

DevOps

Cours 4: Infrastructure as Code II & introduction à l'orchestration de conteneurs

Antoine Balliet

antoine.balliet@dauphine.psl.eu

Dans l'épisode précédent ...

Cloud Provider

- Présentation générale
 - Infrastructure
 - Logiciel
 - Couverture mondiale
 - Services “managés”
 - Estimation des coûts
- Limites et ouverture
 - Concentration des risques :
 - cloud privés
 - cloud hybrides
 - Open Source
 - Contribution des “hyperscalers”
 - Solution cloud open source : déployer son propre cloud

Dans l'épisode précédent ...

Infrastructure as Code

- Présentation Terraform
 - Concept : versionner les ressources des projets Cloud
 - Terraform providers : liens entre le code HCL et les APIs
 - Syntaxe HCL
- Fonctionnement
 - Création / Mise à jour / Destruction des ressources
 - State
 - fait correspondre le code Terraform (HCL) aux ressources cloud
 - contient des données sensibles sur l'infrastructure
 - unique pour un ensemble de ressource donné
 - Pas mis dans Git
 - Collaboratif si partagé entre les développeurs
 - Partagé sur un espace de stockage type "bucket" :
AWS S3 / Google GCS / Azure F (similaire à Google Drive)

Dans l'épisode précédent ...

Infrastructure as Code

- TP Terraform
 - Provider GCP
 - terraform plan / terraform apply
 - CI / CD pour appliquer changements d'infrastructure
 - Stockage du state dans bucket GCS
 - Documentation terraform
 - Modules terraform

Critique du TP Infrastructure as Code

Problèmes permissions CI / CD de l'exemple
pas clair 🐼👉

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

Entièrement
reproductible ? 🤔

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

Structure du code ? 🌀

Problèmes permissions Cloud Build

if 'build.service_account' is specified, the build must either (a) specify 'build.logs_bucket' (b) use the CLOUD_LOGGING_ONLY logging option, or (c) use the NONE logging option

<https://stackoverflow.com/questions/68779751/error-publishing-source-code-from-cloud-build-to-a-bucket-using-triggers>

options:

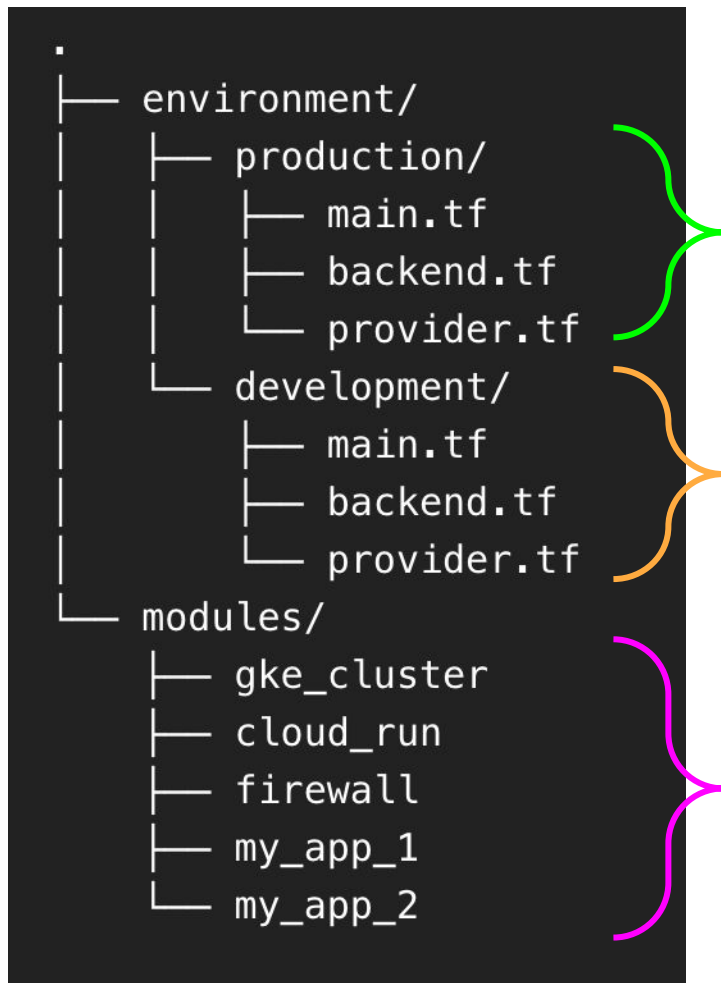
logging: CLOUD_LOGGING_ONLY

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
 - La configuration de la pipeline CI/CD utilisée est celle de la branche courante
 - Les “environnements” dans le jargon Terraform correspondent à des espaces **isolés** d'infrastructure
 - On les matérialise 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

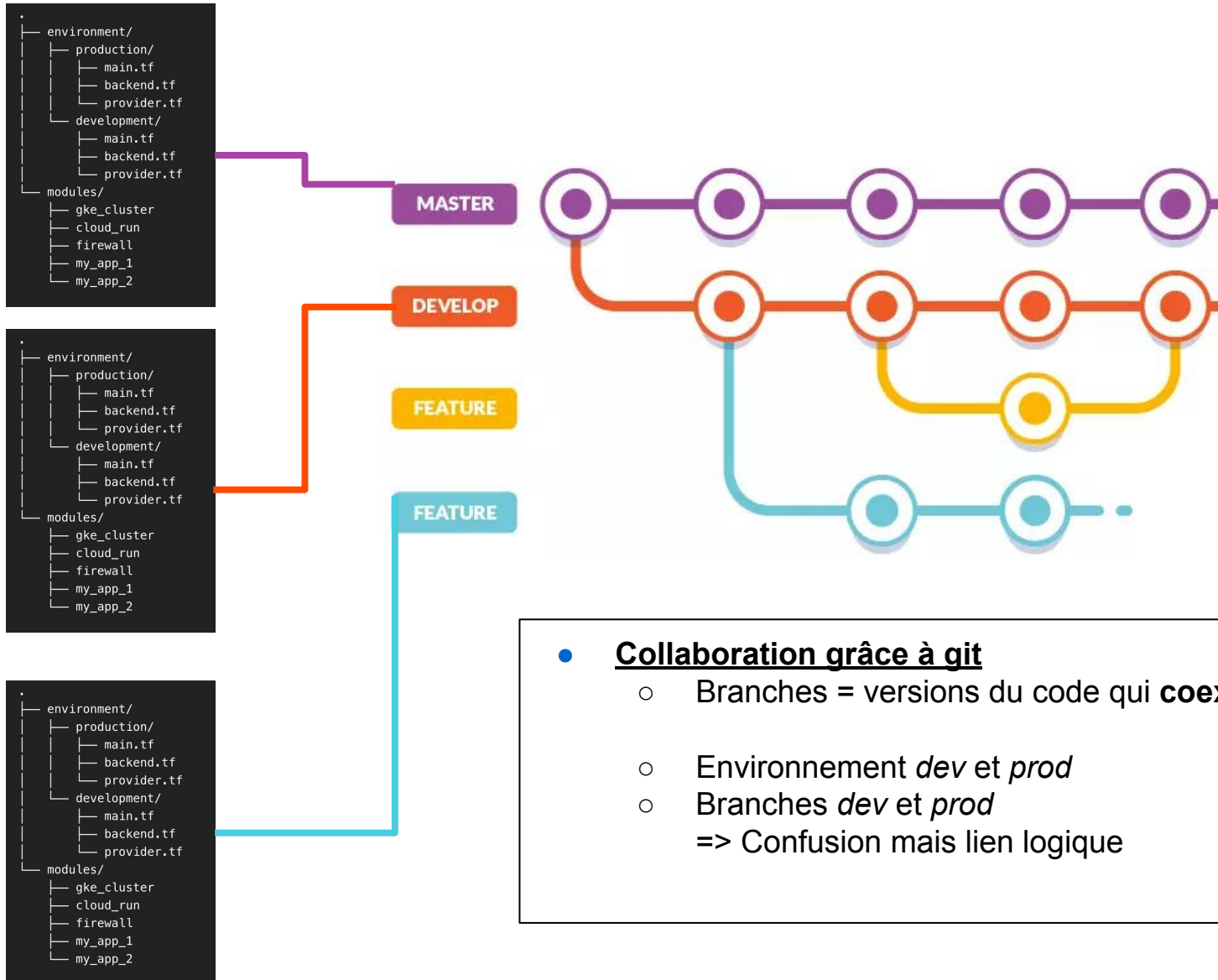
Gérer les versions et environnements du code d'infrastructure

- Structure “standard” d'un projet utilisant terraform

```
.
├── environment/
│   ├── production/
│   │   ├── main.tf
│   │   ├── backend.tf
│   │   └── provider.tf
│   └── development/
│       ├── main.tf
│       ├── backend.tf
│       └── provider.tf
└── modules/
    ├── gke_cluster
    ├── cloud_run
    ├── firewall
    ├── my_app_1
    └── my_app_2
```

```
terraform {
  backend "gcs" {
    bucket = "PROJECT_ID-tfstate"
    prefix = "env/dev"
  }
}
```

Gérer les versions et environnements du code d'infrastructure



- **Collaboration grâce à git**
 - Branches = versions du code qui **coexistent**
 - 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

aballiet / cloud-code-samples

Pull requests Actions Projects Wiki Security Insights Settings

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#) or [learn more about](#)

base repository: aballiet/cloud-code-samples base: v1 head repository: aballiet/cloud-code-samples compare: aballiet-patch-1

✓ **Able to merge.** These branches can be automatically merged.

Discuss and review the changes in this comparison with others. [Learn about pull requests](#)

1 commit 1 file changed

Commits on Dec 13, 2024

Update README.md
aballiet authored now

Ouvrir une PR
sur son repo ou
le repo original

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.](#)

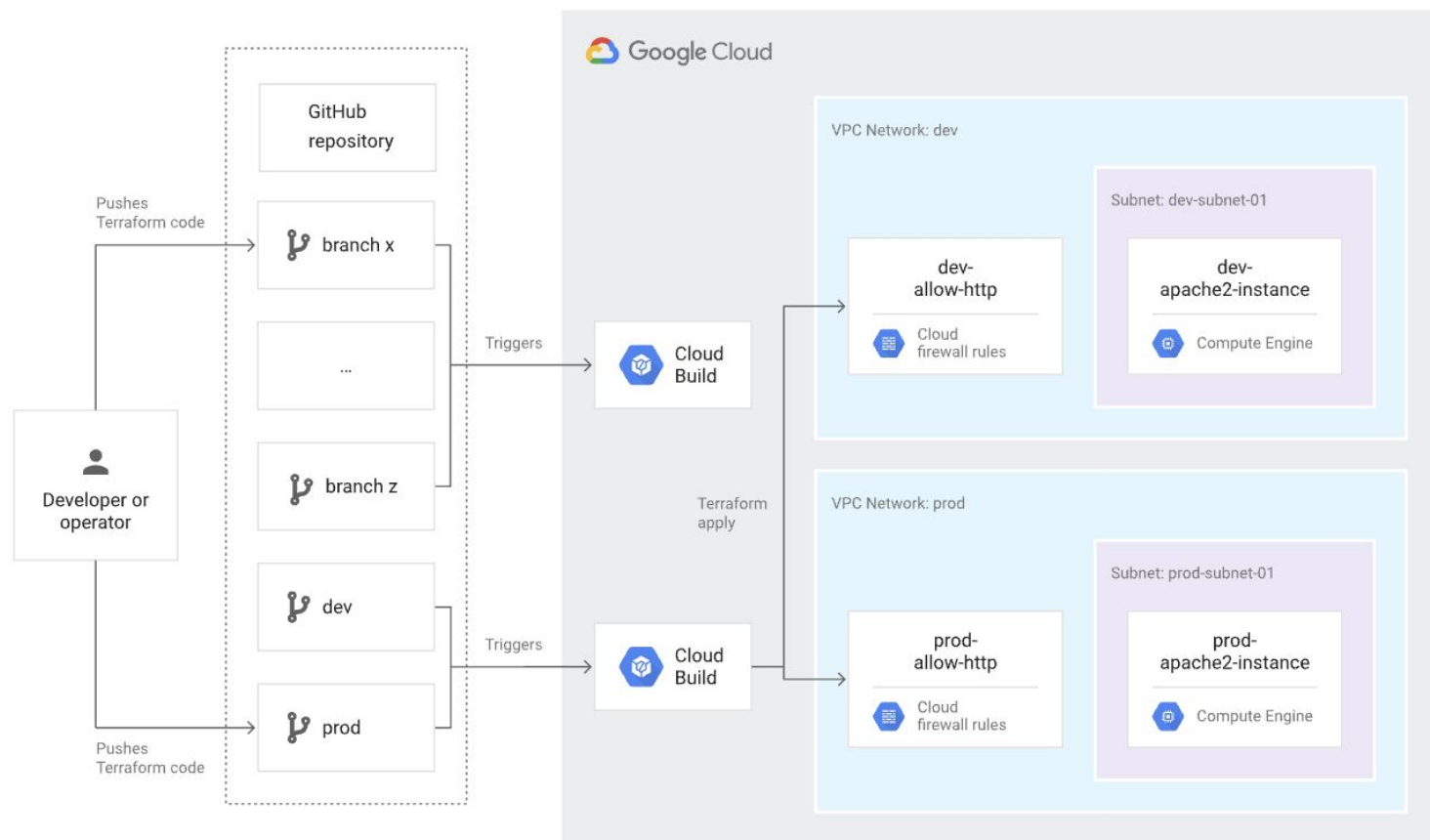
 You are creating a fork in your personal account.

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.

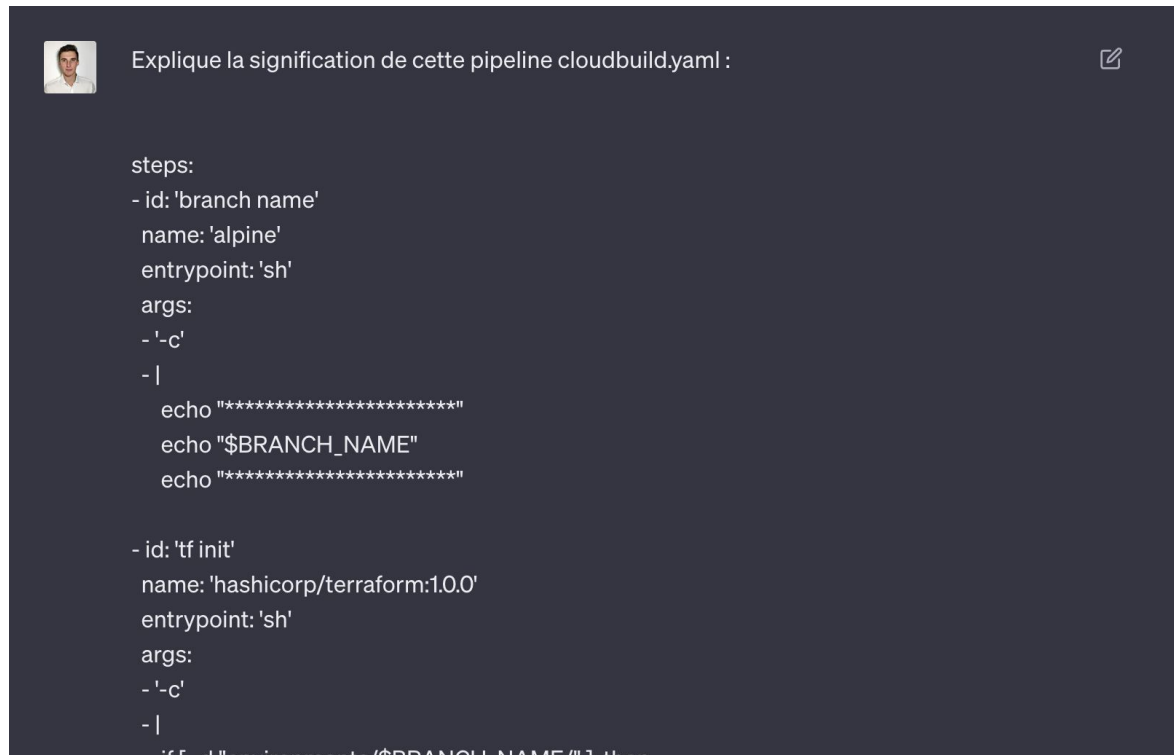
Si aucun problème n'est détecté, vous devez d'abord fusionner les modifications dans la branche `dev`. Cette fusion déclenche un déploiement de l'infrastructure dans l'environnement `dev`, ce qui vous permet de tester cet environnement. Une fois que vous avez vérifié l'infrastructure et que vous êtes 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 ``environments/$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 ``environments/`` 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 ``environments/$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 ``environments/$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 ...

- Infrastructure as Code



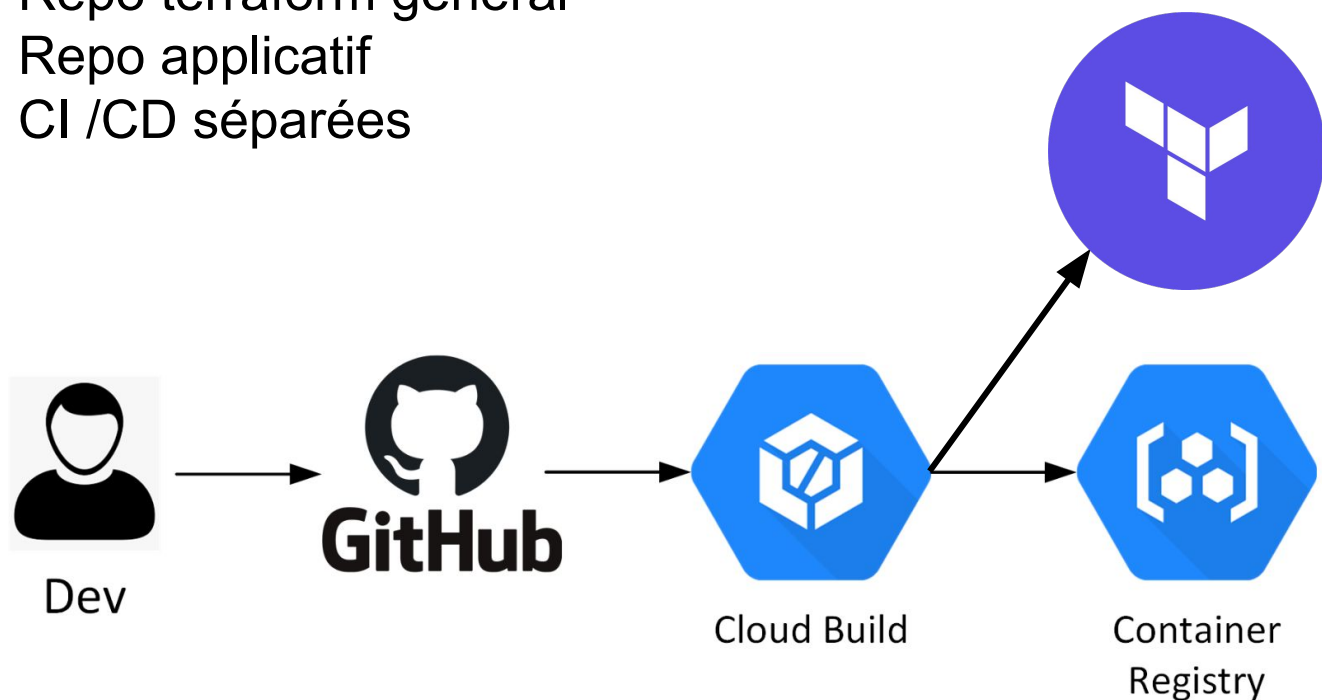
HashiCorp

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



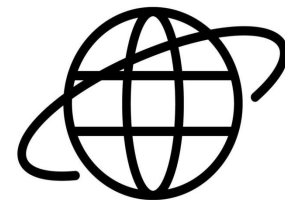
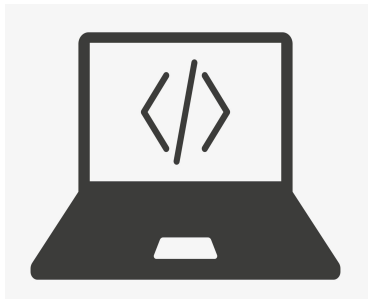
DevOps

Initiation aux microservices

Antoine Balliet

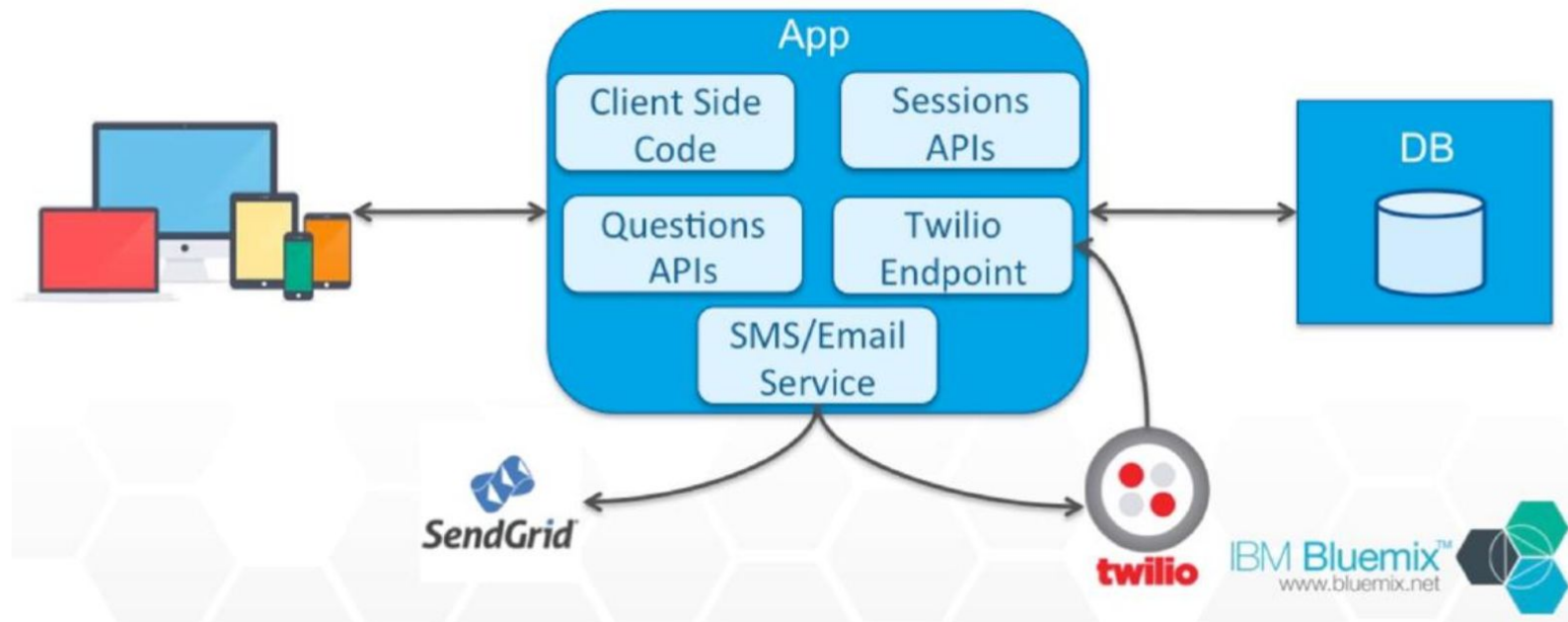
antoine.balliet@dauphine.psl.eu

Vélocité 🚀



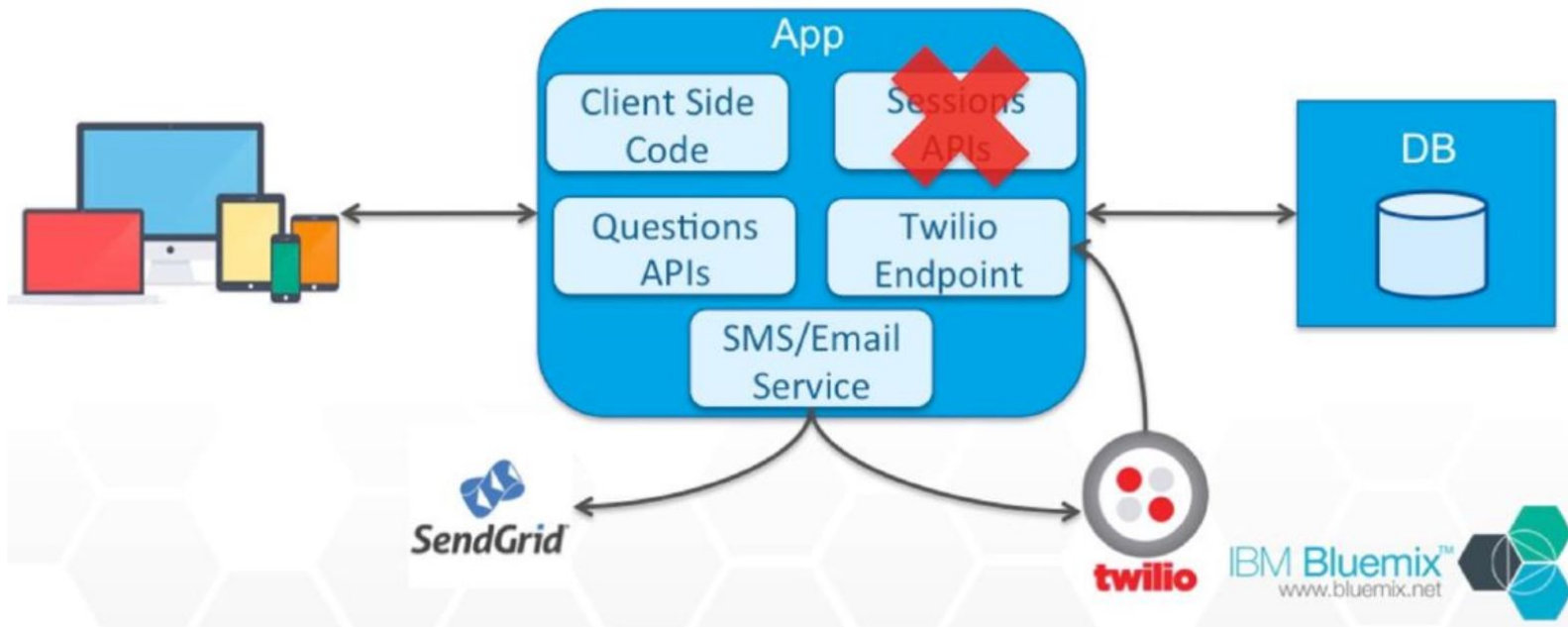
**Charge sur les
ressources ???**

Isolation des applications



Isolation des applications

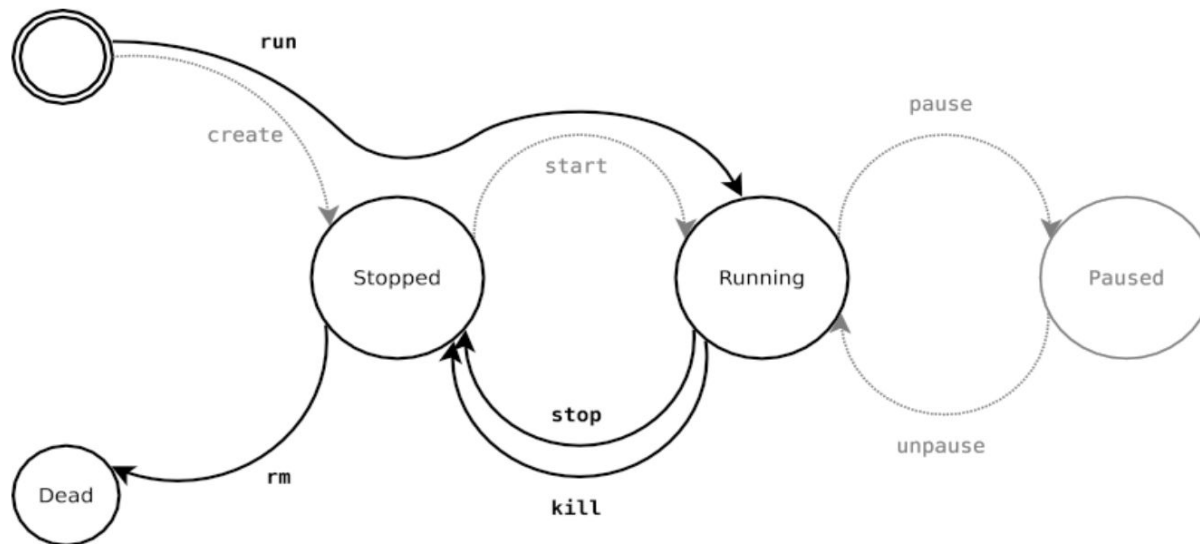
Failure in Monolith !!



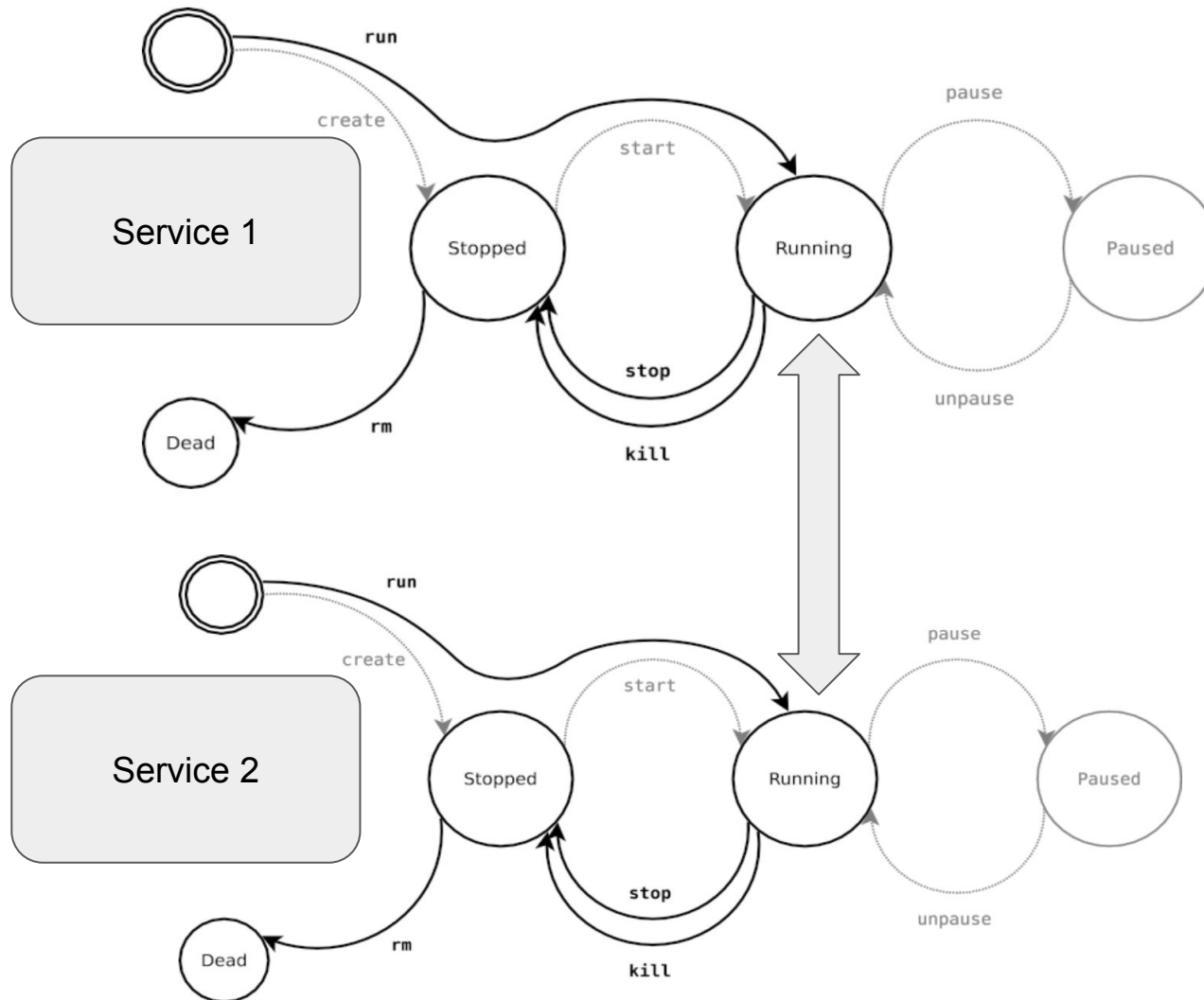
Microservices

- 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



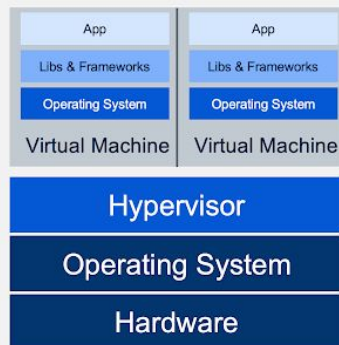
Prêt ?



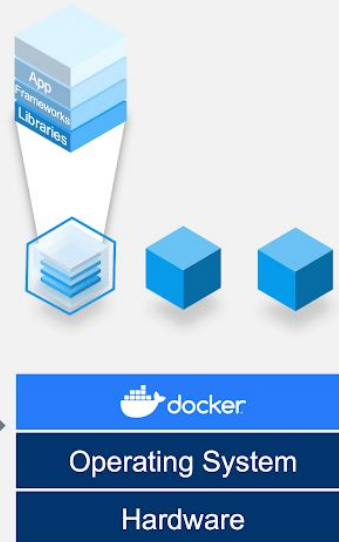
Microservices



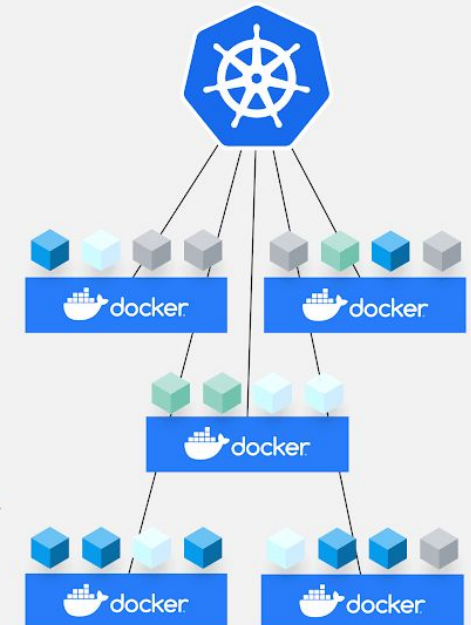
**Traditional
Deployment**



**Virtualized
Deployment**



**Container
Deployment**



**Kubernetes
Deployment**

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

Kubernetes



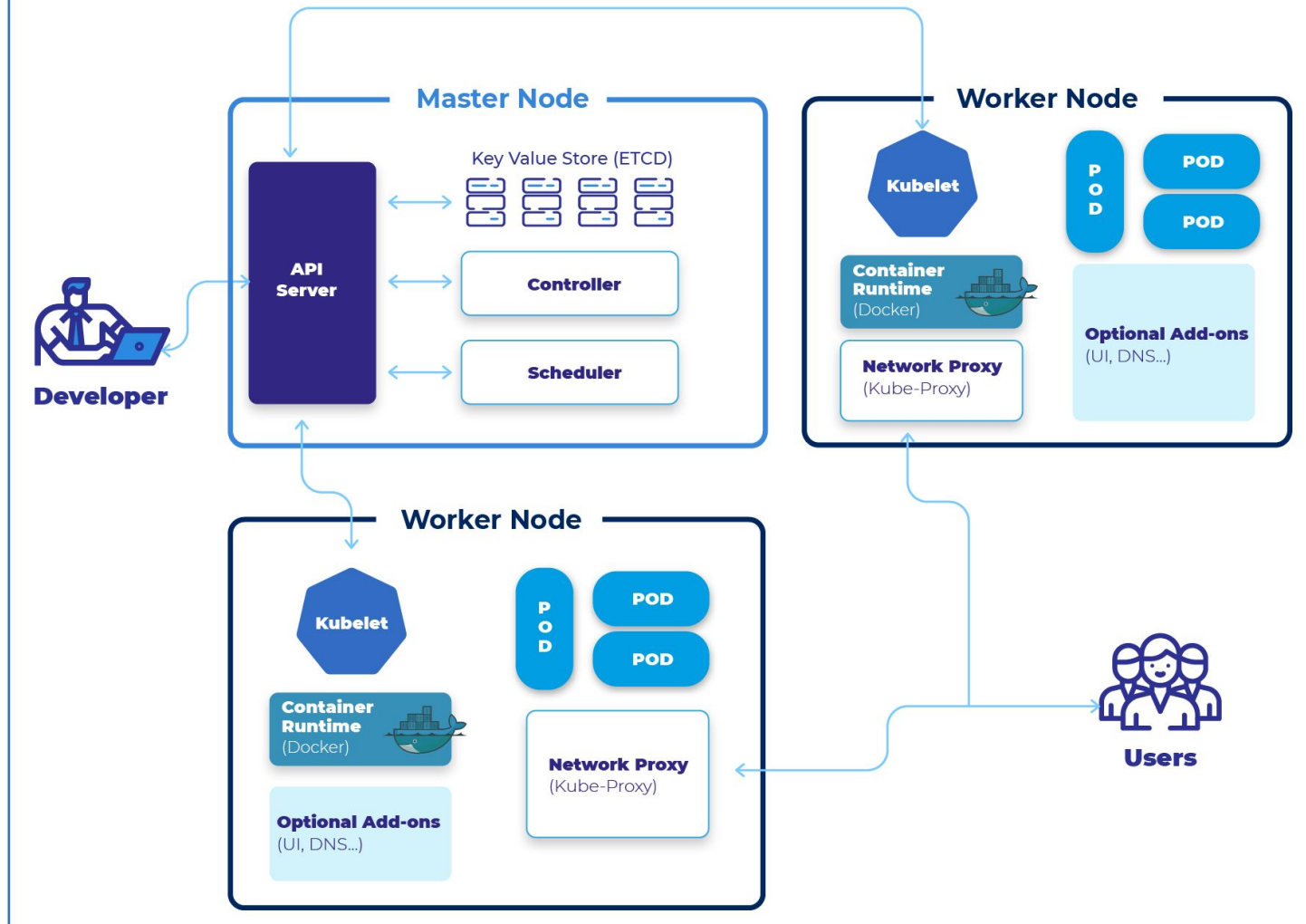
kubernetes

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

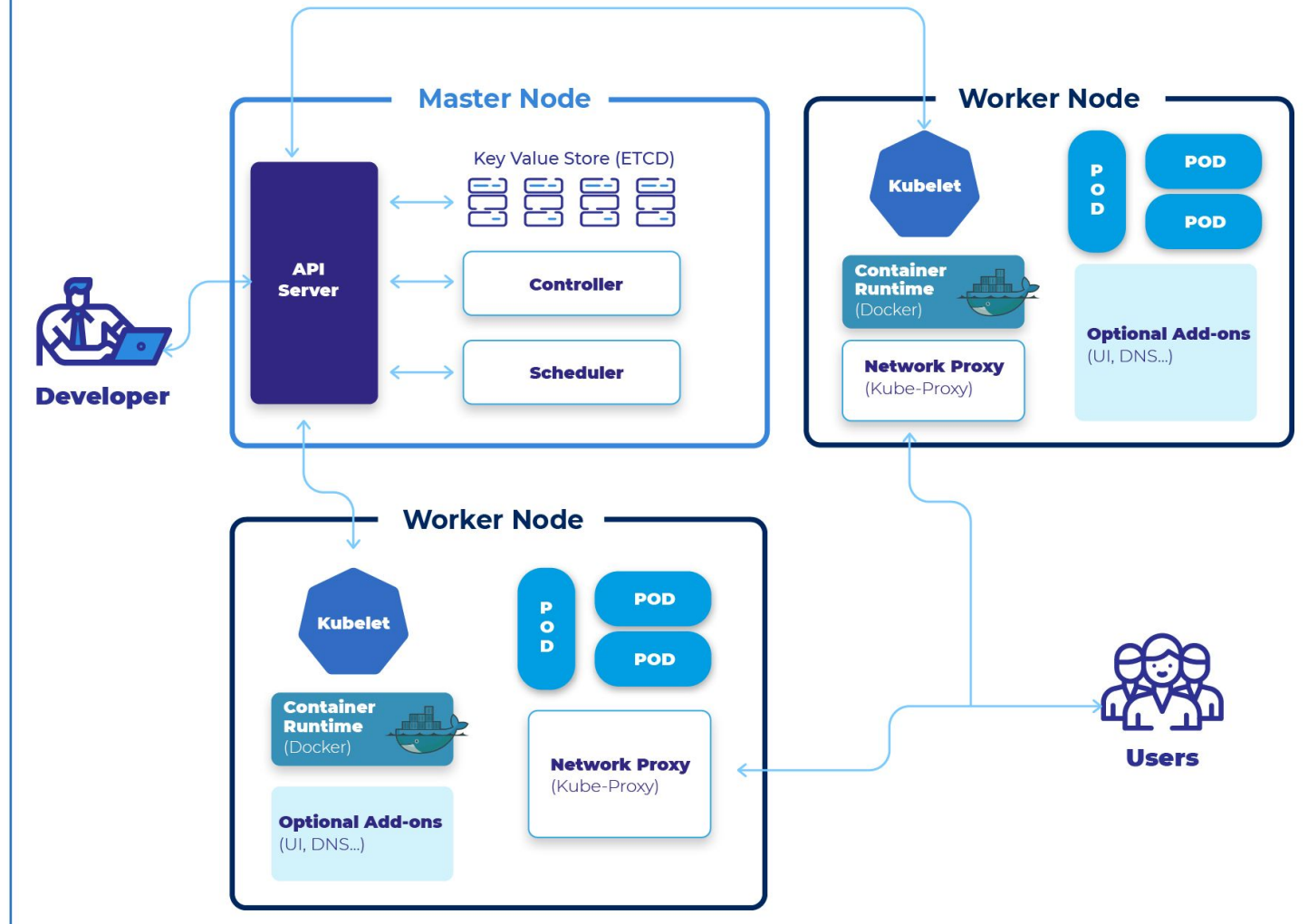
Kubernetes Architecture Diagram



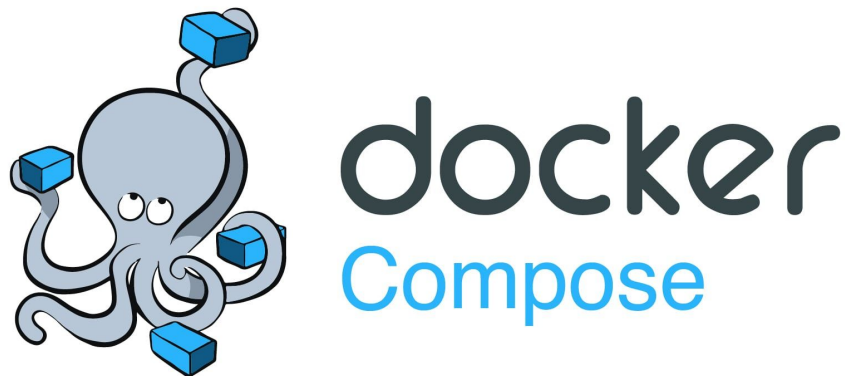
Kubernetes

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

- Finir le TP 3 Partie 2
 - GCP
 - Terraform
- Reprend tous les concepts vus ensemble
- N'hésitez pas à me poser des questions
- Introduction aux microservices
 - Docker Compose
 - Définition d'un ensemble de composants / services pour notre application
 - Kubernetes
 - Déploiement sur GKE : service managé Kubernetes sur GCP

Pour aller plus loin

- Terraformer
 - <https://github.com/GoogleCloudPlatform/terraformer>
- Terragrunt
 - <https://terragrunt.gruntwork.io/>
- Aller plus loin avec Kubernetes x Terraform
 - Kubernetes Terraform provider
 - Helm Terraform provider