelle machine.

- Pas de « drift » entre les environnements : pas d'état caché, pas de scripts d'init qui peuvent diverger. - Déclarativité : - La configuration (modules Nix) décrit l'état final souhaité, pas une suite d'actions impératives. - Flexibilité : - Ajout/suppression de services, changement de version, ajout de paquets... tout est modulaire et paramétrable. - Sécurité : - Moins de dépendance à l'état antérieur de la VM : chaque build part de zéro. - Facilité d'automatisation : - Génération d'images (qcow2, ISO, etc.) automatisée, sans intervention manuelle ni scripts shell fragiles.

Limites de l'approche Ubuntu + cloud-init + Ansible - Imprévisibilité : - Les images Ubuntu officielles sont parfois modifiées, certains paquets ou configurations changent, ce qui peut casser des scripts. - Scripts impératifs : - Ansible applique des actions, mais l'état final dépend de l'ordre d'exécution, de l'état de l'image, etc. - Tests/rebuilds plus difficiles : - Pour tester un changement, il faut souvent relancer tout le pipeline (Terraform + Ansible), et le résultat n'est pas toujours identique.

3.3 LibVirt vs Tofu

Avantages - Contrôle granulaire et dynamique : - Le wrapper Python permet de piloter la création/destruction de VMs à la volée, d'intégrer de la logique métier, de la validation, etc. - Intégration directe avec la configuration Python : - Le modèle de cluster (Pydantic) est directement utilisé pour générer les VMs, sans traduction dans un autre langage (HCL). - Extensibilité : - Possibilité d'ajouter des hooks, des opérations post-création, de l'orchestration complexe, etc., plus difficile à faire avec Terraform. - Pas de dépendance à un backend cloud : - Libvirt est natif, léger, et adapté au lab/local/dev : pas besoin d'API cloud, ni de provider Terraform parfois instable.

Limites de Terraform - Moins flexible pour l'orchestration dynamique : - Terraform est fait pour décrire un état d'infrastructure, pas pour orchestrer des workflows dynamiques ou répondre à des événements. - Double maintenance (HCL + Ansible + Python) : - Nécessite de maintenir les fichiers HCL (Terraform), les playbooks Ansible, et éventuellement du code Python pour la logique métier.

3.4 Python vs Ansible

Avantages - Typage fort et validation : - Les modèles Pydantic valident la configuration dès le chargement : erreurs détectées avant même le provisionnement. - Unification de la logique : - Toute la logique de configuration, de validation, d'orchestration est dans le même langage, facilement testable. - Réutilisabilité et modularité : - Possibilité de générer dynamiquement des configurations, de les adapter, de les versionner, etc.

Limites d'Ansible - Impératif et fragile : - Les playbooks sont séquentiels, parfois non idempotents, et la validation du résultat est plus difficile. - Multiplication des outils : - Nécessite de maintenir des rôles, des inventaires, des variables, etc., souvent dispersés.

3.5 StaticIP vs DHCP

3.6 Resource manager vs Schedulers

4 Conception

4.1 Holistic Configuration

hydra

4.2 Reproducibility by design

4.3 Automated Provisioning

4.4 VM Configuration

4.5 Image Provisionner

4.6 Declarative vs Imperative Configuration

5 Implementation

5.1 Slow Iteration Cycle

5.2 Nix K8S Hostname Problem

image building, image provisioning, image runtime,

The thing is that I really need to define the hostname at provision time ! I cannot do it at runtime ! otherwise it will fail because nix is immutable. I insist on the fact that I need the hostname to be handled by nix because k8s needs networking.hostname to initialize itself. If we configure another method it will work at the os level but the applications won't be able to use it properly. So I would love to have a way to pass the hostname to virt install somehow and have it in the nix configuration. This is super important because I need only 1 generic image for the worker. Otherwise I would need to build n images for n workers.

You're right, setting the hostname at runtime in a purely Nix-driven, immutable way is challenging because NixOS rebuilds the entire system when configuration changes. However, there are cleaner approaches than the hostname-init script that avoids the reboot loop, while still setting the hostname at "provision time" (before the system fully boots). The key is to leverage systemd's capabilities and avoid modifying the NixOS configuration after the initial installation.

The provisioning workflow for Kubernetes clusters involves distinct phases: build time and provision

time. During build time, the operating system, network, and DNS configurations are established, and certificates for Kubernetes components are generated. Both master and worker node images are configured with these settings. At provision time, the master node is deployed first, generating a secret used for secure communication within the cluster. Worker nodes are subsequently provisioned, joining the cluster using this secret to authenticate their membership. However, a critical issue arises with the static VM images used for worker nodes. These images, created during the build phase, have a fixed hostname embedded within them, such as 'k8s-worker'. When multiple worker nodes are provisioned from the same static image, they inherit this identical hostname, leading to conflicts within Kubernetes. Kubernetes mandates unique hostnames for each node to ensure proper identification and management. Consequently, while the first worker node with a given hostname can successfully join the cluster, additional nodes with the same hostname are rejected, preventing them from joining. This limitation severely impacts scalability and operational functionality, as Kubernetes cannot differentiate between nodes with identical hostnames. To resolve this issue, a dynamic mechanism for hostname configuration is required, enabling each worker node to receive a unique identifier at provision time without relying on static image settings. This adjustment would allow all nodes to join the Kubernetes cluster successfully, enhancing scalability and ensuring robust cluster operations.

While provisioning Kubernetes worker nodes on NixOS using a flake-based configuration and remote automation via Python over SSH, we encountered a reproducible but non-deterministic failure during the execution of the following command: `sudo nixos-rebuild switch --flake /etc/nixos/current`

This command intermittently fails with a non-zero exit code during the `building the system configuration` step when executed as part of an automated script. Surprisingly, some configuration changes—notably the hostname update—appear to take effect, despite the failure. When the exact same command is re-executed manually via SSH, it completes successfully, applying the full configuration and starting all required services.

A deeper analysis revealed the underlying cause is linked to the startup sequence of the Kubernetes services on the worker node, in particular the interaction between `certmgr`, the hostname change, and the bootstrap token mechanism:

- The node's hostname is updated as part of the NixOS rebuild.
- Simultaneously, Kubernetes-related services such as `certmgr`, `kubelet`, and `flannel` are started.
- `certmgr` immediately attempts to request TLS certificates from the Kubernetes master node, using a bootstrap token for authentication.
- This request fails with errors such as `invalid token` or `authentication error. Giving up without retries`, as the master node does not recognize the node (due to the just-updated hostname) or the token is not yet registered.
- Because certificate retrieval fails, dependent services such as `kubelet` and `flannel` cannot

start properly, causing the rebuild to report a failure.

The correct provisioning strategy involves splitting the process into discrete steps:

1. Apply the initial NixOS configuration to set the hostname, but without enabling Kubernetes services.
2. Manually generate and register a valid bootstrap token on the Kubernetes master node.
3. Inject this token into the worker node's configuration and then re-apply the full configuration with Kubernetes services enabled.

This sequence ensures that the hostname is recognized by the master node and that the certificate request from `certmgr` can be authenticated successfully. Notably, private keys used in the certificate process are generated locally and are never transmitted, ensuring secure provisioning. This failure mode highlights the sensitivity of NixOS declarative infrastructure to ordering and timing during initial cluster bootstrap.

5.3 Allocate disk with libvirt with proper size

An intriguing aspect of deploying NixOS images with Libvirt is the handling of disk volumes. The whole purpose of using Libvirt was to be able to provision a virtual machine with an arbitrary disk size defined in the Hydra's configuration and build the OS image one time. However, NixOS's declarative nature and the way it manages disk volumes can lead to unexpected behaviors, particularly when it comes to resizing disks.

I noticed a mismatch between the allocated disk size and the usable filesystem space using the `createXMLFrom` method. Although the volume was created with a size of 256 GiB, the filesystem inside the cloned image remained limited to its original size defined in the image's Nix configuration, which in this case was 16 GiB. This behavior occurs because `createXMLFrom` replicates both the data and metadata of the base image, including the partition table and filesystem, without automatically resizing them to fit the new volume size.

As a result, system tools highlithed in listings 1 reported inconsistent disk usage. The `lsblk` command showed that the partition was still set to 255.8 GiB, and the `df -h` command indicated only 16 GiB of usable space on the filesystem.

This mismatch between the reported disk size and the available capacity was because the file system within the volume was not resized after the partition was extended.

To resolve this issue, I had to ensure that both the partition and the filesystem were resized in a way compatible with NixOS. Since the system is managed declaratively through the Nix configuration, it is not possible to simply resize the file system with tools like `resize2fs`. Instead, I had to adjust the disk configuration directly in NixOS configuration. After rebuilding the system with nixos-rebuild, the

```
[odancona@k8s-master:/etc/nixos/current-systemconfig]$ lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
vda    253:0    0   256G  0 disk
├─vda1 253:1    0   236M  0 part
├─vda2 253:2    0  1007K  0 part
└─vda3 253:3    0 255.8G  0 part /nix/store

[odancona@k8s-master:/etc/nixos/current-systemconfig]$ df -h /
Filesystem      Size  Used Avail Use% Mounted on
/dev/vda3        16G  8.0G  6.7G  55% /
```

Listing 1: Disk size mismatch after volume creation

filesystem was correctly recognized with the full capacity. This approach ensures the disk resizing is consistent with NixOS's declarative model and avoids breaking system assumptions.

6 Tests

6.1 Topic 1: Tests

7 Future Work

8 Conclusion