



ÉCOLE MOHAMMADIA D'INGÉNIEURS

DEVSECOPS

ATELIER N°5 : WORKFLOW GIT, MERGE REQUESTS ET
DÉTECTION DES VULNÉRABILITÉS

Compte-rendu Atelier N°5

Elèves :

Sami FAOUZI

Encadré par :

Pr. Asmaa Retbi

2ème année Génie Informatique

15 décembre 2025

Table des matières

1 Objectif du TP (résumé)	2
2 Étape 1 — Clonage du dépôt GitLab	2
2.1 Clonage du projet	2
2.2 Vérification de la structure du projet	2
3 Étape 2 — Vérification et exécution du projet	3
3.1 Build et démarrage	3
3.2 Test de l'API	3
3.3 Analyse du projet	4
4 Étape 3 — Crédation du pipeline GitLab CI (.gitlab-ci.yml)	4
5 Étape 4 — Crédation et protection des branches	5
5.1 Crédation des branches principales	5
5.2 Protection des branches sur GitLab	5
6 Étape 5 — Détection des vulnérabilités (branche security-analysis)	5
6.1 Modification du pipeline pour intégrer la sécurité	5
6.2 Merge Request de sécurité	6
6.3 Lancement et suivi du pipeline	6
7 Étape 6 — Analyse des résultats (SAST + Secret Detection)	7
7.1 Récupération des rapports (artifacts JSON)	8
7.2 Tableau de synthèse des findings (demandé par la prof)	9
7.3 Pourquoi ces findings sont des failles ?	11
8 Étape 7 — Correction des vulnérabilités (1 vulnérabilité = 1 branche = 1 MR)	11
8.1 7.1 Correction SAST : SQL Injection	11
8.2 7.2 Correction Secret Detection : secret hardcodé	12
9 Étape 8 — Fusion finale : develop → main (Rebase & Merge)	13
9.1 Comparatif rapide : Merge vs Rebase	14
9.2 Rebase final (test d'intégration) : develop → main	14
9.3 Merge final : develop → main	14
10 Étape 9 — Synthèse	15

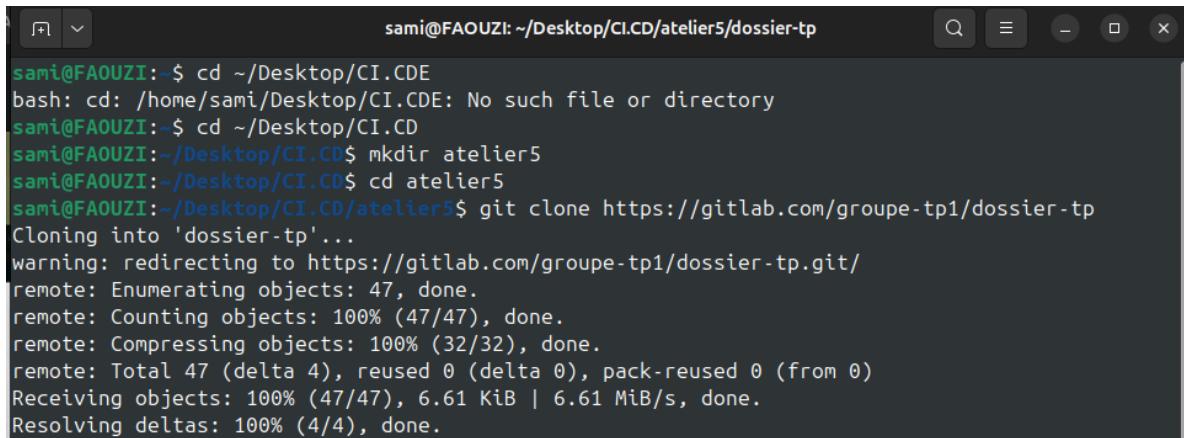
1 Objectif du TP (résumé)

Mettre en place un workflow Git propre (branches + MR + pipeline) et démarrer la détection automatisée de vulnérabilités (SAST, secrets, dépendances) via GitLab CI.

2 Étape 1 — Clonage du dépôt GitLab

2.1 Clonage du projet

Clonage du dépôt GitLab dans un répertoire local afin de récupérer le projet et démarrer les manipulations.



```
sami@FAOUZI:~$ cd ~/Desktop/CI.CDE
bash: cd: /home/sami/Desktop/CI.CDE: No such file or directory
sami@FAOUZI:~$ cd ~/Desktop/CI.CD
sami@FAOUZI:~/Desktop/CI.CD$ mkdir atelier5
sami@FAOUZI:~/Desktop/CI.CD$ cd atelier5
sami@FAOUZI:~/Desktop/CI.CD/atelier5$ git clone https://gitlab.com/groupe-tp1/dossier-tp
Cloning into 'dossier-tp'...
warning: redirecting to https://gitlab.com/groupe-tp1/dossier-tp.git/
remote: Enumerating objects: 47, done.
remote: Counting objects: 100% (47/47), done.
remote: Compressing objects: 100% (32/32), done.
remote: Total 47 (delta 4), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (47/47), 6.61 KiB | 6.61 MiB/s, done.
Resolving deltas: 100% (4/4), done.
```

FIGURE 1 – Clonage du dépôt GitLab en local.

2.2 Vérification de la structure du projet

Vérification rapide de l’arborescence du projet après clonage.

```
sami@FAOUZI:~/Desktop/CI.CD/atelier5/dossier-tp$ tree
.
├── pom.xml
├── README.md
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           └── devsecops
    │   │               ├── DevsecopsDemoApplication.java
    │   │               └── VulnerableController.java
    │   └── resources
    │       ├── application.properties
    │       ├── data.sql
    │       └── schema.sql
    └── test
        └── java
            └── com
                └── example
                    └── devsecops
                        └── DevsecopsDemoApplicationTests.java
    └── target
        ├── classes
        │   ├── application.properties
        │   └── com
        │       └── example
        │           └── devsecops
        │               ├── DevsecopsDemoApplication.class
        │               └── VulnerableController.class
        └── data.sql
        └── schema.sql
    └── test-classes
        └── com
            └── example
                └── devsecops
                    └── DevsecopsDemoApplicationTests.class
22 directories, 14 files
```

FIGURE 2 – Structure du projet après clonage.

3 Étape 2 — Vérification et exécution du projet

3.1 Build et démarrage

Compilation / lancement local pour confirmer que l’application démarre correctement.

```
E)
2025-12-15 16:01:22.353 DEBUG 12727 ... [main] o.s.jdbc.datasource.init.ScriptUtils : Executed SQL script from URL [file:/home/sami/Desktop/CI.CD/atelier5/dossier-tp/target/classes/data.sql] in 1 ms.
2025-12-15 16:01:22.405 INFO 12727 ... [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8082 (http) with context path ''
2025-12-15 16:01:22.412 INFO 12727 ... [main] c.e.devsecops.DevsecopsDemoApplication : Started DevsecopsDemoApplication in 1.492 seconds (JVM running for 1.732)
2025-12-15 16:01:22.413 INFO 12727 ... [main] o.s.b.a.ApplicationAvailabilityBean : Application availability state LivenessState changed to CORRECT
2025-12-15 16:04:37.372 INFO 12727 ... [nio-8082-exec-1] o.a.c.c.T[Tomcat].[localhost].[/] : Application availability state ReadinessState changed to ACCEPTING_TRAFFIC
2025-12-15 16:04:37.372 INFO 12727 ... [nio-8082-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2025-12-15 16:04:37.373 INFO 12727 ... [nio-8082-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
[[INFO]
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 17:42 min
[INFO] Finished at: 2025-12-15T16:18:52+01:00
[INFO]
```

FIGURE 3 – Build réussi (application prête à être testée).

3.2 Test de l’API

Test d’un endpoint de l’API via navigateur pour valider le bon fonctionnement.

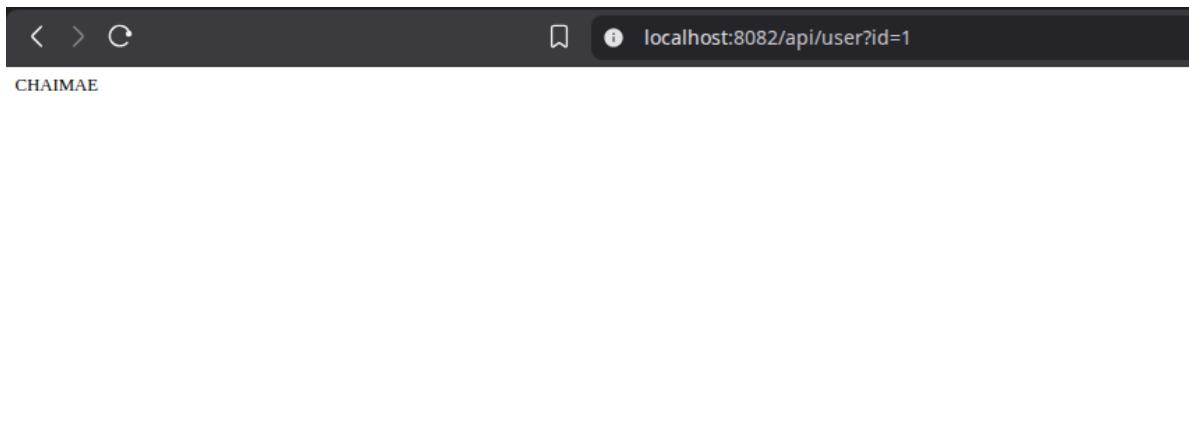


FIGURE 4 – Test de l’API dans le navigateur.

3.3 Analyse du projet

Le projet fourni est une application Spring Boot volontairement vulnérable, conçue pour pratiquer l’analyse DevSecOps et la détection automatique de failles. Le fichier `pom.xml` joue un rôle central : il définit l’identité du projet Maven, les dépendances (frameworks et bibliothèques), la configuration de compilation ainsi que l’exécution des tests, ce qui en fait également un point critique pour identifier d’éventuelles dépendances vulnérables (CVE). La logique applicative est exposée via un contrôleur (ex. `VulnerableController`) qui illustre des mauvaises pratiques de sécurité typiques (secrets codés en dur, requêtes SQL non sécurisées pouvant mener à une injection, etc.) afin de permettre leur détection par les outils. L’application s’appuie sur une base H2 configurée dans les fichiers de ressources pour exécuter facilement les scénarios en local. La structure globale respecte le standard Spring : code dans `src/main/java`, configuration dans `src/main/resources` et tests dans `src/test/java`, ce qui facilite l’automatisation du build et des tests dans un pipeline CI.

4 Étape 3 — Création du pipeline GitLab CI (.gitlab-ci.yml)

Création d’un pipeline minimal avec au moins deux stages (`build` et `test`) afin d’automatiser la compilation et l’exécution des tests.

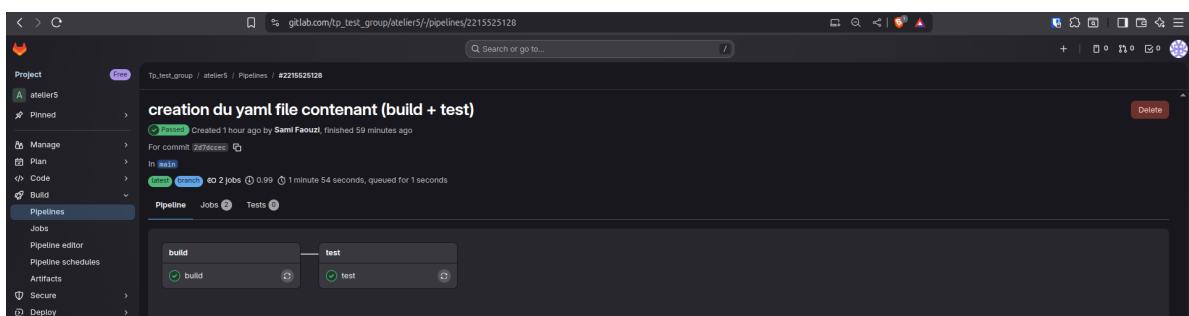


FIGURE 5 – Pipeline GitLab CI exécuté avec succès (statut réussi).

5 Étape 4 — Création et protection des branches

5.1 Crédit des branches principales

Mise en place de l'arborescence des branches (travail par branches, éviter les push directs sur `main`).

```
sami@FAOUIZI:~/Desktop/CI.CD/atelier5/dossier-tp$ git checkout -b feature/init
Switched to a new branch 'feature/init'
sami@FAOUIZI:~/Desktop/CI.CD/atelier5/dossier-tp$ git branch -a
  develop
* feature/init
  main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
  remotes/origin2/develop
  remotes/origin2/main
sami@FAOUIZI:~/Desktop/CI.CD/atelier5/dossier-tp$
```

FIGURE 6 – Branches créées via commandes Git.

5.2 Protection des branches sur GitLab

Activation de la protection des branches sur GitLab pour imposer les bonnes pratiques (MR obligatoire, éviter force-push, etc.).

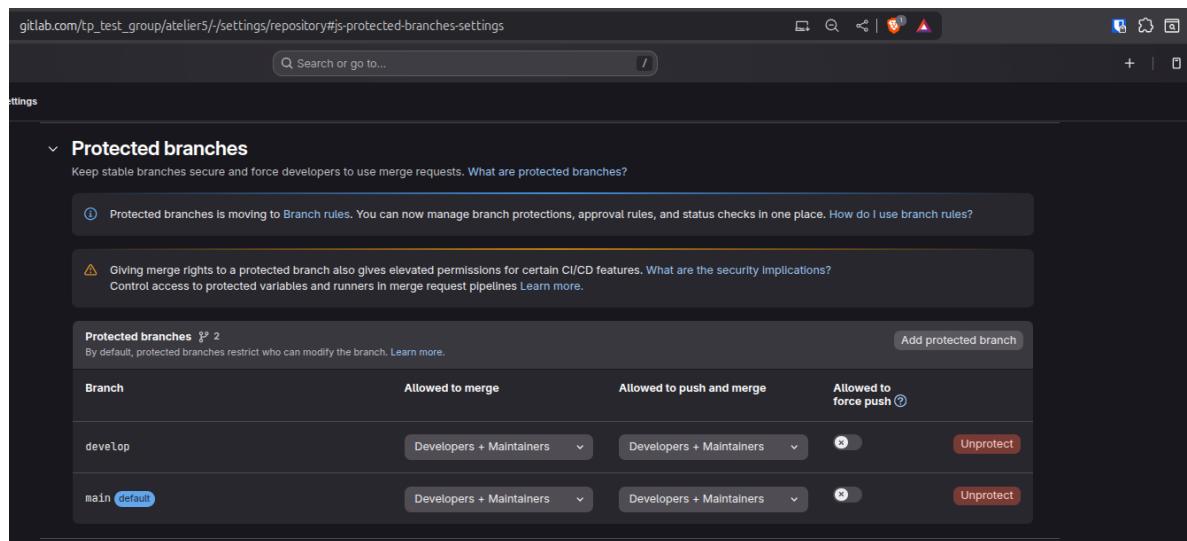


FIGURE 7 – Configuration des branches protégées dans GitLab.

6 Étape 5 — Détection des vulnérabilités (branche **security-analysis**)

6.1 Modification du pipeline pour intégrer la sécurité

Création d'une branche dédiée à l'analyse sécurité et modification du fichier `.gitlab-ci.yml` pour intégrer les outils de détection (SAST, Secret Detection, et analyse des dépendances).

```
sami@FAOUIZI:~/Desktop/CI.CD/atelier5/dossier-tp$ git checkout -b feature/security-analysis
Switched to a new branch 'feature/security-analysis'
sami@FAOUIZI:~/Desktop/CI.CD/atelier5/dossier-tp$ nano .gitlab-ci.yml
sami@FAOUIZI:~/Desktop/CI.CD/atelier5/dossier-tp$ nano .gitlab-ci.yml
sami@FAOUIZI:~/Desktop/CI.CD/atelier5/dossier-tp$ git add .gitlab-ci.yml
sami@FAOUIZI:~/Desktop/CI.CD/atelier5/dossier-tp$ git commit -m "Add security scans: SAST, Secret Detection, OWASP Dependency-Check"
[feature/security-analysis 8b7f594] Add security scans: SAST, Secret Detection, OWASP Dependency-Che ck
 1 file changed, 16 insertions(+)
sami@FAOUIZI:~/Desktop/CI.CD/atelier5/dossier-tp$ git push -u origin2 feature/security-analysis
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 500 bytes | 500.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote:
remote: To create a merge request for feature/security-analysis, visit:
remote: https://gitlab.com/tp_test_group/atelier5/-/merge_requests/new?merge_request%5Bsource_branch%5D=feature%2Fsecurity-analysis
remote:
To https://gitlab.com/tp_test_group/atelier5.git
 * [new branch]      feature/security-analysis -> feature/security-analysis
branch 'feature/security-analysis' set up to track 'origin2/feature/security-analysis'.
sami@FAOUIZI:~/Desktop/CI.CD/atelier5/dossier-tp$
```

FIGURE 8 – Modification du fichier YAML sur la branche **security-analysis**.

6.2 Merge Request de sécurité

Ouverture d'une Merge Request vers la branche **develop** afin d'exécuter le pipeline de sécurité et centraliser la validation.

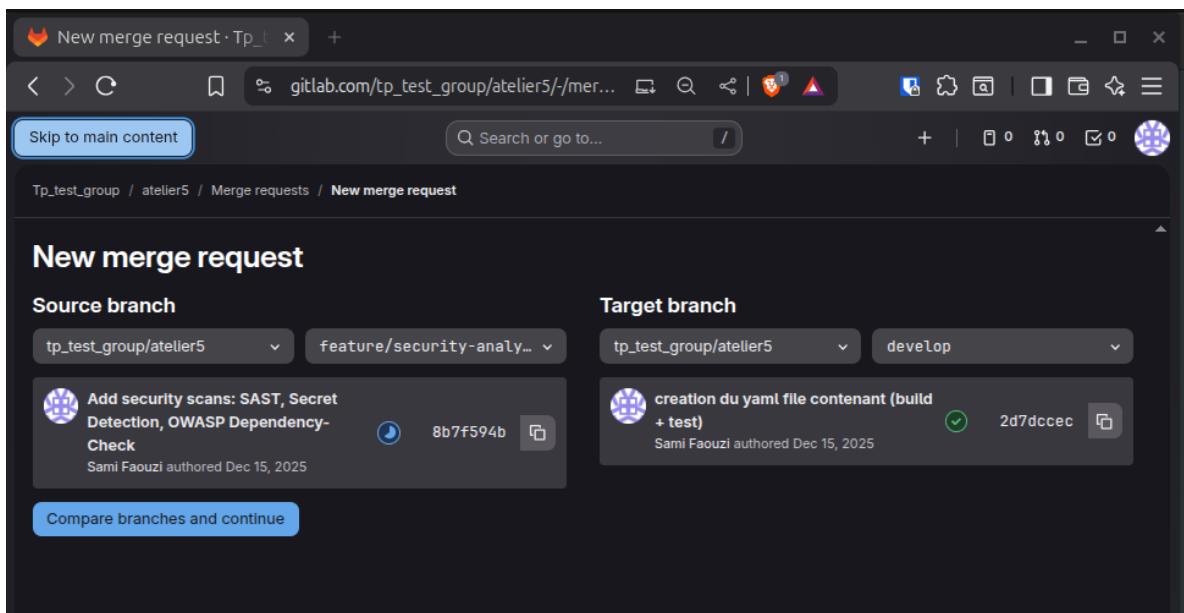


FIGURE 9 – Configuration / création de la Merge Request de sécurité.

6.3 Lancement et suivi du pipeline

Lancement du pipeline CI et suivi de l'exécution des jobs.

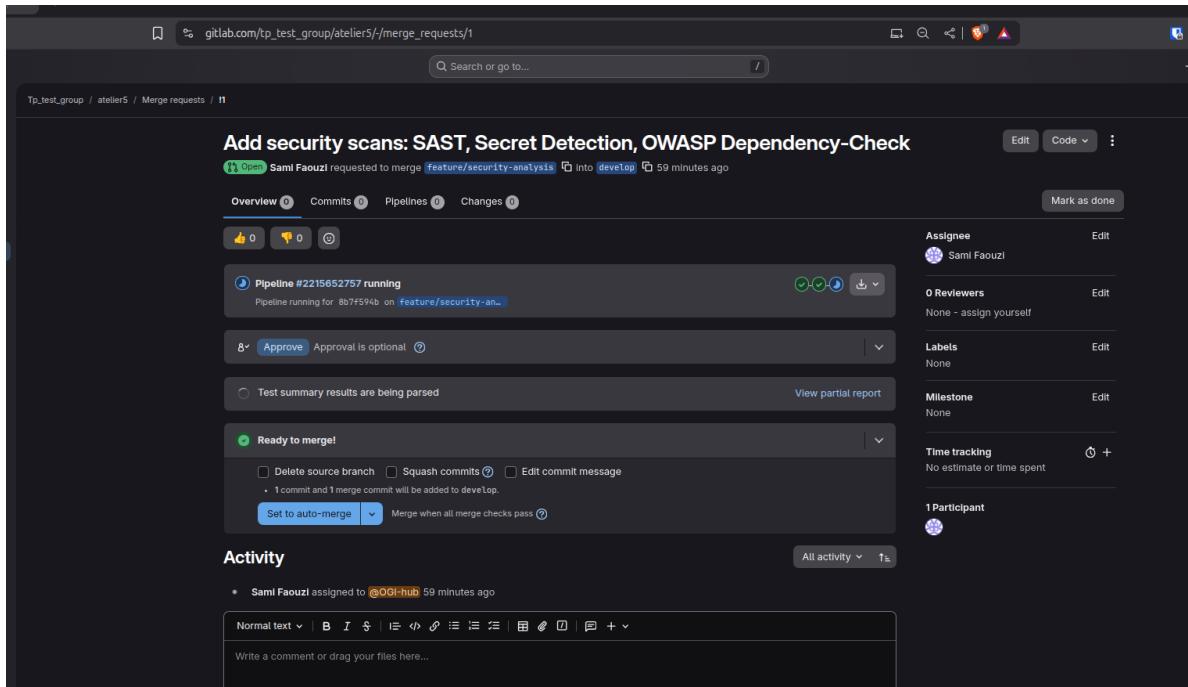


FIGURE 10 – Lancement du pipeline CI.

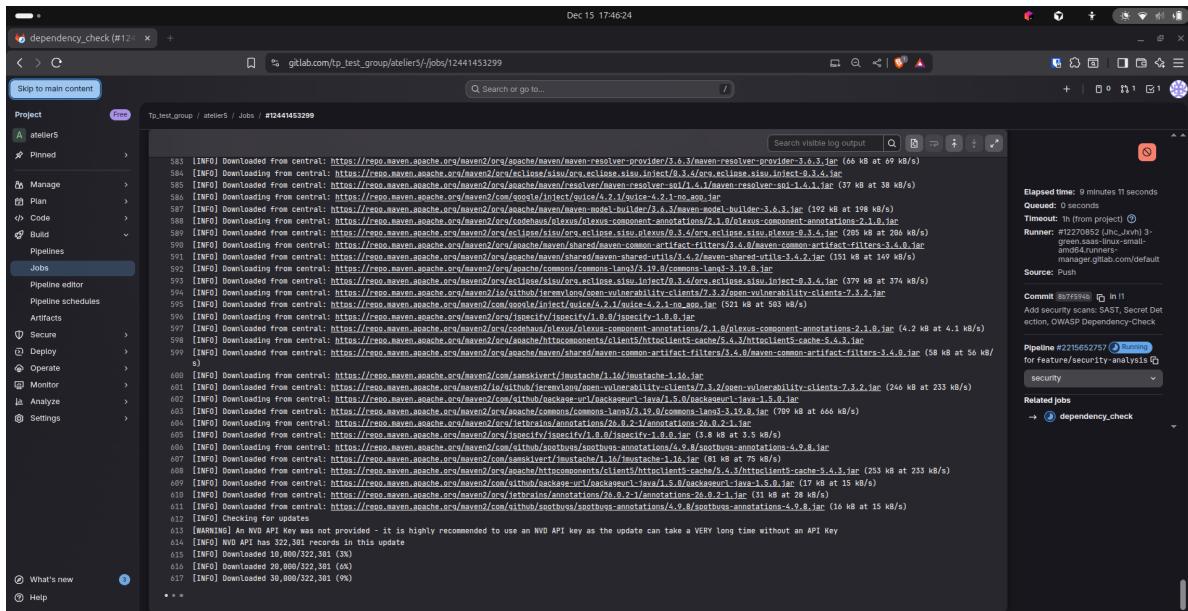


FIGURE 11 – Progression du pipeline (TIME OUT APRES UNE HEURE).

7 Étape 6 — Analyse des résultats (SAST + Secret Detection)

Après exécution des pipelines de sécurité, nous analysons les résultats fournis par GitLab afin d'identifier : (i) les vulnérabilités **SAST** et (ii) les **secrets codés en dur** détectés par *Secret Detection*.

7.1 Récupération des rapports (artifacts JSON)

Les résultats détaillés des scans sont disponibles via les *artifacts* des jobs de sécurité (téléchargement des rapports JSON depuis la Merge Request / pipeline).

The screenshot shows a GitLab pipeline interface. At the top, the URL is `gitlab.com/tp_test_group/atelier5/-/pipelines/2215826792/`. Below it, the pipeline path is `Tp_test_group / atelier5 / Pipelines / #2215826792`. The main title is **Dependency-Check removed from yaml file**. A green button indicates the pipeline is **Passed**, created 59 minutes ago by **Sami Faouzi** and finished 59 minutes ago. It was triggered for commit `e4a1ffee`. One related merge request is listed: `I1 Add security scans: SAST, Secret Detection, OWASP Dependency-Check`. The pipeline status shows **latest** and **branch** with **60 4 jobs** running, queued for 1 second. Below this, there are tabs for **Pipeline**, **Jobs 4**, and **Tests 0**. The pipeline diagram shows a flow from a **build** stage to a **test** stage. The **build** stage has one job, and the **test** stage has three jobs: **secret_detection**, **semgrep-sast**, and **test**. All jobs are marked with a green checkmark.

FIGURE 12 – Pipeline de sécurité exécuté (jobs SAST et Secret Detection).

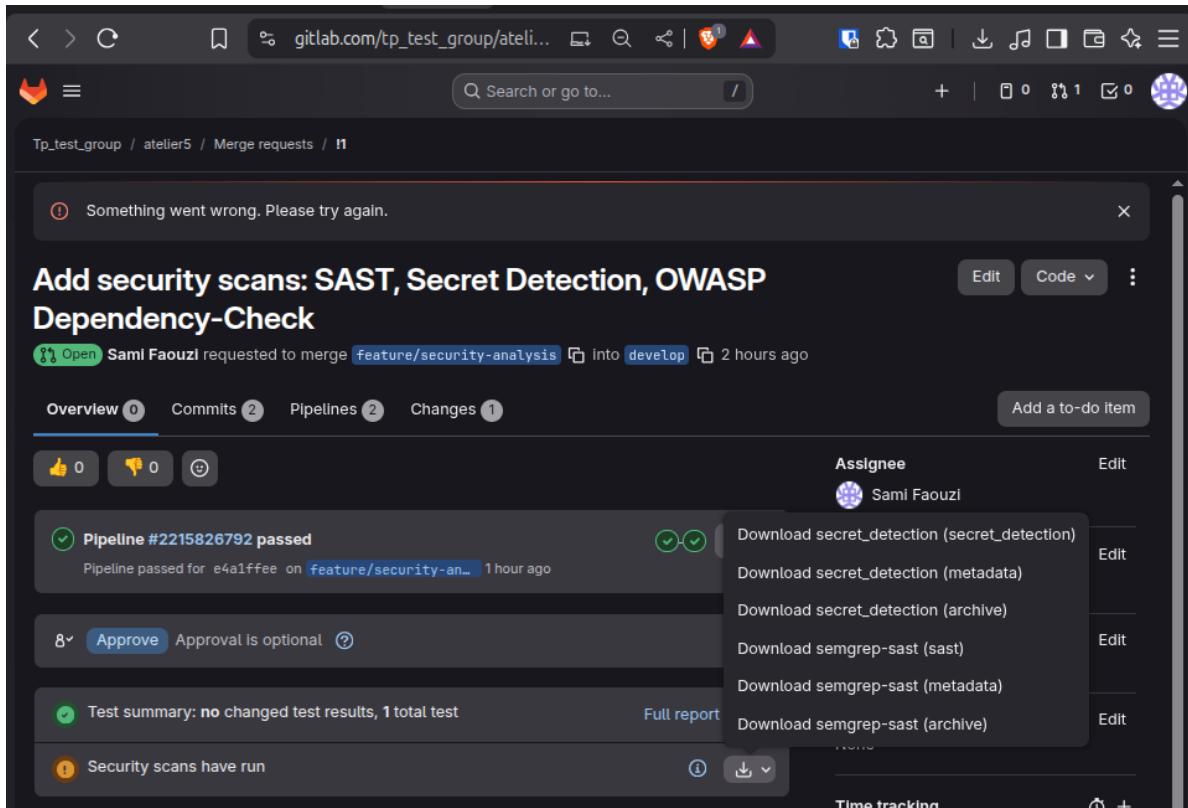


FIGURE 13 – Téléchargement / récupération des rapports JSON (SAST / Secret Detection).

7.2 Tableau de synthèse des findings (demandé par la prof)

Conformément à la consigne, nous remplissons uniquement les lignes **Finding SAST** et **Finding Secret Detection**.

```

{
  "version": "15.2.2",
  "vulnerabilities": [
    {
      "id": "d6d5964d22723cd3c17ef611a8a6810dd5e753bb9961622d67bf3f506d060a72",
      "category": "sast",
      "name": "Use of hard-coded password",
      "description": "A potential hard-coded password was identified in a database connection string.\nPasswords should not be stored directly in code\\nbut loaded from secure locations such as a Key Management System (KMS).\\n\\nThe purpose of using a Key Management System is so access can be audited and keys easily\\nrotated\\nin the event of a breach. By hardcoding passwords, it will be extremely difficult to determine\\nwhen or if, a key is compromised.\\n\\nThe recommendation on which KMS to use depends on the environment the application is running:\\n\\n- For Google Cloud Platform consider [Cloud Key Management](https://cloud.google.com/kms/docs)\\n- For Amazon Web Services consider [AWS Key Management](https://aws.amazon.com/kms/)\\n- For on-premise or other alternatives to cloud providers, consider [Hashicorp's\\nVault](https://www.vaultproject.io/)\\n- For other cloud providers, please see their documentation\\n",
      "cve": "semgrep_id:find_sec_bugs.DMI_CONSTANT_DB_PASSWORD-1.HARD_CODE_PASSWORD-3:23:23",
      "severity": "Critical",
      "scanner": {
        "id": "semgrep",
        "name": "Semgrep"
      },
      "location": {
        "file": "src/main/java/com/example/devsecops/VulnerableController.java",
        "start_line": 23
      },
      "identifiers": [
        {
          "type": "semgrep_id",
          "name": "find_sec_bugs.DMI_CONSTANT_DB_PASSWORD-1.HARD_CODE_PASSWORD-3",
          "value": "find_sec_bugs.DMI_CONSTANT_DB_PASSWORD-1.HARD_CODE_PASSWORD-3"
        }
      ]
    }
  ]
}

```

FIGURE 14 – Exemple de findings SAST localisés dans [VulnerableController.java](#).

```

{
  "version": "15.2.2",
  "vulnerabilities": [
    {
      "id": "717499c483c255a952b795f455644ea611421067aaa4a6ff62d74a5a9049cdcd",
      "category": "secret_detection",
      "name": "AWS Access Key ID",
      "description": "An AWS Access Key ID was detected. AWS Access Key IDs come in different types: long-term IAM user access keys\\n(starting with AKIA), temporary STS credentials (starting with ASIA), or AWS STS service bearer tokens. These credentials are paired with Secret Access Keys to authenticate programmatic requests to AWS services. A malicious actor with access to both the Access Key ID and"
    }
  ]
}

```

FIGURE 15 – Finding Secret Detection : clé AWS détectée dans [.env](#).

outil	Problème détecté	Fichier	Sévérité
SAST (Semgrep)	Injection SQL (CWE-89) : requête construite dynamiquement (risque d'injection)	src/main/java/com/example/devsecops/VulnerableController.java	High
Secret Detection (Gitleaks)	AWS Access Key ID exposée (secret hardcodé)	.env	Critical

TABLE 1 – Synthèse des findings demandés (SAST + Secret Detection).

7.3 Pourquoi ces findings sont des failles ?

- **Injection SQL (SAST)** : une concaténation de paramètres utilisateur dans une requête SQL peut permettre à un attaquant de modifier la requête (lecture/altération de données, contournement de contrôles), impactant confidentialité et intégrité.
- **Secret AWS (Secret Detection)** : exposer une clé (même “fake” dans le TP) simule une fuite de credential ; dans un cas réel, cela permettrait un accès non autorisé aux ressources cloud et peut entraîner compromission ou coûts non maîtrisés.

8 Étape 7 — Correction des vulnérabilités (1 vulnérabilité = 1 branche = 1 MR)

Cette étape applique la règle de travail : **chaque vulnérabilité est corrigée dans une branche dédiée**, via une Merge Request vers `develop`, avec une pipeline CI validée.

8.1 7.1 Correction SAST : SQL Injection

Objectif : supprimer la construction dynamique de requête SQL et la remplacer par une requête sécurisée (paramétrée).

Étapes réalisées

1. Créer une branche dédiée à la correction (depuis `develop`) : `fix/sast-sql-injection`.
2. Identifier le code vulnérable signalé par SAST dans `VulnerableController.java`.
3. Remplacer la concaténation SQL par une requête paramétrée (ex : PreparedStatement / requête paramétrée selon l'implémentation du projet).
4. Commit + push sur la branche de correction.
5. Ouvrir une MR `fix/sast-sql-injection -> develop`.
6. Vérifier que la pipeline passe et que le finding SAST (SQL Injection) disparaît.

```

{
  "version": "15.2.2",
  "vulnerabilities": [],
  "scan": {
    "analyzer": {
      "id": "secrets",
      "name": "secrets",
      "url": "https://gitlab.com/gitlab-org/security-products/analyzers/secrets"
    },
    "scanner": {
      "id": "gitleaks",
      "name": "Gitleaks",
      "url": "https://github.com/gitleaks/gitleaks"
    },
    "vendor": {
      "name": "GitLab",
      "version": "8.28.0"
    }
  },
  "scanner": {
    "id": "gitleaks",
    "name": "Gitleaks",
    "url": "https://github.com/gitleaks/gitleaks"
  },
  "type": "secret_detection",
  "start_time": "2025-12-15T22:58:47",
  "end_time": "2025-12-15T22:58:49",
  "status": "success"
}

```

FIGURE 16 – Après correction : le scan SAST ne remonte plus la vulnérabilité SQL Injection.

8.2 7.2 Correction Secret Detection : secret hardcodé

Objectif : supprimer le secret du dépôt et le remplacer par une variable d'environnement (côté GitLab CI/CD).

Étapes réalisées

1. Créer une branche dédiée : `fix/secret-vulnerability` (ou nom équivalent).
2. Supprimer le secret du dépôt :
 - retirer le fichier `.env` du tracking Git (ne plus versionner),
 - ajouter `.env` dans `.gitignore`.
3. Remplacer l'utilisation du secret dans le code/config par une **variable d'environnement**.
4. Dans GitLab : **Settings > CI/CD > Variables** :
 - créer la variable (ex : `AWS_ACCESS_KEY_ID`),
 - cocher **Masked** (et **Protected** si demandé).

5. Commit + push sur la branche.
6. Ouvrir une MR [fix/secret-vulnerability -> develop](#).
7. Vérifier pipeline OK + disparition de l'alerte Secret Detection.

```

version: "15.2.2"
vulnerabilities: []
scan:
  analyzer:
    id: "secrets"
    name: "secrets"
    url: "https://gitlab.com/gitlab-org/security-products/analyzers/secrets"
  vendor:
    name: "GitLab"
    version: "7.21.0"
  scanner:
    id: "gitleaks"
    name: "Gitleaks"
    url: "https://github.com/gitleaks/gitleaks"
  vendor:
    name: "GitLab"
    version: "8.28.0"
    type: "secret_detection"
    start_time: "2025-12-15T20:36:17"
    end_time: "2025-12-15T20:36:18"
    status: "success"
  
```

FIGURE 17 – Après correction : la fuite de secret est supprimée et le statut est en succès.

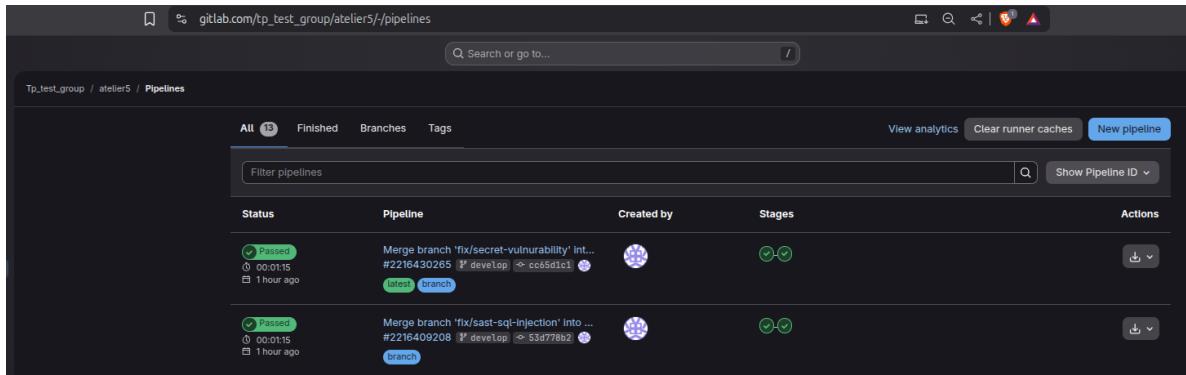


FIGURE 18 – Pipelines OK après merge des branches de correction vers develop.

9 Étape 8 — Fusion finale : develop → main (Rebase & Merge)

Oui : il ne s'agit pas seulement du *rebase*. Le slide demande un **comparatif Rebase vs Merge** et une **fusion finale de develop vers main**. On teste d'abord l'intégration via **Rebase**, puis on réalise aussi une fusion via **Merge** (avec capture + commandes + pipeline verte).

9.1 Comparatif rapide : Merge vs Rebase

Critère	Merge	Rebase
Historique	Conserve l'historique (commit de merge)	Historique linéaire (réécriture des commits)
Risque	Moins risqué (pas de réécriture)	Peut nécessiter force-push (attention collaboration)
Lisibilité	Historique réel mais plus "bruité"	Historique propre, linéaire

TABLE 2 – Comparatif synthétique Merge vs Rebase.

9.2 Rebase final (test d'intégration) : develop → main

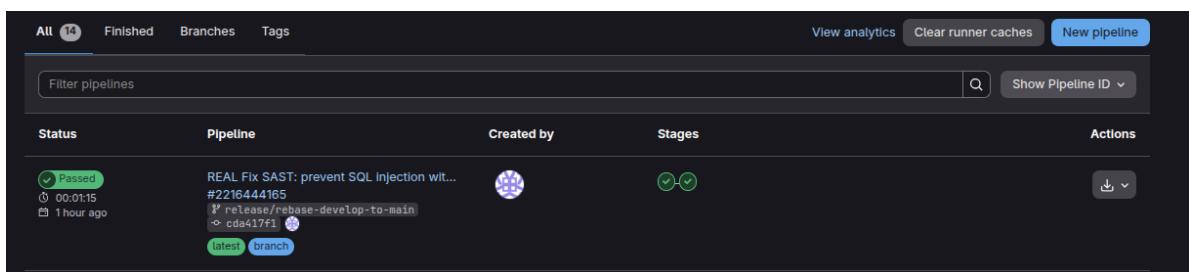


FIGURE 19 – Capture du rebase final `develop -> main`.

```
sami@FAOUIZI:~/Desktop/CI.CD/atelier5/dossier-tp$ git checkout develop
git pull
git checkout -b release/rebase-develop-to-main
git rebase origin2/main
git push -u origin2 release/rebase-develop-to-main
Switched to branch 'develop'
Your branch is up to date with 'origin2/develop'.
remote: Enumerating objects: 2, done.
remote: Counting objects: 100% (2/2), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
```

FIGURE 20 – Commandes Git utilisées pour le rebase final (succès).

9.3 Merge final : develop → main

Après validation par rebase (test), on effectue également la fusion finale via **merge** pour répondre explicitement à la consigne “Merge/Rebase develop → main”.

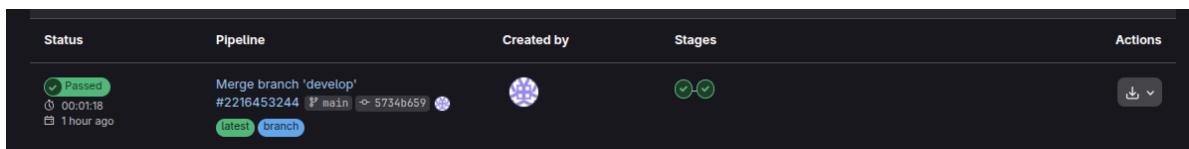


FIGURE 21 – Capture du merge final vers `main`.

```
sami@FAOUZZI:~/Desktop/CI.CD/atelier5/dossier-tp$ git checkout main
git pull
git merge --no-ff develop      # merge final vers main
git push origin2 main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)
warning: redirecting to https://gitlab.com/groupe-tp1/dossier-tp.git/
Already up to date.
Merge made by the 'ort' strategy.
.gitlab-ci.yml                  |  8 ++++++++
src/main/java/com/example/devsecops/VulnerableController.java | 15 +++++++-----+
2 files changed, 16 insertions(+), 7 deletions(-)
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 220 bytes | 220.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
To https://gitlab.com/tp_test_group/atelier5.git
  2d7dcce..5734b65  main -> main
sami@FAOUZZI:~/Desktop/CI.CD/atelier5/dossier-tp$
```

FIGURE 22 – Commandes Git utilisées pour le merge final (succès).

10 Étape 9 — Synthèse

Ce TP a permis de mettre en place un workflow DevSecOps complet basé sur GitLab :

- **Workflow Git propre** : travail par branches, Merge Requests vers `develop`, puis intégration finale vers `main`.
- **CI automatisée** : stages `build` et `test` validant la qualité fonctionnelle avant intégration.
- **Détection de vulnérabilités** : exécution des scans **SAST (Semgrep)** et **Secret Detection (Gitleaks)**.
- **Traitemennt des failles** selon la règle $1 \text{ vulnérabilité} = 1 \text{ branche} = 1 \text{ MR}$:
 - correction de l'**Injection SQL** (SAST) via une branche dédiée et vérification de disparition du finding,
 - correction du **secret AWS** (Secret Detection) en supprimant le secret du dépôt et en utilisant une variable GitLab CI/CD.
- **Release final** : comparaison **Rebase vs Merge** et intégration de `develop` dans `main` avec pipeline verte.