

DevOps

Cours 5: Initiation aux microservices & orchestration de conteneurs

Antoine Balliet

antoine.balliet@dauphine.psl.eu

Dans l'épisode précédent ...

Infrastructure as Code

- Suite cours Terraform (démon)
 - Provider GCP
 - terraform plan / terraform apply
 - Compréhension du state
 - *depends_on* : spécifier l'ordre de traitement des ressources
- TP Terraform
 - Utilisation d'un projet exemple de Google
 - CI / CD pour appliquer changements d'infrastructure
 - Stockage du state dans bucket
 - Documentation terraform
 - Modules terraform

Critique du TP Infrastructure as Code

Fonctionnement du code et de la CI / CD de l'exemple pas clair 🐣👉

Quels problèmes avons-nous constaté pendant le TP Infrastructure as Code?

Reproductible ? 🤔

Dans quel ordre construire notre CI/CD:
docker build après terraform ? 🔍

Environnement de dev ?

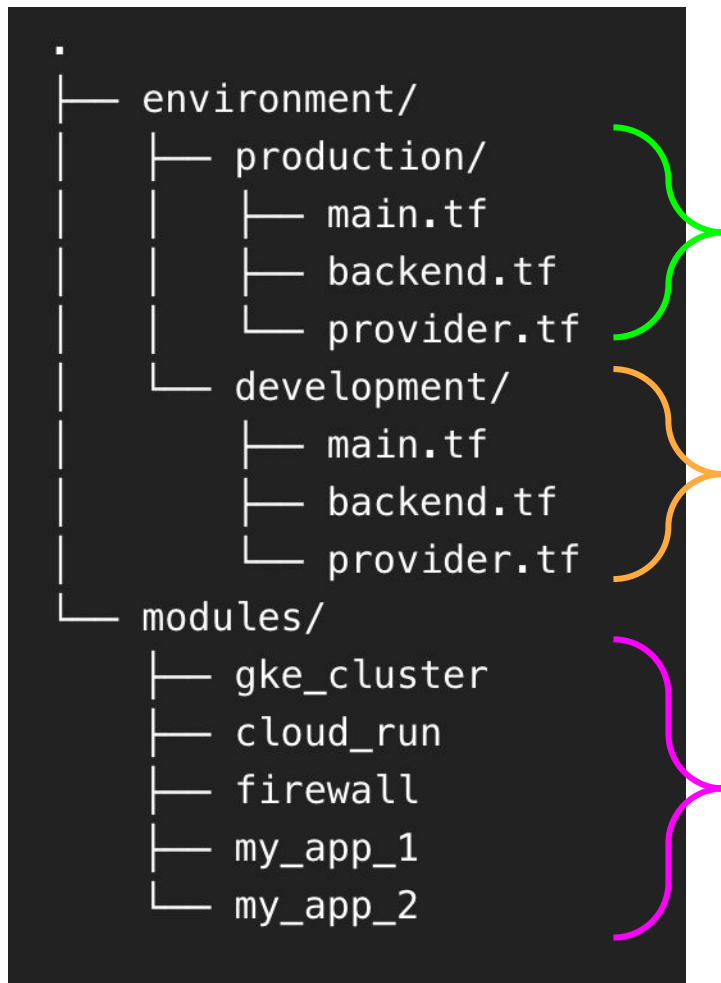


Gérer les versions et environnements du code d'infrastructure

- **Code / “Environnement” Terraform vs “branches” git**
 - Le code est versionné dans git
 - Les versions du code qui coexistent sont matérialisées dans des branches
 - Les “environnements” dans le jargon Terraform correspondent à des espaces **isolés** d'infrastructure
 - On les matérialise en général pas des dossiers séparés
 - La CI / CD peut appliquer des règles suivant la branche (version de notre code git) sur laquelle on a fait des modifications

Gérer les versions et environnements du code d'infrastructure

- Structure “standard” d'un projet utilisant terraform



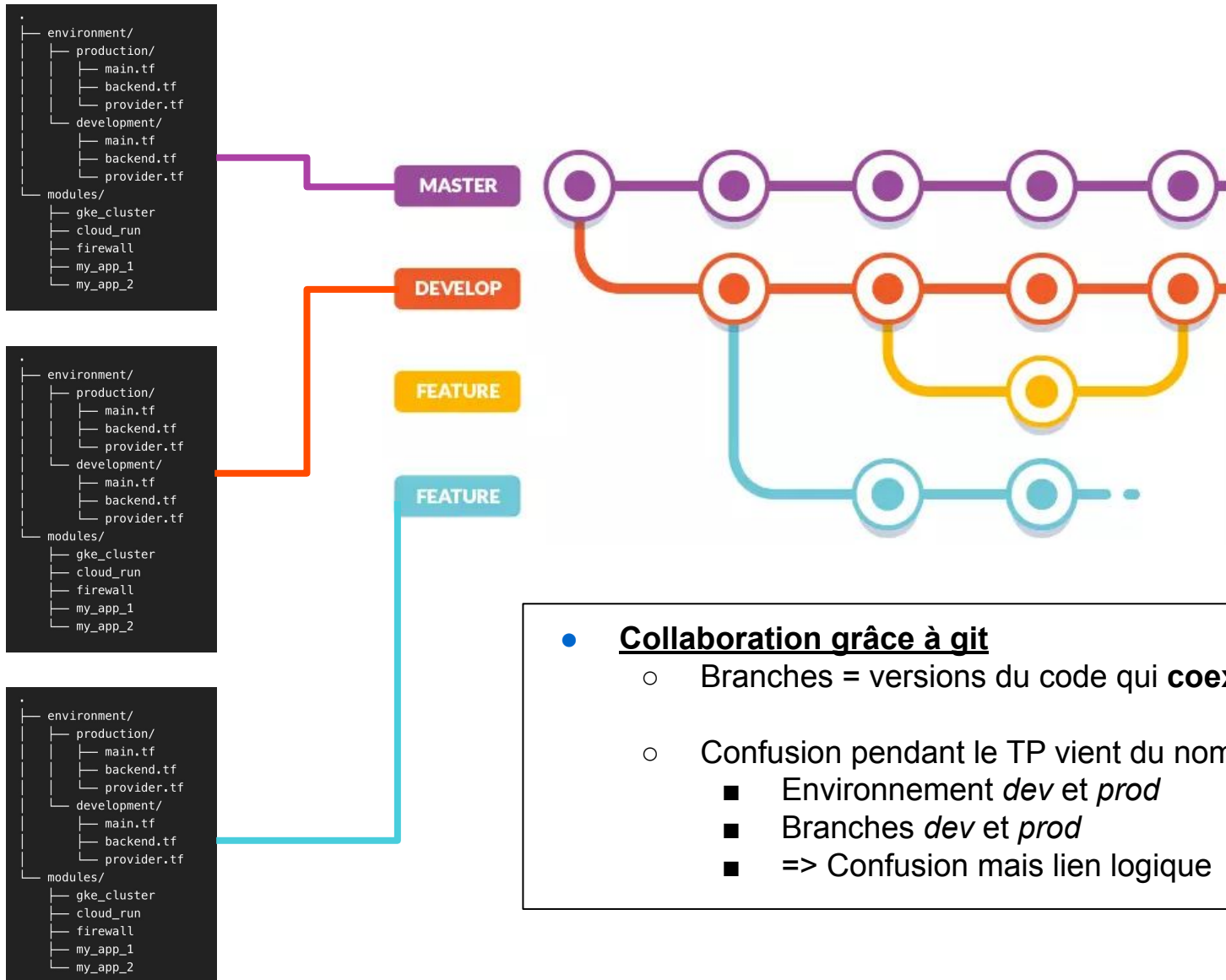
plan et apply dans les dossiers d'environnements

- Environnements isolés
 - Projets Cloud séparés
 - Prefix / suffix + Isolation réseau
- Un *state* par environnement :
 - 1 terraform apply / plan => 1 state

Code mutualisé entre les environnements : “fonctions”

- Modularité
 - Définir des configuration partagées / réutilisables
 - Définir l'ensemble des composants nécessaire au déploiement d'une application

Gérer les versions et environnements du code d'infrastructure



- **Collaboration grâce à git**

- Branches = versions du code qui **coexistent**
- Confusion pendant le TP vient du nom des objets
 - Environnement *dev* et *prod*
 - Branches *dev* et *prod*
 - => Confusion mais lien logique

Structure du projet d'exemple du TP 3 de Google

- Note rapide sur les fork


Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks.](#)

Required fields are marked with an asterisk ().*


Owner *

Repository name *

 aballiet ▾

 /


solutions-terraform-clou

 solutions-terraform-cloudbuild-gitops is available.

By default, forks are named the same as their upstream repository. You can customize the name to distinguish it further.

Description (optional)

☒ **Copy the `dev` branch only**
Contribute back to GoogleCloudPlatform/solutions-terraform-cloudbuild-gitops by adding your own branch. [Learn more.](#)

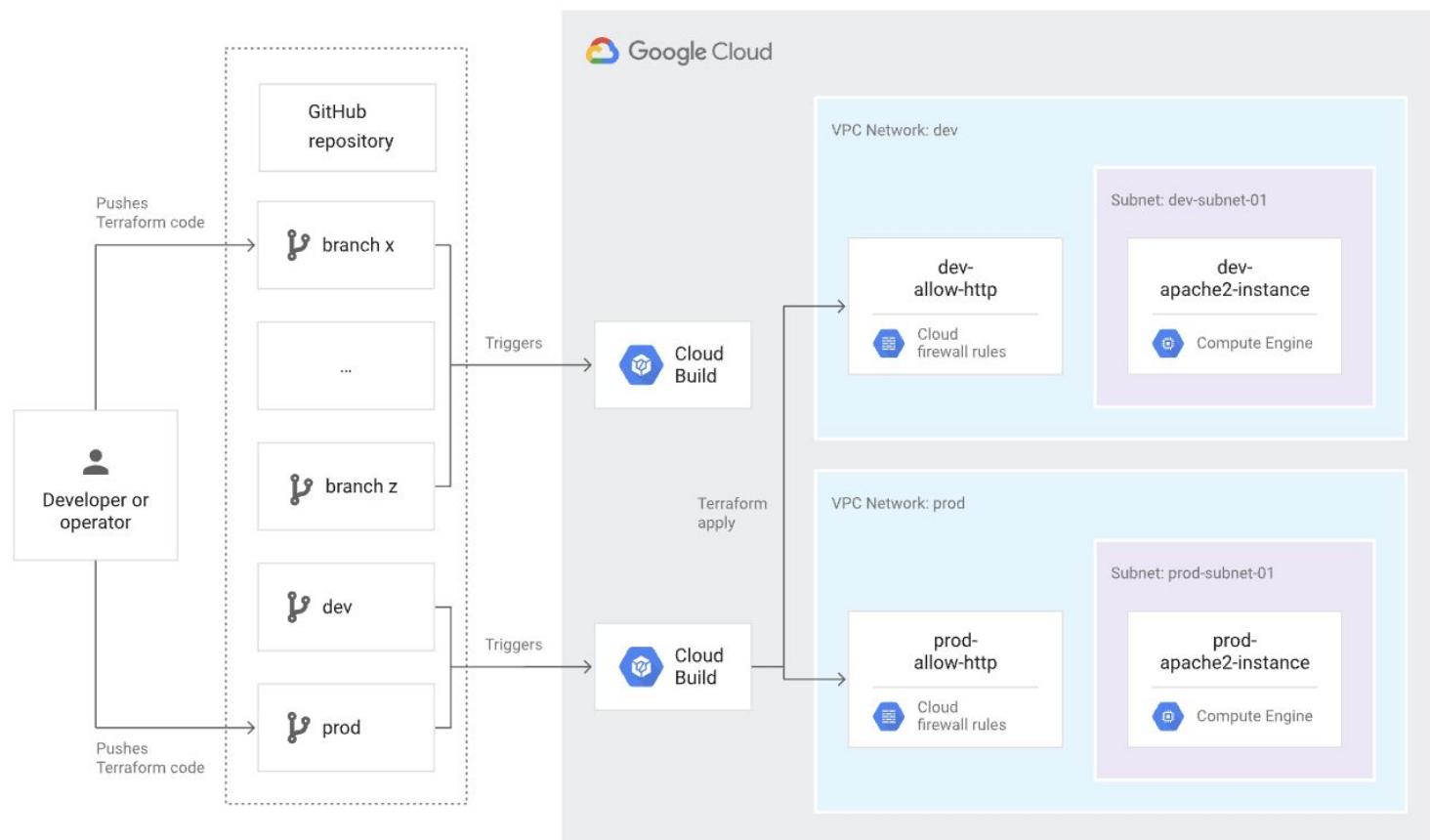
 You are creating a fork in your personal account.

Create fork

Copie qu'une seule
branche (dev) !

Structure du projet d'exemple du TP 3 de Google

★ **Remarque :** Pour plus de simplicité, ce tutoriel ne met en œuvre que des environnements **dev** et **prod** à l'aide de VPC. Vous pouvez étendre ce comportement pour effectuer des déploiements dans d'autres environnements et pour créer des projets dans la hiérarchie de votre organisation, si nécessaire.



<https://cloud.google.com/docs/terraform/resource-management/managing-infrastructure-as-code?hl=fr#architecture>

Structure du projet d'exemple du TP 3 de Google

- *cloudbuild.yaml*
 - \$BRANCH_NAME
 - Terraform init

```
steps:
- id: 'branch name'
  name: 'alpine'
  entrypoint: 'sh'
  args:
  - '-c'
  - |
    echo "*****"
    echo "$BRANCH_NAME"
    echo "*****"

- id: 'tf init'
  name: 'hashicorp/terraform:1.0.0'
  entrypoint: 'sh'
  args:
  - '-c'
  - |
    if [ -d "environments/$BRANCH_NAME/" ]; then
      cd environments/$BRANCH_NAME
      terraform init
    else
      for dir in environments/*/
      do
        cd ${dir}
        env=${dir%*/}
        env=${env#*/}
        echo ""
        echo "***** TERRAFORM INIT *****"
        echo "***** At environment: ${env} *****"
        echo "*****"
        terraform init || exit 1
        cd ../../
      done
    fi
```

Structure du projet d'exemple du TP 3 de Google

- *cloudbuild.yaml*
 - Terraform plan

```
# [START tf-plan]
- id: 'tf plan'
  name: 'hashicorp/terraform:1.0.0'
  entrypoint: 'sh'
  args:
  - '-c'
  - |
    if [ -d "environments/$BRANCH_NAME/" ]; then
      cd environments/$BRANCH_NAME
      terraform plan
    else
      for dir in environments/*/
      do
        cd ${dir}
        env=${dir%*/}
        env=${env#*/}
        echo ""
        echo "***** TERRAFOM PLAN *****"
        echo "***** At environment: ${env} *****"
        echo "*****"
        terraform plan || exit 1
        cd ../../
      done
    fi
# [END tf-plan]
```

Structure du projet d'exemple du TP 3 de Google

- *cloudbuild.yaml*
 - Terraform apply

```
# [START tf-apply]
- id: 'tf apply'
  name: 'hashicorp/terraform:1.0.0'
  entrypoint: 'sh'
  args:
  - '-c'
  - |
    if [ -d "environments/$BRANCH_NAME/" ]; then
      cd environments/$BRANCH_NAME
      terraform apply -auto-approve
    else
      echo "***** SKIPPING APPLYING *****"
      echo "Branch '$BRANCH_NAME' does not represent an official environment."
      echo "*****"
    fi
# [END tf-apply]
```

Structure du projet d'exemple du TP 3 de Google

- *cloudbuild.yaml*

Le processus commence lorsque vous envoyez le code Terraform à la branche `dev` ou `prod`. Dans ce scénario, Cloud Build se déclenche, puis applique les fichiers manifestes Terraform pour obtenir l'état souhaité dans l'environnement concerné. D'autre part, lorsque vous transférez du code Terraform vers une autre branche, par exemple une branche de fonctionnalité, Cloud Build s'exécute pour exécuter `terraform plan` mais rien n'est appliqué aux environnements.

Idéalement, les développeurs ou les opérateurs doivent faire des propositions d'infrastructure à des [branches non protégées](#), puis les soumettre par le biais de [demandes d'extraction](#). L'[application GitHub Cloud Build](#), décrite plus loin dans ce tutoriel, déclenche automatiquement les tâches de compilation et associe les rapports `terraform plan` à ces demandes d'extraction. Ainsi, vous pouvez discuter des modifications potentielles et les examiner avec les collaborateurs, et ajouter des commits de suivi avant que les modifications ne soient fusionnées dans la branche de base.

Si aucun problème n'est soulevé, vous devez d'abord fusionner les modifications dans la branche `dev`. Cette fusion déclenche un déploiement d'infrastructure dans l'environnement `dev`, ce qui vous permet de tester cet environnement. Une fois que vous avez testé votre infrastructure et que vous êtes sûr de la fiabilité de votre déploiement, vous devez fusionner la branche `dev` dans la branche `prod` pour déclencher l'installation de l'infrastructure dans l'environnement de production.

- Documentation cloud build

- <https://cloud.google.com/build/docs/configuring-builds/create-basic-configuration>
- <https://cloud.google.com/build/docs/configuring-builds/substitute-variable-values>

Structure du projet d'exemple du TP 3 de Google

- *cloudbuild.yaml*

Le processus commence lorsque vous envoyez le code Terraform à la branche `dev` ou `prod`. Dans ce scénario, Cloud Build se déclenche, puis applique les fichiers manifestes Terraform pour obtenir l'état souhaité dans l'environnement concerné. D'autre part, lorsque vous transférez du code Terraform vers une autre branche, par exemple une branche de fonctionnalité, Cloud Build s'exécute pour exécuter `terraform plan` mais rien n'est appliqué aux environnements.

Idéalement, les développeurs ou les opérateurs doivent faire des propositions de structure à des branches non protégées, puis les soumettre par le biais de demandes d'extraction. GitHub Cloud Build, décrite plus loin dans ce tutoriel, déclenche automatiquement des tâches de construction des rapports `terraform plan` à ces demandes d'extraction. Ainsi, vous pouvez examiner des modifications et les examiner avec les collaborateurs, et ajouter des commits de suivi pour les modifications apportées dans la branche de base.

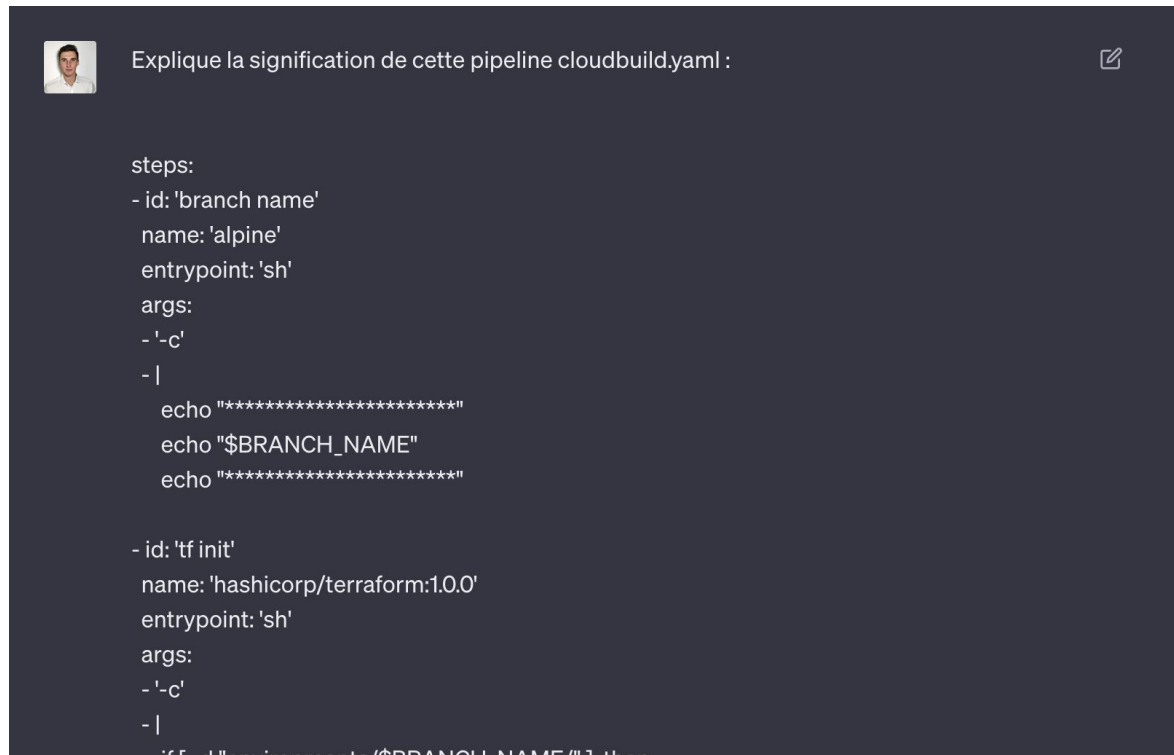
Si aucun problème n'est détecté, vous devez d'abord fusionner les modifications dans la branche `dev`. Cette fusion déclenche un déploiement de l'infrastructure dans l'environnement `dev`, ce qui vous permet de tester cet environnement. Une fois que vous avez vérifié l'infrastructure et que vous êtes sûr de la fiabilité de votre déploiement, vous devez fusionner la branche `dev` dans la branche `prod` pour déclencher l'installation de l'infrastructure dans l'environnement de production.

- Documentation cloud build

- <https://cloud.google.com/build/docs/configuring-builds/create-basic-configuration>
- <https://cloud.google.com/build/docs/configuring-builds/substitute-variable-values>

Structure du projet d'exemple du TP 3 de Google

- *cloudbuild.yaml*
 - Documentation cloud build
 - <https://cloud.google.com/build/docs/configuring-builds/create-basic-configuration>
 - <https://cloud.google.com/build/docs/configuring-builds/substitute-variable-values>
 - ChatGPT 🔥🔥🔥



Explique la signification de cette pipeline cloudbuild.yaml :

```
steps:  
- id: 'branch name'  
  name: 'alpine'  
  entrypoint: 'sh'  
  args:  
  - '-c'  
  - |  
    echo "*****"  
    echo "$BRANCH_NAME"  
    echo "*****"  
  
- id: 'tf init'  
  name: 'hashicorp/terraform:1.0.0'  
  entrypoint: 'sh'  
  args:  
  - '-c'  
  - |  
    if [ -d "environments/$BRANCH_NAME/" ]; then
```


Structure du projet d'exemple du TP 3 de Google

- *cloudbuild.yaml*

terraform apply est exécuté seulement si :

- le nom de la branche correspond à un dossier environnement

SINON on affiche : *"Branch '\$BRANCH_NAME' does not represent an official environment."*

Voici une explication de chaque étape dans ce fichier YAML :

1. Étape 'branch name' :

- Cette étape utilise l'image Docker Alpine.
- L'entrée est définie sur 'sh', ce qui signifie qu'elle exécute un shell pour exécuter les commandes.
- Les arguments sont fournis sous forme de script shell (``-c``) qui affiche le nom de la branche actuelle dans la variable d'environnement ``$BRANCH_NAME``.

2. Étape 'tf init' :

- Cette étape utilise l'image Docker 'hashicorp/terraform:1.0.0', qui contient Terraform version 1.0.0.
- Elle exécute également un shell.
- Le script shell vérifie d'abord si le répertoire ``environnements/$BRANCH_NAME/`` existe. Si c'est le cas, il se déplace dans ce répertoire et exécute ``terraform init``. Sinon, il parcourt tous les sous-répertoires sous ``environnements/`` et exécute ``terraform init`` pour chacun.

3. Étape 'tf plan' :

- Cette étape utilise également l'image Docker 'hashicorp/terraform:1.0.0'.
- Comme précédemment, elle exécute un script shell qui effectue un ``terraform plan``. Si le répertoire ``environnements/$BRANCH_NAME/`` existe, il est utilisé. Sinon, il parcourt tous les sous-répertoires et exécute ``terraform plan`` pour chacun.

4. Étape 'tf apply' :

- Cette étape utilise l'image Docker 'hashicorp/terraform:1.0.0'.
- Elle exécute un script shell qui, s'il trouve le répertoire ``environnements/$BRANCH_NAME/``, exécute ``terraform apply -auto-approve``. Sinon, il affiche un message indiquant que l'application n'est pas effectuée car la branche ne représente pas un environnement officiel.

🔄 Regenerate

Dans l'épisode précédent ...

- Terraform ?
 - Commandes gcloud permettent de définir les ressources 🤔

gcloud run deploy 

[Send feedback](#)

NAME

gcloud run deploy - create or update a Cloud Run service

SYNOPSIS

```
gcloud run deploy [[ SERVICE ] --namespace = NAMESPACE ] [ --args = [ ARG ,... ] [ --async ]
[ --command = [ COMMAND ,... ] [ --concurrency = CONCURRENCY ] [ --cpu = CPU ] [ --ingress = INGRESS ;
default="all" ] [ --max-instances = MAX_INSTANCES ] [ --memory = MEMORY ]
[ --min-instances = MIN_INSTANCES ] [ --platform = PLATFORM ; default="managed" ] [ --port = PORT ]
[ --service-account = SERVICE_ACCOUNT ] [ --tag = TAG ] [ --timeout = TIMEOUT ] [ --no-traffic ]
[ --[no-]use-http2 ] [ --clear-env-vars | --env-vars-file = FILE_PATH |
--set-env-vars = [ KEY = VALUE ,... ] | --remove-env-vars = [ KEY ,... ] --update-env-vars = [ KEY = VALUE ,... ] ]
[ --clear-labels | --remove-labels = [ KEY ,... ] --labels = [ KEY = VALUE ,... ] |
--update-labels = [ KEY = VALUE ,... ] [ --clear-secrets | --set-secrets = [ KEY = VALUE ,... ] |
--remove-secrets = [ KEY ,... ] --update-secrets = [ KEY = VALUE ,... ] [ --connectivity = CONNECTIVITY
--clear-config-maps | --set-config-maps = [ KEY = VALUE ,... ] | --remove-config-maps = [ KEY ,... ]
--update-config-maps = [ KEY = VALUE ,... ] [ --image = IMAGE | --source = SOURCE ]
[ --[no-]allow-unauthenticated --breakglass = JUSTIFICATION --clear-vpc-connector
--[no-]cpu-boost --[no-]cpu-throttling --description = DESCRIPTION
--execution-environment = EXECUTION_ENVIRONMENT --revision-suffix = REVISION_SUFFIX
--[no-]session-affinity --vpc-connector = VPC_CONNECTOR --vpc-egress = VPC_EGRESS
--add-cloudsql-instances = [ CLOUDSQL-INSTANCES ,... ] | --clear-cloudsql-instances |
--remove-cloudsql-instances = [ CLOUDSQL-INSTANCES ,... ] |
--set-cloudsql-instances = [ CLOUDSQL-INSTANCES ,... ] --binary-authorization = POLICY |
--clear-binary-authorization --clear-encryption-key-shutdown-hours |
--encryption-key-shutdown-hours = ENCRYPTION_KEY_SHUTDOWN_HOURS --clear-key | --key = KEY
--clear-post-key-revocation-action-type |
--post-key-revocation-action-type = POST_KEY_REVOCATION_ACTION_TYPE [ --region = REGION |
--cluster = CLUSTER --cluster-location = CLUSTER_LOCATION | --context = CONTEXT
--kubeconfig = KUBECONFIG ] [ G_CLOUD_WIDE_FLAG ... ]
```


Dans l'épisode précédent ...

- Infrastructure as Code



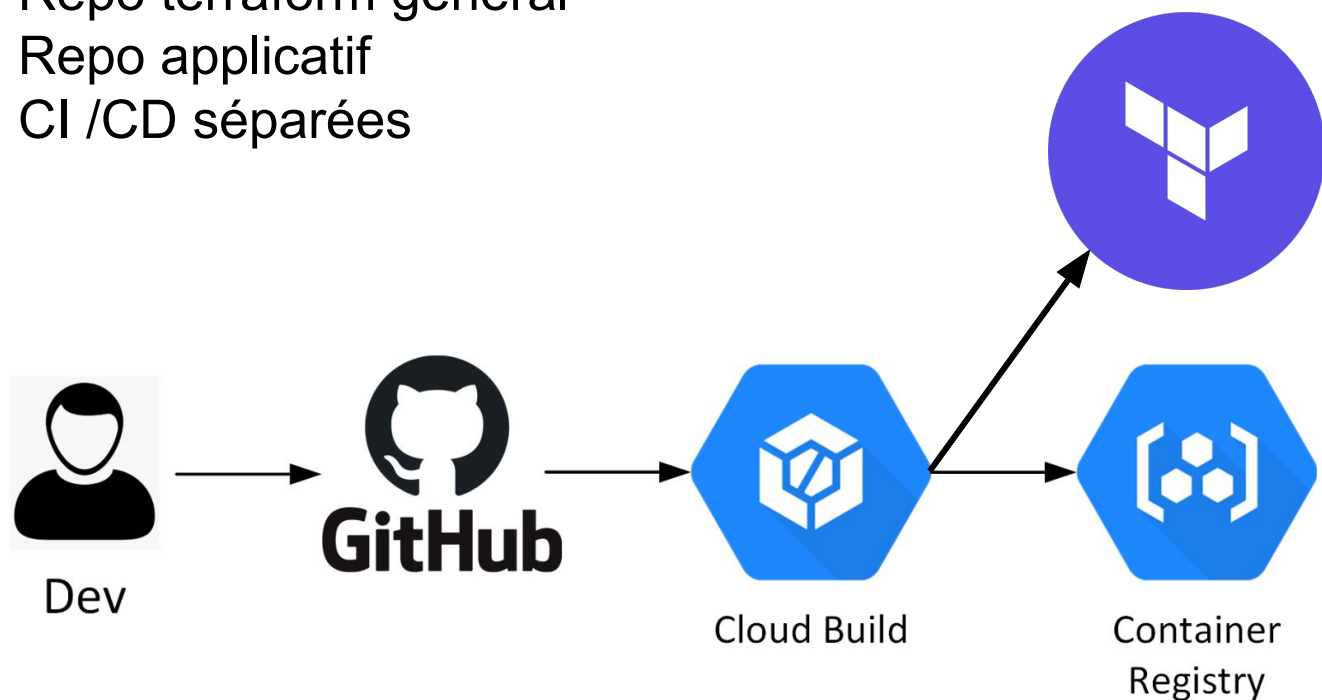
HashiCorp

Terraform

- Utilise les API du cloud provider
- Versionner les configurations
- Collaboratif
- Garantie un état reproductible (en théorie)
- Vérification des changements d'infrastructure

Dans l'épisode précédent ...

- Pipeline complètement reproductible ?
 - Les objets doivent parfois déjà exister
 - Multiple provider terraform
- Découper les projets ?
 - Repo terraform général
 - Repo applicatif
 - CI /CD séparées



TP

- Finir le TP 4
 - Documentation : autonomie
 - GCP
 - Terraform
 - Reprend tous les concepts vus ensemble
 - N'hésitez pas à me poser des questions

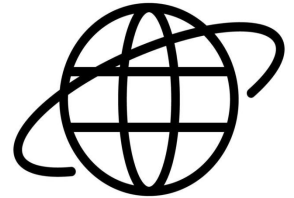
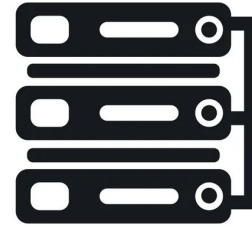
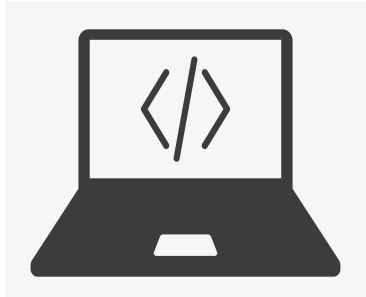
DevOps

Initiation aux microservices

Antoine Balliet

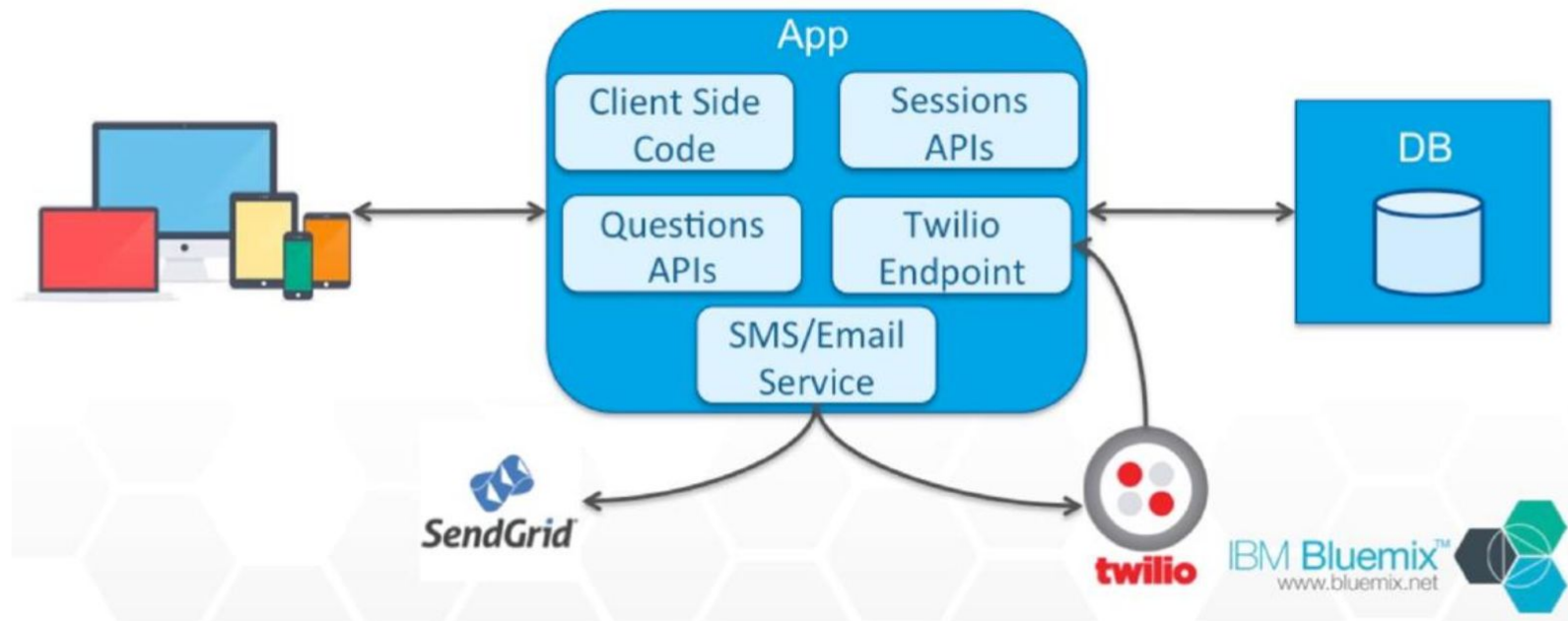
antoine.balliet@dauphine.psl.eu

Vélocité 🚀



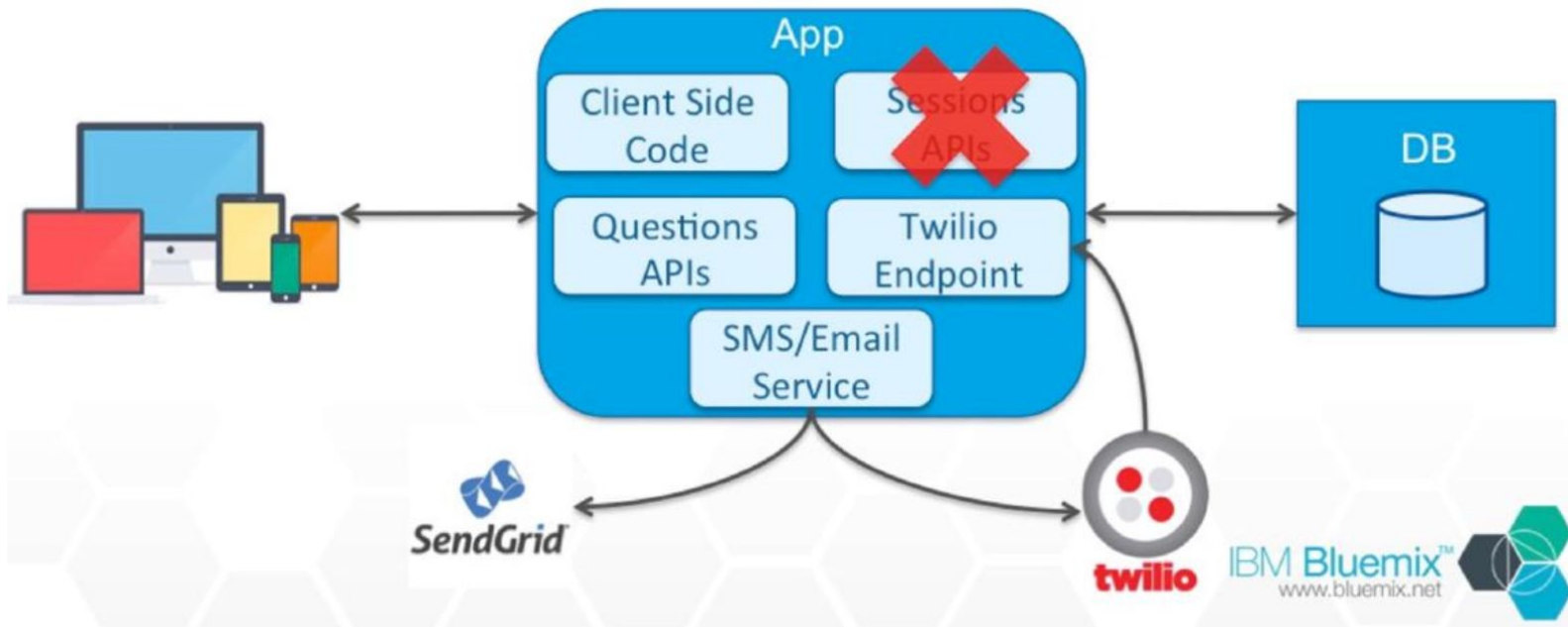
**Charge sur les
ressources ???**

Isolation des applications



Isolation des applications

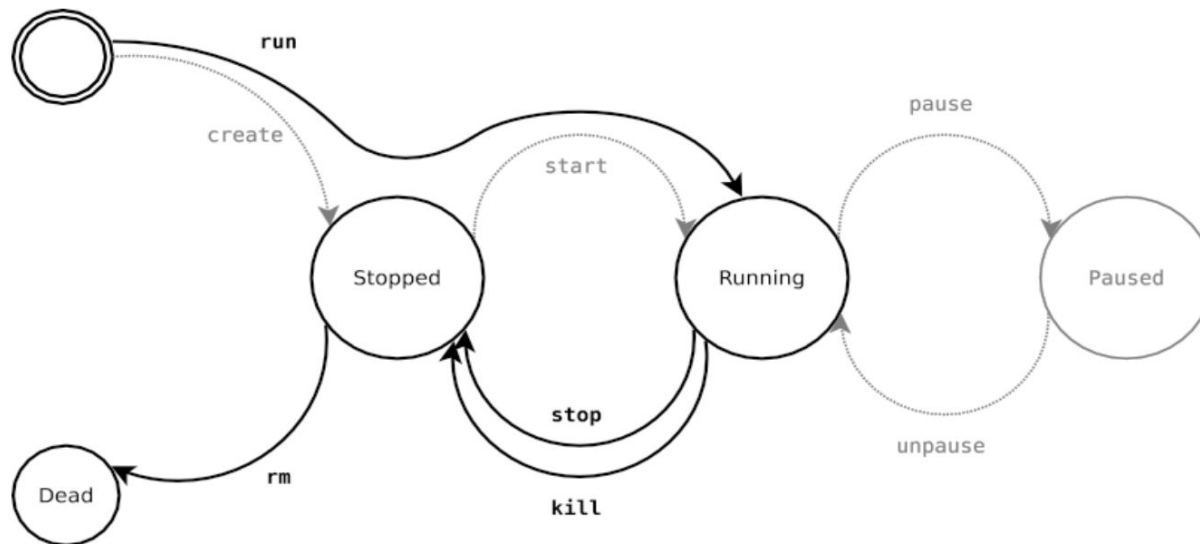
Failure in Monolith !!



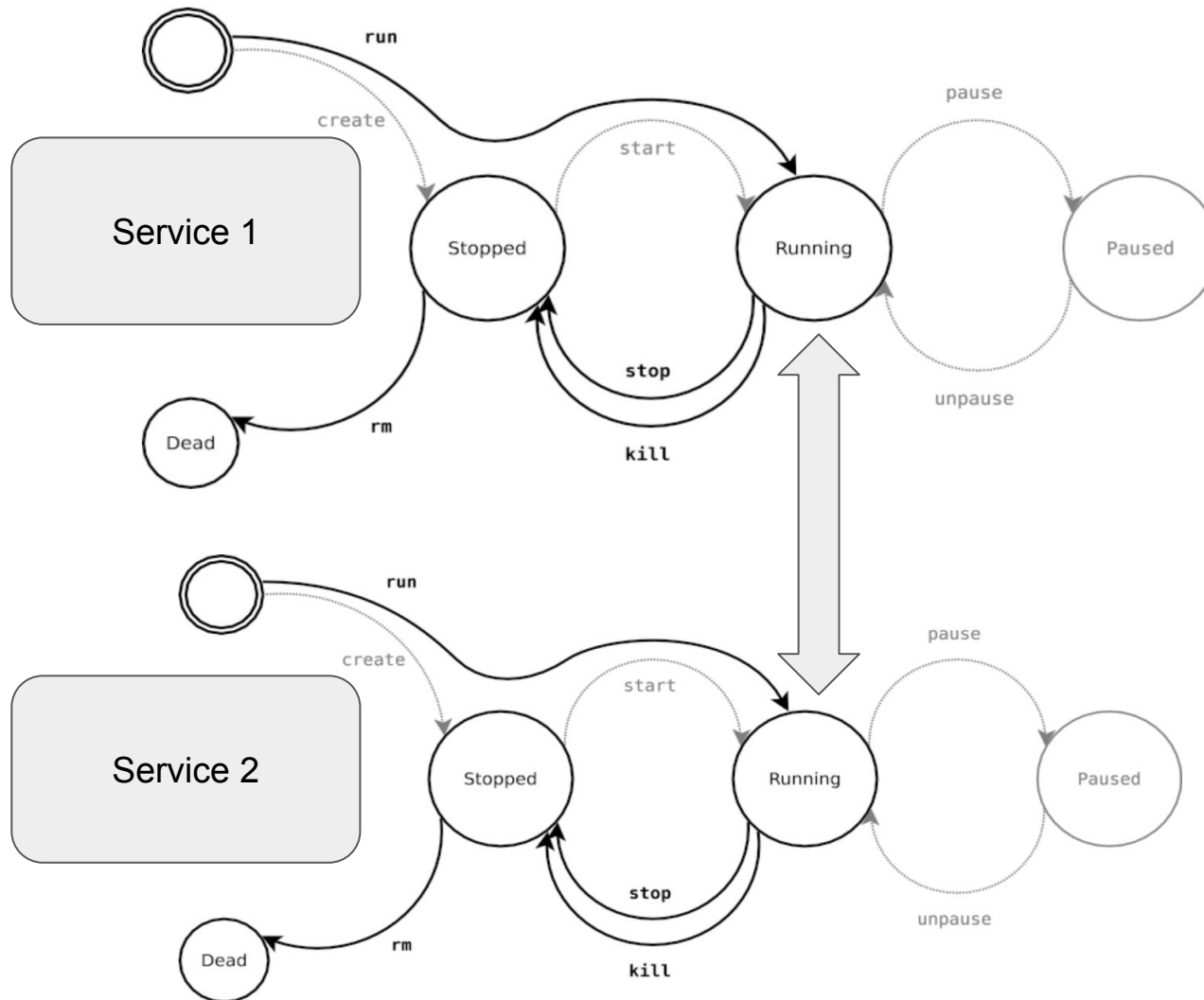
Microservices

- Services isolés
 - Configuration propre
 - Déploiement distincts
 - Docker image dédiée
- Nouvelles possibilités
 - Gérer les ressources alloués par service
 - Code source pouvant être séparé : plusieurs repos
- Inconvénient
 - Communication entre services
 - “Contrats” entre les équipes
 - Complexité plus grande du déploiement

Cycle de vie conteneur



Microservices



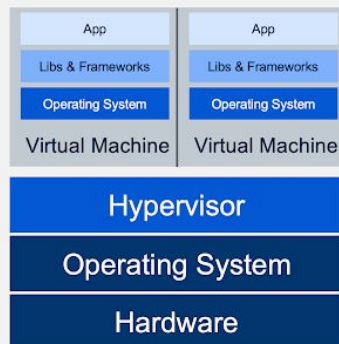
Prêt ?



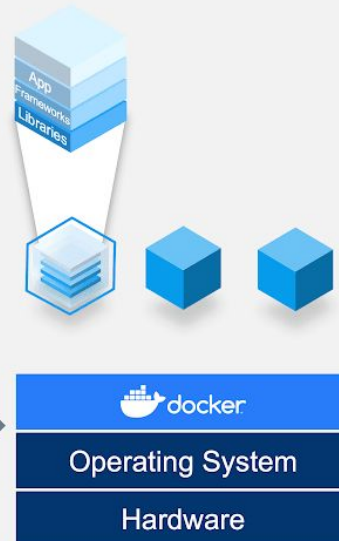
Microservices



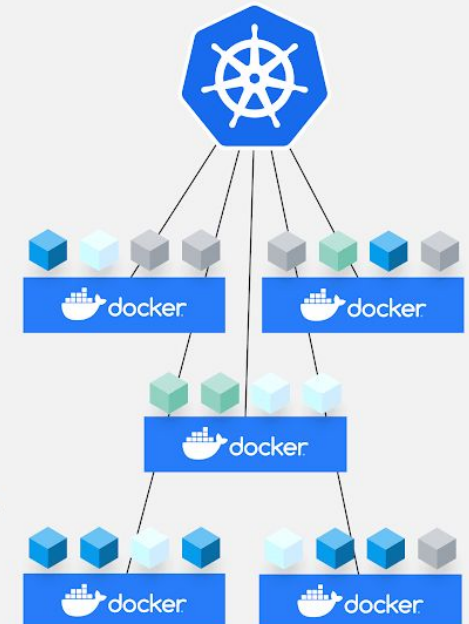
**Traditional
Deployment**



**Virtualized
Deployment**



**Container
Deployment**



**Kubernetes
Deployment**

**Kubernetes & Docker work
together to build & run
containerized applications**

Kubernetes



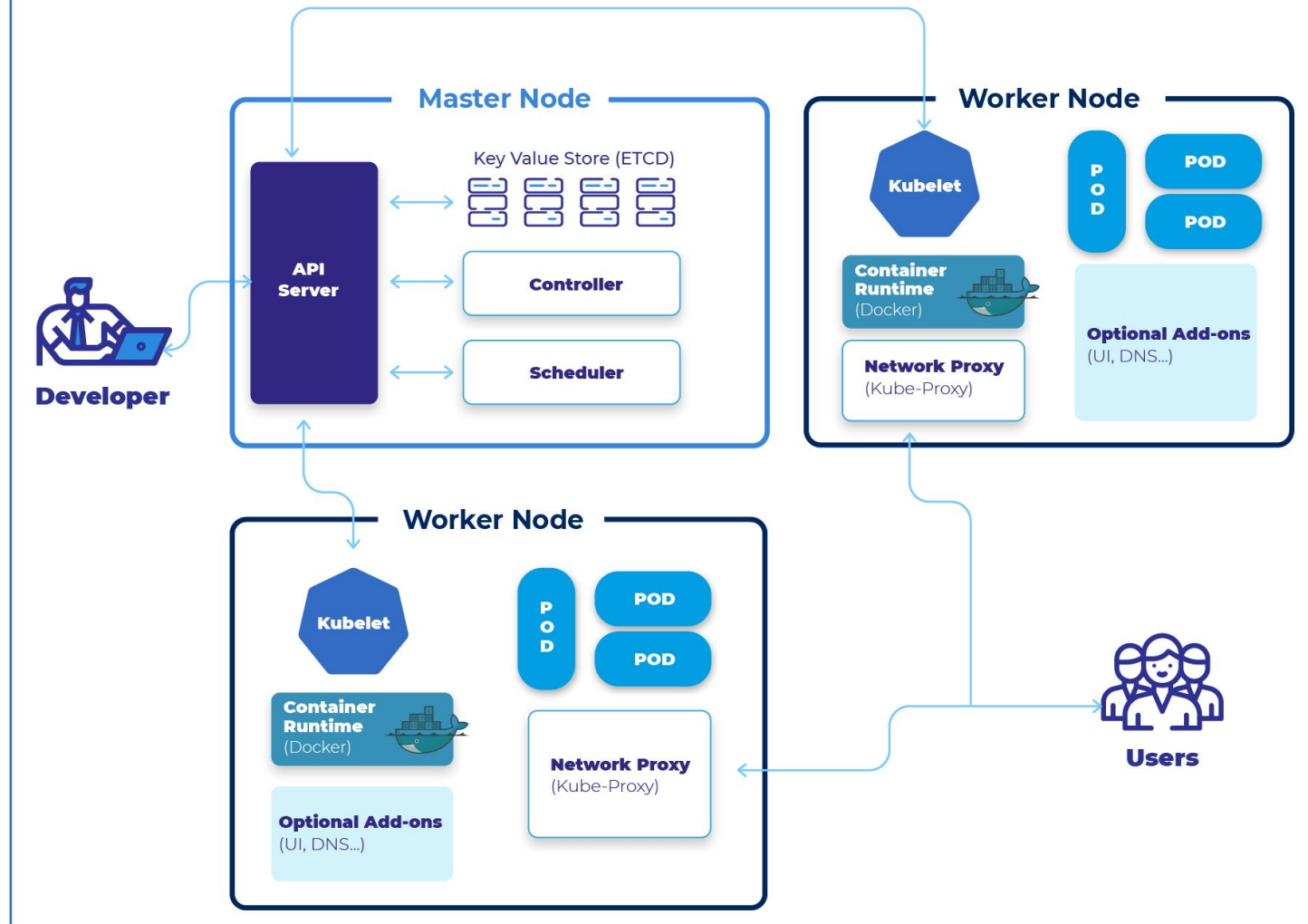
kubernetes

Kubernetes (Projet Open Source initié par Google après Borg 2004)

- **Orchestrateur de conteneurs à l'échelle**
- **Exécuter des applications Docker à conteneurs multiples** sur des pool de machines

Kubernetes

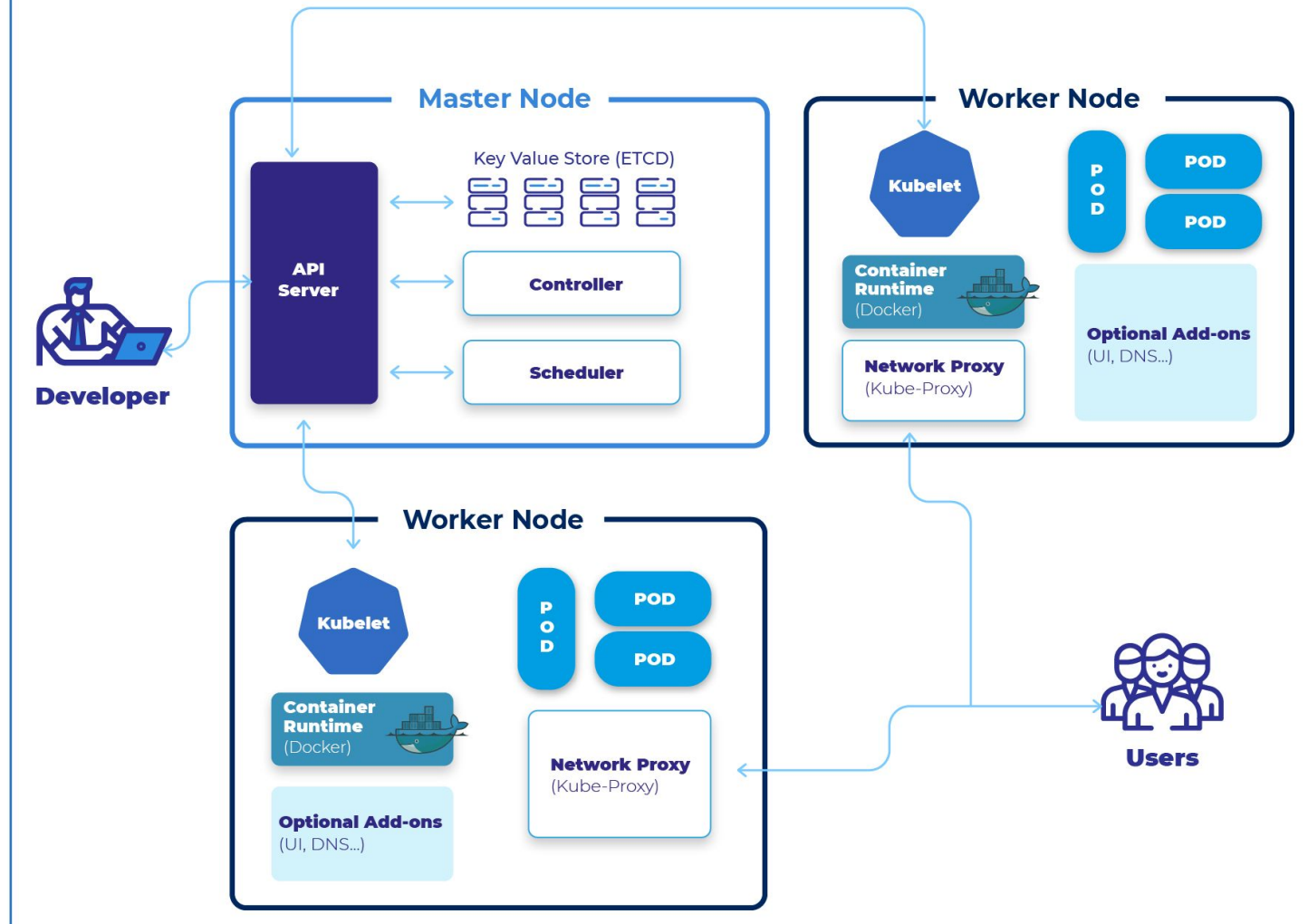
Kubernetes Architecture Diagram



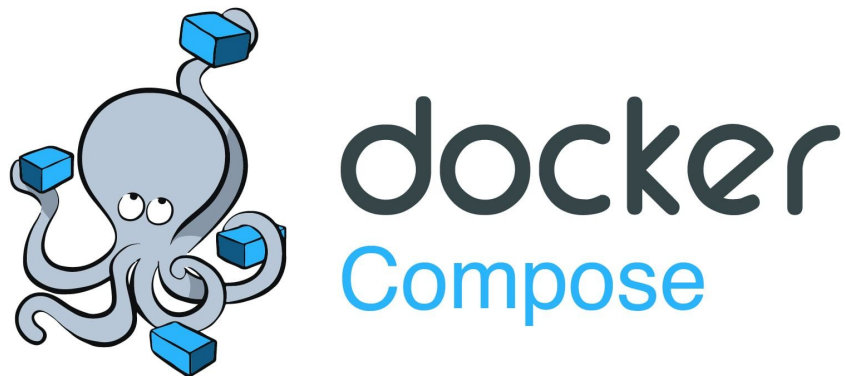
Kubernetes

Développement en local ? 🧐

Kubernetes Architecture Diagram



Docker compose



Docker Compose : **définir un ensemble de services / composants de notre applications**

- associer plusieurs conteneurs
- définir les paramètres docker pour composer tout un système
 - nom d'image existante (DockerHub)
 - binding de port
 - volumes
 - réseau

TP

- Introduction aux microservices
 - Docker Compose
 - Définition d'un ensemble de composants / services pour notre application
 - Kubernetes
 - Déploiement sur GKE : service managé Kubernetes sur GCP
 - Aller plus loin avec Terraform
 - Kubernetes Terraform provider
 - Helm Terraform provider

Pour aller plus loin

- TP Kubernetes : <https://labocloud.matbilodeau.dev/laboKube0.html>
 - Toutes les parties pour l'aspect général de Kubernetes
 - Partie 4 aborde les service mesh avec Istio