Documentation Technique de l'Action GitHub TFDefect

Équipe: PFE017

Membres de l'équipe	Code permanent
Hamza Bellahsen	BELH92090100
Amine Merri	MERA79050006
Abdelmoumene Haouari	HAOA25029805
Mohammed Gallizioli	GALM19059709
Mohammed Amine Gadi	GADM81070009

Introduction

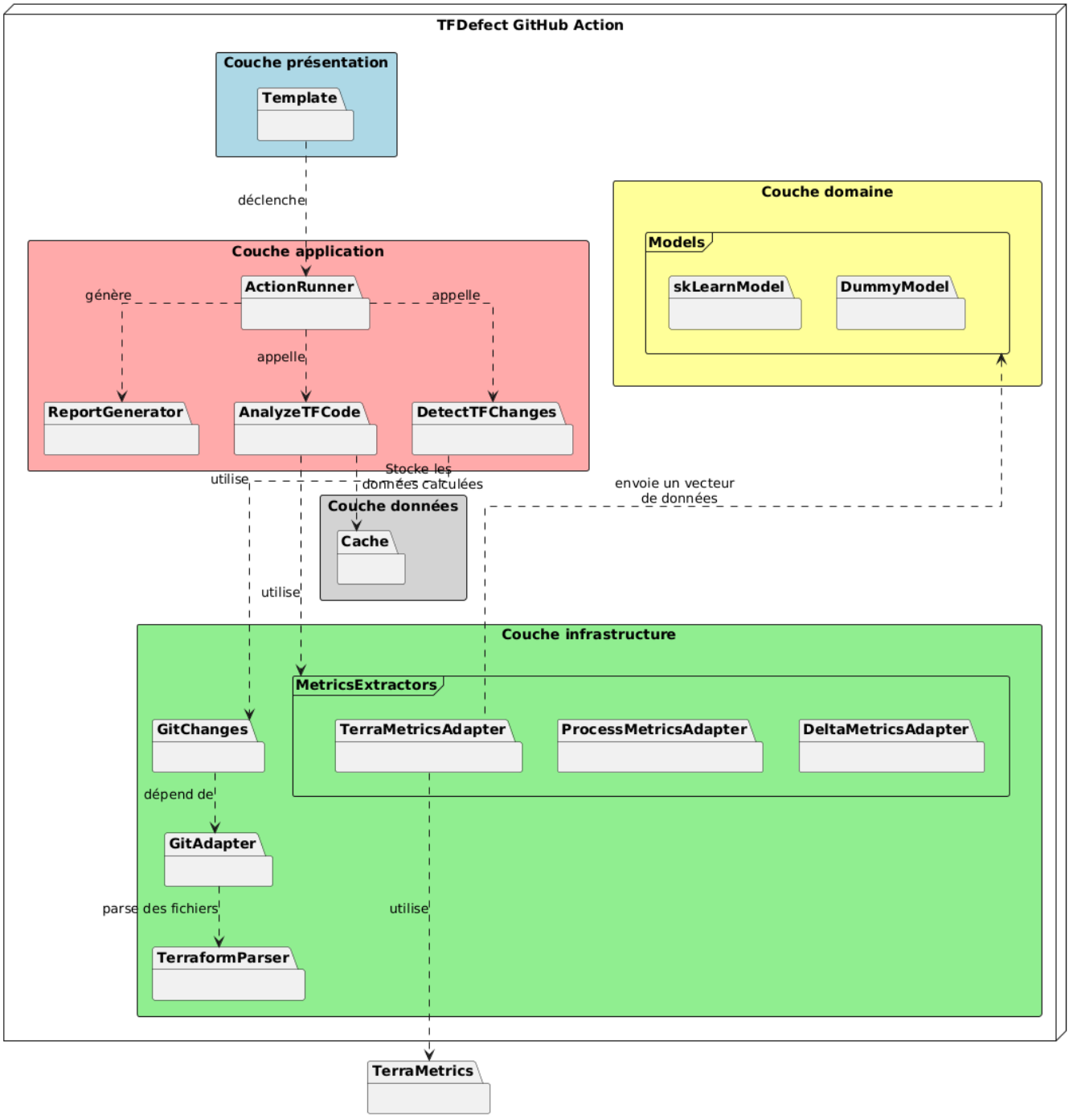
Ce document décrit l'architecture technique du projet en détaillant les différentes couches, composants et technologies utilisées. Son objectif est de fournir une vue d’ensemble claire et structurée de l’architecture de l'Action GitHub, tout en mettant en avant les interactions entre les différentes parties de l’application.

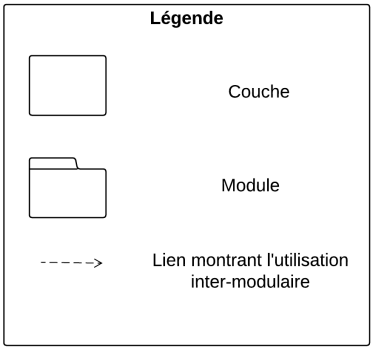
Pour cela, nous présenterons une vue architecturale de type module afin d'illustrer les dépendances fonctionnelles entre les composants du système, ainsi qu'un diagramme de classes détaillant la structure interne et les relations entre les différentes entités logicielles. Nous aborderons également les technologies utilisées telles, que les outils DevOps et les bibliothèques Python.

Vues architecturales et diagrammes

Vue architecturale de type module

La vue architecturale suivante est de type "Utilise". Elle permet de représenter les dépendances fonctionnelles entre les différents composants du système de détection de bogues. Cela permet de comprendre quels modules utilisent quels autres modules pour exécuter leurs fonctionnalités.





Description de l'architecture

Le système est conçu selon les principes de la *Clean Architecture*, permettant une séparation claire des responsabilités et une grande maintenabilité.

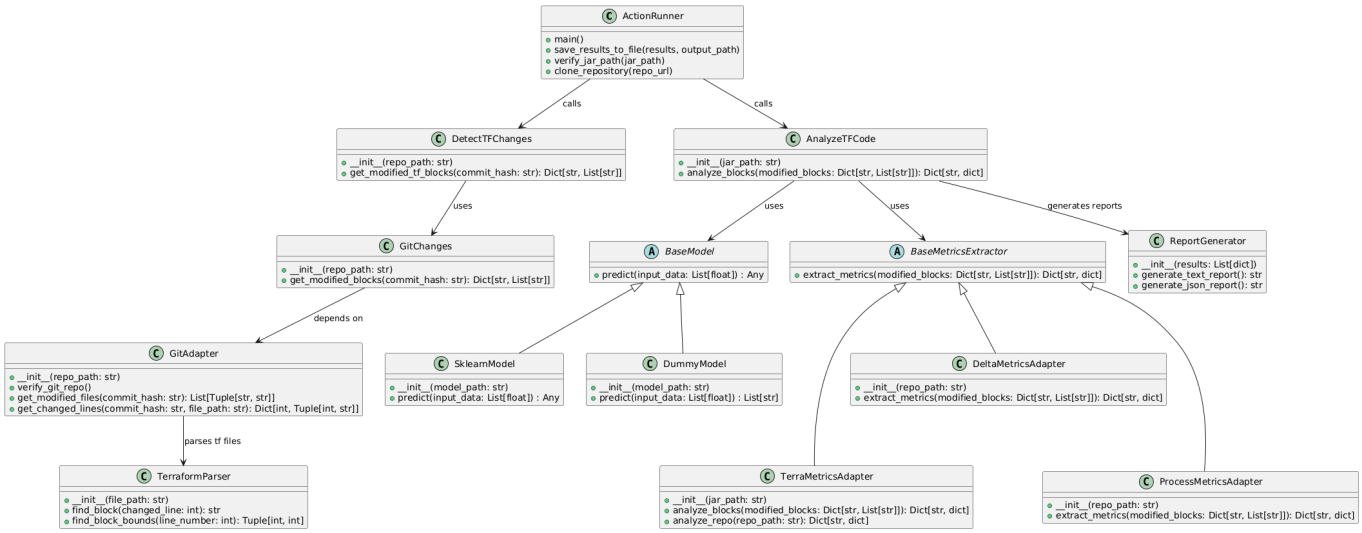
Couches

L'architecture de l'Action GitHub TFDefect se compose des 4 couches suivantes:

- **Couche présentation** : Il s'agit de l'interface utilisateur qui permet d'afficher les résultats et d'interagir avec l'action GitHub.
- **Couche application** : Elle contient les cas d'utilisation spécifiques de l'application. Le module `ActionRunner` permet l'exécution du processus d'analyse. Le module `AnalyzeTFCode` analyse les blocs Terraform modifiés pour en tirer des métriques. Le module `DetectTFChanges` identifie les blocs Terraform modifiés. Quant au module `ReportGenerator`, il génère des rapports à partir des résultats.
- **Couche données**: Elle contient une cache qui stocke les données calculées temporairement pour optimiser les performances et éviter des calculs inutiles.
- **Couche domaine** : C'est le cœur de l'application, où sont définies les règles métiers et les modèles de données.
  - `Models`: Regroupe les modèles d'apprentissage automatique utilisés pour l'analyse des métriques. On y retrouve le module `skLearnModel` qui Implémente un modèle basé sur scikit-learn pour prédire des anomalies ou tendances dans les métriques Terraform. On y retrouve également le module `DummyModel`, qui est un modèle naïf utilisé pour faire des tests.
- **Couche infrastructure** : Elle gère les interactions avec des services externes (Git, fichiers Terraform, etc.). Elle contient les adaptateurs `GitChanges`, `GitAdapter` et `TerraformParser`. Elle contient également le bloc `MetricsExtractors` qui regroupe les 3 méthodes de calcul des métriques suivantes: `TerraMetricsAdapter`, `ProcessMetricsAdapter` et `DeltaMetricsAdapter`.
- **Autre (externe)**:
  - `TerraMetrics`: Outil qui permet de calculer des métriques pour un fichier Terraform

Diagramme de classes

Le diagramme de classes suivant illustre la structure du système en mettant en évidence les différentes classes, leurs responsabilités et leurs relations.



Description des composants

- **ActionRunner** : Gère l'exécution du processus global, y compris la gestion des résultats, la vérification des fichiers JAR et le clonage du dépôt Git.
- **DetectTFChanges** : Identifie les blocs Terraform modifiés dans un commit donné.
- **AnalyzeTFCode** : Analyse les blocs Terraform extraits pour en tirer des métriques ou des informations utiles.
- **BaseModel** : Classe abstraite pour un modèle de prédiction.
- **SklearnModel** : Étend `BaseModel` en utilisant un modèle basé sur `scikit-learn` pour effectuer des prédictions sur des données d'entrée.
- **DummyModel** : Étend `BaseModel` en utilisant un modèle fictif utilisé pour les tests.
- **ReportGenerator** : Génère des rapports textuels ou JSON à partir des résultats de l'analyse.
- **GitChanges** : Récupère les fichiers modifiés dans un commit donné à l'aide de `GitAdapter`.
- **GitAdapter** : Interface permettant d'interagir avec un dépôt Git (vérification, extraction des fichiers modifiés et des lignes de code changées).
- **TerraformParser** : Analyse les fichiers Terraform pour identifier et extraire des blocs spécifiques.
- **BaseMetricsExtractor**: Classe abstraite de base pour les méthodes de calcul de métriques.
- **TerraMetricsAdapter** : Calcule des métriques spécifiques à partir des blocs Terraform à l'aide de TerraMetrics.
- **ProcessMetricsAdapter**: Autre méthode de calcul des métriques spécifiques à partir des blocs Terraform.
- **DeltaMetricsAdapter**: Autre méthode de calcul des métriques spécifiques à partir des blocs Terraform.

Technologies utilisées

Le projet est développé en Python. Ce langage est utilisé pour exécuter l'Action GitHub, analyser les fichiers Terraform, interagir avec Git et appliquer des modèles de Machine Learning pour détecter les bogues.

Librairies utilisées:

Gestion des Dépôts Git

- **GitPython** : Permet d'interagir avec les dépôts Git, récupérer les fichiers modifiés et extraire l'historique des commits.

Gestion des Logs et Exécution de Commandes

- **Logging** : Utilisé pour enregistrer et suivre les événements et erreurs du programme.
- **Subprocess** : Permet d'exécuter des commandes externes et d'interagir avec des processus système.

Manipulation des Fichiers et Formats de Données

- **Tempfile** : Permet de créer et gérer des fichiers temporaires.
- **JSON** : Utilisé pour stocker et échanger des résultats d'analyse sous forme structurée.

Machine Learning et Analyse des Données

- **Scikit-learn** : Utilisé pour charger et exécuter les modèles de Machine Learning.

Tests et Assurance Qualité

- **pytest** : Framework utilisé pour écrire et exécuter des tests unitaires, d'intégration et fonctionnels.

Annotations et Sécurité

- **Typing** : Fournit des annotations de type pour améliorer la lisibilité et la robustesse du code.

Outils DevOps et CI/CD

- **GitHub Actions** : Automatisation des workflows (tests, analyse du code) après chaque commit.
- **YAML**: Utilisé pour les workflows GitHub Actions.

Gestion des dépendances et environnements

- **requirements.txt** : Liste des dépendances Python.
- **setup.py** : Script d'installation pour le package Python.
- **Pipenv** : Isolation des dépendances du projet Python.

Conclusion

Ce document a présenté l’architecture technique de l’Action GitHub en détaillant ses différentes couches, composants et technologies utilisées. En adoptant une approche basée sur la Clean Architecture, le système assure une séparation claire des responsabilités, ce qui facilite sa maintenance et son extensibilité.

La vue modulaire de type "Utilise" a illustré les interactions entre les composants et les dépendances fonctionnelles du système, tandis que le diagramme de classes a permis de mieux comprendre la structure interne et les relations entre les différentes entités logicielles.

Ainsi, cette architecture garantit une exécution fluide, une intégration efficace et une grande évolutivité, offrant une base solide pour l’amélioration continue de l’Action GitHub et son adaptation aux futurs besoins.