GitLab Flow
Pipelines avancés

Plan

Plan

Nécessité d'adopter un workflow

Pour l'instant nos pipelines ne prennent pas en compte les possibilités offertes par Git, à savoir le mécanisme de branches.

L'idée est d'organiser le flux du développement en respectant un processus standardisé pour la création des features, releases, fixes, etc. et d'organiser le CI/CD en conséquence.

Principes généraux d'un workflow basé sur Git

- Permettre un travail collectif efficace
- Suivre les modifications apportées au code source simplement
- Introduire des points de relecture avant validation (merge requests)
- Distinguer pré-production vs. production
- Automatiser le déploiement et la création de releases, en fonction des branches

GitFlow

Basé sur 2 branches principales : *master* et *develop*

- les branches de fonctionnalités sont créées à partir *develop*
- puis fusionnées dans *develop*, une fois testées et validées
- *develop* est fusionnée dans *master* lorsqu'on souhaite créer une release
- les versions, commits tagués, sont créées sur *master*

GitHub Flow

Basé sur une seule branche principale (e.g. *master*)

- les branches de fonctionnalités sont créées à partir de *master*
- puis fusionnées dans *master*, une fois testées et validées
- les versions, commits tagués, sont créées sur *master*

GitLab Flow

Une variante de GitFlow qui ne fixe pas le nom et le nombre de branches pour la pré/production : pas de branche *develop* ▸ GitLab Flow

Il en existe deux variantes :

Avec des branches d'environnements : *environment branches*

- Deux branches principales : *main* et *production*
- les branches de fonctionnalités sont créées à partir de *main*
- puis testées/déployées dans autant de branches d'environnements (pré-production) que nécessaires : tests, acceptation, ...
- puis fusionnées dans la branche *main*, puis dans la branche *production*

Avec des branches *releases*

- Idem mais avec une branche principale et des branches de version (long terme)
- $\Rightarrow$ convient lorsqu'on produit du logiciel téléchargeable

GitLab Flow / Principes de mises en œuvre

Bonnes pratiques

▸ GitLab Flow best practices

1. Toujours utiliser des branches annexes pour développer
2. Effectuer tous les tests pour chaque commit : un push déclenche un pipeline
3. Effectuer une code review pour chaque merge request
4. Le déploiement est automatisé en fonction des branches, e.g. *production*
5. Les tags sont créés par les développeurs, pas par le CI
6. Les releases sont basées sur les tags : une release par tag
7. Éviter le rebase lors de push, pour un suivi clair de l'historique
8. Les développeurs partent de *main* et ciblent *main* pour chaque nouvelle feature
9. Fixer les bugs dans *main*, puis dans les branches de releases (cherry-pick)
10. Les messages de commit doivent dire explicitement pourquoi il a été réalisé, pas seulement comment ▸ How to Write a Git Commit Message

Plan

Environments

GitLab permet de définir des *environnements* ▸ Environments
Ils permettent de spécifier le contexte dans lequel un job (déploiement/tests) s'exécute et de suivre facilement le résultat des opérations

Par exemple pour le déploiement, on peut créer plusieurs environnements :

- *staging* : pré-production
- *production* : version active
- créés de façon dynamique et unique pour chaque pipeline, par ex. pour tester une feature

Création d'un environnement statique

Prérequis depuis l'UI de GitLab

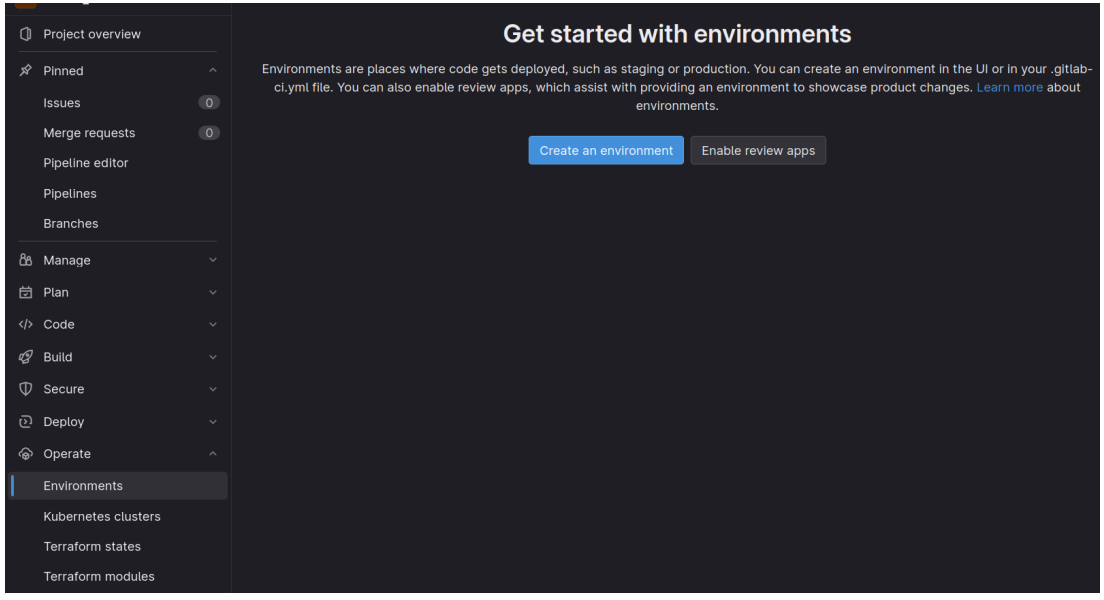
In the UI

Prerequisites:

- You must have at least the Developer role.

To create a static environment in the UI:

1. On the left sidebar, at the top, select **Search GitLab** (**{search}**) to find your project.
2. Select **Operate > Environments**.
3. Select **Create an environment**.
4. Complete the fields.
5. Select **Save**.



Conventions pour le nommage des environnements

Deployment tier of environments

> Introduced in GitLab 13.10.

Sometimes, instead of using an industry standard environment name, like `production`, you might want to use a code name, like `customer-portal`. While there is no technical reason not to use a name like `customer-portal`, the name no longer indicates that the environment is used for production.

To indicate that a specific environment is for a specific use, you can use tiers:

Environment tier	Environment name examples
`production`	Production, Live
`staging`	Staging, Model, Demo
`testing`	Test, QC
`development`	Dev, Review apps, Trunk
`other`	

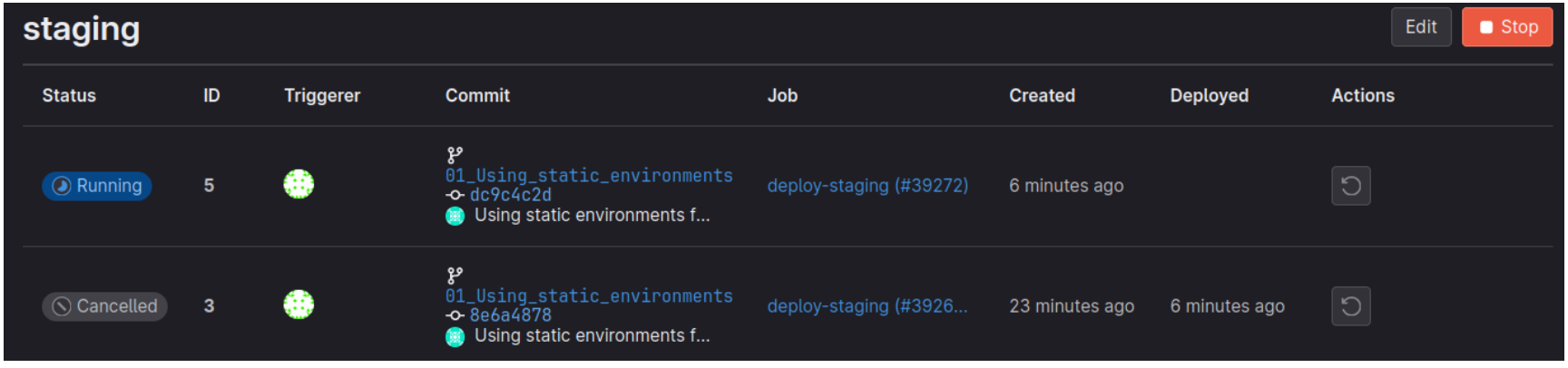
By default, GitLab assumes a tier based on the environment name. Instead, you can use the `deployment_tier` keyword to specify a tier.

Utilisation des environnements

File: .gitlab-ci.yml

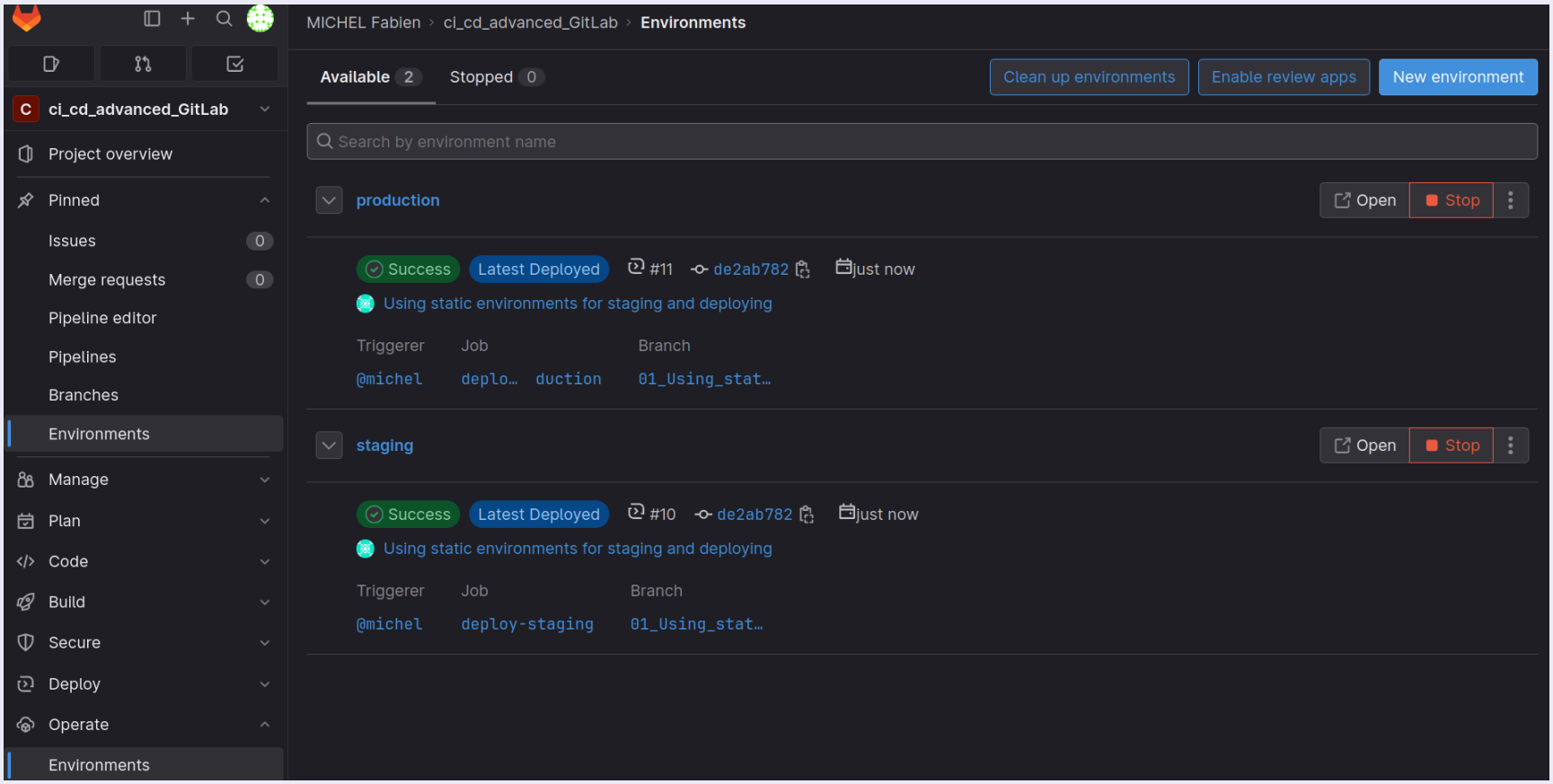
```yaml
default:
  image: eclipse-temurin:11-jdk-alpine

variables:
  VERSION: "2.0"
  PACKAGE_NAME: "hello-$VERSION.tar.gz"
  STAGING_URL: "cicd/staging"
  PRODUCTION_URL: "cicd/production"
  FTP_BASE_DIR: "public_html"

build:
  stage: build
  script:
    - javac hello/Hello.java
  artifacts:
    paths:
      - hello/Hello.class

test:
  stage: test
  script: java hello.Hello  | grep -q 'Hello world'

deploy-staging:
  image: finalgene/openssh
  stage: deploy
  variables:
    TARGET_DIR: "$FTP_BASE_DIR/$STAGING_URL"
  script:
    - echo $PACKAGE_NAME
    - tar cvzf $PACKAGE_NAME --directory=. hello/*class
    - echo "<a href=\"$PACKAGE_NAME\">Last version -> $PACKAGE_NAME</a>" > index.html
    - apk add lftp #install the lftp package on the alpine distro
    - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; mkdir -p $TARGET_DIR; cd $TARGET_DIR; put $PACKAGE_NAME index.html; bye;" sftp://ftpinfo.iutmontp.univ-montp2.fr
  environment:
    name: staging
    url: https://webinfo.iutmontp.univ-montp2.fr/~michel/$STAGING_URL

deploy-production:
  image: finalgene/openssh
  stage: deploy
  variables:
    TARGET_DIR: "$FTP_BASE_DIR/$PRODUCTION_URL"
  script:
    - echo $PACKAGE_NAME
    - tar cvzf $PACKAGE_NAME --directory=. hello/*class
    - echo "<a href=\"$PACKAGE_NAME\">Last version -> $PACKAGE_NAME</a>" > index.html
    - apk add lftp #install the lftp package on the alpine distro
    - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; mkdir -p $TARGET_DIR; cd $TARGET_DIR; put $PACKAGE_NAME index.html; bye;" sftp://ftpinfo.iutmontp.univ-montp2.fr
  environment:
    name: production
```

Visualisation des activités par environnement



Résultats du déploiement

Open permet d'aller directement sur l'URL



Exemple sur ▸ https://webinfo.iutmontp.univ-montp2.fr/ michel/cicd/staging/

Plan

Mot-clé *rules*

Il permet de spécifier quand les jobs doivent être exécutés ▸ rules

Certains jobs ne doivent pas être exécutés tout le temps

- *deploy-production* doit être exécuté uniquement pour un tag sur la branche *production*
- *deploy-staging* uniquement pour *main*

Specify when jobs run with `rules`

> Introduced in GitLab 12.3.

Use `rules` to include or exclude jobs in pipelines.

Rules are evaluated in order until the first match. When a match is found, the job is either included or excluded from the pipeline, depending on the configuration. See the `rules` reference for more details.

Future keyword improvements are being discussed in our epic for improving `rules`, where anyone can add suggestions or requests.

Exemples d'utilisation 1

`rules` examples

The following example uses `if` to define that the job runs in only two specific cases:

```yaml
job:
  script: echo "Hello, Rules!"
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      when: manual
      allow_failure: true
    - if: $CI_PIPELINE_SOURCE == "schedule"
```

- If the pipeline is for a merge request, the first rule matches, and the job is added to the merge request pipeline with attributes of:
  - `when: manual` (manual job)
  - `allow_failure: true` (the pipeline continues running even if the manual job is not run)
- If the pipeline is **not** for a merge request, the first rule doesn't match, and the second rule is evaluated.
- If the pipeline is a scheduled pipeline, the second rule matches, and the job is added to the scheduled pipeline. No attributes were defined, so it is added with:
  - `when: on_success` (default)
  - `allow_failure: false` (default)
- In **all other cases**, no rules match, so the job is **not** added to any other pipeline.

Exemples d'utilisation 2

Alternatively, you can define a set of rules to exclude jobs in a few cases, but run them in all other cases:

```yaml
job:
  script: echo "Hello, Rules!"
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      when: never
    - if: $CI_PIPELINE_SOURCE == "schedule"
      when: never
    - when: on_success
```

- If the pipeline is for a merge request, the job is **not** added to the pipeline.
- If the pipeline is a scheduled pipeline, the job is **not** added to the pipeline.
- In **all other cases**, the job is added to the pipeline, with `when: on_success`.

⚠ If you use a `when` clause as the final rule (not including `when: never`), two simultaneous pipelines may start. Both push pipelines and merge request pipelines can be triggered by the same event (a push to the source branch for an open merge request). See how to prevent duplicate pipelines for more details.

Exemples d'utilisation 3

Skip job if the branch is empty

Use `rules:changes:compare_to` to avoid running a job when the branch is empty, which saves CI/CD resources. Compare the branch to the default branch, and if the branch:

- Doesn't have changed files, the job doesn't run.
- Has changed files, the job runs.

For example, in a project with `main` as the default branch:

```
job:
  script:
    - echo "This job only runs for branches that are not empty"
  rules:
    - if: $CI_COMMIT_BRANCH
      changes:
        compare_to: 'refs/heads/main'
        paths:
          - '*'
```

The rule for this job compares all files and paths (`*`) in the current branch against the default branch `main`. The rule matches and the job runs only when there are changes to the files in the branch.

Utilisation classiques de variables prédéfinies

Other commonly used variables for `if` clauses:

- `if: $CI_COMMIT_TAG`: If changes are pushed for a tag.
- `if: $CI_COMMIT_BRANCH`: If changes are pushed to any branch.
- `if: $CI_COMMIT_BRANCH == "main"`: If changes are pushed to `main`.
- `if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH`: If changes are pushed to the default branch. Use when you want to have the same configuration in multiple projects with different default branches.
- `if: $CI_COMMIT_BRANCH =~ /regex-expression/`: If the commit branch matches a regular expression.
- `if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH && $CI_COMMIT_TITLE =~ /Merge branch.*/`: If the commit branch is the default branch and the commit message title matches a regular expression. For example, the default commit message for a merge commit starts with `Merge branch`.
- `if: $CUSTOM_VARIABLE !~ /regex-expression/`: If the custom variable `CUSTOM_VARIABLE` does **not** match a regular expression.
- `if: $CUSTOM_VARIABLE == "value1"`: If the custom variable `CUSTOM_VARIABLE` is exactly `value1`.

Utilisation classiques de variables prédéfinies

The following table lists some of the variables that you can use, and the pipeline types the variables can control for:

- Branch pipelines that run for Git `push` events to a branch, like new commits or tags.
- Tag pipelines that run only when a new Git tag is pushed to a branch.
- Merge request pipelines that run for changes to a merge request, like new commits or selecting the **Run pipeline** button in a merge request's pipelines tab.
- Scheduled pipelines.

Variables	Branch	Tag	Merge request	Scheduled
`CI_COMMIT_BRANCH`	Yes			Yes
`CI_COMMIT_TAG`		Yes		Yes, if the scheduled pipeline is configured to run on a tag.
`CI_PIPELINE_SOURCE = push`	Yes	Yes		
`CI_PIPELINE_SOURCE = schedule`				Yes
`CI_PIPELINE_SOURCE = merge_request_event`			Yes	
`CI_MERGE_REQUEST_IID`			Yes	

Réutilisation de règles

Il est possible de réutiliser des règles afin de factoriser le code et d'améliorer sa lisibilité

Reuse rules in different jobs

> Introduced in GitLab 14.3.

Use `!reference` tags to reuse rules in different jobs. You can combine `!reference` rules with regular job-defined rules:

```
.default_rules:
  rules:
    - if: $CI_PIPELINE_SOURCE == "schedule"
      when: never
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH

job1:
  rules:
    - !reference [.default_rules, rules]
  script:
    - echo "This job runs for the default branch, but not schedules."

job2:
  rules:
    - !reference [.default_rules, rules]
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
  script:
    - echo "This job runs for the default branch, but not schedules."
    - echo "It also runs for merge requests."
```

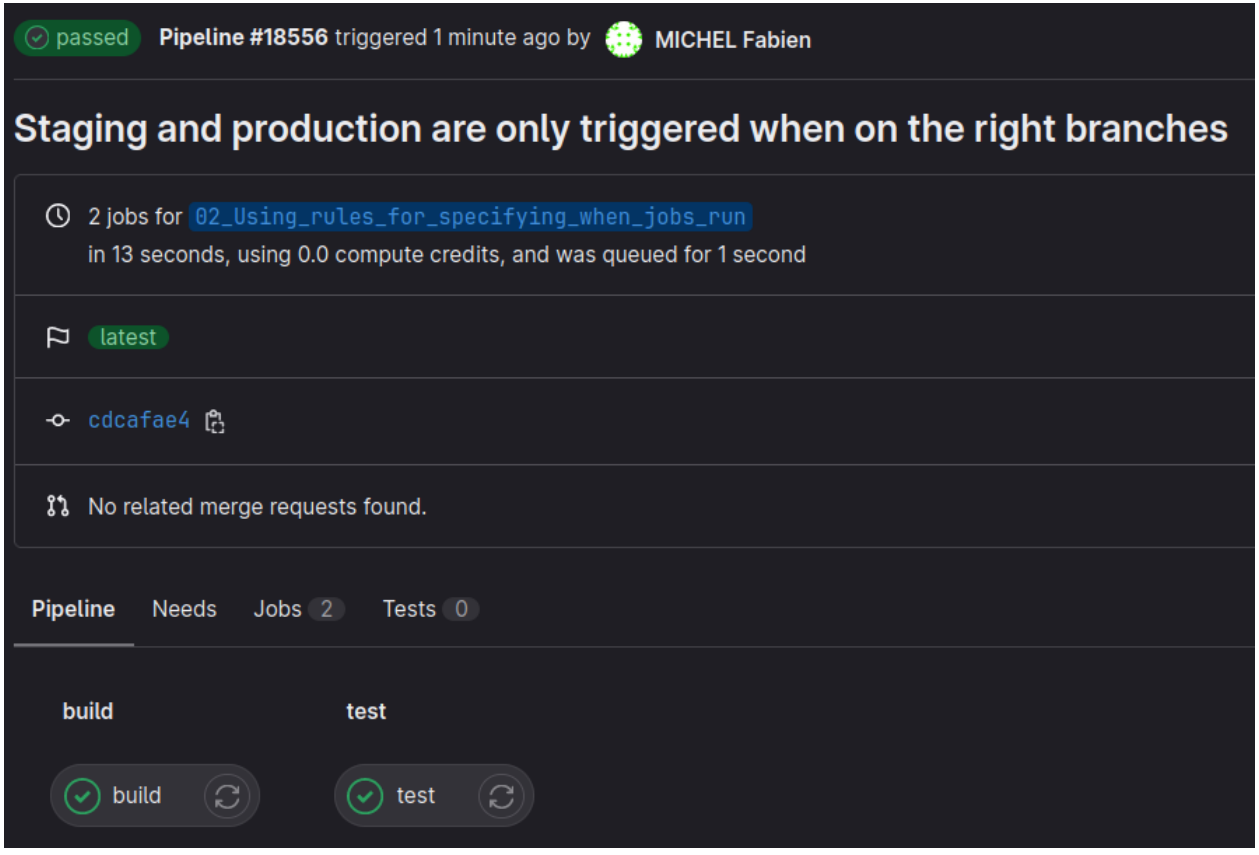
Application sur notre pipeline

On modifie le pipeline pour s'aligner sur un GitLab Flow (*environment*)

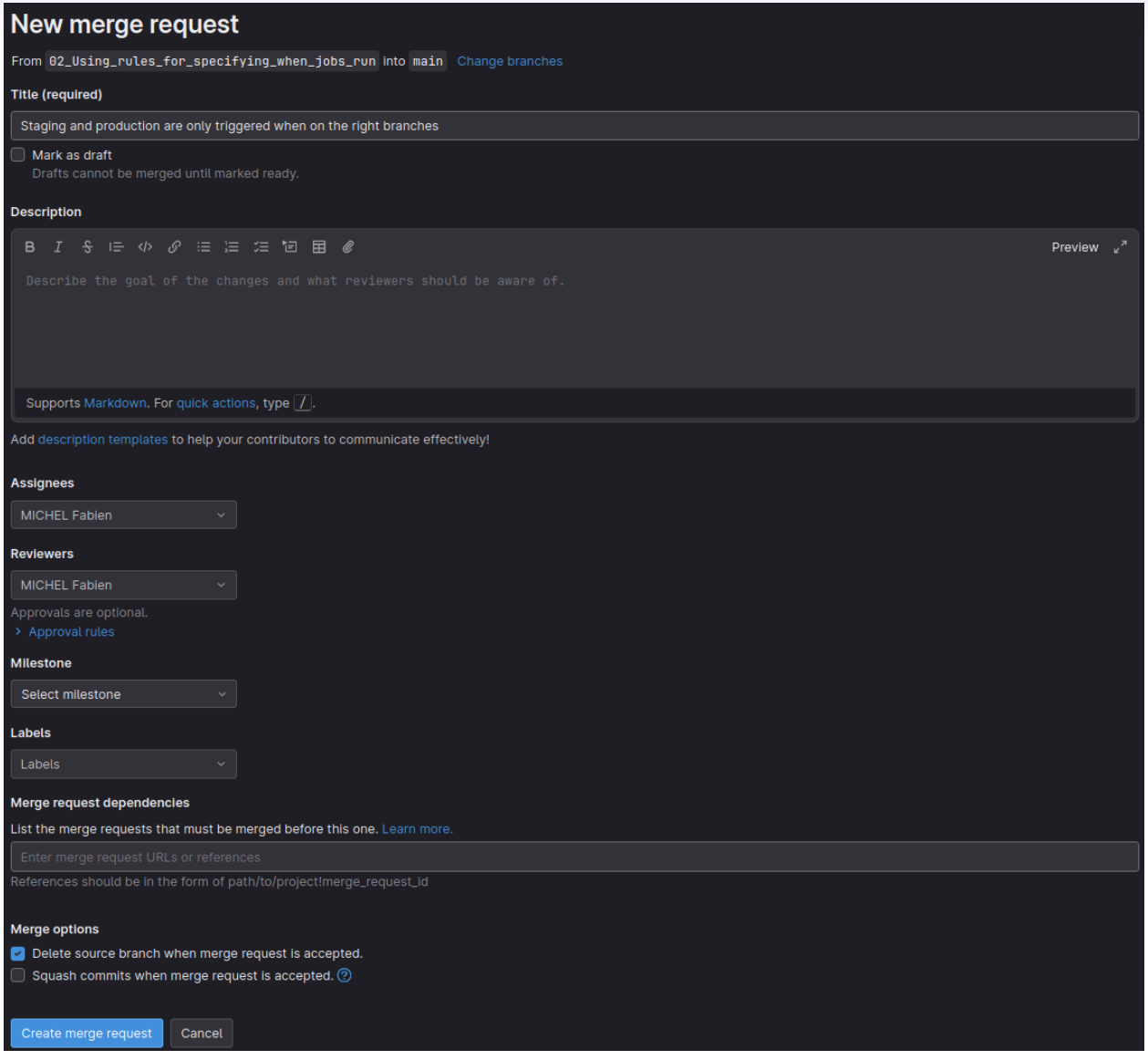
```
  deploy-staging:
    image: finalgene/openssh
    stage: deploy
    variables:
      TARGET_DIR: "$FTP_BASE_DIR/$STAGING_URL"
    script:
      - echo $PACKAGE_NAME
      - tar cvzf $PACKAGE_NAME --directory=. hello/*class
      - echo "<a href=\"$PACKAGE_NAME\">Last version -> $PACKAGE_NAME</a>" > index.html
      - apk add lftp #install the lftp package on the alpine distro
      - echo $TARGET_DIR
      - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; mkdir -p $TARGET_DIR; cd $TARGET_DIR;
put $PACKAGE_NAME index.html; bye;" sftp://ftpinfo.iutmontp.univ-montp2.fr
    environment:
      name: staging
      url: https://webinfo.iutmontp.univ-montp2.fr/~michel/$STAGING_URL/
    rules:
      - if: $CI_COMMIT_BRANCH == "main"

  deploy-production:
    image: finalgene/openssh
    stage: deploy
    variables:
      TARGET_DIR: "$FTP_BASE_DIR/$PRODUCTION_URL"
    script:
      - echo $TARGET_DIR
      - echo $PACKAGE_NAME
      - tar cvzf $PACKAGE_NAME --directory=. hello/*class
      - echo "<a href=\"$PACKAGE_NAME\">Last version -> $PACKAGE_NAME</a>" > index.html
      - apk add lftp #install the lftp package on the alpine distro
      - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; mkdir -p $TARGET_DIR; cd $TARGET_DIR;
put $PACKAGE_NAME index.html; bye;" sftp://ftpinfo.iutmontp.univ-montp2.fr
    environment:
      name: production
    rules:
```

Les jobs sont maintenant filtrés en fonction du contexte

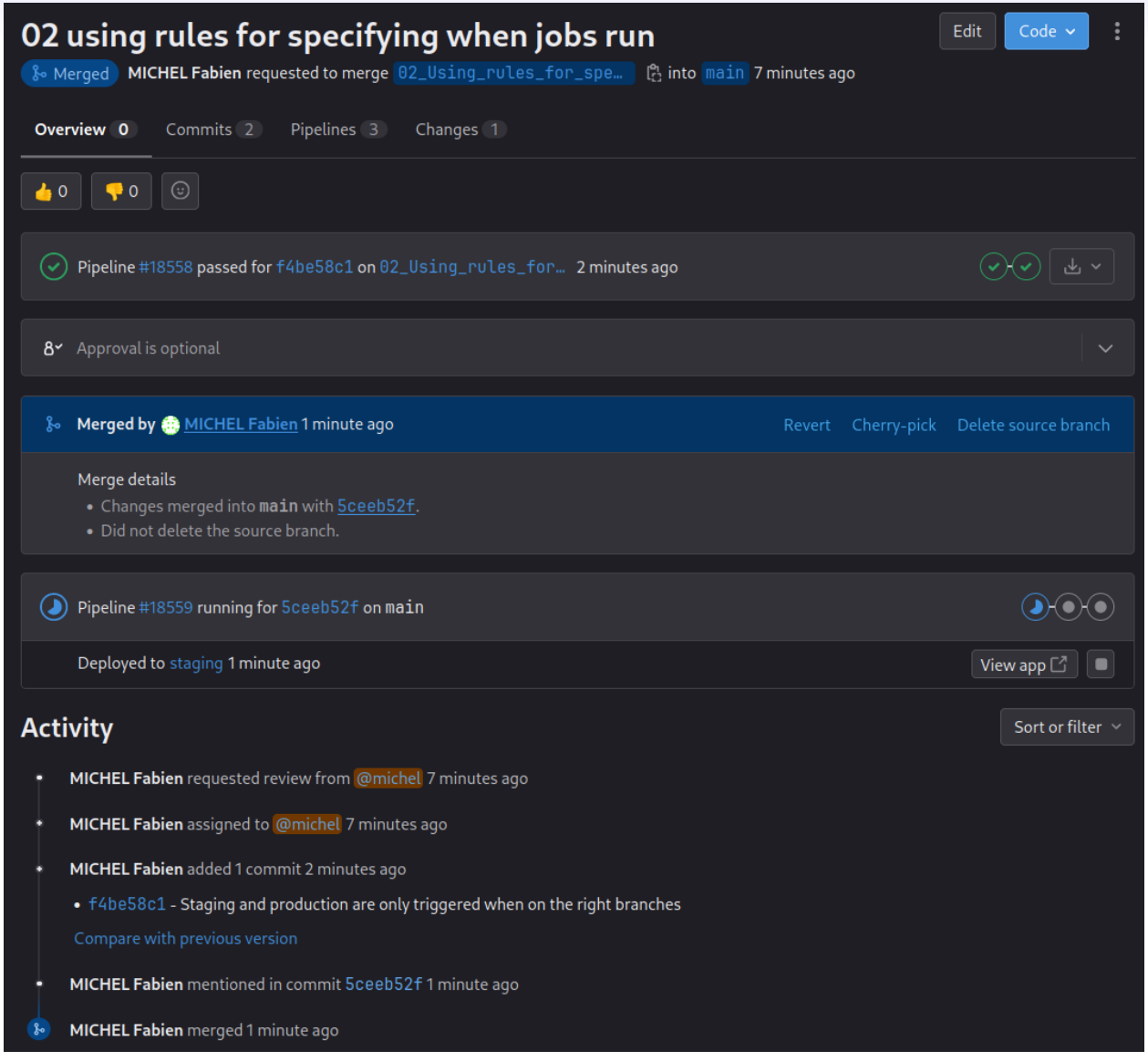
Les commits sur la branche *feature* ne déclenchent plus les déploiements :



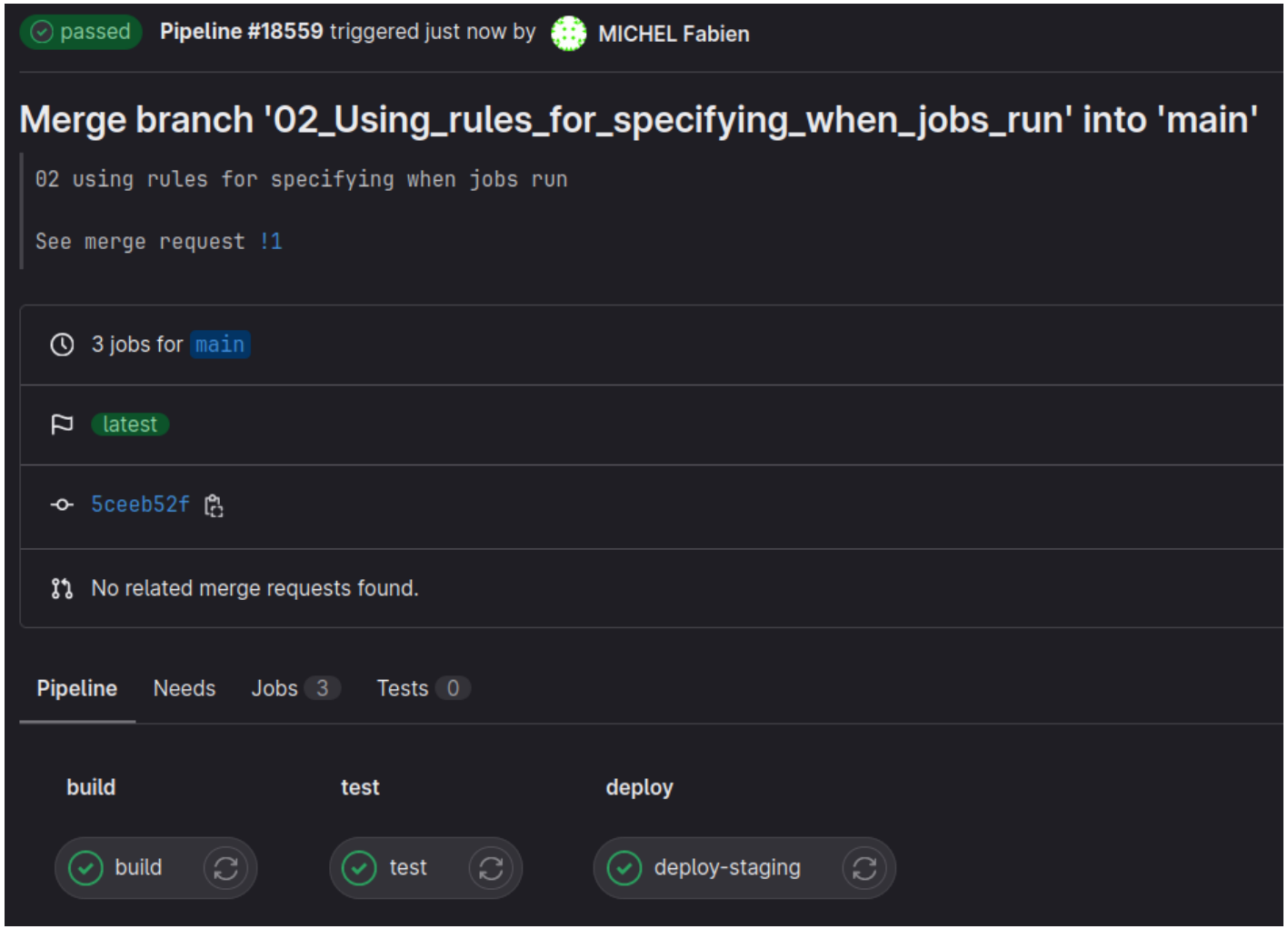
Création d'une merge request sur main



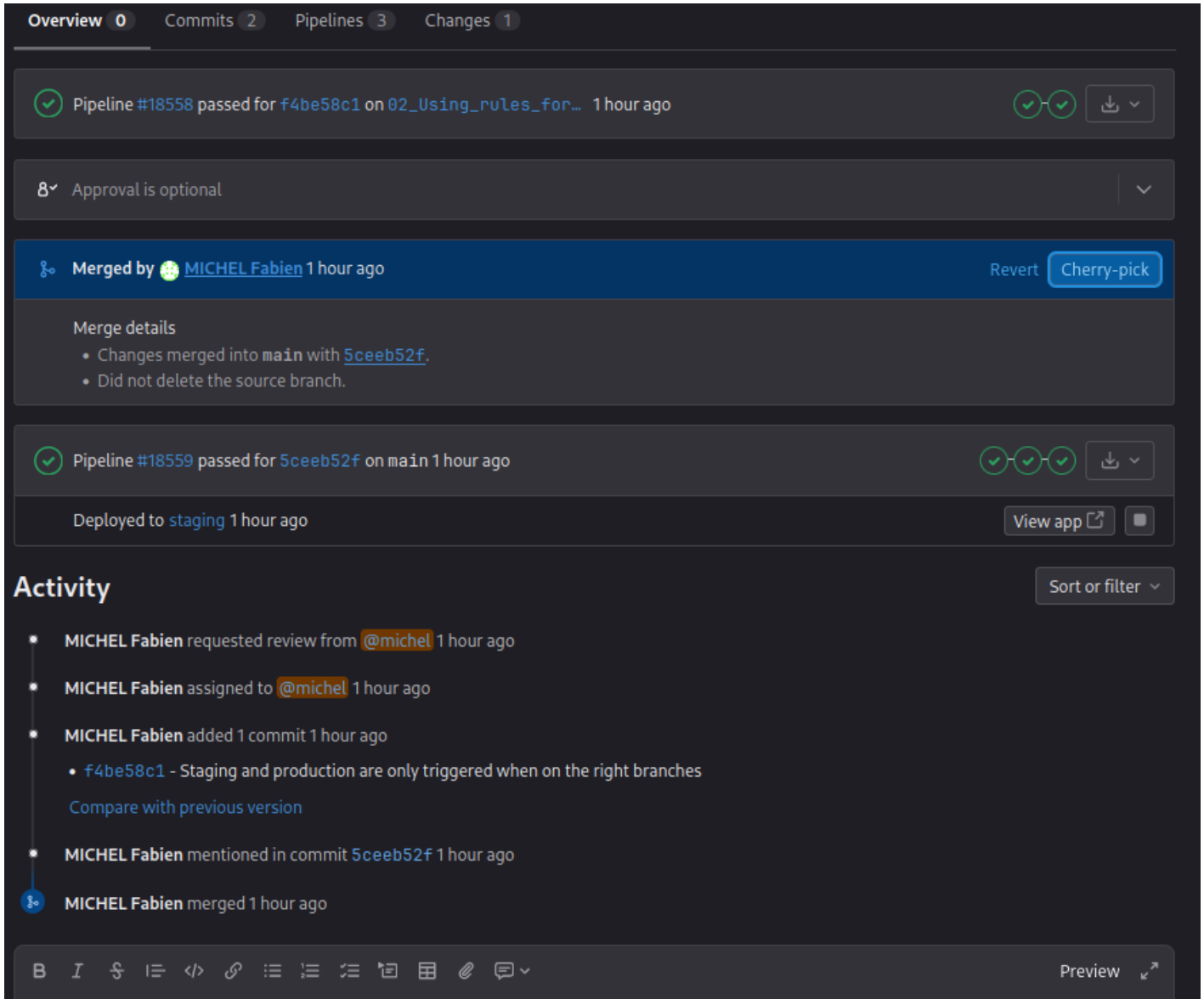
Déclenchement du déploiement sur staging



Exécution du pipeline sur la branche *main*

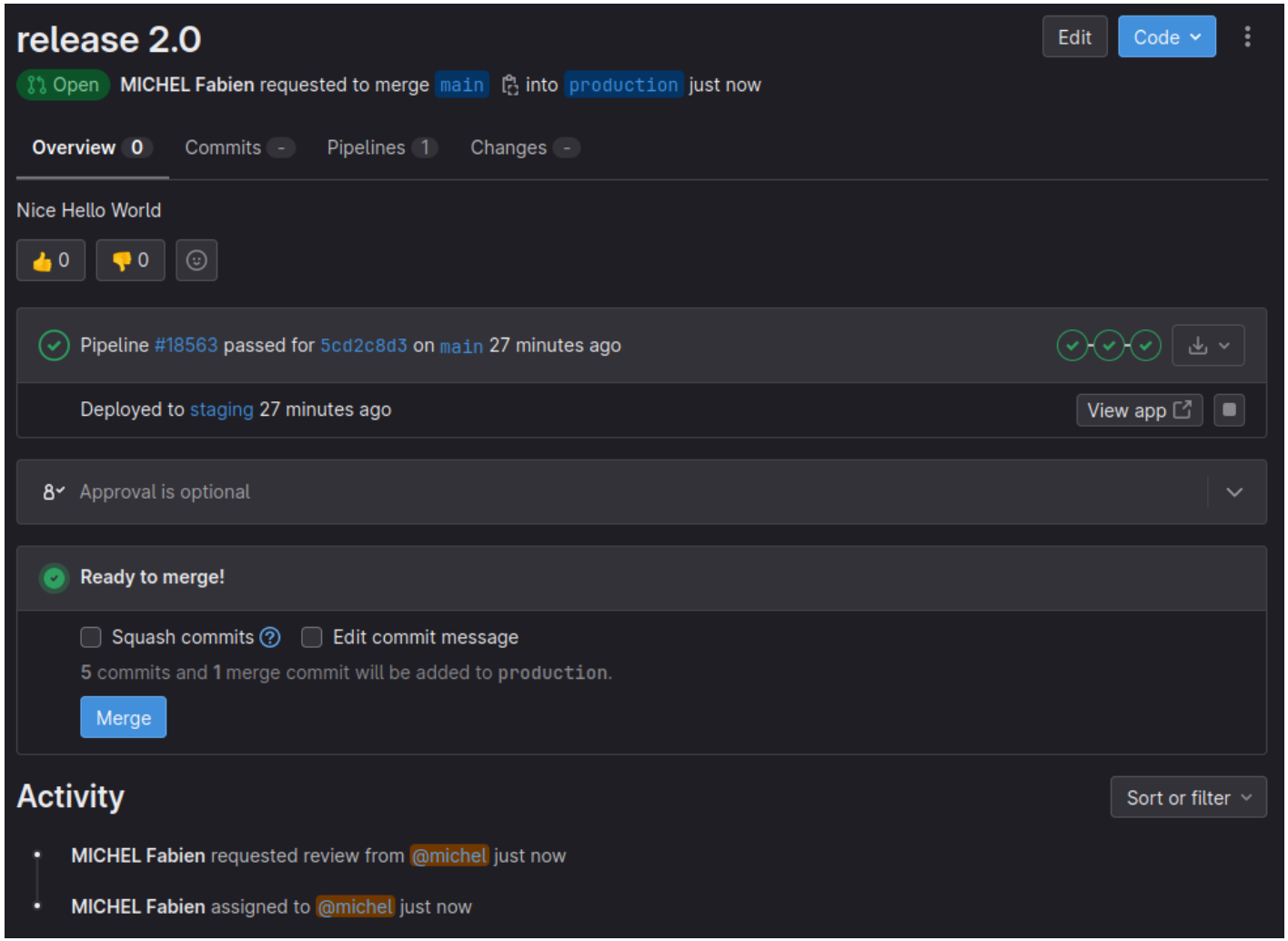


Fin du merge de la merge request ⇒ pré-prod OK



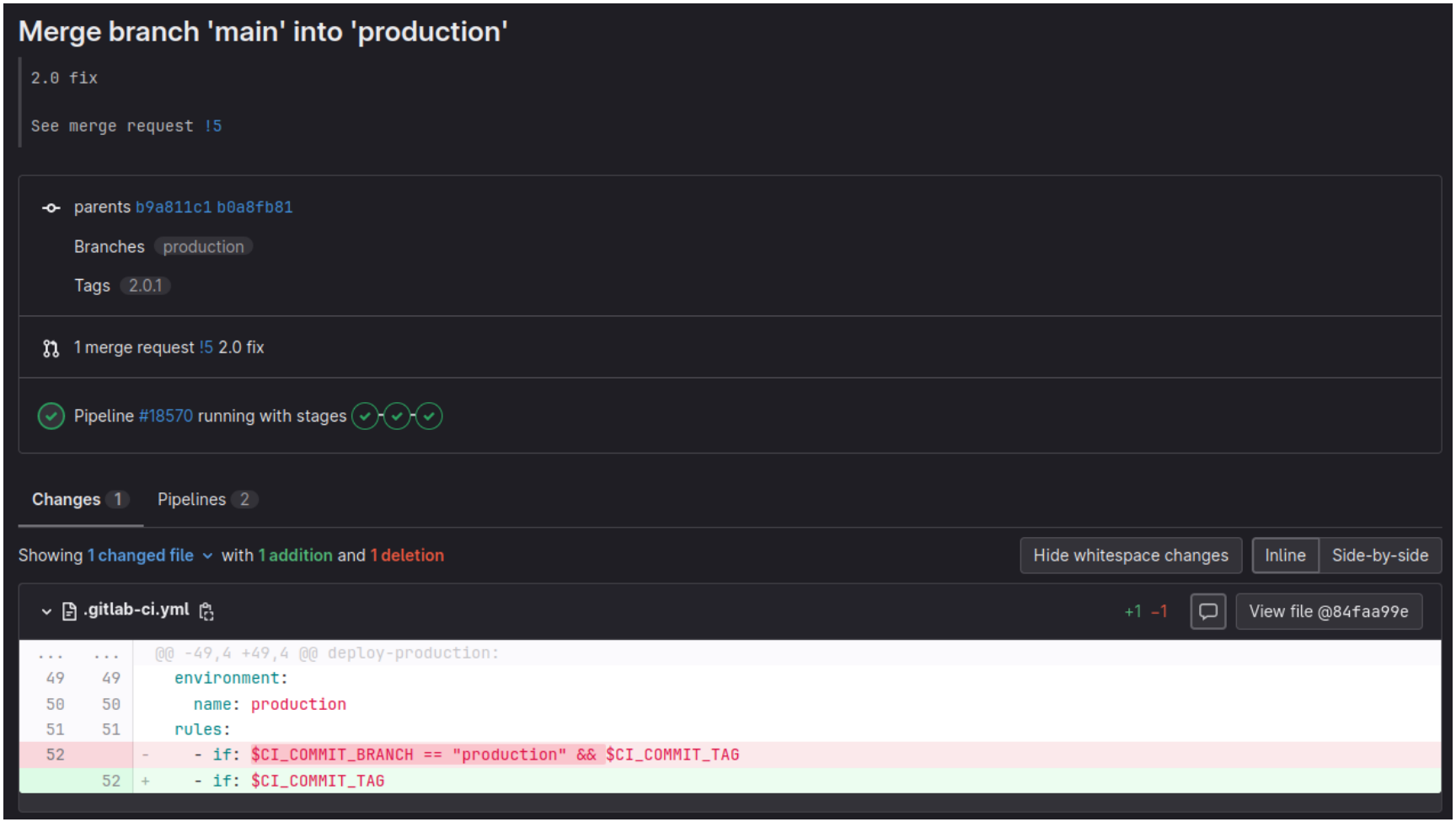
Mise en production

Il faut maintenant faire une merge request de *main* vers *production*



Mais on a une petite erreur dans le job…

Le bloc *rules* ne pouvait pas être déclenché par la création d'un tag
Une fois corrigée dans une branche feature, on recommence les MR, puis la création d'un tag sur *production* déclenche bien le pipeline



Déclencher un job manuellement

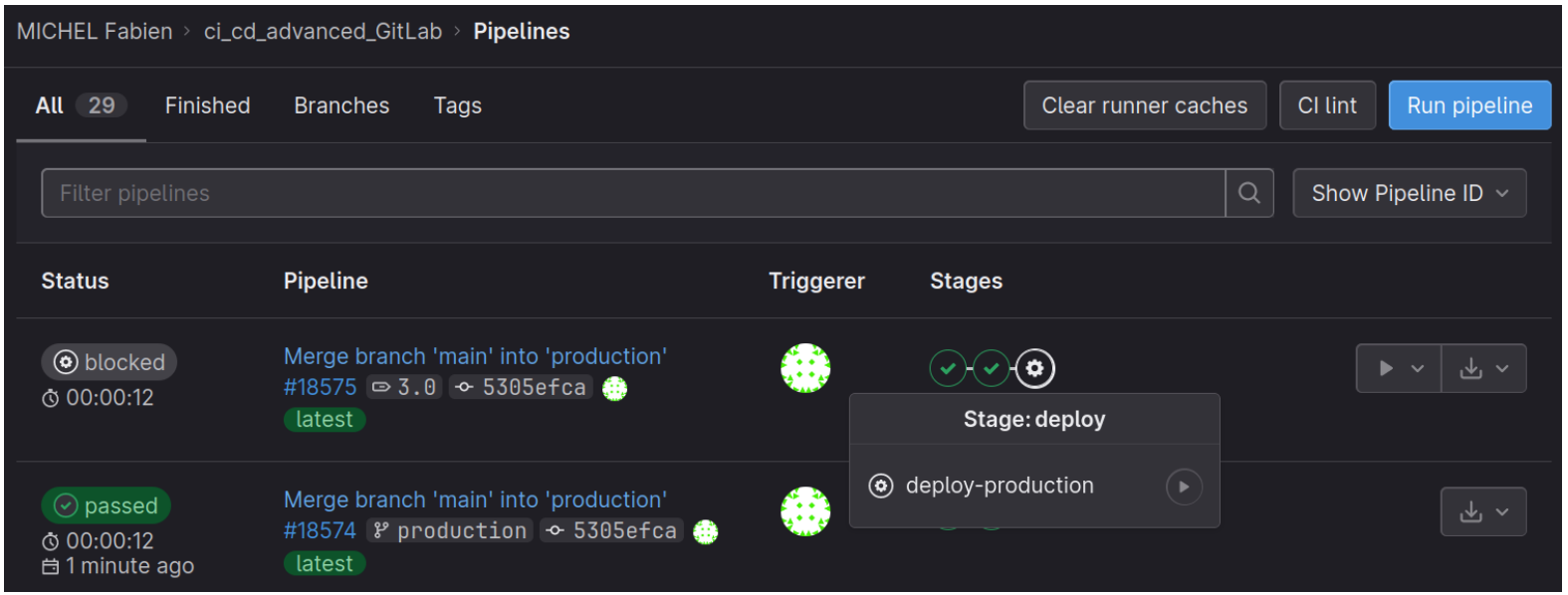
Il est possible de définir qu'un job est uniquement déclenché manuellement, depuis l'UI de GitLab avec le mot-clé *when* dans le bloc *rules*

Par exemple pour déclencher le déploiement en production au moment le plus propice, et pas automatiquement en fin de pipeline (les jobs précédents peuvent être longs)

```
deploy-production:
  image: finalgene/openssh
  stage: deploy
  variables:
    TARGET_DIR: "$FTP_BASE_DIR/$PRODUCTION_URL"
  script:
    - tar cvzf $PACKAGE_NAME --directory=. hello/*class
    - echo "<a href=\"$PACKAGE_NAME\">Last version -> $PACKAGE_NAME</a>" > index.html
    - apk add lftp #install the lftp package on the alpine distro
    - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; mkdir -p $TARGET_DIR; cd $TARGET_DIR;
put $PACKAGE_NAME index.html; bye;" sftp://ftpinfo.iutmontp.univ-montp2.fr
  environment:
    name: production
  rules:
    - if: $CI_COMMIT_TAG
      when: manual
```

Déclencher un job manuellement

Le déploiement n'est plus automatique :



Plan

Environnements dynamiques

Il est possible de créer des environnements éphémères dynamiquement, e.g. pour tester le résultat d'une feature avant un merge.

On a un squelette proposé dans la section environment de GitLab, avec le bouton **Enable Review Apps** :

Enable Review Apps

Review apps are dynamic environments that you can use to provide a live preview of changes made in a feature branch.

To configure a dynamic review app, you must:

1. Have access to infrastructure that can host and deploy the review apps.
2. Install and configure a runner to do the deployment.
3. Add a job in your CI/CD configuration that:
   - Only runs for feature branches or merge requests.
   - Uses a predefined CI/CD variable like `$(CI_COMMIT_REF_SLUG)` to dynamically create the review app environments. For example, for a configuration using merge request pipelines:

```
deploy_review:
  stage: deploy
  script:
    - echo "Add script here that deploys the code to your infrastructure"
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://$CI_ENVIRONMENT_SLUG.example.com
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
```

4. Recommended: Set up a job that manually stops the Review Apps.

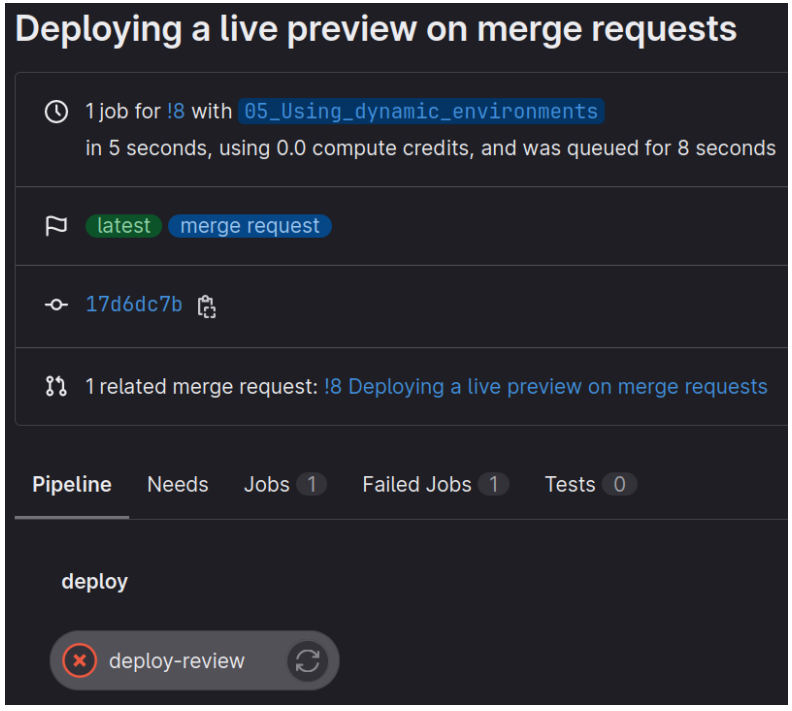
Dans notre pipeline

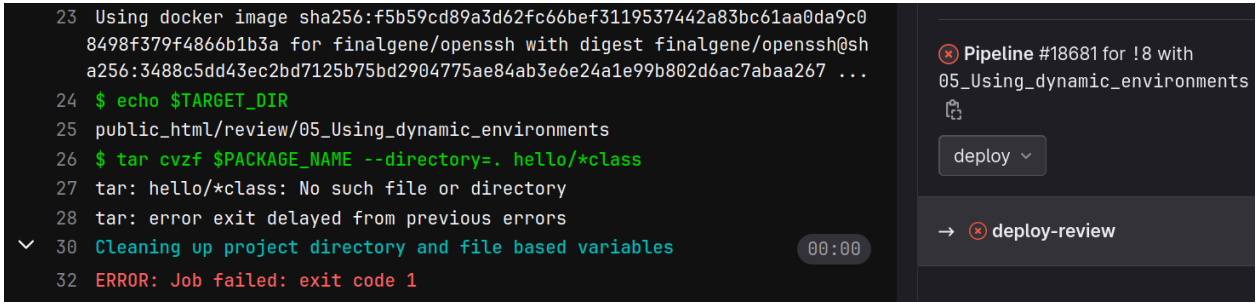
On ajoute un job `deploy-review` qui sera déclenché par une merge request :

```yaml
build:
  stage: build
  script:
    - javac hello/Hello.java
  artifacts:
    paths:
      - hello/Hello.class

test:
  stage: test
  script: java hello.Hello  | grep -q 'Hello world'

deploy-review:
  image: finalgene/openssh
  stage: deploy
  variables:
    TARGET_DIR: "$FTP_BASE_DIR/review/$CI_COMMIT_REF_NAME"
  script:
    - echo $TARGET_DIR
    - tar cvzf $PACKAGE_NAME --directory=. hello/*class
    - echo "<a href=\"$PACKAGE_NAME\">Testing version -> $PACKAGE_NAME</a>" > index.html
    - apk add lftp #install the lftp package on the alpine distro
    - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; mkdir -p $TARGET_DIR; cd $TARGET_DIR; put $PACKAGE_NAME index.html; by
e;" sftp://ftpinfo.iutmontp.univ-montp2.fr
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://webinfo.iutmontp.univ-montp2.fr/~michel/review/$CI_COMMIT_REF_NAME
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
```

Création de la merge request

Elle déclenche bien le job, mais il reste quelque chose à faire. . .





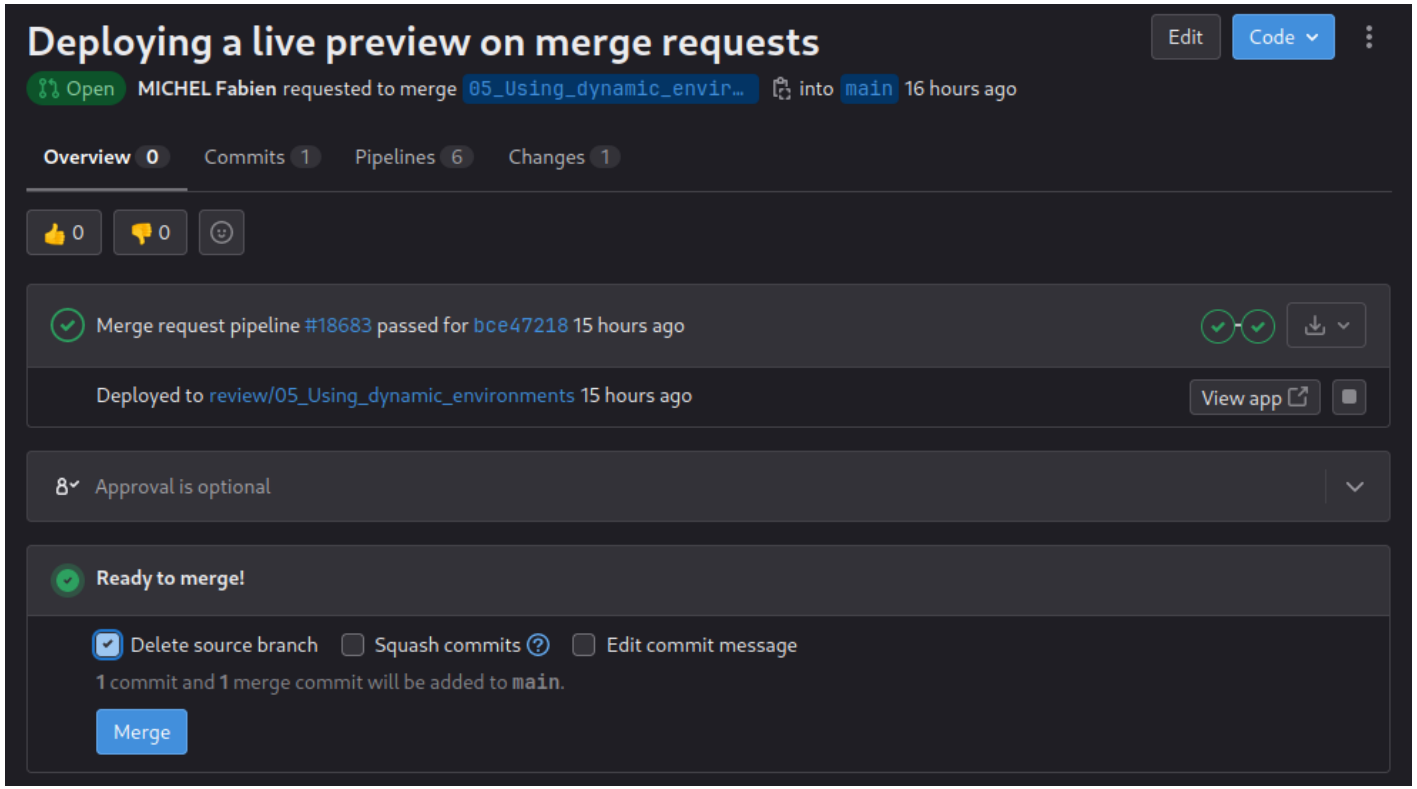
Ajout d'une règle au job `build`

Pour que notre job `deploy-review` fonctionne, il a besoin de l'artefact produit par le job `build` : il faut qu'il soit lancé !
⇒ On utilise pour cela la règle `when: always`

```yaml
build:
  stage: build
  script:
    - javac hello/Hello.java
  artifacts:
    paths:
      - hello/Hello.class
  rules:
  - when: always

test:
  stage: test
  script: java hello.Hello  | grep -q 'Hello world'

deploy-review:
  image: finalgene/openssh
  stage: deploy
  variables:
    TARGET_DIR: "$FTP_BASE_DIR/review/$CI_COMMIT_REF_NAME"
  script:
    - echo $TARGET_DIR
    - tar cvzf $PACKAGE_NAME --directory=. hello/*class
    - echo "<a href=\"$PACKAGE_NAME\">Testing version -> $PACKAGE_NAME</a>" > index.html
    - apk add lftp #install the lftp package on the alpine distro
    - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; mkdir -p $TARGET_DIR;
e;" sftp://ftpinfo.iutmontp.univ-montp2.fr
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://webinfo.iutmontp.univ-montp2.fr/~michel/review/$CI_COMMIT_REF_NAME
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
```

Nouveau résultat pour la merge request

Les deux jobs ont bien été lancés



Stopper et nettoyer un environnement dynamique

Une fois la MR validée ou fermée, l'environnement peut être stoppé, puis effacé. ▸ Stop an environment

In the following example, a `deploy_review` job calls a `stop_review` job to clean up and stop the environment.

- Both jobs must have the same `rules` or `only/except` configuration. Otherwise, the `stop_review` job might not be included in all pipelines that include the `deploy_review` job, and you cannot trigger `action: stop` to stop the environment automatically.
- The job with `action: stop` might not run if it's in a later stage than the job that started the environment.
- If you can't use merge request pipelines, set the `GIT_STRATEGY` to `none` in the `stop_review` job. Then the runner doesn't try to check out the code after the branch is deleted.

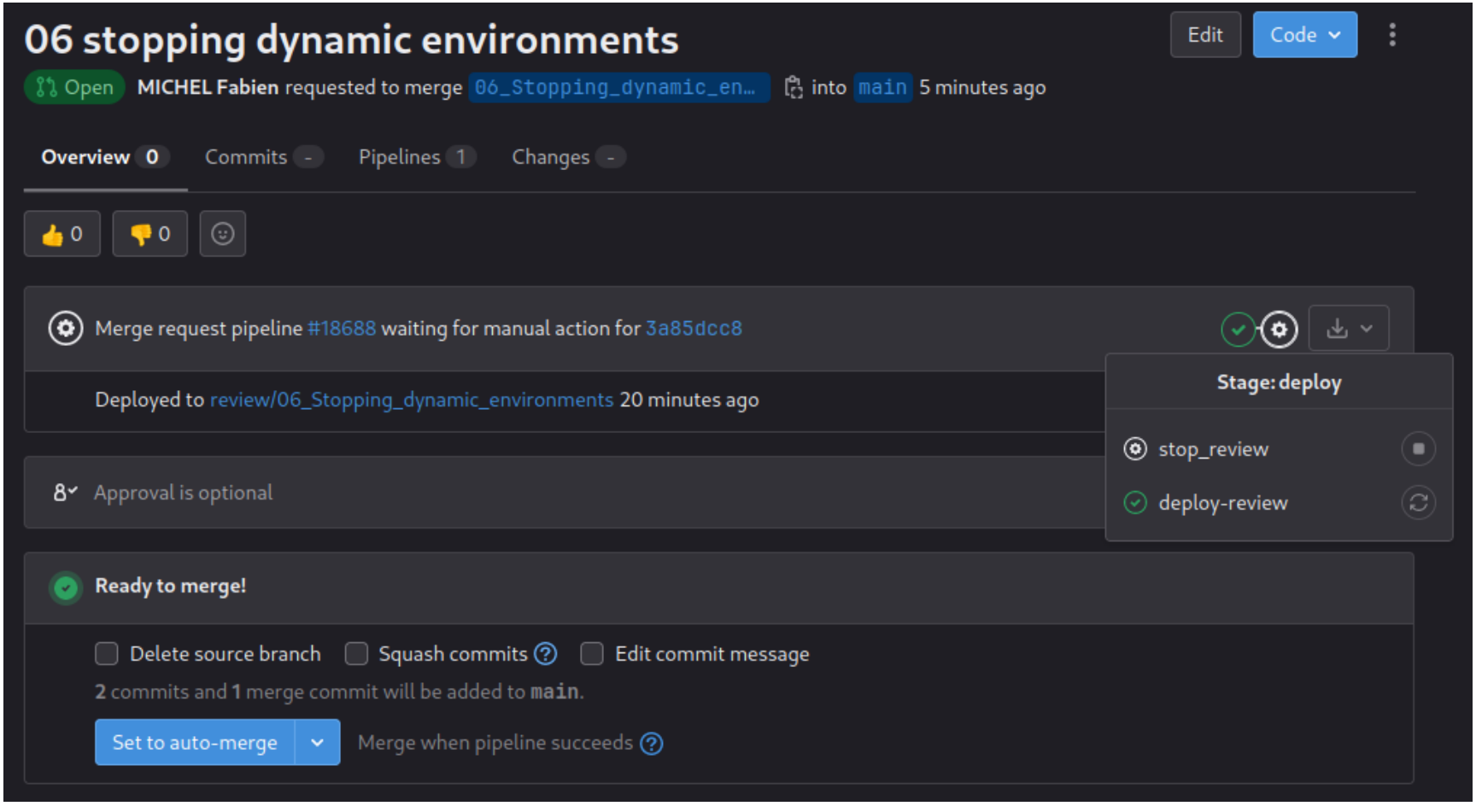
```yaml
deploy_review:
  stage: deploy
  script:
    - echo "Deploy a review app"
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    url: https://$CI_ENVIRONMENT_SLUG.example.com
    on_stop: stop_review

stop_review:
  stage: deploy
  script:
    - echo "Remove review app"
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    action: stop
  when: manual
```

Ajout du job `stop_review` dans le pipeline

Il consiste dans un simple nettoyage du répertoire correspondant

```yaml
deploy-review:
  image: finalgene/openssh
  stage: deploy
  variables:
    TARGET_DIR: "$FTP_BASE_DIR/review/$CI_COMMIT_REF_NAME"
  script:
    - tar cvzf $PACKAGE_NAME --directory=. hello/*class
    - echo "<a href=\"$PACKAGE_NAME\">Testing version -> $PACKAGE_NAME</a>" > index.html
    - apk add lftp #install the lftp package on the alpine distro
    - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; mkdir -p $TARGET_DIR; cd $TARGET_DIR; put $PACKAGE_NAME index.htm
.univ-montp2.fr
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://webinfo.iutmontp.univ-montp2.fr/~michel/review/$CI_COMMIT_REF_NAME
    on_stop: stop_review
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"

stop_review:
  image: finalgene/openssh
  stage: deploy
  variables:
    TARGET_DIR: "$FTP_BASE_DIR/review/$CI_COMMIT_REF_NAME"
  script:
    - apk add lftp
    - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; rm -rf $TARGET_DIR; bye;" sftp://ftpinfo.iutmontp.univ-montp2.fr
  environment:
    name: review/$CI_COMMIT_REF_NAME
    action: stop
  rules:
    - if: $CI_MERGE_REQUEST_ID
      when: manual
```

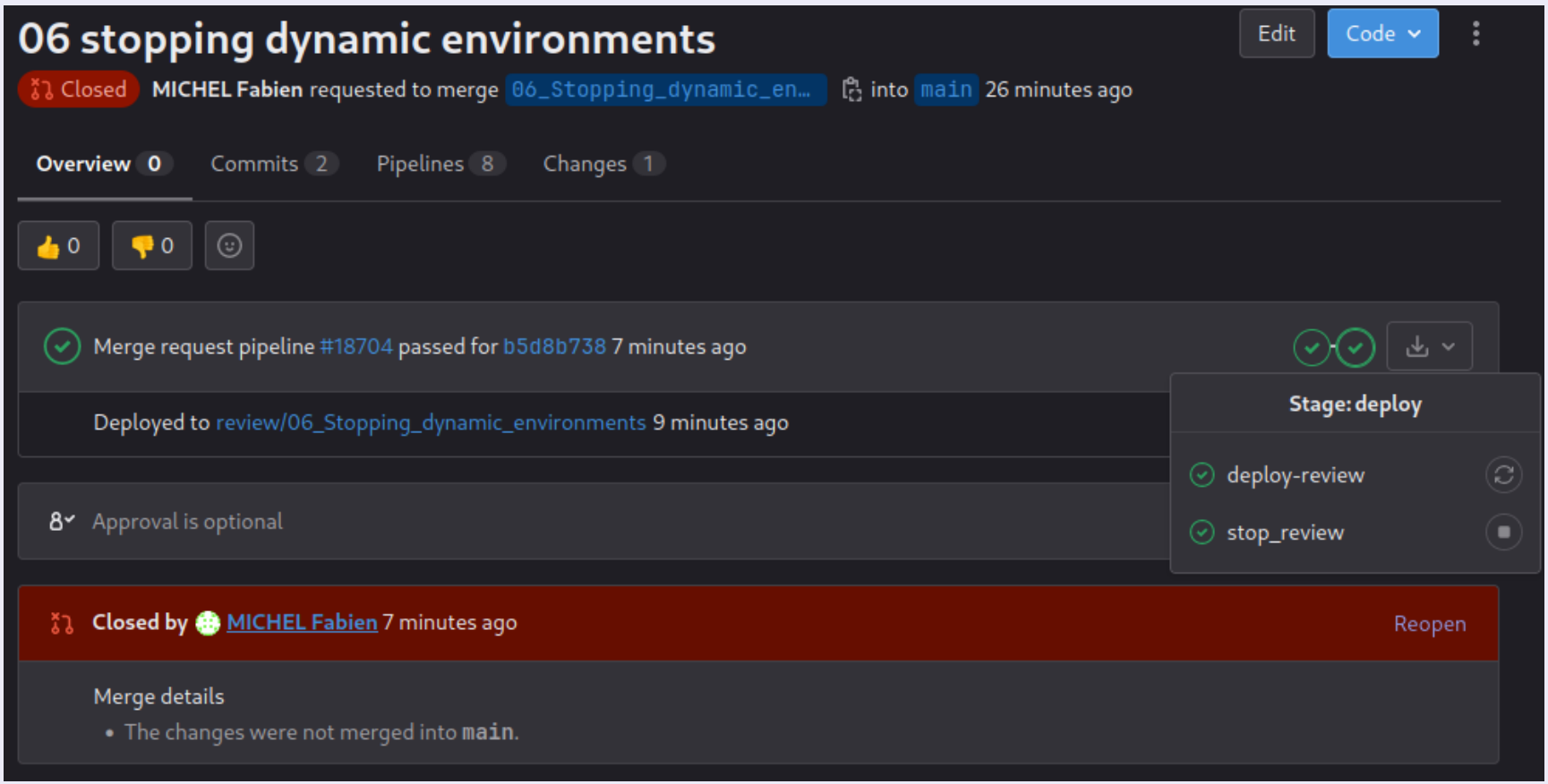
Apparition du job `stop_review` dans la MR



Déclenchement du job `stop_review`

Il sera déclenché manuellement : par (1) merge/close de la MR ou (2) un clic sur le job.

Exemple avec un *close* de la MR



Plan

Beaucoup de choses sont à améliorer

```yaml
default:
  image: eclipse-temurin:11-jdk-alpine

variables:
  VERSION: "4.0"
  PACKAGE_NAME: "hello-$VERSION.tar.gz"
  STAGING_URL: "cicd/staging"
  PRODUCTION_URL: "cicd/production"
  FTP_BASE_DIR: "public_html"

build:
  stage: build
  script:
    - javac hello/Hello.java
  artifacts:
    paths:
      - hello/Hello.class
  rules:
  - when: always

test:
  stage: test
  script: java hello.Hello  | grep -q 'Hello world'

deploy-review:
  image: finalgene/openssh
  stage: deploy
  variables:
    TARGET_DIR: "$FTP_BASE_DIR/cicd/review/$CI_COMMIT_REF_NAME"
  script:
    - tar cvzf $PACKAGE_NAME --directory=. hello/*class
    - echo "<a href=\"$PACKAGE_NAME\">Testing version -> $PACKAGE_NAME</a>" > index.html
    - apk add lftp #install the lftp package on the alpine distro
    - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; mkdir -p $TARGET_DIR; cd $TARGET_DIR; put $PACKAGE_NAME index.html; bye;" sftp://ftpinfo.iutmontp.univ-montp2.fr
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://webinfo.iutmontp.univ-montp2.fr/~michel/cicd/review/$CI_COMMIT_REF_NAME
    on_stop: stop_review
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"

stop_review:
  image: finalgene/openssh
  stage: deploy
  variables:
    TARGET_DIR: "$FTP_BASE_DIR/cicd/review/$CI_COMMIT_REF_NAME"
  script:
    - apk add lftp
    - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; rm -rf $TARGET_DIR; bye;" sftp://ftpinfo.iutmontp.univ-montp2.fr
  environment:
    name: review/$CI_COMMIT_REF_NAME
    action: stop
  rules:
    - if: $CI_MERGE_REQUEST_ID
      when: manual

deploy-staging:
  image: finalgene/openssh
  stage: deploy
  variables:
    TARGET_DIR: "$FTP_BASE_DIR/$STAGING_URL"
  script:
    - tar cvzf $PACKAGE_NAME --directory=. hello/*class
    - echo "<a href=\"$PACKAGE_NAME\">Last version -> $PACKAGE_NAME</a>" > index.html
    - apk add lftp #install the lftp package on the alpine distro
    - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; mkdir -p $TARGET_DIR; cd $TARGET_DIR; put $PACKAGE_NAME index.html; bye;" sftp://ftpinfo.iutmontp.univ-montp2.fr
  environment:
    name: staging
    url: https://webinfo.iutmontp.univ-montp2.fr/~michel/$STAGING_URL/
  rules:
    - if: $CI_COMMIT_BRANCH == "main"

deploy-production:
  image: finalgene/openssh
  stage: deploy
  variables:
    TARGET_DIR: "$FTP_BASE_DIR/$PRODUCTION_URL"
  script:
    - tar cvzf $PACKAGE_NAME --directory=. hello/*class
    - echo "<a href=\"$PACKAGE_NAME\">Last version -> $PACKAGE_NAME</a>" > index.html
    - apk add lftp #install the lftp package on the alpine distro
    - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; mkdir -p $TARGET_DIR; cd $TARGET_DIR; put $PACKAGE_NAME index.html; bye;" sftp://ftpinfo.iutmontp.univ-montp2.fr
  environment:
    name: production
  rules:
    - if: $CI_COMMIT_TAG
      when: manual
```

Beaucoup de choses sont à améliorer

Défauts potentiels

1. Il y a trop de lignes semblables
2. Le numéro de version ne devrait pas être toujours identique
3. `PRODUCTION_URL` / `STAGING_URL`... ne devraient pas être là

Améliorations possibles

1. Solutions pour la factorisation
   - Utiliser le mot-clé `include` pour factoriser include
   - Utiliser un moteur de production comme Gradle
   - Chercher ou créer une image docker adaptée aux besoins de déploiement
2. Utiliser le contexte pour nommer la version : feature / pré-prod / tags
3. Utiliser des variables définies dans l'UI de GitLab

Ce que nous n'avons pas vu

L'utilisation du mot-clé `workflow` pour définir le comportement du pipeline en fonction du contexte ▸ workflow

Examples of `workflow:name`:

A simple pipeline name with a predefined variable:

```yaml
workflow:
  name: 'Pipeline for branch: $CI_COMMIT_BRANCH'
```

A configuration with different pipeline names depending on the pipeline conditions:

```yaml
variables:
  PROJECT1_PIPELINE_NAME: 'Default pipeline name'  # A default is not required.

workflow:
  name: '$PROJECT1_PIPELINE_NAME'
  rules:
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
      variables:
        PROJECT1_PIPELINE_NAME: 'MR pipeline: $CI_MERGE_REQUEST_SOURCE_BRANCH_NAME'
    - if: '$CI_MERGE_REQUEST_LABELS =~ /pipeline:run-in-ruby3/'
      variables:
        PROJECT1_PIPELINE_NAME: 'Ruby 3 pipeline'
    - when: always  # Other pipelines can run, but use the default name
```

Conclusion

Nécessité de suivre un workflow

- $\Rightarrow$ Le développement doit suivre un workflow
- $\Rightarrow$ Le GitLab flow existe en deux versions : environnement et releases
- $\Rightarrow$ Les environnements de GitLab permettent de simplifier le suivi du workflow

Les rules permettent d'exécuter les jobs en fonction du workflow

- L'exécution des jobs est en fonction du contexte : feature / pré-prod / tags
- On utilise pour ce la les variables prédéfinies, e.g. `CI_COMMI_TAG`