Infrastructures As Code
Fondamentaux

Plan

JOUR 1

Introduction

I. Origines et évolution du DevOps

II. Infrastructure as Code : Principes et concepts

III. Premiers outils d'Infrastructure as Code

IV. L'émergence des outils modernes : Ansible et Terraform

V. Limites et contraintes de l’IaC

-- Travaux pratiques - Terraform --

JOUR 2

Quizz

VI. Vers l'automatisation complète avec IaC et DevOps

Bibliographie

-- Travaux pratiques - Ansible --

Ménage de fin de TP

Infrastructure as Code

Introduction

Quelques définitions

DevOps

DevOps permet la coordination et la collaboration des rôles autrefois cloisonnés (développement, opérations informatiques, ingénierie qualité et sécurité) pour créer des produits plus performants et plus fiables. En adoptant une **culture** DevOps ainsi que des pratiques et outils DevOps, les équipes peuvent mieux répondre aux besoins des clients, accroître la confiance suscitée par les applications qu'elles développent, et atteindre plus rapidement les objectifs de leur entreprise.
https://azure.microsoft.com/fr-fr/resources/cloud-computing-dictionary/what-is-devops

Infrastructure as Code

L'Infrastructure as code (IaC) (littéralement : « infrastructure en tant que code ») est un ensemble de mécanismes permettant de gérer, par des fichiers descripteurs ou des scripts (code informatique), une infrastructure (informatique) virtuelle.
https://fr.wikipedia.org/wiki/Infrastructure_as_code

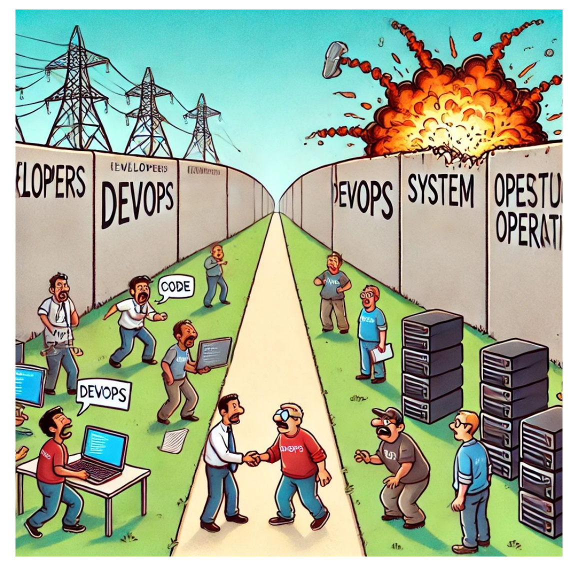
Objectifs

Accélérer le cycle de développement, améliorer la **collaboration** et **automatiser** la gestion des infrastructures

Continuous Delivery

Culture de la Collaboration

Infrastructure as Code



Infrastructure as Code

DevOps - Origines et évolution

DevOps - Origines et évolution

1. **Le contexte pré-DevOps : les silos traditionnels**
    - Séparation stricte entre les équipes de développement et d'exploitation
    - Risques et inconvénients : lenteur, inefficacité, manque de communication

DevOps - Origines et évolution

1. **L'apparition de l'agilité**
   - Influence des méthodologies agiles (années 2000)
   - DevOps comme réponse à la **nécessité d'intégrer des pratiques agiles aux opérations**

HISTORY OF AGILE

1970
WATERFALL MODEL CREATED

1991
SCRUM CONCEIVED

1995
SCRUM CODIFIED AND PUT INTO USE

2001
AGILE MANIFESTO CREATED

IMPLEMENTATION
TESTING
MAINTENANCE

Agile
Define
Test
Analyze
Backlog
Deploy

https://the-agilesphere.com/2016/09/18/the-history-of-agile/

DevOps - Origines et évolution

1. **Naissance du mouvement DevOps (2008-2009)**
   - Conférences et initiatives clés : DevOpsDays en Belgique (2009)
   - Premiers concepts : collaboration, intégration continue, livraison continue (CI/CD)
   - Objectifs : automatisation, rapidité, amélioration de la qualité des déploiements

https://everythingdevops.dev/a-brief-history-of-devops-and-its-impact-on-software-development/

DevOps - Origines et évolution

1. **L'essor du cloud computing et la standardisation**
   - Adoption massive du cloud : AWS, Google Cloud, Azure
   - Importance de l'infrastructure élastique et à la demande pour le DevOps

vmware

amazon web services

docker

Virtualizati
on
1998

Clou
d
200
6

Containe
rs
2013

MicroServic
es

147M Internet
users

1 093M Internet
users

2
802M
Internet
users

3 835M+
Internet users

Source :
https://www.slideshare.net/AmazonWebServices/why-microservices
https://www.internetworldstats.com/emarketing.htm

Infrastructure as Code

Principes et concepts

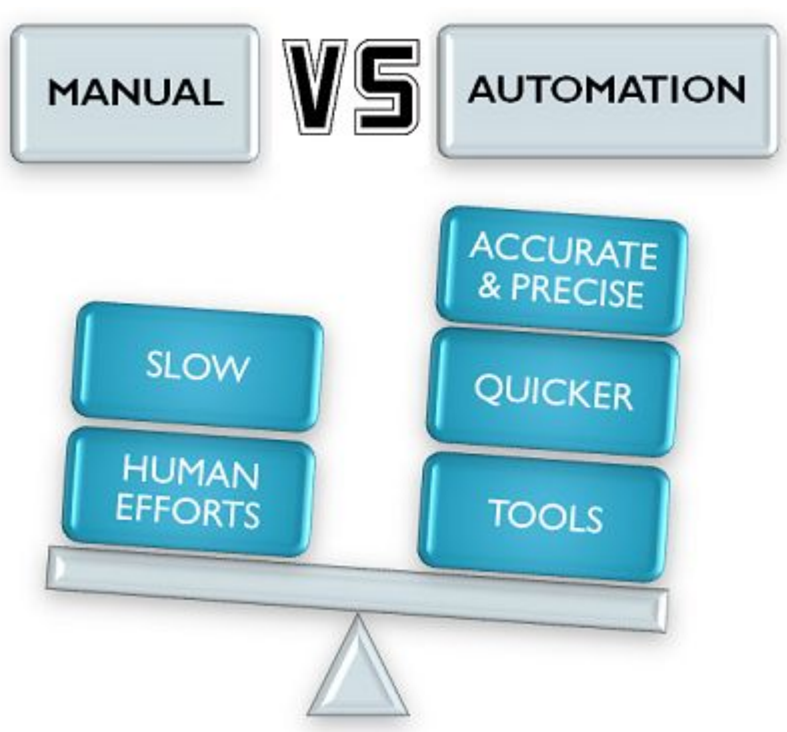
Principes et concepts

Problèmes traditionnels de gestion d'infrastructure

- **Installation d’un serveur**
  - Système d’exploitation
  - Configuration réseau et DNS
  - Services de bases (SSH, NTP, sauvegarde, sécurité, monitoring …)
  - Services haut niveau : base de données, mail, firewall N7, gestion de containers …
  - Applications
- **Bare metal : gros serveur multi-services (typical pet)**
- **Virtual machine : spécialisée par rôle, disponible pour les devs (on my machine) , multiplication des installations (copie de machines, …)**
- **Problèmes divers**
  - Actions manuelles et temps passé
  - Documentation, répétabilité et debug
  - Re Provisioning en cas de crash server / clonage
  - Gestion des mises à jour, du day2 et de l’auditabilité
  - Robustesse, limites et maintenance des scripts
  - Dépendance aux équipes infrastructure pour disposer de ressources

Principes et concepts

1. **Naissance du concept d'Infrastructure as Code**
   - Gestion des ressources d'infrastructure via du code : versionnement, réutilisation, et automatisation



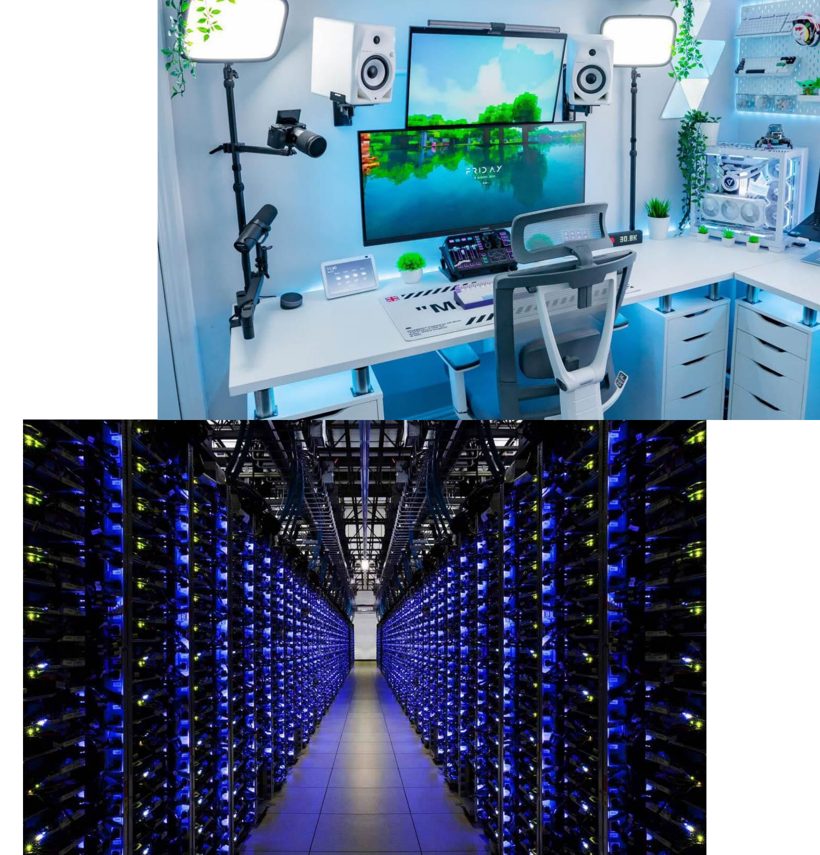
Principes et concepts

1. **Les principes clés d'IaC**
   - Reproductibilité : créer des environnements cohérents et fiables
   - Versioning : suivi des modifications via des systèmes de contrôle de version
   - Idempotence : application de configurations de multiples fois sans changer l'état des ressources
   - Déclaratif vs impératif
   - Infrastructure immuable

Principes et concepts

Reproductibilité

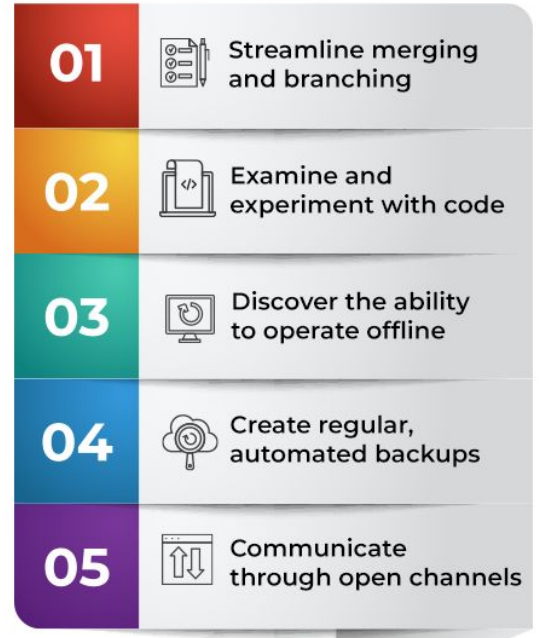
- Automatiser pour reproduire à l'identique
- Pet vs cattle



Principes et concepts

Versioning : suivi des modifications via des systèmes de contrôle de version

BENEFITS OF VERSION CONTROL

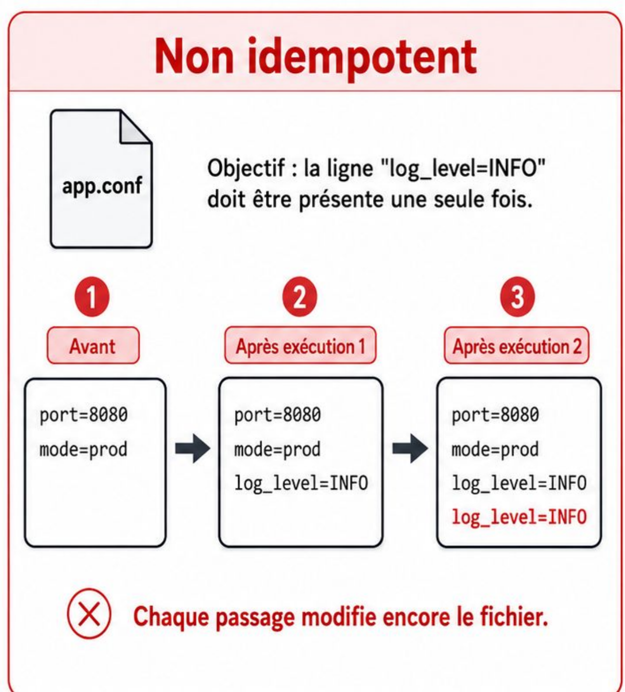


https://www.spiceworks.com/tech/devops/articles/what-is-version-control/

Principes et concepts

Idempotence (ré-entrance)

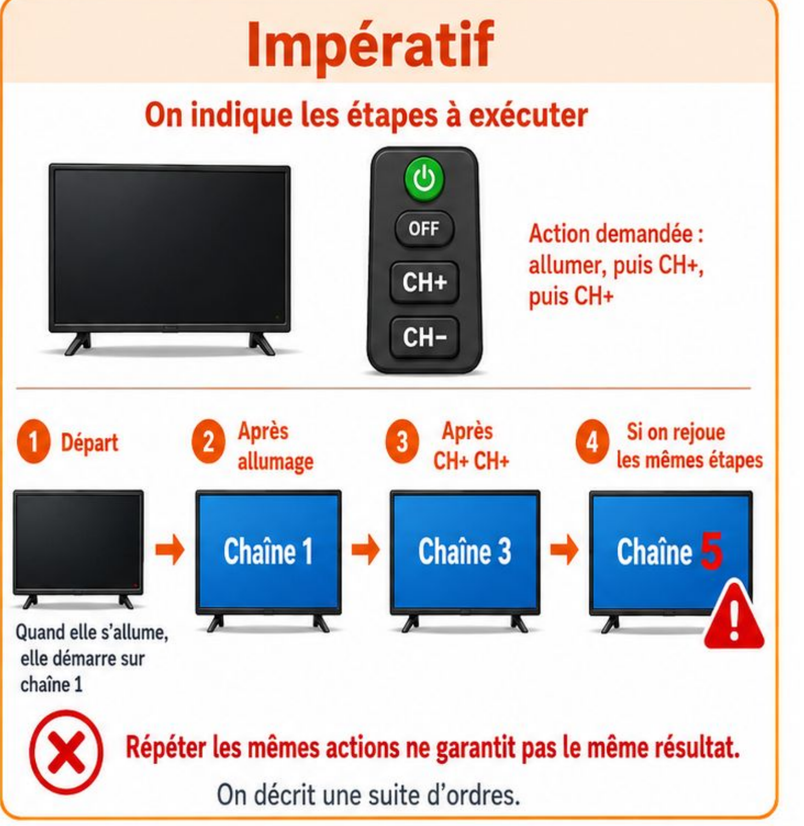
- **rm <file>**
- **rm -f <file>**



Principes et concepts

Déclaratif et impératif

- **Notion d'état désiré et état courant (réconciliation)**
- **Laisser l'outil décider des étapes et dépendances pour arriver à l'état désiré**



Principes et concepts

Infrastructure immuable

Update App

Mutable Infrastructure

Instance A
App v1.0
OS v1.0

Reboot

Instance A
App v1.1
App v1.0
OS v1.0

Immutable Infrastructure

Instance A
App v1.0
OS v1.0

Destroy
X
+
Provision

Instance B
App v1.1
OS v1.0

https://www.opsramp.com/guides/why-kubernetes/infrastructure-as-code/

Principes et concepts - DevOps et IaC

1. **L'intégration d'IaC dans la culture DevOps**
   - Automatisation des pipelines de déploiement et de gestion de la configuration
   - Gestion du code Iac en mode DevOps

Bénéfices de l'IaC: amener le dev à l'ops

- **Grâce à Infra as Code, les ops deviennent des codeurs d'infra en lieu et place d'experts système & réseau à l'aise en shell.**

- **On amène les pratiques de développement côté ops :**
  - Versioning / Release management (GIT)
  - Revue de code par les pairs (merge / pull requests)
  - Test automatisés
  - Intégration continue
  - Tests de performances

- **Mêlé au volet "Culture de la collaboration" le bénéfice pour l'entreprise est énorme.**
  - Contribution forte au PRA / PCA
  - Capacité à faire des déploiements progressif (diminutions des bugs).
  - Capacité à remonter bien plus vite les machines crashées
  - Agilité apportée à l'infrastructure qui devenait un frein aux méthodes agiles côté dev.
  - Globalement les entreprises qui y sont passées y voient un vrai intérêt économique.

Infrastructure as Code

Premiers outils (historique)

Premiers outils d'Infrastructure as Code

Les ancêtres de l'IaC : Shell scripts et gestionnaires de configuration

- Utilisation de scripts Bash/Python pour automatiser les configurations
  - Moins d'erreurs / oublis, plus facile à lancer que suivre une doc
  - Plus rapide mais développement long (RAZ du serveur pour tester).
- Problèmes : scripts non reproductibles, dépendance à des systèmes spécifiques
  - Ces scripts finissent souvent par être très longs donc très complexes à maintenir
  - Idempotence / Répétabilité
  - Complexe d'assurer la compatibilité entre packages (versions)
  - Un script par OS / version, ou un script très complexe
  - Les fichiers de configuration des différents service sont très difficiles à modifier proprement via du scripting
  - Templating très limité (envsubst), python jinja n'arrive que beaucoup plus tard
  - Configuration réseau / DNS par script souvent dangereuse, en cas d'échec ou d'erreur dans le script on peut perdre l'accès à la machine
  - A part dans quelques contextes particuliers on est toujours dépendant de la disponibilité des ops

CFEngine

- Le plus vieux, l'acteur historique (**1993**) créé par Mark Burgees
- Écrit en C, il est encore aujourd'hui plus performant que les autres systèmes qui reposent sur Ruby ou Python.
- Fonctionne avec un serveur et un agent par machine qui communiquent entre eux.
  - On doit donc d'abord déployer un serveur, et l'installation de chaque machine commence par le déploiement d'un agent.
- Basé sur un DSL (*Domain Specific language*) interne
- Nécessite une phase de compilation du DSL pour l'exécution.

Avantages :

- modèle d'autonomie : chaque nœud corrige sa configuration vers l'état désiré.
- Déclaratif
- Utilisé dans de grands environnements pour assurer la gestion automatique des serveurs à travers des politiques de configuration.
- Impact : a posé les bases conceptuelles des outils modernes (idempotence)

Inconvénients :

Très verbeux, communauté en disparition, dépendance au serveur, peu de réutilisabilité du code

- Pour aller plus loin :
  - http://articles.mongueurs.net/magazines/linuxmag95.html

CFEngine : Exemple

```
# cat /var/cfengine/inputs/cf.editfiles
control:

  actionsequence = ( editfiles shellcommands )
  AddInstallable = ( sysctl )

  # Cfengine refuse d'éditer les fichiers au dessus d'une
  # certaine taille et la limite est souvent un peu basse
  editfilesize = ( 30000 )

classes:

  # on définit une classe postgres s'il est installé
  postgres = ( IsDir(/usr/lib/postgresql/) )

editfiles:

  # on ajoute cfengine dans les services si il n'y est pas
  # déjà
  any::
    {
      /etc/services
      SetLine "cfengine 5308/tcp         # cfengine port"
      # on ajoute la ligne précédente si l'expression
      # rationnelle suivante n'est pas trouvée dans le fichier
      AppendIfNoLineMatching "^cfengine.*"
    }

  # on augmente shmall et shmmax pour les serveurs postgresql
  postgres::
    {
      /etc/sysctl.conf
      # on ajoute ces lignes si elles n'existent pas déjà
      AppendIfNoSuchLine    "kernel.shmall = 134217728"
      AppendIfNoSuchLine    "kernel.shmmax = 134217728"
      # et on définit la classe sysctl
      DefineClasses "sysctl"
    }

shellcommands:

  # si sysctl est défini alors on recharge /etc/sysctl.conf
  sysctl::
      "/sbin/sysctl -p"
```

Puppet

- Sorti en **2005**, Inspiré de CFEngine, écrit en Ruby
- Outil référence de l'écosystème Redhat
- Mode de fonctionnement principal : un serveur et un agent par machine qui communiquent entre eux.
- Peut aussi fonctionner sans serveur (ce n'est pas son mode privilégié/encouragé)
  - Nécessite de faire un système maison via SSH
- Basé sur un DSL interne
- Le DSL (Manifests) est interprété par Ruby. Templates ERB.
- Intègre son propre système de tests automatisés

Avantages :

Déclaratif, Modules custom en ruby, forte adoption ces dernières années / grosse communauté & documentations, strict

Inconvénients :

DSL Très verbeux, manque de souplesse, disparaît progressivement au profit d'Ansible

- Pour aller plus loin :
  - https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-112/Les-sysadmins-jouent-a-la-poupee
  - http://www.puppet.com/
- **Mots clefs : Manifests, Classes**

Puppet

```
node 'myserver' {
    class { 'postgresql::server': }

    postgresql::server::db { 'db':
        user     => 'username',
        password => 'password',
    }
}
```

```
if $osfamily != 'windows' {
  class { 'rsyslog::client': }
}
```

```
node 'myserver' {
    user { 'username':
      ensure => 'present',
      groups => ['wheel'],
    }
}
```

```
[root@master manifests]# pwd
/etc/puppetlabs/code/environments/production/modules/firewall/manifests
[root@master manifests]# tree
.
├── init.pp                 # class firewall
├── linux
│   ├── archlinux.pp        # class firewall::linux::archlinux
│   ├── debian.pp           # class firewall::linux::debian
│   ├── gentoo.pp           # class firewall::linux::gentoo
│   └── redhat.pp           # class firewall::linux::redhat
├── linux.pp                # class firewall::linux
└── params.pp               # class firewall::params
```

Chef

- Sorti en **2009**, nouveau paradigme programmatique, écrit en Ruby
- Mode de fonctionnement au choix : solo ou serveur.
- Basé sur un DSL interne + Ruby
- Permet de mélanger du DSL à du code ruby pur
- Gros progrès dans la réutilisabilité du code (Définitions, ressources …)
- Templates ERB.

Avantages :

Les développeurs l'apprennent très vite, très souple, l'utilisation de Ruby au sein d'un cookbook permet d'effectuer des tâches complexes impossibles à réaliser avec un DSL.

Inconvénients :

Sa souplesse est aussi son inconvénient : mélanger du DSL et du Ruby amène vite à du code illisible. Temps d'exécution assez lent (Ruby). En train de disparaître au profit d'Ansible.

- Pour aller plus loin :
  - https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-112/Les-sysadmins-jouent-a-la-poupee
  - http://www.puppet.com/
- **Mots clefs : Rôles, Cookbooks**

Chef

```ruby
unless node.datadog[:enabled]
  package 'datadog-agent' do
    action :remove
  end

  if node.monit.enabled
    monit_service 'datadog-agent' do
      action :delete
    end
  end

  return
end

package 'datadog-agent' do
  version node.datadog[:old_version]
  action :purge
end
```

```ruby
service "cron" do
  supports :status => true, :restart => true, :reload => true
  action [ :enable, :start ]
end
```

```ruby
proxys = find_proxy_nodes.to_a.
         sort_by { |name, server| name }.
         map { |name, server| server}
proxy_index = proxys.find_index local_node
```

```ruby
template "/etc/cron.d/reload_node_#{node.fasterize_engine.proxy[:name]}_cluster" do
  owner "root"
  mode 0644
  source "reload_node_cluster.crontab.erb"
  cookbook "nodejs_app"
  variables :service_name => node.fasterize_engine.proxy[:name],
            :reload_hour => 6 + proxy_index / 6,
            :reload_minute => (proxy_index % 6) * 10,
            :reload => true
  notifies :restart, resources(:service => "cron")
end
```

Salt Stack

Sorti en **2011**, inspiré de Puppet, écrit en Python

Outil de gestion de configuration open source, particulièrement utilisé dans des environnements nécessitant une grande scalabilité et une vitesse d'exécution rapide.

Mode de fonctionnement principal : architecture maître/minions (un serveur central contrôle plusieurs nœuds) avec possibilité de fonctionner en mode sans serveur (Salt SSH).

Basé sur le modèle d'exécution à distance et des fichiers d'état (State Files) écrits en YAML combiné avec Jinja pour les templates.

Utilise ZeroMQ pour des communications ultra-rapides entre le maître et les minions.

Intègre un large éventail de modules pré-construits (appelés "grains", "pillars" et "states") pour la gestion d'infrastructure et d'automatisation, ainsi qu'un orchestrateur de tâches avancé.

Avantages :

Architecture flexible (mode maître/minions ou SSH), rapidité d'exécution, forte scalabilité, gestion fine des événements et automatisation à grande échelle, bonne compatibilité multi-plateformes.

Inconvénients :

Peut devenir complexe à configurer et à maintenir dans de grands environnements, documentation parfois inégale, moins "clé-en-main" que des alternatives comme Ansible.

Pour aller plus loin :

https://saltproject.io/

Salt Stack

1. tmux

```
{% set configs = pillar.get('supervisor', {}).get('configs', []) %}
supervisor:
  pkg:
    - installed
  service:
    - running
    - enable: true
    - watch:
      - file: /etc/supervisor/conf.d/*.conf

{% for config in configs %}
/etc/supervisor/conf.d/{{config}}.conf:
  file.managed:
    - source: salt://supervisor/config.jinja
    - template: jinja
    - context:
      config: {{config}}
    - mode: 644
    - require:
      - pkg: supervisor
{% endfor %}
```

Premiers outils d'Infrastructure as Code

Limites de ces premiers outils

1. Complexité des configurations

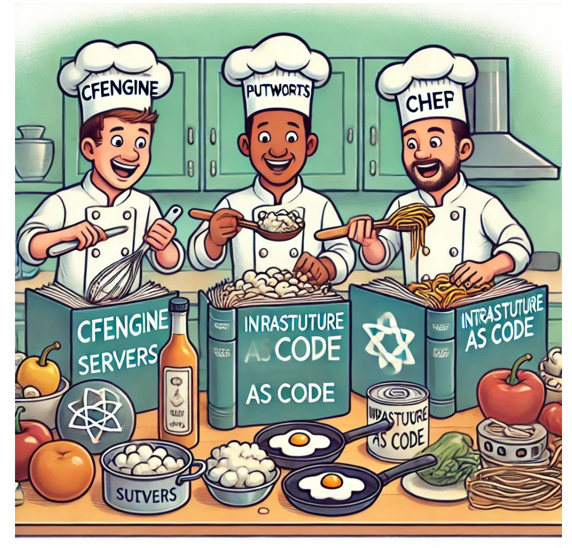
- **Langages déclaratifs complexes :** DSL spécifique ou langage peu commun en infra (Ruby)
- **Des configurations redondantes et difficiles à maintenir**
- **Multiples niveaux d'abstraction**

2. Difficultés à scaler

- **Problèmes de performance** dans des environnements larges
- **Latence dans les changements**

3. Gestion centralisée nécessaire

- **Infrastructures maître/esclave** : Cette approche entraînait une dépendance à ce serveur maître.
- **Single point of failure (SPoF)** : En cas de défaillance du serveur maître,
- **Besoins en infrastructure supplémentaire** : mise en place de serveurs spécifiques pour héberger l'outil lui-même



Infrastructure as Code

L'émergence des outils modernes : Ansible et Terraform

L'émergence des outils modernes : Ansible et Terraform

Ansible

Sorti en **2012**, écrit en Python

- Outil de gestion de configuration et d'automatisation d'infrastructure, popularisé par sa simplicité d'utilisation et son fonctionnement sans agent.
- Mode de fonctionnement principal : exécution par SSH (pas d'agent à installer sur les machines gérées), ce qui le rend particulièrement facile à déployer.
- Basé sur un langage déclaratif simple, utilisant des fichiers YAML appelés "Playbooks" pour définir les tâches et les configurations à appliquer.
- Ansible est également utilisé pour l'orchestration de tâches complexes et le déploiement d'applications multi-nœuds.
- Dispose d'un vaste écosystème de modules natifs pour interagir avec différents systèmes et services cloud.
- Poussé et supporté par un acteur majeur de l'open-source en entreprise : Redhat

Avantages :
Simplicité et rapidité de mise en œuvre, pas d'agent, extensibilité via des modules Python, forte adoption dans le monde DevOps, bonne documentation et large communauté.
Inconvénients :
Peut devenir lent pour des tâches complexes sur un grand nombre de machines en raison de son mode d'exécution par SSH, moins adapté à des environnements nécessitant un contrôle d'état constant comme Puppet ou Chef.

L'émergence des outils modernes : Ansible et Terraform

Terraform

Sorti en **2014**, écrit en Go

- Outil de gestion d'infrastructure as code, principalement utilisé pour provisionner et gérer des infrastructures cloud (comme AWS, Azure, GCP), mais peut aussi gérer des environnements on-premise.
- Mode de fonctionnement principal : déclaratif, basé sur des fichiers de configuration écrits en HCL (HashiCorp Configuration Language), permettant de définir l'infrastructure désirée.
- Fonctionne de manière indépendante des fournisseurs cloud et permet de gérer des infrastructures multi-cloud avec un seul outil.
- Utilise un "State File" pour suivre l'état de l'infrastructure et appliquer les changements de manière incrémentale, avec un mécanisme de planification avant l'application pour valider les modifications.

Avantages :
Support multi-cloud, excellent pour des environnements hybrides, gestion de versions d'infrastructure, modularité avec les "Modules", communauté active, intégration avec de nombreux providers.
Inconvénients :
Gestion de l'état centralisé qui peut parfois causer des problèmes de synchronisation, syntaxe HCL parfois déroutante pour les débutants, nécessite des compétences en infrastructure pour des configurations avancées, **ne fonctionne que si une API est disponible**.

L'émergence des outils modernes : Ansible et Terraform

Comparaison Ansible vs Terraform

Critère	Ansible	Terraform
Langage	YAML (Playbooks)	HCL (HashiCorp Configuration Language)
Mode de Fonctionnement	Sans agent (via SSH)	Déclaratif, gère un fichier d'état
Domaine Principal	**Configuration des serveurs** (gestion des logiciels, fichiers, services)	**Provisioning d'infrastructures** (création de ressources cloud)
Gestion d'État	Sans état (idempotence gérée par la tâche elle-même)	Suivi d'état via un fichier de state
Avantages	Facile à déployer, large communauté, modules natifs	Très adapté aux infrastructures complexes, versionnement intégré
Inconvénients	Moins performant pour le provisioning d'infrastructure	Moins adapté à la gestion fine de la configuration des serveurs

Usage recommandé :

- **Terraform** pour **provisionner** des infrastructures (cloud, réseaux, machines).
- **Ansible** pour **configurer** les systèmes (services, packages, fichiers).
- … même si chaque outil tend à essayer de couvrir le périmètre de son concurrent

Terraform alternatives

- **Août 2023**
  - Terraform est passé en BSL (Business Source Licence)
  - La licence BSL, n'est pas libre. Vous ne pouvez pas commercialiser de produits concurrents à HashiCorp en se basant sur le code source.

- **Regain d'intérêt pour des solutions concurrentes**
  - Open TOFU : https://opentofu.org/ (Fork 100% compatible)
  - Pulumi : https://www.pulumi.com/ (Langages Python, GO, Java et passerelle pour les providers terraform)

Infrastructure as Code

Limites et contraintes de l’IaC

Limites & Contraintes de l'IaC

- **Release management : Le code d'infra doit suivre en version le code applicatif.**
  - Il faut donc aussi créer un système de versions sur le code d'infra, qui doit suivre celui des applications déployées.

- **L'outillage de tests est encore pauvre.**
  - Il existe quelques framework de tests (Serverspec, Inspec, Bats …)
  - Ils permettent de tester qu'un package est installé, qu'un service est actif, qu'un port est en écoute
  - Mais dès qu'on veut tester des choses un peu plus complexe on envoi des commandes shell et analyse leur retour en le parsant via du code.
    - On en vient vite à se faire ses propres outils.

- **Optimisations**
  - Sur beaucoup de systèmes l'infra as code est utilisé en "live" pour ajouter des machines en cas de pic (auto scaling).
  - On arrive vite à devoir écrire beaucoup, beaucoup de code / configuration.
  - Il faut faire attention au code écrit, on doit pouvoir l'exécuter sur une machine de prod active sans phagocyter toutes ses ressources.
    - Ou être capable de désactiver la machine, la mettre à jour, la réactiver (automatiquement)
  - Le compromis entre vitesse de déploiement et code qui fait tout est parfois complexe à trouver.

Limites & Contraintes de l’IaC

- **Gestion des mise à jour :**
  - La mise à jours de packages / services / OS est un sujet réellement complexe en Infra as Code
    - On pourrait vouloir mettre un “apt-get upgrade” dans son infra as code, mais doit-on réellement laisser le serveur se mettre à jour tout seul ? En réalité on a tendance à fixer les versions.

- **Désinstallation**
  - Ecrire du code qui désinstalle tout
  - Penser à retirer ce code quand c’est fini …
  - Débat sur l’écriture de code de désinstallation pour une seule utilisation.
  - Ce code doit marcher même sur une machine ou il n’y a rien à faire.

- **Le mythe du rollback**
  - Infra As Code devrait théoriquement permettre de revenir en arrière.
  - Dans les faits cela implique de prévoir ce code de retour en arrière.
    - Il existe quelques tentatives d’outillage
      - Chef-rewind (Permet de surcharger des ressources).
  - Dans la plupart des cas, on profite du fait que notre code est versionné : on réinitialise puis réinstalle la machine en reculant de n commits.

Limites & Contraintes

Le piège ultime : le coût du cloud

- **Infra as Code permet de créer beaucoup de ressources très vite**
  - **Et de les oublier …**
  - Tout ce que l'on automatise en création doit l'être en destruction.
    - Y compris dans les procédures internes.
      “Oh j'ai oublié de lancer la commande de destruction de mon infrastructure de tests de performances iso-prod le mois dernier…”. 50 000€ imprévus.
    - On va même jusqu'à automatiser la destruction de ressources inutilisées (ce qui n'est pas toujours simple à déterminer).

- **Facturation à la minute/seconde/heure peut être piégeuse**
  - Le coût à l'heure paraît faible mais 1 mois = 720 à 744 heures.
    Et on ne prend rarement qu'une seule VM ou un cluster de base de données à un seul noeud….

- **Ressources “on-demand” vs “reserved”**
  - Tout réserver est contre productif quand on fait de l'auto-scaling
    - Il faut donc faire des calculs fins pour savoir ce qu'on réserve
  - Serverless complexifie encore plus ces estimations budgétaires.

- **Services managés saint-graal si cher …**
  - Le secret de ces services ? Votre cloud provider fait massivement de l'infrastructure as code
  - Toujours comparer aux coûts d'exploitation
  - De nombreuses entreprises ont sur EC2 ou concurrent des services qui existent en managés sur le même Cloud.

Limites & Contraintes : coûts, suite

Il faut donc tout optimiser :

- Auto scaling à tous les niveaux
  - L'infrastructure doit constamment s'adapter à la charge réelle
  - Si l'infrastructure grossit beaucoup c'est bon signe pour le business, mais à part quelques géants du web personne n'est sous pleine charge 24/24 365/365 donc pourquoi payer l'infra maximale toute l'année, tous les jours y compris la nuit ?
- La bande passante en sortie (vers internet) qui est facturée au Go.
- Les échanges au sein du même sous-réseau interne sont gratuits mais, entre plusieurs zones de disponibilités, ils sont facturés au prix fort. (La haute disponibilité d'une plateforme devient ruineuse si on optimise pas son architecture pour localiser au maximum les échanges réseaux.)

- Ne pas oublier que : **adapter une architecture à l'infra as code puis à du Cloud optimisé aux petits oignons peut prendre des mois (**voir des années pour des projets mastodontes.)
  - Tout ce code est long à écrire / tester / éprouver
  - L'entreprise doit drastiquement changer d'organisation avec toute ces automatisations
    - Les étapes validations/signatures managériales sont un des freins les plus courants à une automatisation complète
    - Certains collaborateurs voir même managers peuvent être directement attaqués/menacés par l'arrivée d'Infra as Code qui vient les remplacer

Limites & Contraintes : panel de compétences

- **Infra as Code c'est l'automatisation**
  - Du provisionning / deprovisionning
  - De l'installation / configuration des machines
  - Du monitoring / de la collecte et analyse de métriques
  - Des prévisions budgétaires
  - d'APIs tierces
  - de services tierces
  - …

- En plus de devoir maîtriser **l'infrastructure et le système** un développeur infra as code doit avoir de bonnes bases en **réseau** et un minimum de savoir faire **en sécurité.**
  - Le SRE doit en plus de ces compétences être un architecte capable d'inventer des solutions en mode dégradés (Bdd ou service indisponible) , de la reprise sur erreur … ce qui demande d'agir à tous les niveaux de l'architecture et donc d'avoir une vision technique d'ensemble.
- Il doit aussi souvent savoir coder et écrire des tests dans un à plusieurs langages pour les outils qu'il faudra créer pour en intégrer/orchestrer d'autres.
  - Cela donne encore plus de sens au mot "DevOps"

- **Vu la largeur du panel de compétences requis, il est indispensable d'être redoutable dans le parcours et la recherche de documentation car un cerveau humain "normal" ne peut pas retenir tout ça.**
- **Attention au syndrome du copier/coller de code non maîtrisé sur les docs Stack Overflow !**

Infrastructure as Code

Travaux pratiques - Terraform

Infrastructure as Code

Jour 2

Questionnaire day1

Concepts vus (et retenus ?)

Questionnaire - day1

- **Qu'est ce que l'idempotence ?**
  - Le fait de rendre deux ressources automatisées identiques
  - Le fait de pouvoir lancer deux fois de suite l'outil d'infra as code
  - Une operation a le meme effet qu'on l'applique une ou plusieurs fois
  - Une fonction d'infra as code
- **Le declaratif en infra as code …**
  - Représente l'état final attendu
  - Représente les étapes pour arriver à l'état final
  - Concerne les variables de mon infra as code
  - est un DSL utilisé dans certains outils
- **L'impératif en infra as code est bien imagé par**
  - Les scripts maisons
  - ansible
  - terraform
  - aws

Questionnaire - day1

- **L'auto scaling consiste à**
  - Gérer l'elasticité de la plateforme en permettant à un cloud d'ajouter / supprimer des machines en fonction de métriques
  - Rendre son code d'infra compatible avec un nombre variable de machines
  - Surveiller la charge pour être prêt à ajouter des serveurs à la main
  - Déployer ses serveurs dans un cloud public
- **La gestion de version permet de**
  - Discuter avec ses collègues d’une modification avant déploiement
  - D’utiliser le même numéro de version pour le code applicatif et infrastructure
- **Terraform permet**
  - De gérer les ressources "bas niveau" de tous type et de s'intégrer à n'importe quelle API
  - D'installer et configurer les serveurs
  - D'adapter l'atmosphère d'une planète dans un jeu vidéo

Questionnaire - day1

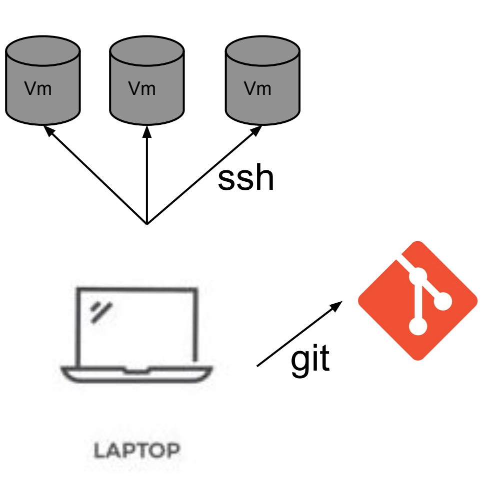
- **Infra as code permet**
  - D'automatiser tout ce qui gravite autour des applications
  - Uniquement de déployer et configurer les serveurs
  - De se passer de Devops ou de SRE dans son équipe
- **Un développeur infra as code n'a pas besoin de savoir programmer**
  - Vrai
  - Faux
- **Infra as code est indispensable pour rentrer dans une démarche CI / CD**
  - Oui
  - Non
- **Les outils d'infra as code gèrent les rollback**
  - Oui
  - Non
- **Infra as Code ne nécessite pas de tests automatisés**
  - Vrai
  - Faux

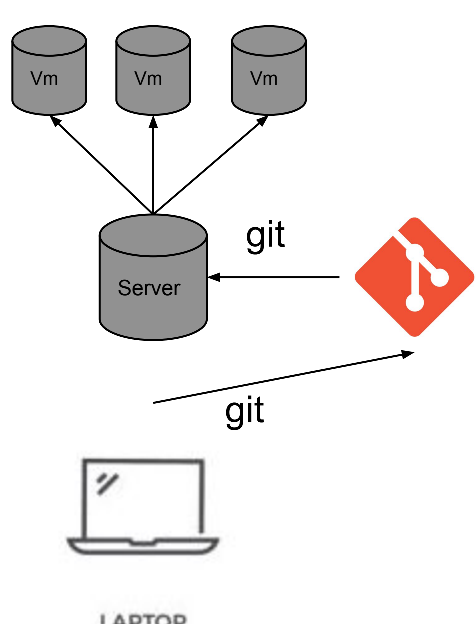
Infrastructure as Code

Vers l'automatisation complète avec IaC et DevOps

Vers l'automatisation complète avec IaC et DevOps

1. **CI/CD et intégration de l'IaC dans les pipelines**
   - Notion de mode solo / serveur (impacts d'un mode hybdride)





- **Attention aux droits nécessaires dans chaque cas (ne pas devenir un rebond ayant les droits sur tout le SI)**
- **Moins rapide pour itérer en mode server**
- **Mode solo à bannir en production**
- **Accès aux logs et auditabilité renforcés en mode server**
- **Orchestration des différents codes (manuel en mode solo, pour server ?)**

Vers l'automatisation complète avec IaC et DevOps

1. **CI/CD et intégration de l'IaC dans les pipelines**
   - Distinction CI et CD
   - IaC intervient plutôt sur CD …
   - … mais le build d'une image IaaS ou PaaS est plutôt en CI

TEST
COMMIT
CODE
RELATED CODE
UNIT TEST
BUILD
INTEGRATION TESTS
CI PIPELINE
REVIEW
STAGING
PRODUCTION
CD PIPELINE

Vers l'automatisation complète avec IaC et DevOps

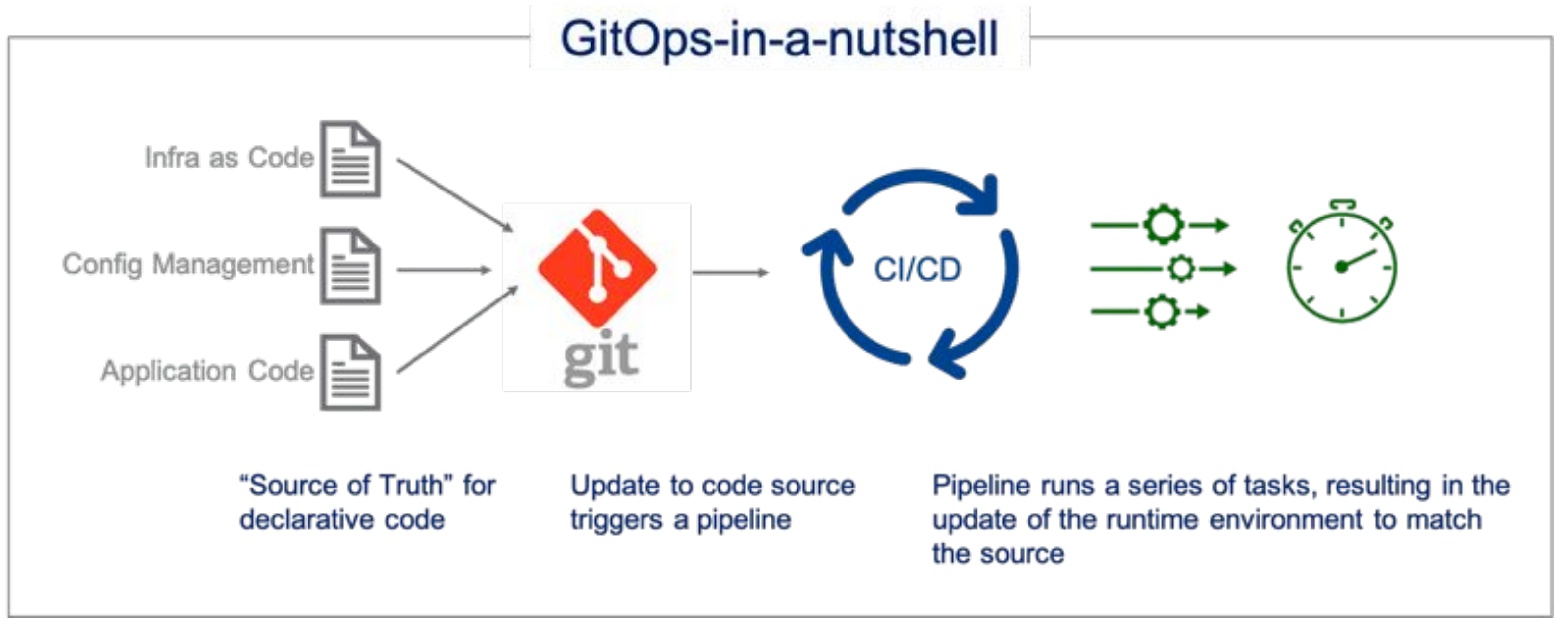
1. **CI/CD et intégration de l'IaC dans les pipelines**
    - Déploiements automatisés de l'infrastructure avec des outils comme Jenkins, GitLab CI
    - Tests automatiques des configurations IaC avant mise en production
    - Limitation des droits nécessaires (jenkins slave, Gitlab Runners)

INTEGRATION
AUTOMATED TEST
LOAD TEST
QA TESTING
FUNCTIONAL TEST
OS TEST
QA APPROVALS
UAT
STAGING
PRODUCTION
DISASTER RECOVERY

https://www.linkedin.com/pulse/building-continuous-delivery-pipeline-using-jenkins-inedo/

Vers l'automatisation complète avec IaC et DevOps

1. **IaC et GitOps : convergence vers la gestion via git**
   - Utilisation de Git comme source de vérité pour l'infrastructure
     i. Mécanismes collaboratifs et validation (pull request / merge request et approval)
   - Automatisation de la gestion des infrastructures via des commits et des pipelines de déploiement



https://dev.to/rsiv/achieve-gitops-on-day-one-with-iac-automation-1eb9

2 types d’outils (fonctions) principaux pour l’IAC

Gestion du provisionning :

Création/destruction de l’infrastructure

- Suivi de l’état de ce qui est déployé avec un state file.
- Cible : VMs, réseaux, stockage, bases de données…
- Exemple : Terraform, Pulumi, AWS CloudFormation

Gestion de la configuration :

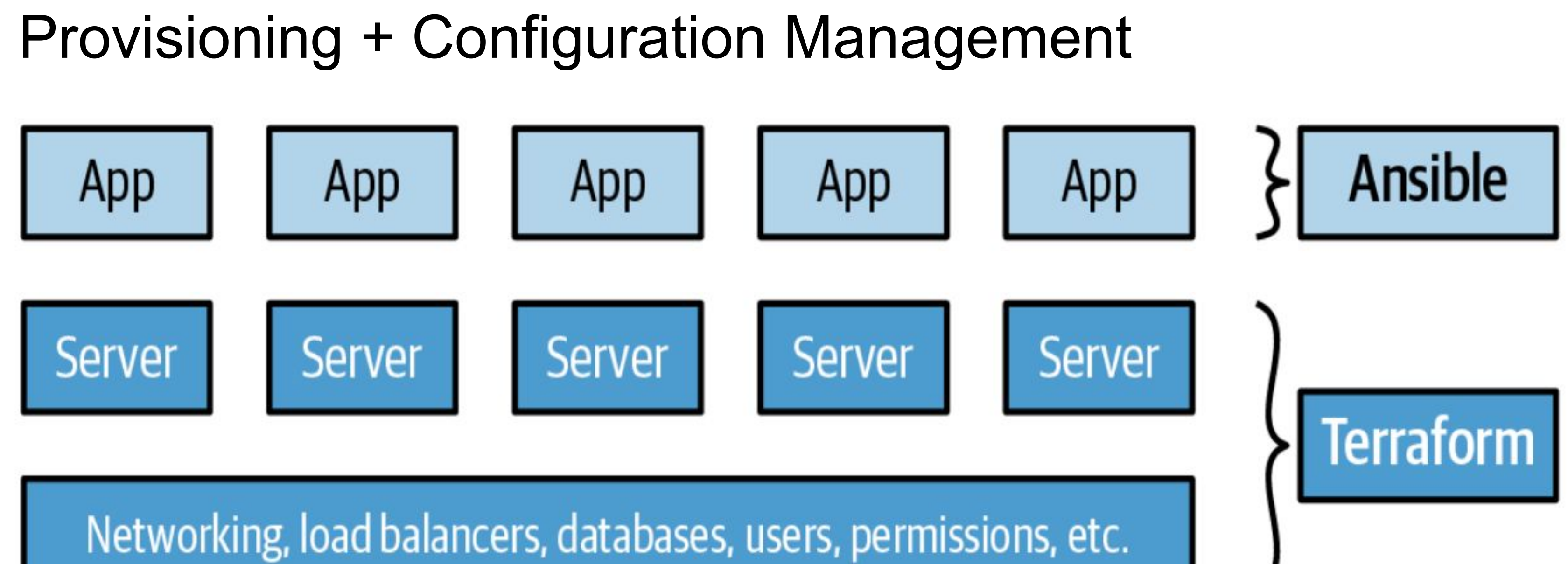
Configuration des serveurs, services ..

Composants cibles : OS, paquets, fichiers, services, utilisateurs

- 2 architectures selon les outils :
  - Agent en mode pull : Chef, Puppet
  - Agentless (serveur qui push): Ansible

Quelques exemples d'architectures IAC (1/3)

Provisioning + Configuration Management



- \+ : Pas d'infrastructure à déployer pour ces 2 outils
- \+ : Fonctionnent bien ensemble (tags Terraform -> inventaire Ansible)
- \- : Maintenance plus difficile si l'infrastructure grossit beaucoup.

Quelques exemples d'architectures IAC (2/3)

Provisioning + Server Templating

VM | VM | VM | VM | VM — HashiCorp Packer

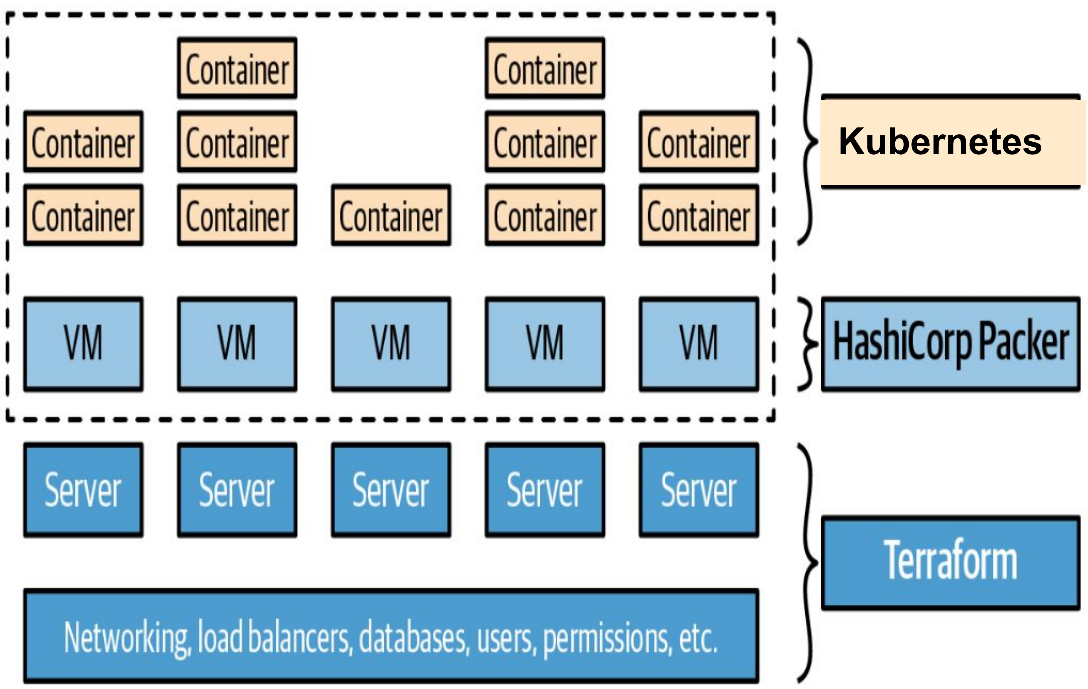
Server | Server | Server | Server | Server

Networking, load balancers, databases, users, permissions, etc. — Terraform

+ : Infrastructure immuable
- : Stratégies de déploiement avec Terraform limités (pas de blue-green natifs)
- : Les VMs peuvent prendre du temps à builder et se redéployer.

Quelques exemples d'architectures IAC (3/3)

Provisioning + Server Templating + Orchestration



- + : Les images container buildent rapidement.
- + : Stratégies de déploiement de Kubernetes (blue-green etc ..)
- - : Complexité de l'infrastructure (K8S) et des couches à maîtriser pour le debug.

Choix des outils

	Chef	Puppet	Ansible	Pulumi	CloudFormation	Heat	Terraform
Source	Open	Open	Open	Open	Closed	Open	Open
Cloud	All	All	All	All	AWS	All	All
Type	Config mgmt	Config mgmt	Config mgmt	Provisioning	Provisioning	Provisioning	Provisioning
Infra	Mutable	Mutable	Mutable	Immutable	Immutable	Immutable	Immutable
Paradigm	Procedural	Declarative	Procedural	Declarative	Declarative	Declarative	Declarative
Language	GPL	DSL	DSL	GPL	DSL	DSL	DSL
Master	Yes	Yes	No	No	No	No	No
Agent	Yes	Yes	No	No	No	No	No
Paid Service	Optional	Optional	Optional	Must-have	N/A	N/A	Optional
Community	Large	Large	Huge	Small	Small	Small	Huge
Maturity	High	High	Medium	Low	Medium	Low	Medium

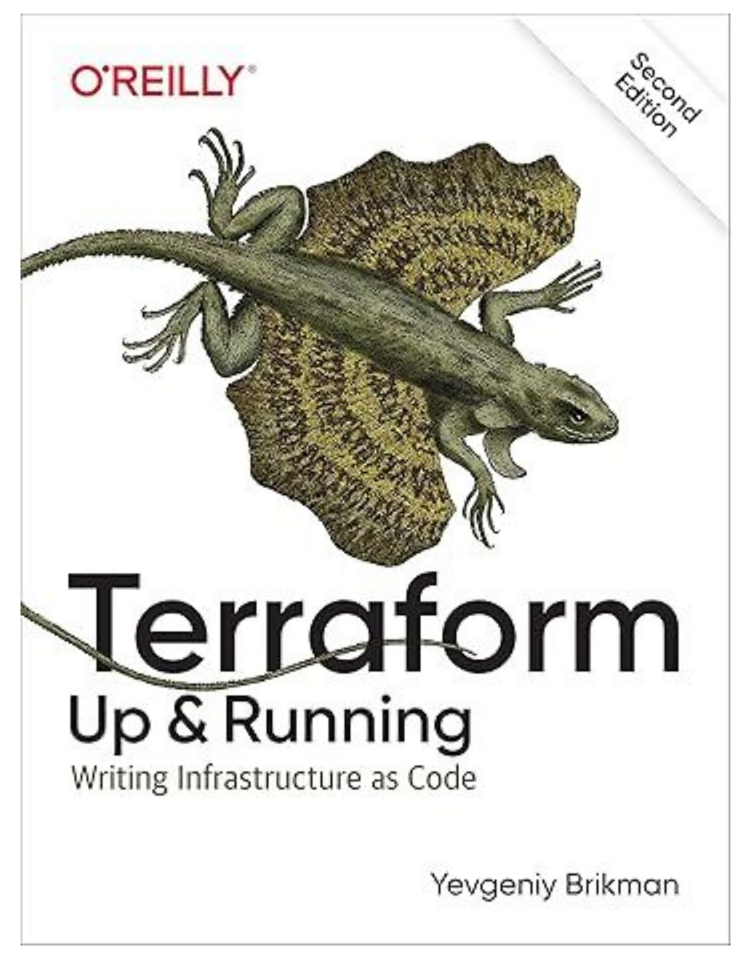
Infrastructure as Code

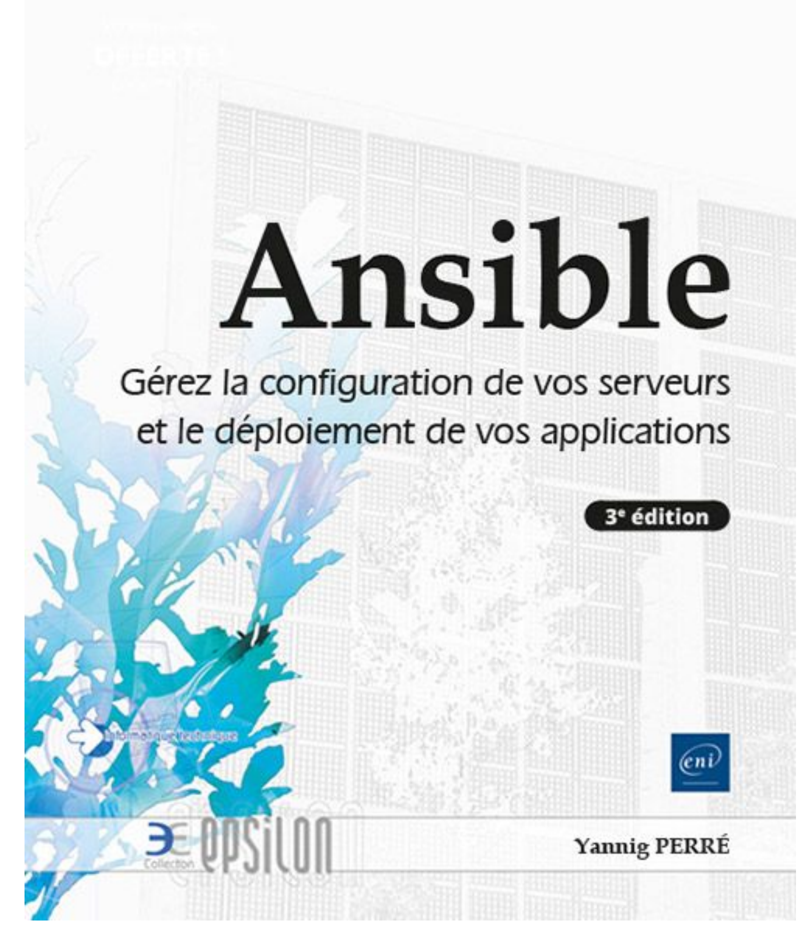
Travaux pratiques - Ansible

Infrastructure as Code

Bibliographie

Bibliographie





Cas concret DEMOBOARD

Exemple concret - demoboard

Internet

Demoboard - Terraform initial AWS architecture

AWS Region

VPC 10.1.0.0/16

Availability Zone A

Availability Zone B

Availability Zone C

Internet Service

Public subnet 10.1.10.0/24

NAT GW + EIP

Public ALB lb_front
HTTP :80 -> front0/front1

EC2 front0 nginx

Public subnet 10.1.11.0/24

NAT GW + EIP

EC2 front1 nginx

Security groups
- SSH :22
- HTTP :80
- API :8000
- PostgreSQL :5432
- Redis :6379

Private subnet 10.1.20.0/24

Private subnet 10.1.21.0/24

Internal ALB lb_api
HTTP :8000 -> api0/api1

EC2 api0 FastAPI

EC2 worker0 Python

EC2 api1 FastAPI

EC2 worker1 Python

EC2 db PostgreSQL

EC2 redis Redis

Empty at initial deployment

Reserved for TP extension: add subnets and replicas

Exemple concret - demoboard

Personal laptop

web browser

Virtual Machine

web browser

frontend (80)

worker-service

api-service (8000)

redis

Database

Ménage en fin de TP

terraform destroy

Peut-être
Cédric Klapisch (1999)