Transposition d'environnements distribués reproductibles avec NixOS Compose

Quentin Guilloteau, Jonathan Bleuzen, Millian Poquet, Olivier Richard

▶ **To cite this version:**

Quentin Guilloteau, Jonathan Bleuzen, Millian Poquet, Olivier Richard. Transposition d'environnements distribués reproductibles avec NixOS Compose. COMPAS 2022 - Conférence francophone d'informatique en Parallélisme, Architecture et Système, Jul 2022, Amiens, France. pp.1-9. hal-03696485v1

HAL Id: hal-03696485
https://hal.science/hal-03696485v1

Submitted on 16 Jun 2022 (v1), last revised 24 Nov 2022 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Transposition d'environnements distribués reproductibles avec *NixOS Compose*

Quentin Guilloteau, Jonathan Bleuzen, Millian Poquet, Olivier Richard

Univ. Grenoble Alpes, Inria, CNRS, LIG, F-38000 Grenoble France
Prénom.Nom@inria.fr

Résumé
La mise au point des environnements logiciels pour l'expérimentation des systèmes distribuées est un processus itératif et souvent laborieux. Pouvoir tester simplement les environnements sur la machine de l'utilisateur ou sur la plateforme cible rend les cycles de développement plus rapides et fluides. Le caractère reproductible de ces environnements est également primordial pour une contribution scientifique rigoureuse. Nous introduisons dans ce papier *NixOS Compose*, un outil basé sur la distribution *NixOS* pour la génération reproductible d'environnements distribués. Il permet d'effectuer des déploiements virtualisés (docker, qemu) ou physiques (*Grid'5000*) avec une description unique. Nos expériences montrent une amélioration en temps de construction des images de l'ordre de 5 et reconstruction d'un facteur 7 par rapport à Kameleon, la solution usuelle pour construire des images sur la plateforme *Grid'5000*.

Mots-clés : Reproductibilité, Environnements, Systèmes Distribués, *NixOS*

---

1. Introduction

La communauté scientifique traverse une crise de la reproductibilité depuis une dizaine d'années, et l'informatique n'y fait pas exception. En 2015, Collberg et al. [5] ont étudié la reproductibilité de 402 articles expérimentaux publiés à des conférences et journaux en *système*. Chaque article étudié comportait un lien vers le code source qui a permis de réaliser l'expérience de l'article. Sur ces 402 papiers, 46 % n'étaient pas reproductibles. Les causes étaient principalement que *(i)* le code n'était pas accessible, *(ii)* le code ne compilait pas ou ne s'exécutait pas, et *(iii)* les expériences nécessitaient du matériel spécifique.
Dans cet article, nous nous attaquons principalement au problème *(ii)* dans le cadre des systèmes distribués. Notre objectif est de faire en sorte que **toute** la pile logicielle directement impliquée dans l'exécution d'une expérience informatique puisse être compilée, déployée et exécutée telle qu'elle l'est aujourd'hui, que l'on ré-exécute l'expérience demain ou dans 10 ans. Nous souhaitons pour cela exploiter pleinement l'approche déclarative de configuration de système introduite par la distribution Linux *NixOS* [9]. Une configuration *NixOS* est une description reproductible et reconstructible de tout l'environnement logiciel du système, en espace utilisateur comme en espace noyau. Nous étendons ici ce concept à la description de la configuration de tout un système distribué dans un nouvel outil nommé *NixOS Compose*.
NixOS Compose est un outil pour définir et faire évoluer des environnements distribués. Il permet de réaliser des déploiements sur plateformes physiques (*e.g.*, Grid'5000 [1]) comme des déploiements virtualisés (via Docker ou qemu), en exposant **exactement la même interface**

aux utilisateurs. Cette fonctionnalité, couplée au fait qu'introduire une variation dans un environnement et le redéployer soit rapide avec *NixOS Compose*, permet de considérablement améliorer l'expérience utilisateur. Ce point peut paraître négligeable à première vue, mais nous pensons qu'il est primordial que l'expérience utilisateur soit satisfaisante pour que les *bonnes* pratiques en terme de reproductibilité d'expériences soient adoptées par la communauté.

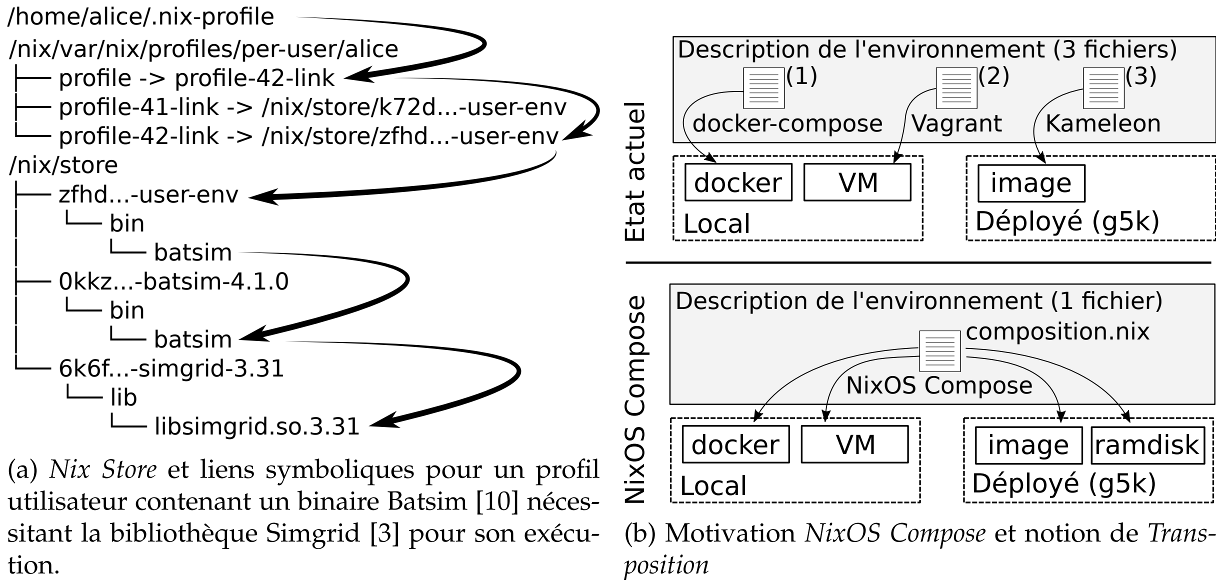
2. État de l'art

2.1. Reproductibilité d'un environnement logiciel local

Il existe plusieurs approches pour encapsuler un environnement logiciel. Une solution serait d'en fournir une image conteneur ou une machine virtuelle. Cela a l'avantage d'être portable, mais introduit un coût de virtualisation. De plus cette virtualisation limite les usages. En effet, il n'est par exemple pas possible avec des conteneurs de choisir la version du noyau Linux.
Courtes et al. [7] mettent en avant les gestionnaires de paquets fonctionnels tels que *Nix* [8] et *Guix* [6] comme de bons candidats pour partager des environnements logiciels complexes entre utilisateurs. Ces deux approches sont similaires, mais *Nix* est le projet avec l'écosystème le plus complet et avec le plus de support de la communauté. Dans la suite nous nous intéresserons donc à *Nix*.
Nix est un gestionnaire de paquets fonctionnel. Chaque paquet est défini avec une fonction *sans effet de bord*. Ainsi, en connaissant les sources d'entrée et cette fonction, *Nix* est capable de reproduire exactement le même paquet, et par conséquent le même environnement. Cela résout partiellement le problème du *Dependency Hell*, ou enfer des dépendances, car *Nix* exige d'expliciter les dépendances (et leur version) nécessaires pour la construction du paquet.
Cette solution se limite aux logiciels en espace utilisateur. Cependant, l'environnement de la machine hôte est également important. La distribution Linux *NixOS* basée sur *Nix*, répond à ce problème. Avec *NixOS*, le système d'exploitation entier (*i.e.*, noyau, applications, fichiers de configurations, services, etc.) est construit par *Nix* à partir d'une expression descriptive.

2.2. Reproductibilité d'un environnement logiciel distribué

En milieu distribué, la notion de reproductibilité se complexifie avec le nombre de machines. En effet, pour assurer la reproductibilité de l'environnement logiciel, il faut déployer un environnement sur les machines. De tels environnements sont encapsulés dans les images avec des outils tels que Kameleon [12]. Cependant, ces images sont coûteuses à construire mais également longues à déployer. Une fois les images déployées, l'expérimentateur doit mettre en place les relations entres les différents noeuds (un montage NFS différent de `/home` entre les noeuds réservés par exemple). Cette étape de configuration est habituellement réalisée après le boot via un script. Ce script, écrit par exemple avec *Execo* [11], n'est usuellement valide que pour une plateforme d'expérimentation. Une solution serait d'encapsuler une partie de cette configuration dans la génération des images.
La mise en place de ces environnements est un processus itératif chronophage. Réduire la durée des cycles de développement est donc un point crucial. Des outils tels que *EnOSlib* [4] ou Vagrant vont dans la bonne direction puisqu'ils permettent de décrire la configuration de l'environnement de manière déclarative en abstrayant les plateformes cibles (*Providers*). Cela rend possible de facilement tester localement les environnements développés. Cependant, ces technologies ne garantissent pas la reproductibilité des piles logicielles.
Une solution est de combiner ces approches avec des outils tels que *NixOS*. Des projets comme *NixOps*, *deploy-rs* ou *Disnix(OS)* permettent d'activer de nouvelles configurations *NixOS* sur des machines cibles, mais uniquement si ces machines sont déjà sous *NixOS*— contrairement à



(a) *Nix Store* et liens symboliques pour un profil utilisateur contenant un binaire Batsim [10] nécessitant la bibliothèque Simgrid [3] pour son exécution.

(b) Motivation *NixOS Compose* et notion de *Transposition*

FIGURE 1 – Schémas explicatif des profiles *Nix* et motivation de *NixOS Compose*

NixOS Compose qui n'a pas cette limitation.

Suite à ce bilan, il nous semble nécessaire de proposer une solution permettant de déployer des environnements reproductibles dans un système distribué avec des cycles de développement rapides.

3. Présentation de *NixOS Compose*

Un **environnement logiciel** est un ensemble de programmes, librairies, configurations et services (modules noyau). Pour qu'un environnement soit reproductible, il est nécessaire que les versions de ses composantes soient connues et également reproductibles.

Nous définissons la **notion de *Transposition*** comme la fonction permettant de déployer un environnement défini de *manière unique* sur plusieurs plateformes de natures différentes. On peut typiquement vouloir d'abord déployer un environnement sur des Machines Virtuelles (VM) locales pour tester et développer, puis déployer l'environnement sur une plateforme distribuée sans travail supplémentaire, en utilisant la même définition de l'environnement.

3.1. Concepts et fonctionnement de *Nix* & *NixOS*

Un paquet *Nix* est représenté par une **fonction** décrivant les sources et dépendances du paquet, ainsi que comment il doit être construit. *Nix* va récupérer ou reconstruire les dépendances puis exécuter, *en isolation*, les commandes pour générer le paquet. Les paquets sont alors entreposés de manière *isolée* et identifiés de manière unique via un hash dans un répertoire spécial appelé le *Nix Store*. Cela permet à différentes versions du même paquet de cohabiter sur la machine hôte. Un **profil** système *Nix* est une configuration du système (paquets, `initrd`, etc). Un profil peut également définir des systèmes de fichiers comme NFS et les monter automatiquement au boot. Une image *NixOS* peut contenir plusieurs profils et *Nix* est capable de passer d'un profil à un autre (*switch*) en changeant des liens symboliques et en redémarrant des services `systemd`. La figure 1a résume les notions de profils et *Nix Store*.

3.2. Concepts de *NixOS Compose*

NixOS Compose se base sur *Nix* et *NixOS* pour appliquer la notion de *Transposition* aux systèmes distribués. Cela permet, comme illustré en figure 1b, d'avoir une description unique d'un environnement tout en pouvant le déployer sur des plateformes cibles de types différents.
Un **rôle** est un type de configuration lié à la mission d'un nœud. Dans le cas d'une expérience du type client-serveur il y aurait deux rôles : client et serveur. Une **composition** est un fichier *Nix* décrivant la configuration *NixOS* de tous les rôles de l'environnement. Enfin, un **déploiement** attribue à chaque nœud le rôle qu'il doit prendre.
Une ***flavour*** est une plateforme cible pour la mise en place de l'environnement. *NixOS Compose* propose déjà plusieurs *flavours* : *(i)* `docker` générant une configuration `docker compose`[1], *(ii)* `vm-ramdisk` pour des VMs qemu, *(iii)* `g5k-ramdisk` générant un `initrd` pour le déploiement rapide en mémoire sans besoin de redémarrage total de la machine cible, ou encore *(iv)* `g5k-image` générant une tarball d'une véritable image pour le déploiement sur *Grid'5000*.
L'utilisatrice choisit une *flavour* en fonction de la plateforme sur laquelle elle se trouve. Ainsi, dans la phase de développement en local elle peut utiliser `docker` ou `vm-ramdisk`.
Certaines *flavours* sont limitées dans leur usage. Des expériences nécessitant de lancer des machines virtuelles ou de changer le noyau Linux par exemple seront impossibles avec `docker`.

3.3. Fonctionnement de *NixOS Compose*

Nix permet à partir d'une expression de générer une configuration *NixOS* notamment les phases de `boot` et `init`. Ces phases sont stockées dans le *Nix Store* et peuvent donc être utilisées par des outils tels que *NixOS Compose*.
NixOS Compose fonctionne en deux étapes : *(i) Construction* : à partir de la composition fournie par l'utilisatrice, et en utilisant les modules, *NixOS Compose* construit avec *Nix* une image comportant tous les profils des rôles défini, puis *(ii) Déploiement* : mise en place de l'environnement construit en fonction de la plateforme cible et activation du profil lié au rôle déployé. La partie de déploiement est réalisée avec Python.

4. Usages comparés de *NixOS Compose*

4.1. Test en environnement local

NixOS Compose propose plusieurs *flavours* pouvant être exécutées localement, par exemple : `docker` et `vm-ramdisk`. Le workflow est le suivant : *(i)* construction de l'image : `nxc build -f docker` par exemple, *(ii)* démarrage : `nxc start`, et enfin *(iii)* connections aux conteneurs/VM : `nxc connect [node]`. Les étapes d'écriture et de tests locaux représentent le processus itératif de la mise en place d'expériences. Réaliser ces étapes de manière locale et légère (docker, VM) aide l'expérimentatrice à l'élaboration rapide de ces dernières.

4.2. Test en environnement distribué

Une fois l'environnement mis au point avec des *flavours* locales, il peut être testé sur un environnement distribué. Une fois sur la frontale *Grid'5000*, *NixOS Compose* propose plusieurs approches pour le déploiement : `g5k-ramdisk` et `g5k-image`. Le workflow est le suivant : *(i)* construction de l'image : `nxc build -f g5k-image` (ou `g5k-ramdisk`), *(ii)* réservation des ressources avec OAR [2] : `oarsub sleep 3600`, *(iii)* déploiement : `nxc start`, et enfin *(iv)* connections aux noeuds : `nxc connect [node]`. Hormis l'étape *(ii)* qui est spécifique à *Grid'5000*, **le workflow est identique** à celui dans un contexte local présenté en section 4.1. Ceci

1. `https://docs.docker.com/compose/`

est l'avantage de la notion de *Transposition*. La figure 3 en annexe résume le workflow global de *NixOS Compose*.

5. Évaluation

Dans cette section nous cherchons à évaluer *NixOS Compose* et à le comparer à Kameleon [12], la solution usuelle pour construire des images sur *Grid'5000*. Cette évaluation portera sur le temps de construction et reconstruction des images ainsi que sur leur taille.

5.1. Dispositif expérimental

Les expériences ont été réalisées sur la grappe `dahu` de la plateforme *Grid'5000*. Les machines de cette grappe ont 2 CPUs Intel Xeon Gold 6130 avec 16 coeurs/CPU et 192 Gio de mémoire. Les disques sont des SSD SATA Samsung MZ7KM240HMHQ0D3 avec une capacité de 240 Go formatés en `ext4`.

5.2. Protocole expérimental

Le but de l'expérience est d'estimer les gains de temps permis par *NixOS Compose* pour la mise au point d'un environnement. Cela comprend la construction, la modification et la reconstruction d'une image. Pour cela nous construisons d'abord une image minimale (`base`) avec *NixOS Compose* et Kameleon. Nous mesurons le temps de construction de l'image ainsi que sa taille et celle de l'ensemble des résultats produits (*artefacts*). Puis, nous ajoutons le paquet `hello` à la recette de l'image (`base + hello`) et construisons à nouveau en mesurant identiquement.
Pour *NixOS Compose* nous prenons soin d'avoir un *Nix Store* local vide avant de construire l'image de `base`. Kameleon dispose d'un mécanisme de cache indirect via le proxy HTTP Polipo, mais nous n'avons pas réussi à faire marcher l'interaction entre Kameleon et Polipo sur *Grid'5000*. Polipo n'est plus maintenu car son utilité est devenue discutable, puisque aujourd'hui presque aucun trafic n'est envoyé de manière non chiffrée, y compris les données transférées par les gestionnaires de paquets. Nous pensons donc que la désactivation de ce cache dans notre expérience n'a qu'un impact très mineur sur les résultats que nous observons.
Nous avons choisi `grid5000/debian11-x64-nfs` comme image de base pour Kameleon car elle fournit l'essentiel des applications pour une expérience distribuée.

5.3. Temps de construction des images

Les résultats sont visibles en figure 2. Nous avons testé la construction des images sur le NFS `/home` du site *Grid'5000* de Grenoble et en local (sur `/tmp`). Cela permet de comparer la situation actuelle (NFS) à une plus idéale. Nous pouvons constater que le temps de construction d'une image est bien moindre avec *NixOS Compose*. De plus, grâce à l'utilisation du *Nix Store*, le coût de reconstruction de l'image après y avoir rajouté le paquet `hello` est très réduit. Pour *Nix*, le temps de construction passe du simple au double lorsqu'on utilise NFS. Le résultat est plus surprenant pour Kameleon puisque la construction est plus rapide en NFS qu'en local, mais est cohérente avec nos observations de bande passante IO sur la grappe `dahu`. En effet, Kameleon fait de nombreuses requêtes IO de petite taille qui ont une meilleure bande passante en NFS qu'en local puisque *(i)* NFS agrège les petites requêtes IO en de plus grosses, et *(ii)* les serveurs NFS ont probablement de meilleurs disques que les machines de calcul.

5.4. Taille des artefacts et de l'image déployée

Nous mesurons la quantité d'artefacts produits lors de la construction des images. C'est-à-dire les fichiers nécessaires créés pour forger l'image. Kameleon produit plus d'artefacts que *NixOS Compose*. Il est intéressant à noter que le produit de la construction avec Kameleon n'est pas

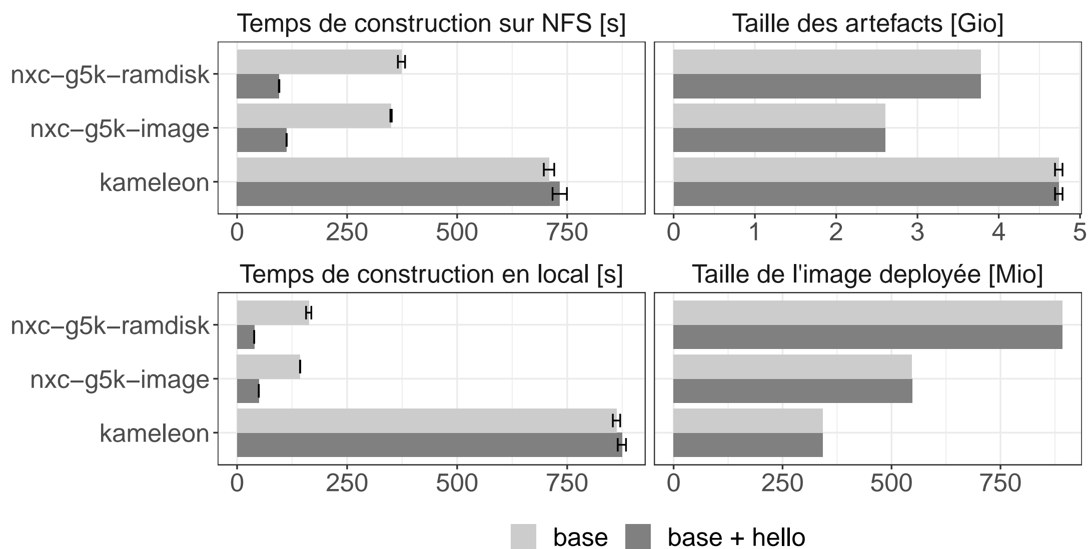


FIGURE 2 – Comparaison entre Kameleon et *NixOS Compose* (`nxc`) lorsque l'on construit une image de `base`, puis que l'on construit une nouvelle image (`base + hello`) qui y ajoute le paquet `hello`. Les barres d'erreurs représentent les intervalles de confiance à 99 % (5 réps).

constant entre deux constructions. Ceci s'explique par la présence de log. L'image produite est cependant identique. En revanche, *NixOS Compose* reconstruit *exactement* les mêmes artefacts.

Nous nous intéressons maintenant à la taille de l'image à déployer. Ici, l'image produite par Kameleon est plus légère que pour *NixOS Compose*. Nous expliquons cela par le manque de minimalisme actuel de notre image de base qui contient une quantité de pilotes matériels inutiles pour les expérimentations (par exemple des pilotes d'imprimantes).

Les temps de déploiement pour les *flavours* `g5k-ramdisk` et `g5k-image` sont respectivement inférieurs à la minute et de l'ordre de plusieurs minutes.

6. Conclusion

Nous avons introduit la notion de *Transposition* ainsi que notre outil *NixOS Compose* appliquant cette notion aux environnements logiciels distribués. *NixOS Compose* permet de réduire considérablement le temps de mise en place d'environnements distribués (*i.e.*, construction, modification et reconstruction des images) par rapport à Kameleon. En se basant sur *Nix* et *NixOS*, les images produites par *NixOS Compose* sont complètement reproductibles. Nous prenons également avantage du *Nix Store* comme un cache local, permettant de reconstruire et faire évaluer rapidement les images.

Cependant, certaines pistes d'améliorations apparaissent. La taille des images *NixOS Compose* ne sont pas encore minimales. Les performances de construction des images sont très liées aux performances du NFS `/home`. Différentes implémentations du *Nix Store* sur *Grid'5000* seraient intéressantes à explorer (*e.g.*, machine dédiée pour la construction et pour héberger un *Nix Store* commun). Permettre aux utilisateurs de construire des images pour des plateformes *Cloud* (par exemple AWS) est également envisagé. Nous voulons enfin étendre notre comparaison à d'autres outils de l'état de l'art comme *EnOSlib* et *Terraform*.

Remerciements

Les expériences présentées dans ce papier ont été réalisées sur le testbed *Grid'5000*, supporté par un groupe d'intérêt scientifique hébergé par l'Inria et incluant le CNRS, RENATER et plusieurs universités ainsi que d'autres organisations (voir `https://www.grid5000.fr`).

A. Un exemple de composition

Dans cette section nous présentons un exemple de composition pour *k3s*[2], une version légère de *Kubernetes*[3] pour l'orchestration de conteneurs. *k3s* a besoin de deux *rôles* : le *serveur* et les *agent*.
Pour la configuration du serveur, nous devons ajouter *k3s* à l'environnement, configurer le service `systemd` associé. et ouvrir un port (6443) du serveur. Pour l'agent, il faut de la même manière ajouter *k3s* à l'environnement et indiquer l'adresse du serveur et le token d'identification. La configuration *Nix* du serveur est alors la suivante :

```
{ pkgs, ... }:
let
  k3sToken = ".....";
in {
  nodes = {
    server = { pkgs, ... }: {
      environment.systemPackages = with pkgs; [ k3s gzip ]; # Paquets
      networking.firewall.allowedTCPPorts = [ 6443 ]; # Ports à ouvrir
      services.k3s = {
        enable = true;                        # Active k3s
        role = "server";                      # Définit le role
        extraFlags = "--token ${k3sToken}"; # Info pour l'enregistrement
      };
    };
    agent = { pkgs, ... }: {
      environment.systemPackages = with pkgs; [ k3s gzip ]; # Paquets
      services.k3s = {
        enable = true;                        # Active k3s
        role = "agent";                       # Définit le role
        serverAddr = "https://server:6443"; # Adresse du server
        token = k3sToken;                     # Token à utiliser
      };
    };
  };
}
```

Notons que utilisons le terme `nodes` dans la composition ci-dessus pour être rétro-compatibles avec les *NixOS tests*[4]. Ces tests permettent de démarrer des machines virtuelles qemu avec les environnements définis et d'exécuter un script Python pour s'assurer de la bonne configuration.

2. `https://k3s.io/`
3. `https://kubernetes.io/`
4. `https://nixos.org/guides/integration-testing-using-virtual-machines.html`

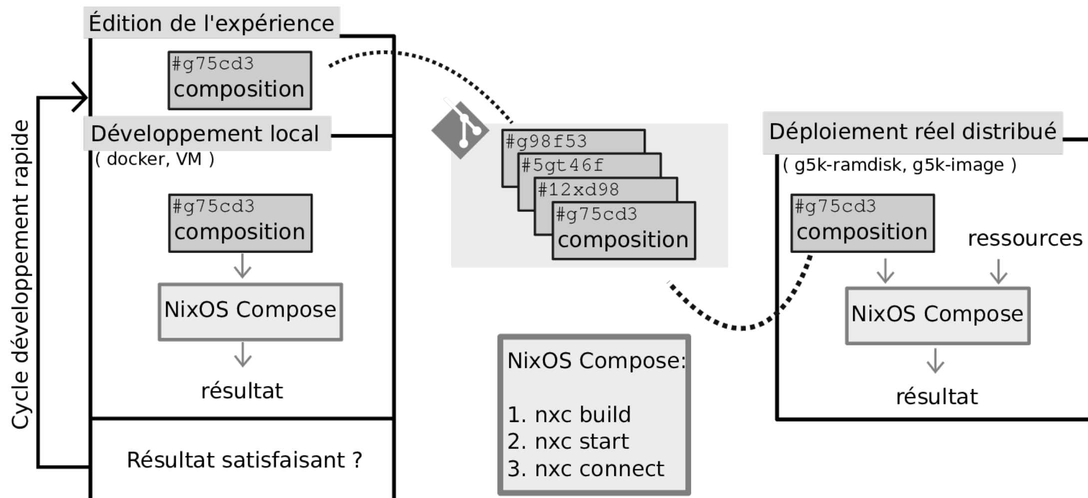


FIGURE 3 – Récapitulatif du workflow de *NixOS Compose*

Une fois la composition construite avec `nxc build` nous pouvons la déployer. Dans ce cas, nous voulons potentiellement avoir plusieurs *agents* et un unique *serveur*. Pour cela nous pouvons fournir un ficher yaml à la commande `nxc start` décrivant la quantité et les hostnames de tous le nœuds.
Par exemple s'il on veut 3 agents et un serveur, le fichier de déploiement ressemblerait à ce qui suit :

```
# en définissant juste les quantités
server: 1
agent: 3
# ou en définissant les hostnames
server: 1
agent:
    - foo
    - bar
    - baz
```

Notez que si aucun fichier de déploiement n'est fournit, *NixOS Compose* déploiement une instance de chaque rôle (un serveur et un agent ici).

B. Schéma du workflow de *NixOS Compose*

La figure 3 représente le workflow de *NixOS Compose*.

Bibliographie

1. Balouek (D.), Carpen Amarie (A.), Charrier (G.), Desprez (F.), Jeannot (E.), Jeanvoine (E.), Lèbre (A.), Margery (D.), Niclausse (N.), Nussbaum (L.), Richard (O.), Pérez (C.), Quesnel (F.), Rohr (C.) et Sarzyniec (L.). – Adding virtualization capabilities to the Grid'5000 testbed. *In : Cloud Computing and Services Science*, éd. par Ivanov (I. I.), van Sinderen (M.), Leymann (F.) et Shan (T.), pp. 3–20. – Springer International Publishing, 2013.

2. Capit (N.), Da Costa (G.), Georgiou (Y.), Huard (G.), Martin (C.), Mounie (G.), Neyron (P.) et Richard (O.). – A batch scheduler with high level components. – In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, pp. 776–783 Vol. 2, Cardiff, Wales, UK, 2005. IEEE.
3. Casanova (H.), Giersch (A.), Legrand (A.), Quinson (M.) et Suter (F.). – Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, vol. 74, n10, juin 2014, pp. 2899–2917.
4. Cherrueau (R.-A.), Delavergne (M.), van Kempen (A.), Lebre (A.), Pertin (D.), Balderrama (J. R.), Simonet (A.) et Simonin (M.). – EnosLib : A Library for Experiment-Driven Research in Distributed Computing. *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, n6, juin 2022, pp. 1464–1477.
5. Collberg (C.), Proebsting (T.) et Warren (A. M.). – Repeatability and Benefaction in Computer Systems Research - A Study and a Modest Proposal. 2015, p. 68.
6. Courtes (L.) et Wurmus (R.). – Reproducible and User-Controlled Software Environments in HPC with Guix. *In : Euro-Par 2015 : Parallel Processing Workshops*, éd. par Hunold (S.), Costan (A.), Giménez (D.), Iosup (A.), Ricci (L.), Gomez Requena (M. E.), Scarano (V.), Varbanescu (A. L.), Scott (S. L.), Lankes (S.), Weidendorfer (J.) et Alexander (M.), pp. 579–591. – Cham, Springer International Publishing, 2015.
7. Courtès (L.). – Functional Package Management with Guix. *arXiv :1305.4584 [cs]*, mai 2013.
8. Dolstra (E.), de Jonge (M.) et Visser (E.). – Nix : A Safe and Policy-Free System for Software Deployment. 2004, p. 14.
9. Dolstra (E.) et Löh (A.). – Nixos : A purely functional linux distribution. *SIGPLAN Not.*, vol. 43, n9, sep 2008, p. 367–378.
10. Dutot (P.-F.), Mercier (M.), Poquet (M.) et Richard (O.). – Batsim : a Realistic Language-Independent Resources and Jobs Management Systems Simulator. – In *20th Workshop on Job Scheduling Strategies for Parallel Processing*, Chicago, United States, mai 2016.
11. Imbert (M.), Pouilloux (L.), Rouzaud-Cornabas (J.), Lebre (A.) et Hirofuchi (T.). – Using the EXECO toolbox to perform automatic and reproducible cloud experiments. – In *1st International Workshop on UsiNg and building ClOud Testbeds (UNICO, collocated with IEEE CloudCom 2013*, Bristol, United Kingdom, décembre 2013. IEEE.
12. Ruiz (C.), Harrache (S.), Mercier (M.) et Richard (O.). – Reconstructable Software Appliances with Kameleon. *ACM SIGOPS Operating Systems Review*, vol. 49, n1, janvier 2015, pp. 80–89.