

# **Lab 5: Continuous Integration (CI) and Continuous Delivery (CD) with Kubernetes**

# Sommaire:

## Sommaire:

### Préparation de l'environnement et des outils

#### Part 1: Continuous Integration (CI)

##### 1.1 Prepare the project

##### 1.2 Create the GitHub Actions workflow

##### 1.3 Test the workflow

##### 1.4 Machine User Credentials and Automatically-Provisioned Credentials

##### 1.5 Example: Configure OIDC with AWS and GitHub Actions

##### 1.6 Running Automated Infrastructure Tests

#### Part 2: Continuous Delivery (CD)

##### 2.1 Remote backend configuration for OpenTofu

##### Configuration du backend distant OpenTofu (S3 + DynamoDB)

##### Migration du state OpenTofu vers un backend distant

##### 2.2 Séparation des rôles : planification vs déploiement

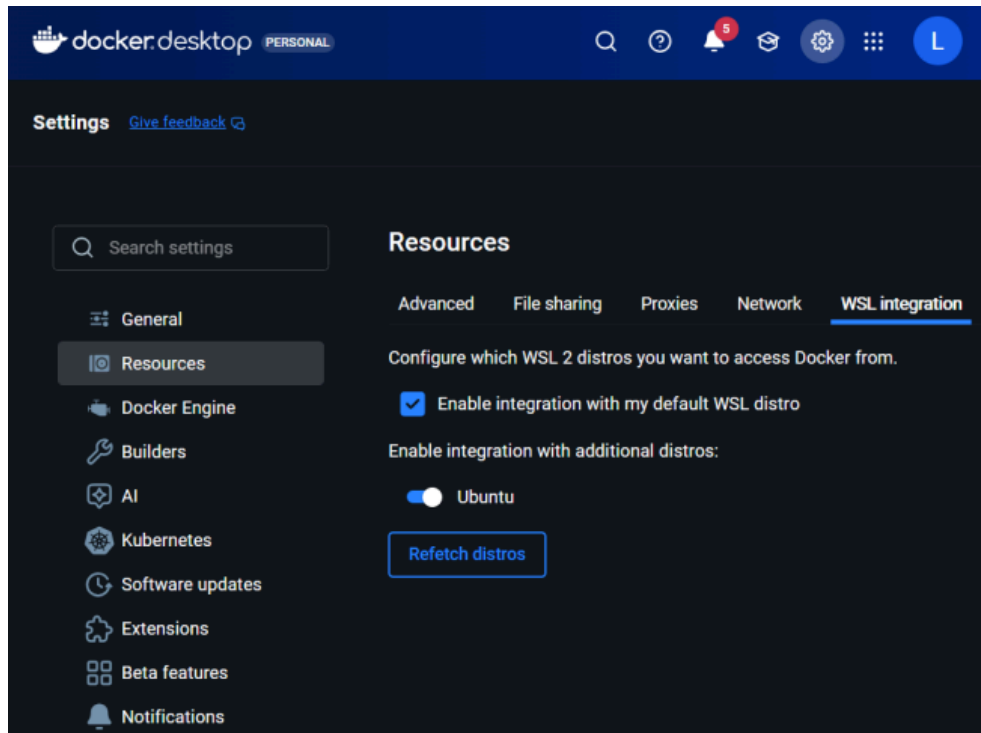
##### 2.3 — Workflow GitHub Actions : Planification du déploiement (tofu plan)

##### 2.4 Workflow GitHub Actions – Application du déploiement (tofu apply)

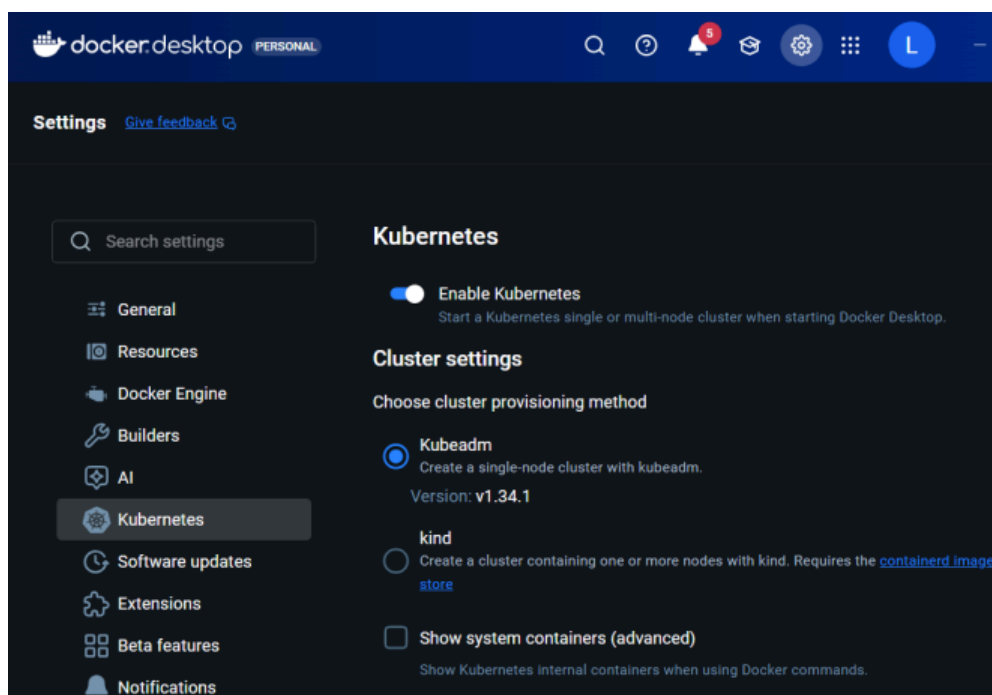
#### Conclusion

# Préparation de l'environnement et des outils

GitHub account AWS account Local Kubernetes cluster (Docker Desktop with Kubernetes enabled or Minikube) Installed and configured tools: Git, Docker, kubectl , OpenTofu, npm , Node.js, aws-cli , flux Utiliser Docker pour WSL



## Activer Kubernetes sur Docker



# Part 1: Continuous Integration (CI)

## 1.1 Prepare the project

Objectif : préparer l'arborescence du Lab 5 en réutilisant l'application Node du Lab 4, afin que les workflows GitHub Actions puissent trouver package.json et exécuter les tests dans le bon dossier.

Je me suis placé à la racine du projet et j'ai synchronisé la branche principale :

```
cd ~/devops_base
git checkout main
git pull origin main
```

J'ai créé le dossier td5 puis copié l'application sample-app du Lab 4 vers le Lab 5 :

```
mkdir -p td5/scripts
cp -r td4/scripts/sample-app td5/scripts/sample-app
```

Vérification rapide : j'ai confirmé que le dossier existe bien et contient l'app (fichiers + tests) via un tree/liste de fichiers.

```
└─views
PS C:\Users\antoi\Desktop\E4\DevOps\2_tds> dir td5/scripts/sample-app

Répertoire:
C:\Users\antoi\Desktop\E4\DevOps\2_tds\td5\scripts\sample-app

Mode                LastWriteTime         Length Name
----                -
d-----          02/12/2025   09:11             views
-a----          29/01/2026   17:17             280 app.js
-a----          29/01/2026   17:17            926 app.test.js
-a----          02/12/2025   09:11            209 build-docker-image.sh
-a----          02/12/2025   09:11            174 Dockerfile
-a----          02/12/2025   09:11        166458 package-lock.json
-a----          02/12/2025   09:11            473 package.json
-a----          02/12/2025   09:11            155 server.js
```

## 1.2 Create the GitHub Actions workflow

L'objectif de cette étape est de mettre en place un workflow GitHub Actions qui lance automatiquement les tests Jest de sample-app à chaque push.

Créer le dossier des workflows à la racine du repo :

```
mkdir -p .github/workflows
```

Créer le fichier `.github/workflows/app-tests.yml` avec ce contenu (c'est l'exemple du lab) :

```
name: Sample App Tests  
on: push
```

```
jobs:  
  sample_app_tests:  
    name: "Run Tests Using Jest"  
    runs-on: ubuntu-latest  
    steps:  
      - uses: actions/checkout@v3  
  
      - name: Install dependencies  
        working-directory: td5/scripts/sample-app  
        run: npm install  
  
      - name: Run tests  
        working-directory: td5/scripts/sample-app  
        run: npm test
```

*on: push* : déclenche à chaque push

*actions/checkout@v3* : récupère le code

Point clé : *working-directory: td5/scripts/sample-app* sinon GitHub Actions ne trouve pas *package.json* → donc *npm install/test* échoue.

## 1.3 Test the workflow

Commande sur main:

```
git add td5/scripts/sample-app .github/workflows/app-tests.yml  
git commit -m "Add sample-app and workflow"  
git push origin main
```

Sur test-workflow (bug puis fix)

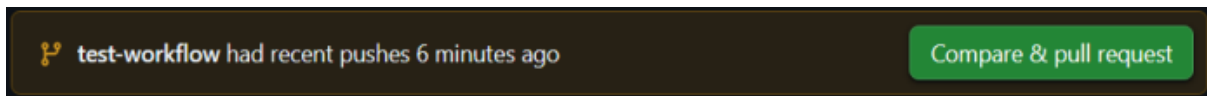
```
git checkout -b test-workflow
```

```
# bug app.js  
git add td5/scripts/sample-app/app.js  
git commit -m "Introduce intentional error"  
git push origin test-workflow
```

puis enfin :

```
# fix app.test.js  
git add td5/scripts/sample-app/app.test.js  
git commit -m "Update response text in test"  
git push origin test-workflow
```

Une fois le workflow en place, j'ai testé son fonctionnement réel. J'ai commencé par commit et push l'application ainsi que le workflow sur la branche main. Cela permet de vérifier que le pipeline fonctionne dans un cas nominal.

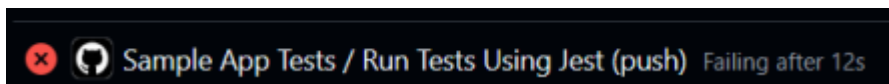


J'ai ensuite créé une branche dédiée :

```
git checkout -b test-workflow
```

Dans cette branche, j'ai introduit volontairement une erreur dans le fichier app.js, en modifiant le texte retourné par l'application.

Après avoir poussé la branche et créé une Pull Request, GitHub Actions s'est déclenché automatiquement. Comme attendu, les tests ont échoué car la valeur retournée ne correspondait plus à celle attendue par le test.



Détail de l'erreur :

```
18      Expected: "Hello, World!"  
19      Received: "DevOps Labs!"
```

J'ai ensuite corrigé le fichier de test `app.test.js` pour qu'il corresponde à la nouvelle valeur retournée par l'application. Après un nouveau commit et push, le workflow s'est relancé automatiquement et a cette fois-ci réussi. Sur la Pull Request, le check `Sample App Tests / Run Tests Using Jest` repasse en success après la correction (voir capture).

```
8 PASS ./app.test.js
9 Test the app
10 ✓ Get / should return DevOps Labs! (22 ms)
11 ✓ Get /name/Bob should return Hello, Bob! (9 ms)
12 ✓ Get /name should sanitize its input (4 ms)
```

## 1.4 Machine User Credentials and Automatically-Provisioned Credentials

Objectif : expliquer les deux façons classiques d'authentifier une automatisation (CI) sur un cloud, et justifier le choix retenu pour le Lab 5.

En CI/CD, on n'utilise jamais les identifiants d'un compte utilisateur réel. On passe par des identités dédiées à l'automatisation avec des droits limités.

Pour automatiser les tests et les déploiements, GitHub Actions doit pouvoir s'authentifier auprès d'AWS de manière sécurisée.

Deux solutions existent:

La première consiste à utiliser un Machine User IAM avec des clés d'accès stockées dans les secrets GitHub. Cette méthode fonctionne mais repose sur des credentials statiques, longue durée, et donc plus exposés en cas de fuite.

La seconde solution utilise des Automatically-Provisioned Credentials via OIDC (OpenID Connect). Dans ce cas, aucune clé AWS permanente n'est stockée. GitHub Actions obtient des identifiants temporaires pour assumer un rôle IAM précis. Cette approche est plus sécurisée et conforme aux bonnes pratiques de sécurité.

Dans la section suivante, je mets en place OIDC entre GitHub et AWS, puis je crée des rôles IAM dédiés (`tests/plan/apply`).

## 1.5 Example: Configure OIDC with AWS and GitHub Actions

Création de la branche demandée :

```
git checkout main
```

```
git pull origin main
git checkout -b opentofu-tests
```

Création du "root module" CI/CD permissions :

```
mkdir -p td5/scripts/tofu/live/ci-cd-permissions
cd td5/scripts/tofu/live/ci-cd-permissions
```

Création du fichier main.tf :

```
provider "aws" {
    region = "us-east-2"
}

module "oidc_provider" {
    source      = "../../modules/github-aws-oidc"
    provider_url = "https://token.actions.githubusercontent.com"
}

module "iam_roles" {
    source              = "../../modules/gh-actions-iam-roles"
    name                = "lambda-sample"
    oidc_provider_arn   = module.oidc_provider.oidc_provider_arn

    enable_iam_role_for_testing = true
    enable_iam_role_for_plan    = true
    enable_iam_role_for_apply   = true

    github_repo      = "liantoine/devops-labs"
    lambda_base_name = "lambda-sample"

    # Choisis des noms stables dès maintenant (même si tu créeras le
    bucket/table plus tard)
    tofu_state_bucket =
"liantoine-devops-tofu-state-021490341961"
    tofu_state_dynamodb_table = "liantoine-devops-tofu-state-locks"
}
```

Création de outputs.tf :

```
output "lambda_test_role_arn" {
    value = module.iam_roles.lambda_test_role_arn
}

output "lambda_deploy_plan_role_arn" {
```

```
    value = module.iam_roles.lambda_deploy_plan_role_arn
}

output "lambda_deploy_apply_role_arn" {
    value = module.iam_roles.lambda_deploy_apply_role_arn
}
```

### Déploiement :

*tofu init*

*tofu apply*

*tofu output -raw lambda\_test\_role\_arn*

*tofu output -raw lambda\_deploy\_plan\_role\_arn*

*tofu output -raw lambda\_deploy\_apply\_role\_arn*

TEST\_ROLE\_ARN = lambda\_test\_role\_arn

PLAN\_ROLE\_ARN = lambda\_deploy\_plan\_role\_arn

APPLY\_ROLE\_ARN = lambda\_deploy\_apply\_role\_arn

Dans cette partie, j'ai mis en place une authentification sécurisée entre GitHub Actions et AWS en utilisant OIDC (OpenID Connect). L'objectif était de permettre à des workflows GitHub d'accéder à AWS sans utiliser de clés statiques.

J'ai utilisé le module OpenTofu fourni dans le TP pour déclarer GitHub comme *Identity Provider* côté AWS. Ce provider permet à GitHub Actions de demander un token temporaire lorsqu'un workflow s'exécute. AWS vérifie alors l'identité du workflow avant d'autoriser l'accès aux ressources.

Ensuite, j'ai créé plusieurs rôles IAM distincts, chacun avec un objectif précis :

- un rôle dédié aux tests d'infrastructure
- un rôle pour la planification des changements
- un rôle pour l'application des changements

Cette séparation m'a permis de comprendre l'importance du principe du moindre privilège, chaque action automatisée dispose uniquement des droits dont elle a besoin.

Une fois les rôles créés, j'ai récupéré leurs ARN et je les ai stockés dans les secrets GitHub. Ainsi, les workflows peuvent assumer dynamiquement le bon rôle au moment de leur exécution, sans exposer d'identifiants sensibles dans le dépôt.

## 1.6 Running Automated Infrastructure Tests

Dans cette étape, l'objectif est de tester automatiquement l'infrastructure de la même manière que le code applicatif.

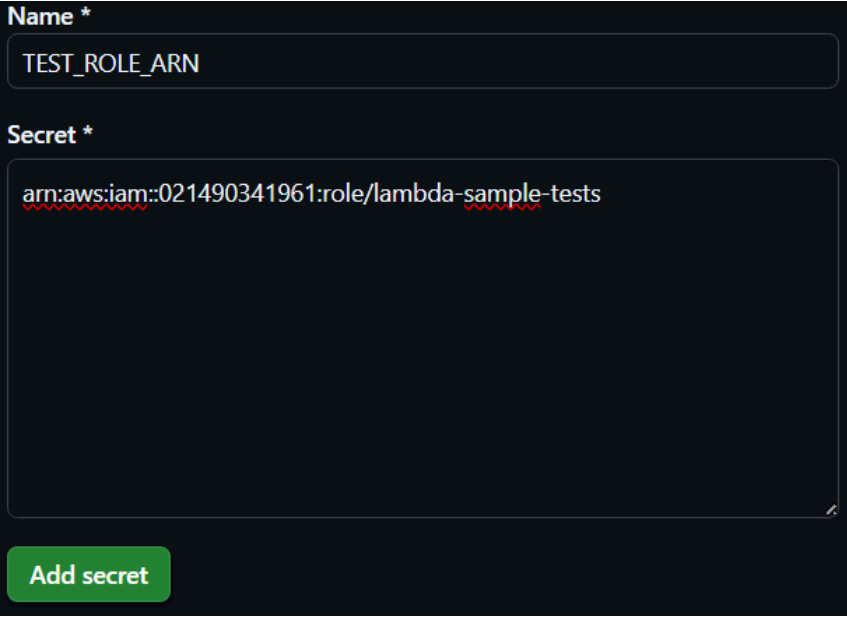
### Running Automated Infrastructure Tests

Dans cette partie, j'ai mis en place des tests automatisés de l'infrastructure afin de vérifier que les configurations OpenTofu sont correctes avant tout déploiement.

L'idée est d'appliquer les principes de l'intégration continue non seulement au code applicatif, mais aussi à l'infrastructure.

Pour cela, j'ai utilisé GitHub Actions afin d'exécuter automatiquement des tests OpenTofu à chaque push. Ces tests permettent de détecter des erreurs de configuration AWS (permissions, ressources manquantes, incohérences) avant qu'elles n'impactent un environnement réel.

Afin d'exécuter les tests d'infrastructure depuis GitHub Actions sans stocker de clés AWS statiques, j'ai utilisé l'authentification OIDC mise en place précédemment. J'ai créé un rôle IAM dédié aux tests d'infrastructure, que GitHub Actions peut assumer temporairement lors de l'exécution du workflow. L'ARN de ce rôle a ensuite été stocké dans les secrets GitHub, ce qui permet au pipeline d'y accéder sans exposer d'informations sensibles dans le dépôt.



The screenshot shows the GitHub 'Add secret' interface. The 'Name' field is labeled 'Name \*' and contains the text 'TEST\_ROLE\_ARN'. The 'Secret' field is labeled 'Secret \*' and contains the text 'arn:aws:iam::021490341961:role/lambda-sample-tests'. At the bottom left, there is a green button labeled 'Add secret'.

Avant d'automatiser les tests dans GitHub Actions, j'ai préparé le code d'infrastructure à tester dans TD5. J'ai vérifié que tous les modules nécessaires étaient présents (notamment api-gateway) et j'ai corrigé les chemins source dans main.tf pour qu'ils pointent vers td5/scripts/tofu/modules.

J'ai ensuite ajouté un fichier variables.tf afin d'utiliser var.name : cela permet au workflow CI de générer des noms uniques via TF\_VAR\_name et d'éviter des conflits de ressources entre exécutions.

Enfin, j'ai validé localement l'ensemble avec tofu init -backend=false -input=false puis tofu test -verbose : les tests passent (2 passed, 0 failed) et l'endpoint HTTP retourne bien 200.

```
Success! 2 passed, 0 failed.  
status_code = 200
```

#### Création du workflow GitHub Actions pour les tests d'infrastructure

J'ai ensuite créé un workflow GitHub Actions dédié aux tests OpenTofu. Ce workflow est déclenché automatiquement à chaque push et réalise les étapes suivantes :

- récupération du dépôt  
configuration des credentials AWS via OIDC
- installation d'OpenTofu
- exécution des tests d'infrastructure avec tofu test

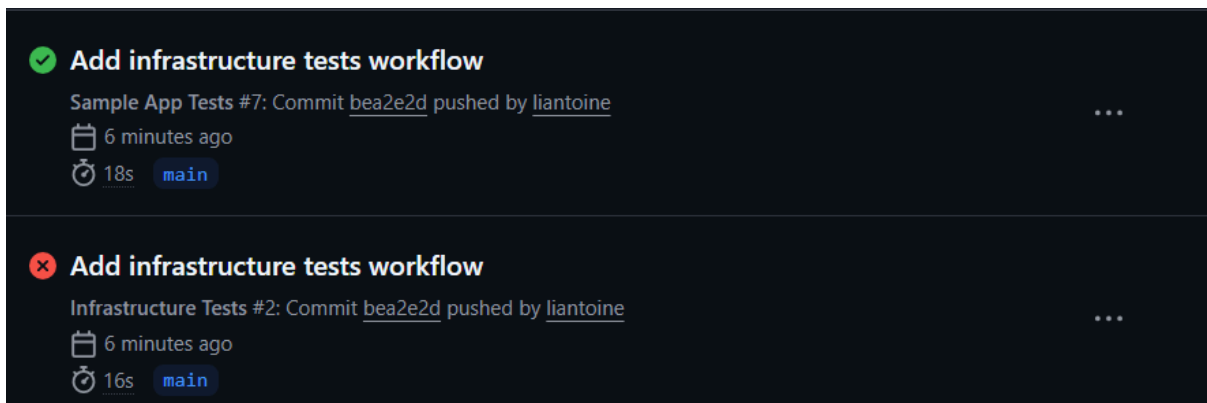
Pour éviter les conflits de ressources AWS entre plusieurs exécutions du pipeline, j'ai utilisé des variables dynamiques fournies par GitHub (github.run\_id / github.run\_number) afin de générer des noms uniques à chaque test.

```
! infra-tests.yml X
C: > Users > antoi > Desktop > E4 > DevOps > 2_tds > .github > workflows > ! infra-tests.yml >
4
5 jobs:
6   opentofu_test:
7     name: "Run OpenTofu tests"
8     runs-on: ubuntu-latest
9
10    permissions:
11      id-token: write
12      contents: read
13
14    steps:
15      - uses: actions/checkout@v3
16
17      - name: Configure AWS credentials (OIDC)
18        uses: aws-actions/configure-aws-credentials@v3
19        with:
20          role-to-assume: ${ secrets.TEST_ROLE_ARN }
21          role-session-name: infra-tests-${ github.run_number }
22          aws-region: us-east-2
23
24      - name: Setup OpenTofu
25        uses: opentofu/setup-opentofu@v1
26
27      - name: Run OpenTofu tests
28        env:
29          TF_VAR_name: lambda-sample-${ github.run_id }
30        working-directory: td5/scripts/tofu/live/lambda-sample
31        run: |
32          tofu init -backend=false -input=false
33          tofu test -verbose
34
```

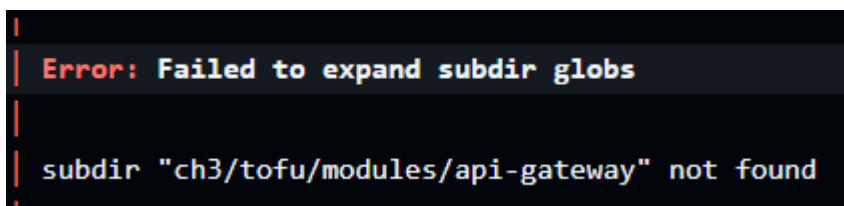
Afin de sécuriser l'accès à AWS, j'ai configuré le workflow pour assumer dynamiquement un rôle IAM à l'aide d'OIDC. Le rôle utilisé pour les tests est référencé via un secret GitHub, ce qui évite toute exposition de clés AWS statiques dans le dépôt. Cette configuration garantit que les tests d'infrastructure sont exécutés de manière sécurisée et contrôlée.

### Exécution et validation des tests d'infrastructure

Une fois le workflow de tests d'infrastructure configuré, j'ai effectué un push afin de déclencher automatiquement son exécution. Le workflow s'est alors lancé dans l'onglet Actions de GitHub. Lors de cette exécution, GitHub Actions a récupéré le dépôt, configuré les credentials AWS via OIDC, installé OpenTofu, puis lancé les tests d'infrastructure à l'aide de la commande `tofu test`. Cette étape permet de vérifier que la configuration OpenTofu est valide et cohérente avant tout déploiement réel sur AWS. L'exécution automatique de ces tests m'a permis de m'assurer que l'infrastructure définie respecte les contraintes attendues et que les permissions AWS sont correctement configurées.



Lors de la première exécution du workflow de tests d'infrastructure, celui-ci a échoué.

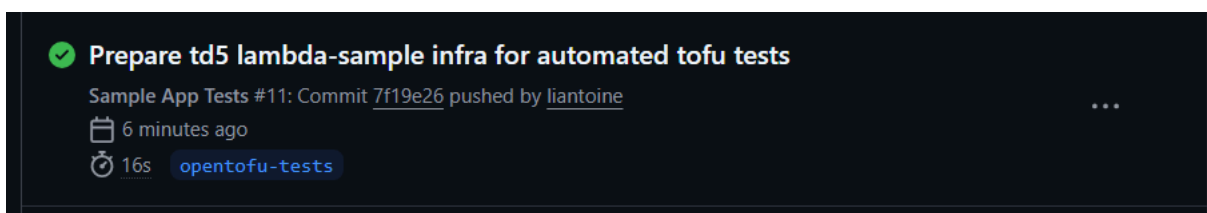


Lors de l'exécution du workflow de tests d'infrastructure, celui-ci a échoué lors de la phase d'exécution de tofu test. Après analyse des logs GitHub Actions, j'ai identifié que l'erreur provenait d'un chemin de module OpenTofu introuvable (ch3/tofu/modules/api-gateway).

Cet échec n'est pas lié à l'authentification AWS ni à la configuration OIDC, qui ont correctement fonctionné. Il s'agit d'un problème de dépendance de module, illustrant l'importance de la cohérence des chemins et des modules utilisés dans une infrastructure as code.

Cette erreur m'a permis de mieux comprendre la dépendance entre les modules OpenTofu et l'importance du respect strict des chemins dans une infrastructure as code.

J'ai corrigé le problème en ajoutant le module manquant api-gateway dans td5/scripts/tofu/modules et en modifiant main.tf pour que le module gateway pointe vers ../../modules/api-gateway (au lieu du chemin ch3/...). Après un nouveau commit et push, le workflow s'est relancé et les tests ont réussi.



On observe ensuite une exécution en succès après correction des chemins de modules.

## Part 2: Continuous Delivery (CD)

### 2.1 Remote backend configuration for OpenTofu

#### Configuration du backend distant OpenTofu (S3 + DynamoDB)

Dans cette étape, j'ai mis en place un backend distant pour stocker l'état OpenTofu de l'infrastructure. Pour cela, j'ai utilisé un bucket S3 afin de centraliser l'état et une table DynamoDB pour gérer le verrouillage et éviter les conflits lors des exécutions concurrentes.

La création de ces ressources a été réalisée à l'aide du module `state-bucket` fourni dans le TP. Lors de la première exécution, la création du bucket a échoué car le nom de bucket S3 doit être unique mondialement. J'ai donc choisi un nom unique en ajoutant un suffixe (ex : année / identifiant), puis la création du bucket S3 et de la table DynamoDB a réussi.

```
provider "aws" {
  region = "us-east-2"
}

module "state" {
  source = "github.com/brikis98/devops-book//ch5/tofu/modules/state-bucket"

  # TODO: fill in your own bucket name!
  name = "fundamentals-of-devops-tofu-state-antoine-2026"
}
```

Apply complete! Resources: 5 added, 0 changed, 1 destroyed.

Outputs:

```
dynamodb_table_name = "fundamentals-of-devops-tofu-state-antoine-2026"
s3_bucket_name      = "fundamentals-of-devops-tofu-state-antoine-2026"
```

#### Migration du state OpenTofu vers un backend distant

Une fois le bucket S3 et la table DynamoDB créés, j'ai configuré OpenTofu pour utiliser un backend distant basé sur S3. Lors de la réinitialisation, OpenTofu a détecté l'existence d'un état local et m'a proposé de le migrer vers le backend S3. Le verrouillage via DynamoDB empêche deux exécutions concurrentes (ex : deux apply) de modifier l'état en même temps.

Après validation de cette migration, l'état a été correctement transféré vers le backend distant. L'état de l'infrastructure est désormais stocké de manière centralisée et sécurisée, ce qui permet une meilleure gestion de l'infrastructure dans un contexte collaboratif.

```

terraform {
  backend "s3" {
    bucket      = "fundamentals-of-devops-tofu-state-antoine-2026"
    key         = "td5/scripts/tofu/live/tofu-state"
    region      = "us-east-2"
    encrypt     = true
    dynamodb_table = "fundamentals-of-devops-tofu-state-antoine-2026"
  }
}

```

```

PS C:\Users\antoi\Desktop\E4\DevOps\2_tds\td5\scripts\tofu\live\tofu-state> tofu init

Initializing the backend...
Do you want to copy existing state to the new backend?
Pre-existing state was found while migrating the previous "local" backend to the
newly configured "s3" backend. No existing state was found in the newly
configured "s3" backend. Do you want to copy this state to the new "s3"
backend? Enter "yes" to copy and "no" to start with an empty state.

Enter a value: yes

Successfully configured the backend "s3"! OpenTofu will automatically
use this backend unless the backend configuration changes.
Initializing modules...

Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v5.100.0

OpenTofu has been successfully initialized!

You may now begin working with OpenTofu. Try running "tofu plan" to see
any changes that are required for your infrastructure. All OpenTofu commands
should now work.

If you ever set or change modules or backend configuration for OpenTofu,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.

```

## 2.2 Séparation des rôles : planification vs déploiement

Dans cette étape, j'ai mis en place une séparation stricte des rôles IAM utilisés par les workflows GitHub Actions. J'ai créé trois rôles IAM distincts via OpenTofu : un rôle pour les tests, un rôle pour la planification (tofu plan) et un rôle pour l'application (tofu apply).

- Test role : permissions minimales pour exécuter tofu test (validation infra)
- Plan role : lecture + génération du plan (pas de modification)
- Apply role : permissions d'écriture nécessaires pour créer/modifier les ressources

Cette séparation permet d'appliquer le principe du moindre privilège et de sécuriser le pipeline de déploiement continu, conformément aux bonnes pratiques DevOps.

```
Plan: 7 to add, 0 to change, 0 to destroy.
```

```
Changes to Outputs:
```

```
+ lambda_deploy_apply_role_arn = (known after apply)
+ lambda_deploy_plan_role_arn  = (known after apply)
+ lambda_test_role_arn         = (known after apply)
```

```
Apply complete! Resources: 6 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
lambda_deploy_apply_role_arn = "arn:aws:iam::021490341961:role/lambda-sample-apply"
lambda_deploy_plan_role_arn  = "arn:aws:iam::021490341961:role/lambda-sample-plan"
lambda_test_role_arn         = "arn:aws:iam::021490341961:role/lambda-sample-tests"
```

Les ARN sont ensuite stockés dans GitHub Secrets : TEST\_ROLE\_ARN, PLAN\_ROLE\_ARN, APPLY\_ROLE\_ARN.

## 2.3 — Workflow GitHub Actions : Planification du déploiement (tofu plan)

J'ai ajouté un workflow GitHub Actions de planification (tofu plan) déclenché à chaque Pull Request vers main. Le workflow s'authentifie sur AWS via OIDC (rôle PLAN\_ROLE\_ARN), exécute tofu init puis tofu plan, et publie le résultat (plan.txt) en artifact.

```
name: Infrastructure Plan (OpenTofu)

on:
  pull_request:
  ...

  role-to-assume: ${ secrets.PLAN_ROLE_ARN }
  aws-region: us-east-2
  role-session-name: infra-plan-${ github.run_number }

- name: Setup OpenTofu
  uses: opentofu/setup-opentofu@v1

- name: OpenTofu init (remote backend)
  working-directory: td5/scripts/tofu/live/lambda-sample
  env:
```

```

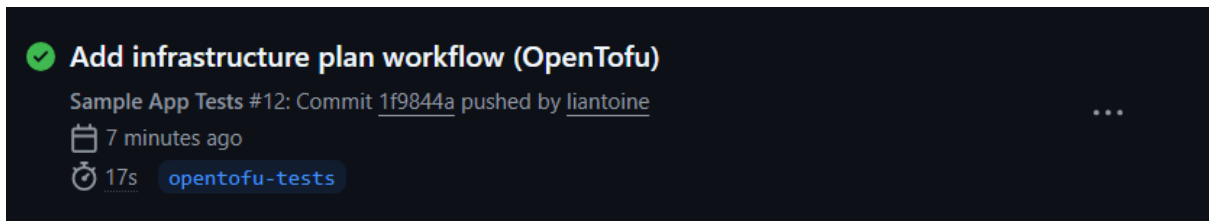
    TF_VAR_name: lambda-sample-${{ github.run_id }}
  run: |
    tofu init -input=false

- name: OpenTofu plan
  working-directory: td5/scripts/tofu/live/lambda-sample
  env:
    TF_VAR_name: lambda-sample-${{ github.run_id }}
  run: |
    tofu plan -input=false -out=plan.tfplan
    tofu show -no-color plan.tfplan > plan.txt

- name: Upload plan artifact
  uses: actions/upload-artifact@v4
  with:
    name: tofu-plan
    path: td5/scripts/tofu/live/lambda-sample/plan.txt

```

Extrait du workflow infra-plan.yml : déclenchement PR + authentification OIDC + exécution du plan.



À chaque Pull Request, GitHub Actions exécute automatiquement le workflow de planification.

Cette étape permet de valider les changements à venir sans modifier l'infrastructure (contrairement à tofu apply).

## 2.4 Workflow GitHub Actions – Application du déploiement (tofu apply)

```
name: Infrastructure Apply (OpenTofu)

on:
  workflow_dispatch:
...
    role-to-assume: ${ secrets.APPLY_ROLE_ARN }
...
    tofu init -input=false
...
    tofu apply -input=false -auto-approve
```

### 2 workflow runs

Event ▾ Status ▾ Branch ▾ Actor ▾

This workflow has a `workflow_dispatch` event trigger.

Run workflow ▾

#### 🔴 Infrastructure Apply

Infrastructure Apply #2: Manually run by [liantoine](#)

...

📅 now

🕒 In progress

main

#### ✅ [Add infrastructure apply workflow \(OpenTofu\)](#)

Sample App Tests #13: Commit [7a67304](#) pushed by [liantoine](#)

...

📅 1 minute ago

🕒 16s

opentofu-tests

J'ai mis en place un workflow GitHub Actions dédié à l'application des changements (tofu apply). Pour éviter tout déploiement involontaire, il est déclenché manuellement via `workflow_dispatch`.

Le workflow s'authentifie sur AWS via OIDC en assumant le rôle `APPLY_ROLE_ARN`, initialise le backend distant (S3/DynamoDB), puis exécute `tofu apply -auto-approve` sur l'infrastructure.

Cette séparation entre plan (PR) et apply (manuel) limite le risque de déploiement non intentionnel et applique le principe du moindre privilège.

## Conclusion

Ce LAB m'a permis de mettre en place une chaîne CI/CD complète avec GitHub Actions, en appliquant les bonnes pratiques DevOps : tests automatiques pour le code applicatif (CI), tests automatiques pour l'infrastructure OpenTofu, et authentification sécurisée à AWS via OIDC sans clés statiques. J'ai également configuré un backend distant (S3 + DynamoDB) afin de centraliser l'état OpenTofu et d'éviter les conflits d'exécution. Enfin, j'ai séparé les rôles IAM (tests / plan / apply) pour respecter le principe du moindre privilège, puis automatisé la planification et l'application du déploiement via workflows GitHub Actions, ce qui garantit des déploiements reproductibles, traçables et sécurisés.