Automatic generation of CI/CD pipelines

Alexandre-Xavier Labonté-Lamoureux
MASc Student
alexandre-xavier.labonte-lamoureux.1@ens.etsmtl.ca

Friday March 7th 2025
Kaloom-TELUS ÉTS IRC f2f Meeting

A little about me

- ~1 year at Arctiq (consulting)
  - Clients: Banks, Payment, Trading, Fintechs and video game companies
  - Certified GitLab Professional Services “Engineer”
  - Mostly been doing GitLab → GitHub migrations or improving pipelines
- Previously: 3 years at Ubisoft
  - SRE/ Platform team, mostly doing cybersecurity in a DevOps environment

The story of automatic CI/CD pipeline generation

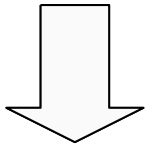
How we got there

ÉTS

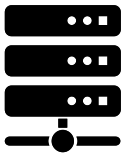
DevOps

Developer

Software

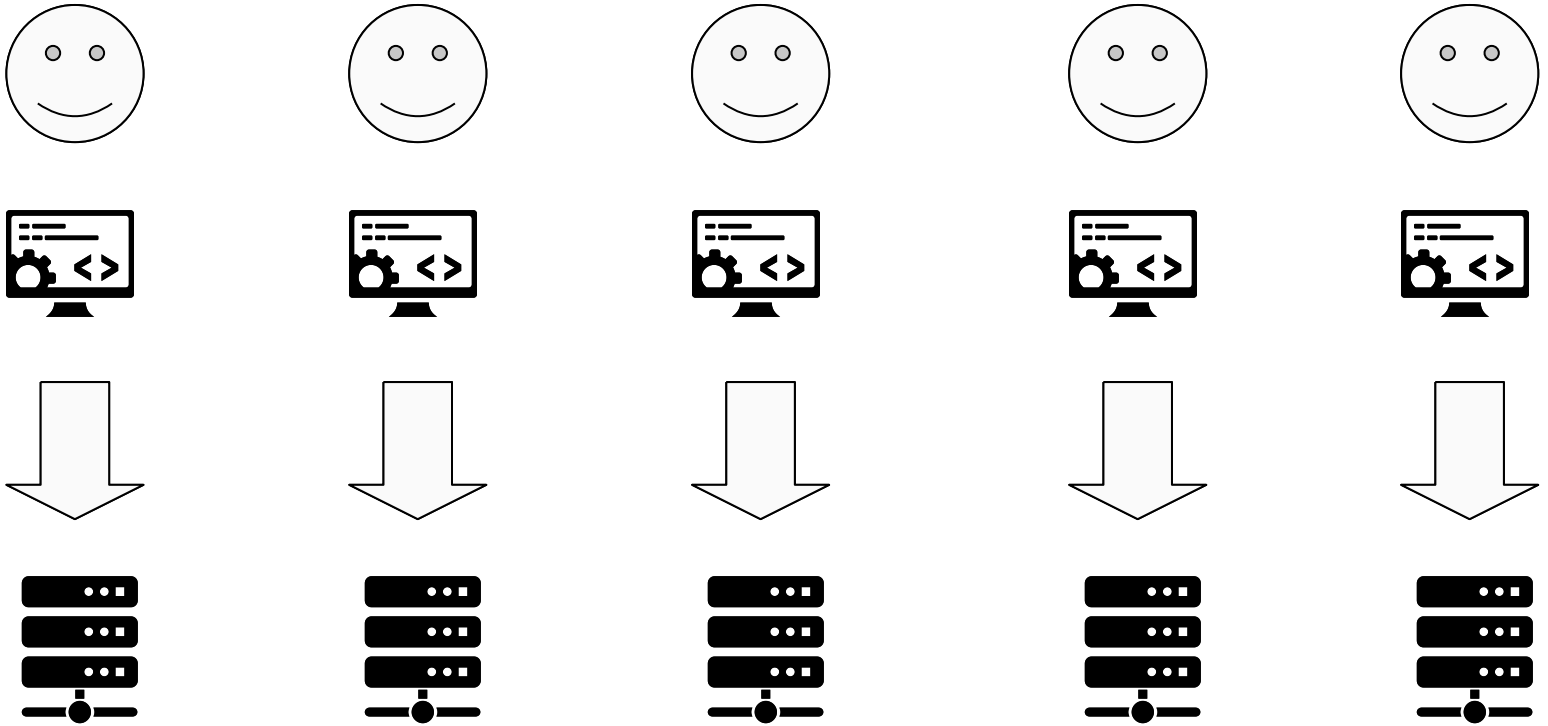


Provisioning and
CI/CD

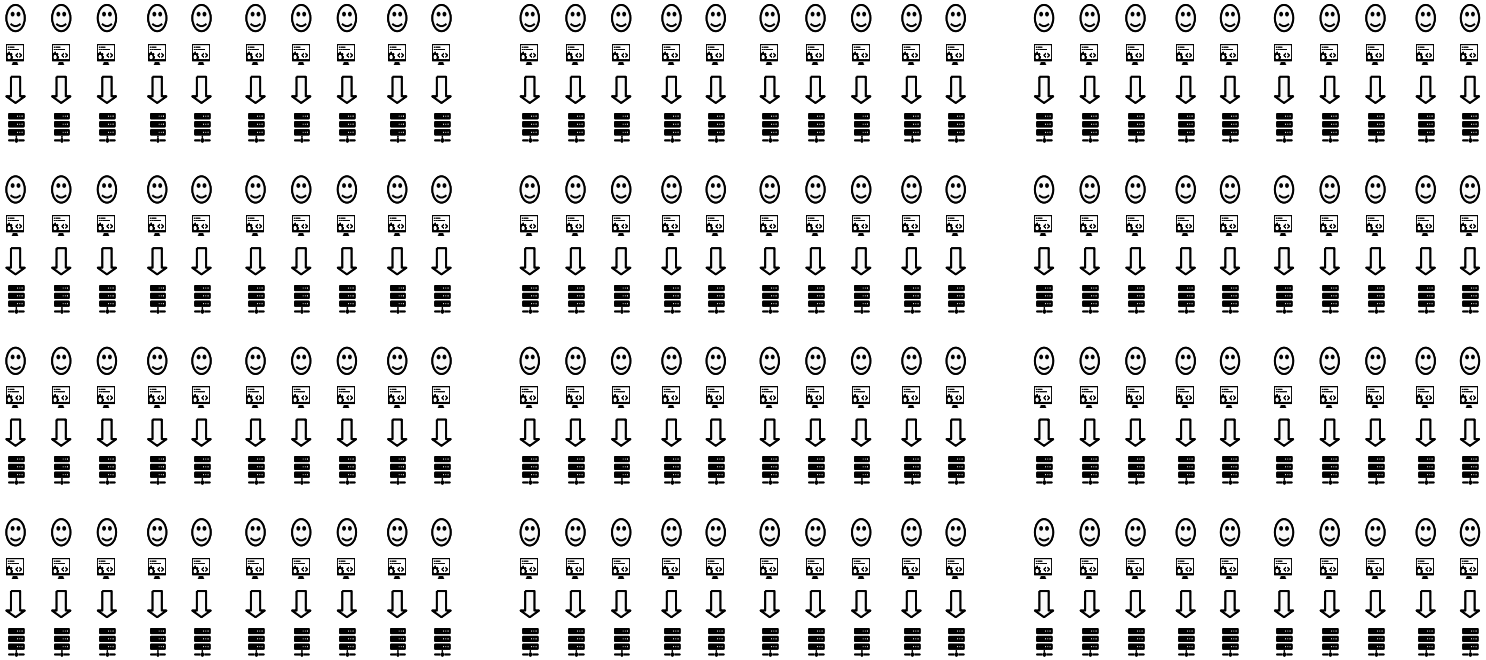


Infrastructure
(network, servers, etc.)

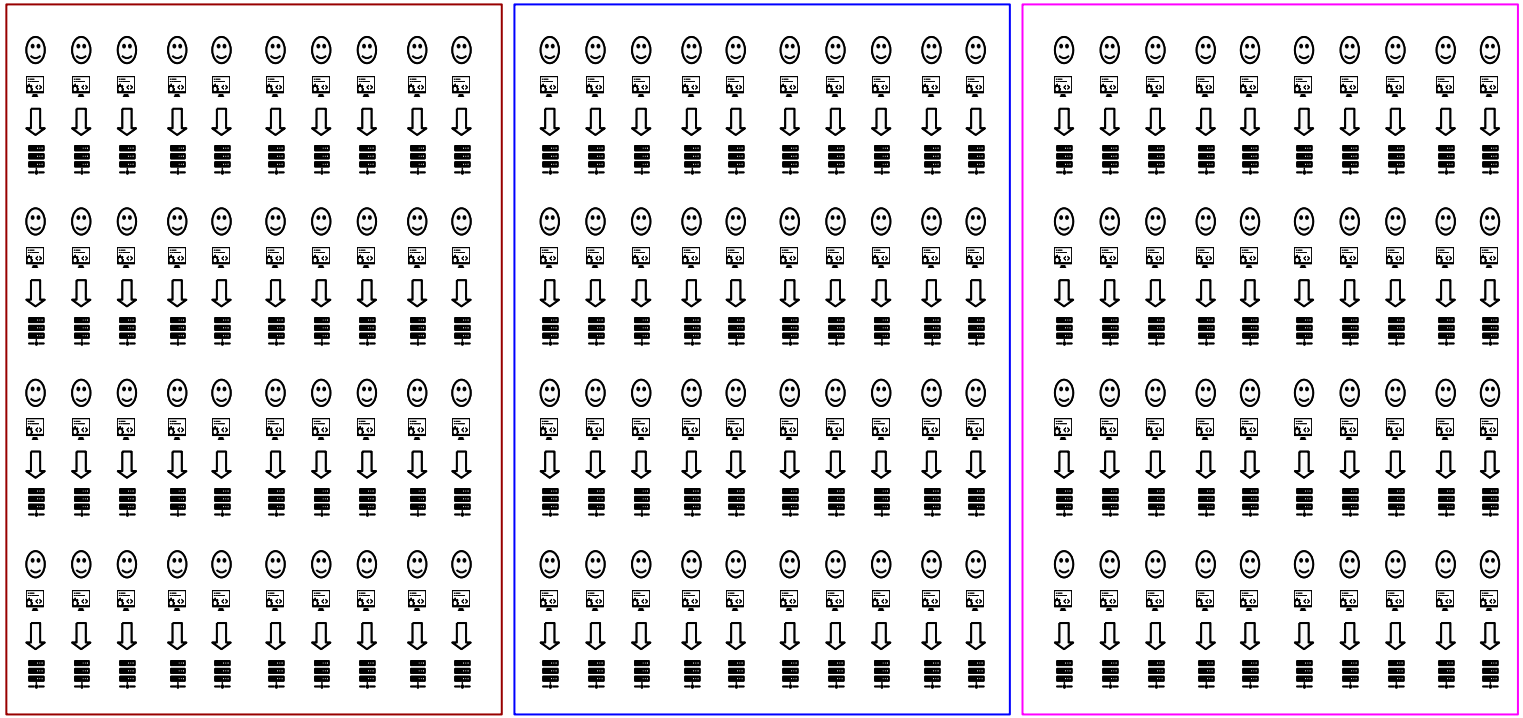
DevOps (at scale?)



DevOps at scale…

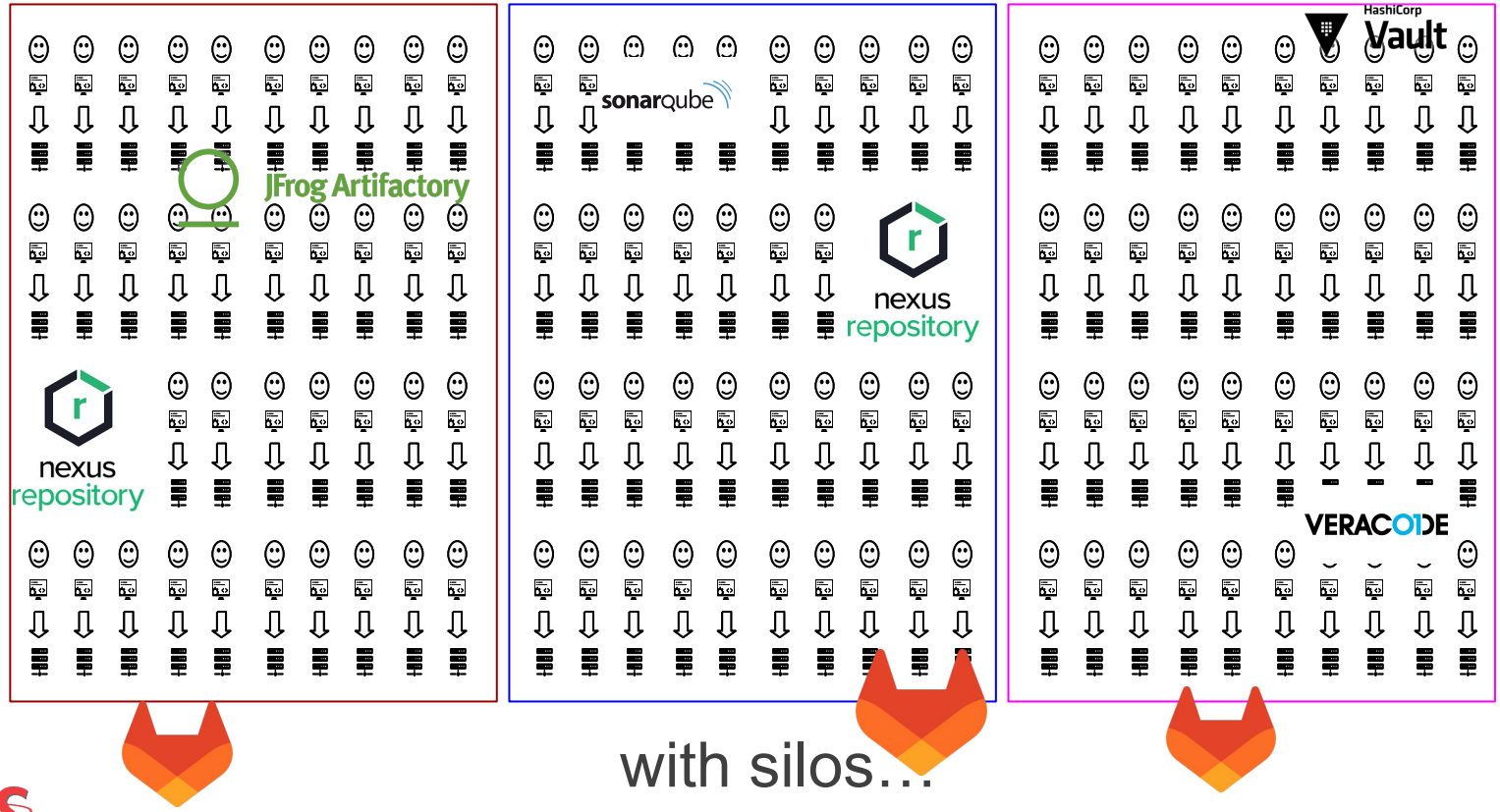


DevOps at scale…



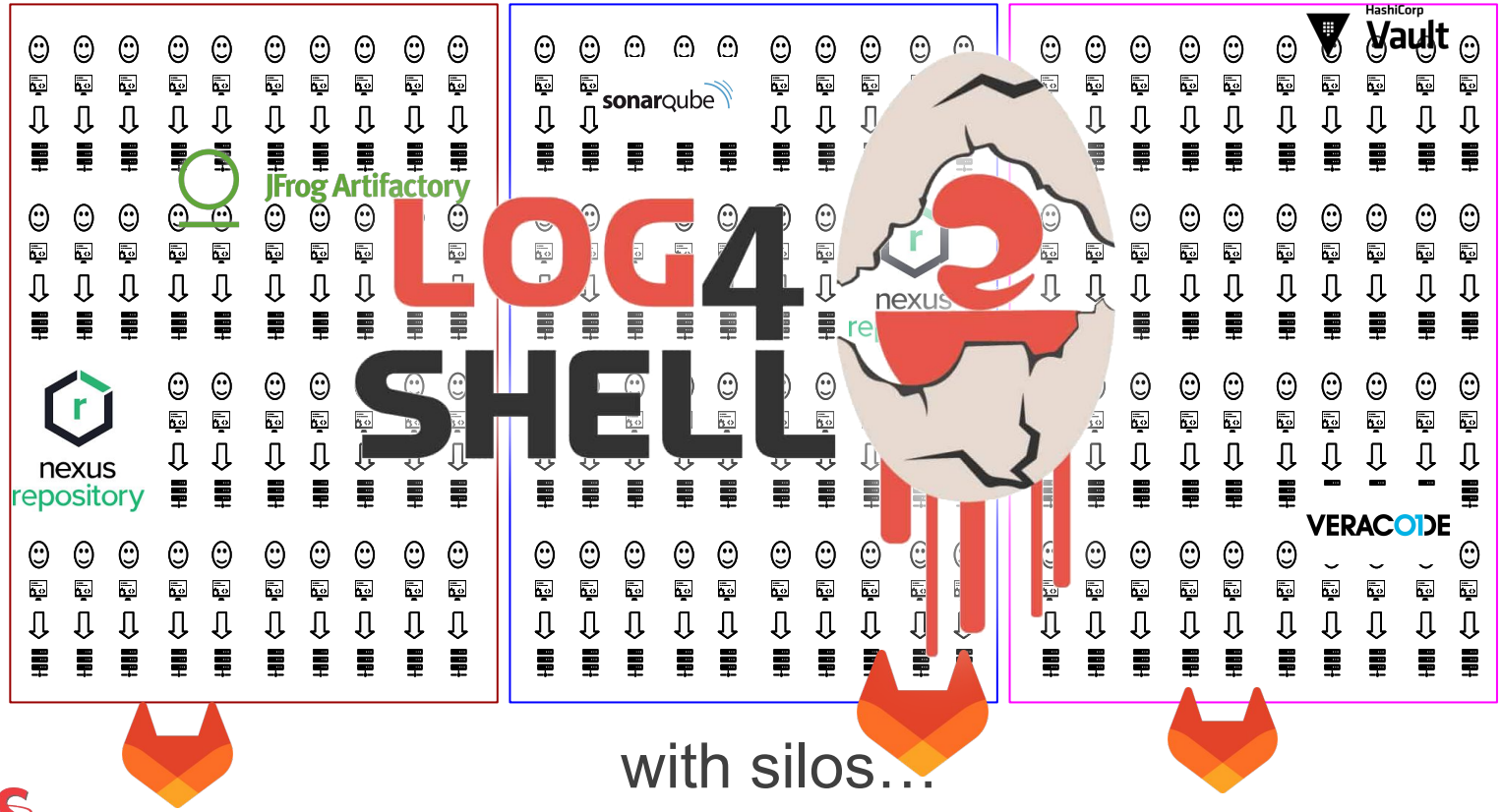
with silos…

DevOps at scale…



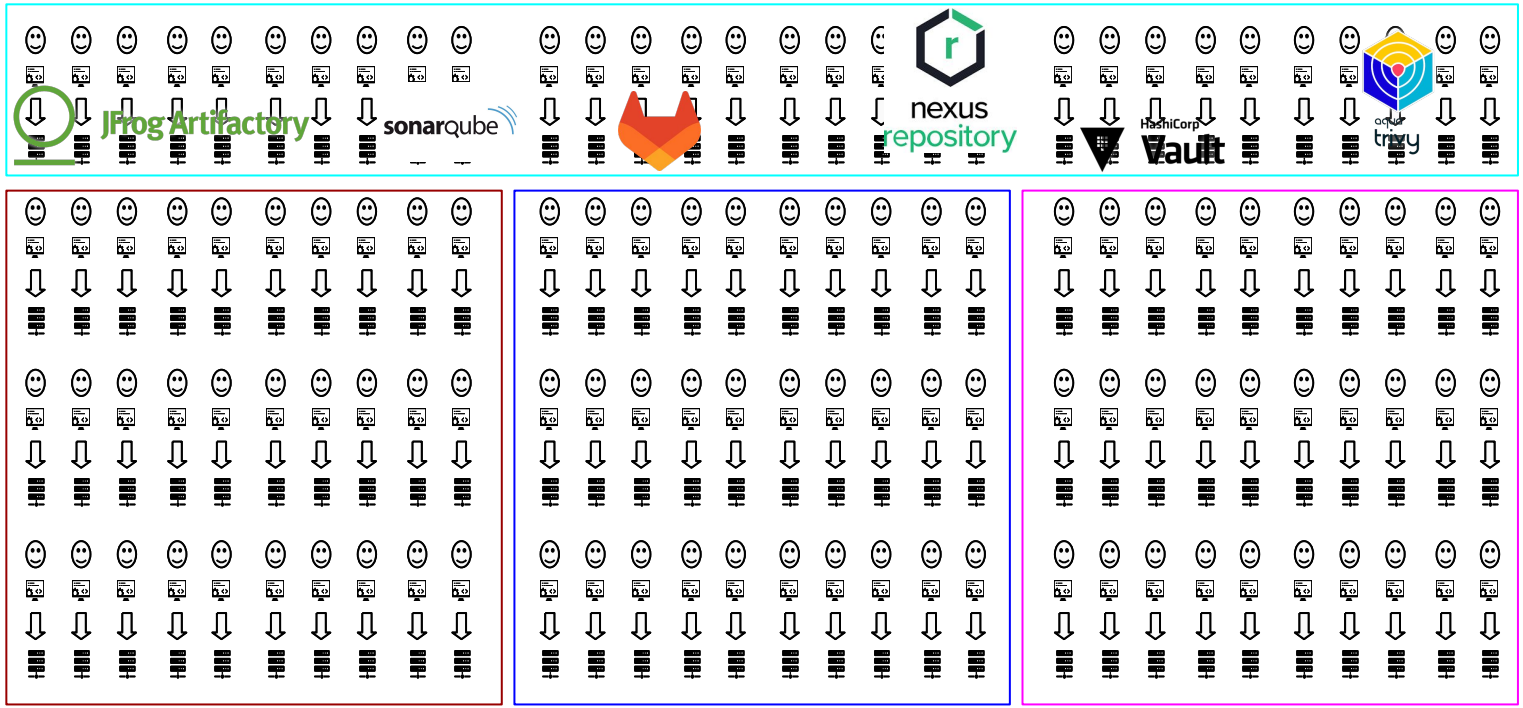
with silos…

DevOps at scale…



with silos…

DevOps at scale…

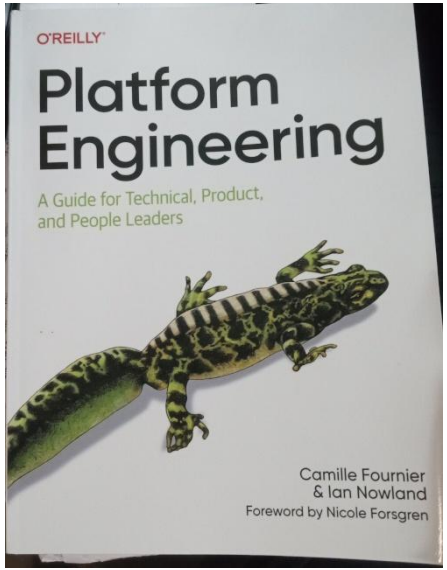


with silos…

ÉTS

How to solve DevOps at scale with Silos

- Regroup common practices under a single team
  - Establish a **Platform Engineering** team
  - Define best practices (Golden paths / Paved roads)
  - Apply the Pareto principle
- Among everything, build a template engine for
  - CI/CD pipelines
  - Infrastructure provisioning code such as Terraform



Attacking CI/CD pipeline complexity

Templates as a solution (maybe?)

GitLab templates

.gitlab-ci.yml 1.60 KiB

```
8  stages:
9  - build
10 - test
11 - review
12 - deploy
13
14 include:
15   - template: Jobs/Build.gitlab-ci.yml
16   - template: Jobs/SAST.gitlab-ci.yml
17   - template: Jobs/Secret-Detection.gitlab-ci.yml
18   - template: Jobs/Dependency-Scanning.gitlab-ci.yml
19   - template: Jobs/Container-Scanning.gitlab-ci.yml
20
21 gemnasium-dependency_scanning:
22   variables:
23     DS_REMEDIATE: "false"
```

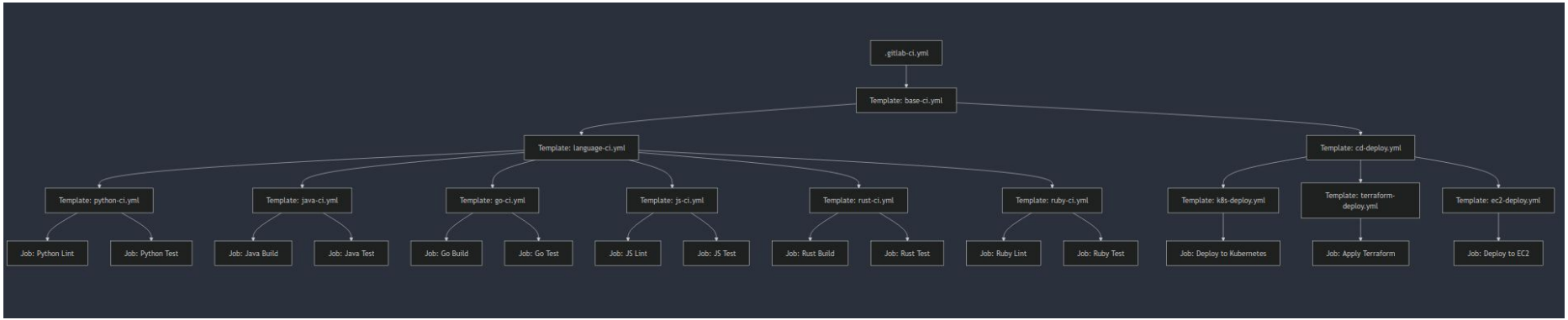
Ugh…

```
configure_system:
  stage: configure
  script:
    - echo "Configuring the deployment system..."
    - if [ "$ENVIRONMENT" == "production" ]; then
          echo "Configuring for PRODUCTION environment...";
          if [ "$BRANCH" == "main" ]; then
              echo "Deploying from the MAIN branch...";
              if [ "$DEPLOY_TYPE" == "full" ]; then
                  echo "Performing a FULL deployment...";
                  if [ "$CONFIRM_DEPLOY" == "true" ]; then
                      echo "Deployment confirmed. Proceeding with FULL production deployment...";
                      ./configure_production.sh --full;
                  else
                      echo "Deployment not confirmed. Aborting FULL production deployment.";
                  fi;
              elif [ "$DEPLOY_TYPE" == "partial" ]; then
                  echo "Performing a PARTIAL deployment...";
                  if [ "$CONFIRM_DEPLOY" == "true" ]; then
                      echo "Deployment confirmed. Proceeding with PARTIAL production deployment...";
                      ./configure_production.sh --partial;
                  else
                      echo "Deployment not confirmed. Aborting PARTIAL production deployment.";
                  fi;
              else
                  echo "Invalid DEPLOY_TYPE specified for production.";
              fi;
          else
              echo "Deployments to PRODUCTION can only be done from the MAIN branch.";
          fi;
      elif [ "$ENVIRONMENT" == "staging" ]; then
          echo "Configuring for STAGING environment...";
          if [ "$BRANCH" == "develop" ]; then
              echo "Deploying from the DEVELOP branch...";
              if [ "$DEPLOY_TYPE" == "full" ]; then
                  echo "Performing a FULL staging deployment...";
                  ./configure_staging.sh --full;
              elif [ "$DEPLOY_TYPE" == "partial" ]; then
                  echo "Performing a PARTIAL staging deployment...";
                  ./configure_staging.sh --partial;
              else
                  echo "Invalid DEPLOY_TYPE specified for staging.";
              fi;
          else
              echo "Deployments to STAGING can only be done from the DEVELOP branch.";
          fi;
      else
          echo "Invalid ENVIRONMENT specified. Must be 'production' or 'staging'.";
      fi
```

Ugh… (2)

```
643 deploy_system:
644   stage: deploy
645   script:
646     - echo "Deploying the system..."
647     - if [ "$ENVIRONMENT" == "production" ]; then
648           echo "Starting PRODUCTION deployment...";
649           if [ "$CONFIRM_DEPLOY" == "true" ]; then
650               echo "Deployment confirmed. Executing production deployment scripts...";
651               ./deploy_production.sh;
652           else
653               echo "Deployment not confirmed. Skipping production deployment.";
654           fi;
655       elif [ "$ENVIRONMENT" == "staging" ]; then
656           echo "Starting STAGING deployment...";
657           ./deploy_staging.sh;
658       else
659           echo "Invalid ENVIRONMENT specified. No deployment will be performed.";
660       fi
661     - echo "Deployment process completed."
```

Ugh… (illustrated)



The process of programming CI/CD pipelines

- Identify a problem or new feature to implement
- Make changes
- Wait for compute to become available
- Wait for compute to run the workload
- Review results of run
- Repeat

Outcome

- Low velocity
  - Slow iterations
  - Time lost waiting
  - Very likely to break something
  - Cannot test all possibilities (can we?)

Outcome

- Testing
  - Can’t test them without running them
  - Need to test every possible scenarios
  - Each “if” block is two possibilities. 2^n where n is the number of possibilities.
  - Complexity grows very fast

Outcome

- Real life example @ Ubisoft
  - More than 1000 test cases
  - Each test case is an entire project that has a different setup
  - Requires a 24 runners farm and an entire day to go through all the tests
  - Tests are executed inside GitLab CI on real runners

Outcome

- Real life example @ Ubisoft
  - Approx 2 weeks required for a change
  - Approx 2 months required for contributions from external teams

Pipeline generator internals

Template engine as the ultimate solution

Project structure

Project structure: templates

ci-templates > ! go.yml

```yaml
image: golang:latest

stages:
  - build
  - test
  - quality
  - security

variables:
  GOPATH: "$HOME/go"
  BINARY_NAME: "app" # default binary name, should be detected automatically

cache:
  paths:
    - "$GOPATH/pkg/mod"

build:
  stage: build
  script:
    - go mod download
    - go build -o $BINARY_NAME ./cmd/*
  artifacts:
    paths:
      - $BINARY_NAME
    expire_in: 1 week
    name: "${CI_PROJECT_NAME}-${CI_COMMIT_REF_NAME}"
```

Project structure: yaml parser

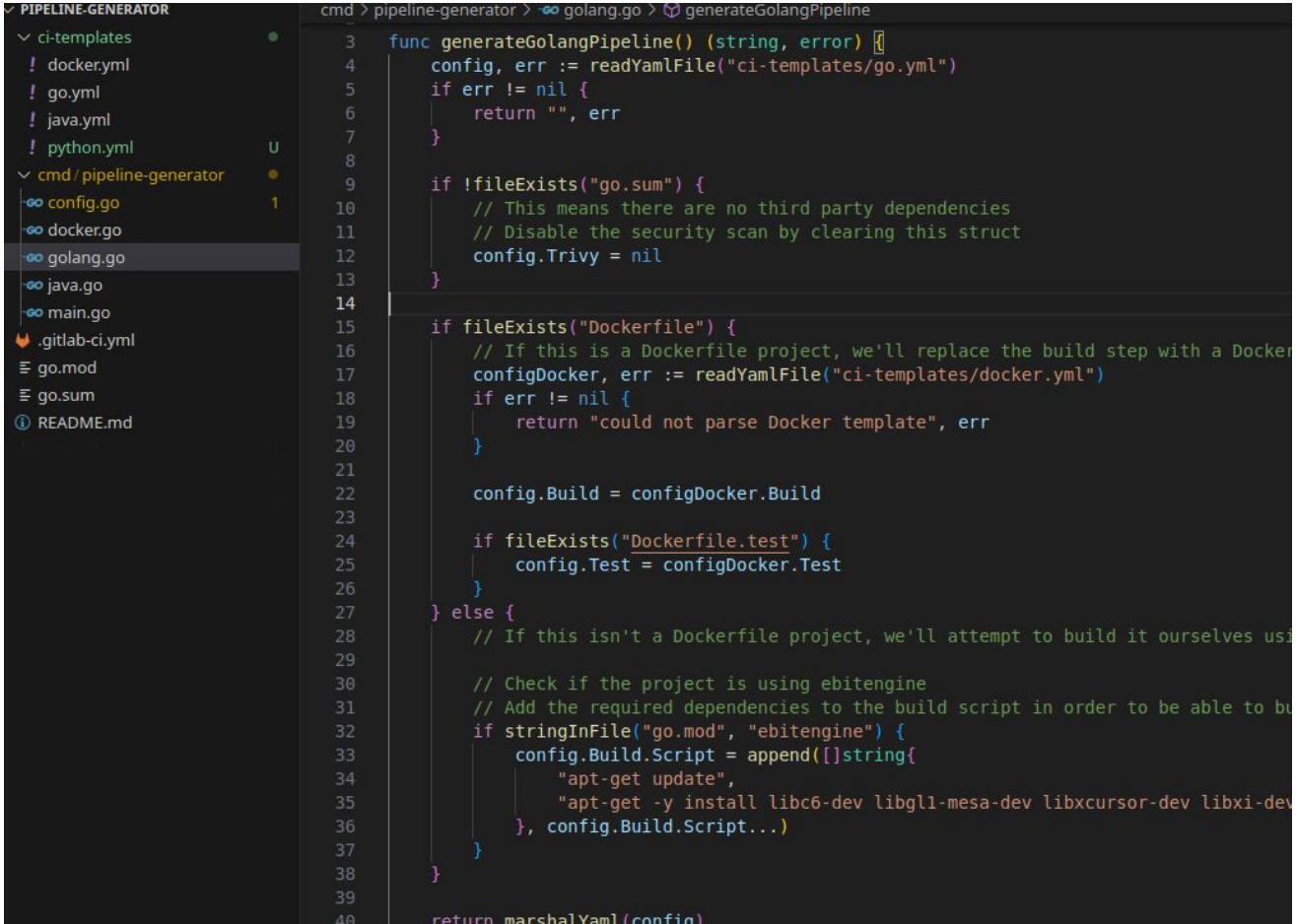
cmd > pipeline-generator > config.go > ...

```go
package main

import (
    "fmt"
    "os"

    "gopkg.in/yaml.v3"
)

type Config struct {
    Image     string            `yaml:"image"`
    Stages    []string          `yaml:"stages"`
    Variables map[string]string `yaml:"variables"`
    Cache     struct {
        Paths []string `yaml:"paths"`
    } `yaml:"cache"`
    Services []string `yaml:"services,omitempty"`
    Build    struct {
        Image     string            `yaml:"image,omitempty"`
        Stage     string            `yaml:"stage"`
        Variables map[string]string `yaml:"variables,omitempty"`
        Services  []string          `yaml:"services,omitempty"`
        Script    []string          `yaml:"script"`
        Artifacts struct {
            Paths    []string `yaml:"paths,omitempty"`
            ExpireIn string   `yaml:"expire_in,omitempty"`
            Name     string   `yaml:"name,omitempty"`
        } `yaml:"artifacts"`
        Only []string `yaml:"only,omitempty"`
    } `yaml:"build"`
    Test *struct {
        Image     string            `yaml:"image,omitempty"`
        Stage     string            `yaml:"stage"`
```

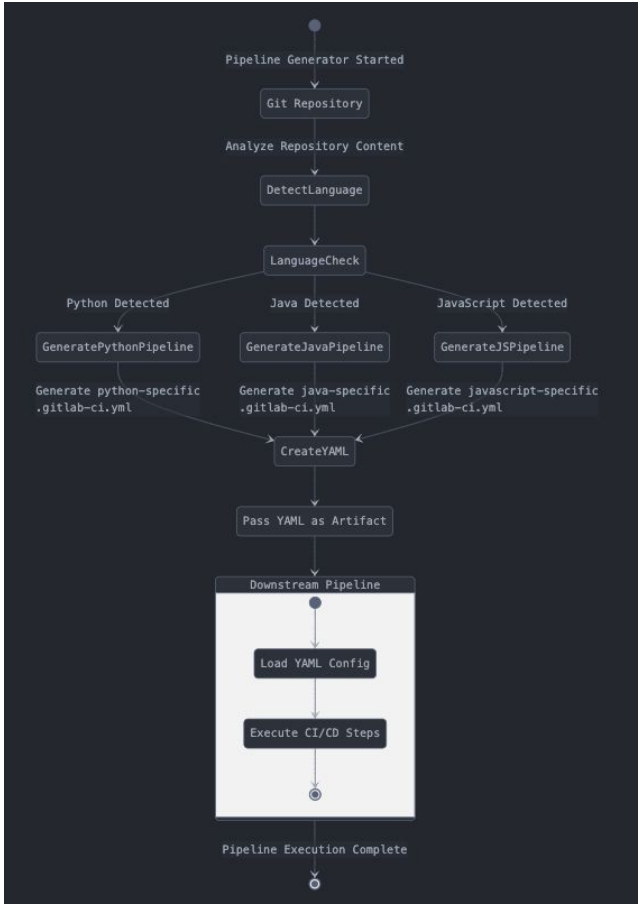
Project structure: language detection

```go
func main() {
    if len(os.Args) < 2 {
        log.Fatalln("Missing argument")
    }

    outputFile := os.Args[1]

    if fileExists("go.mod") {
        log.Println("Generating pipeline for Go project")
        script, err := generateGolangPipeline()
        if err != nil {
            log.Fatalf("Error generating pipeline: %v", err)
        }
        os.WriteFile(outputFile, []byte(script), 0644)
    } else if fileExists("pom.xml") {
        log.Println("Generating pipeline for Java Maven project")
        script, err := generateJavaMavenPipeline()
        if err != nil {
            log.Fatalf("Error generating pipeline: %v", err)
        }
        os.WriteFile(outputFile, []byte(script), 0644)
    } else if fileExists("requirements.txt") {
        log.Println("Generating pipeline for Python project")
        script, err := generatePythonPipeline()
        if err != nil {
            log.Fatalf("Error generating pipeline: %v", err)
        }
        os.WriteFile(outputFile, []byte(script), 0644)
    } else if fileExists("Dockerfile") {
        log.Println("Generating pipeline for Docker project")
        script, err := generateDockerPipeline()
        if err != nil {
            log.Fatalf("Error generating pipeline: %v", err)
        }
        os.WriteFile(outputFile, []byte(script), 0644)
    } else {
```

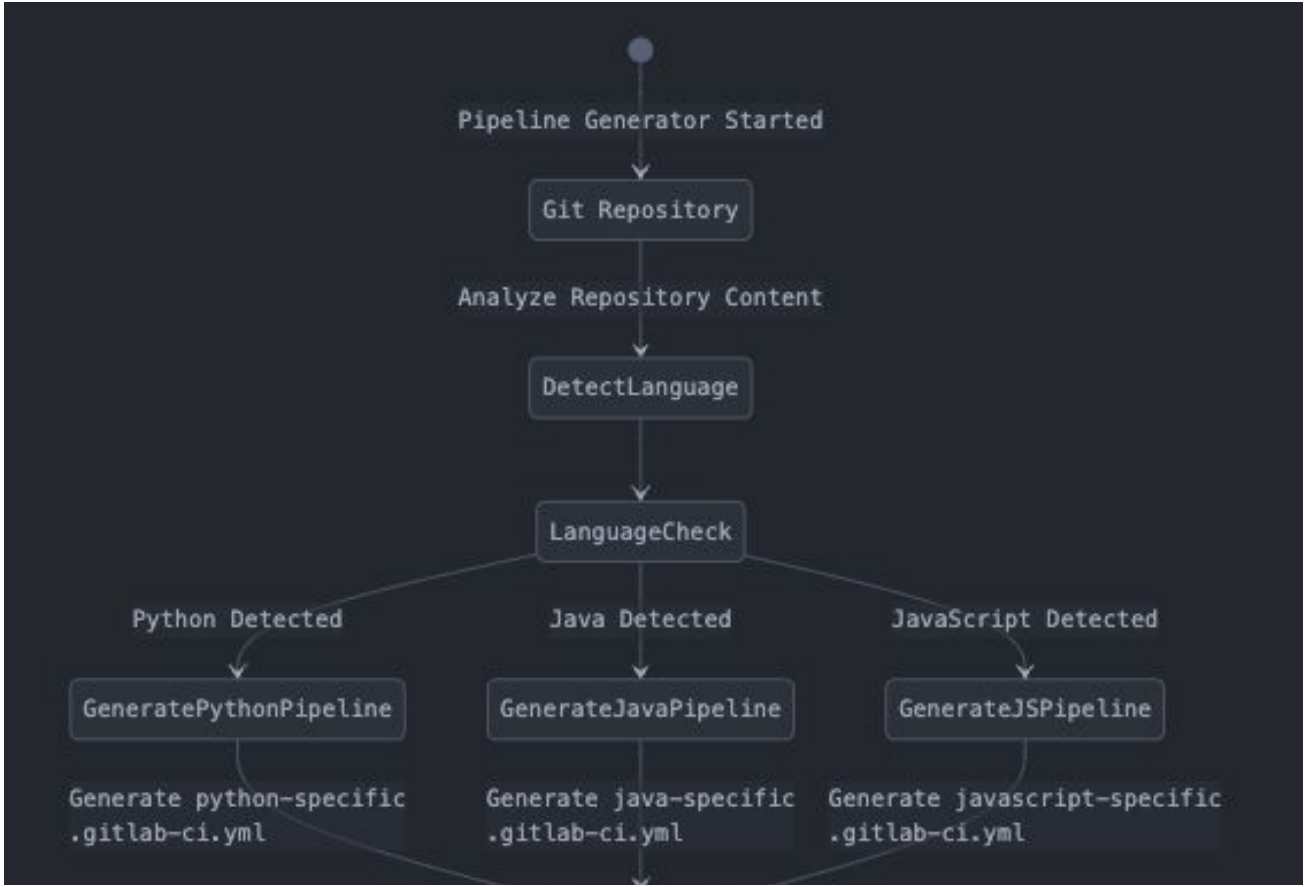
Project structure: language specific code



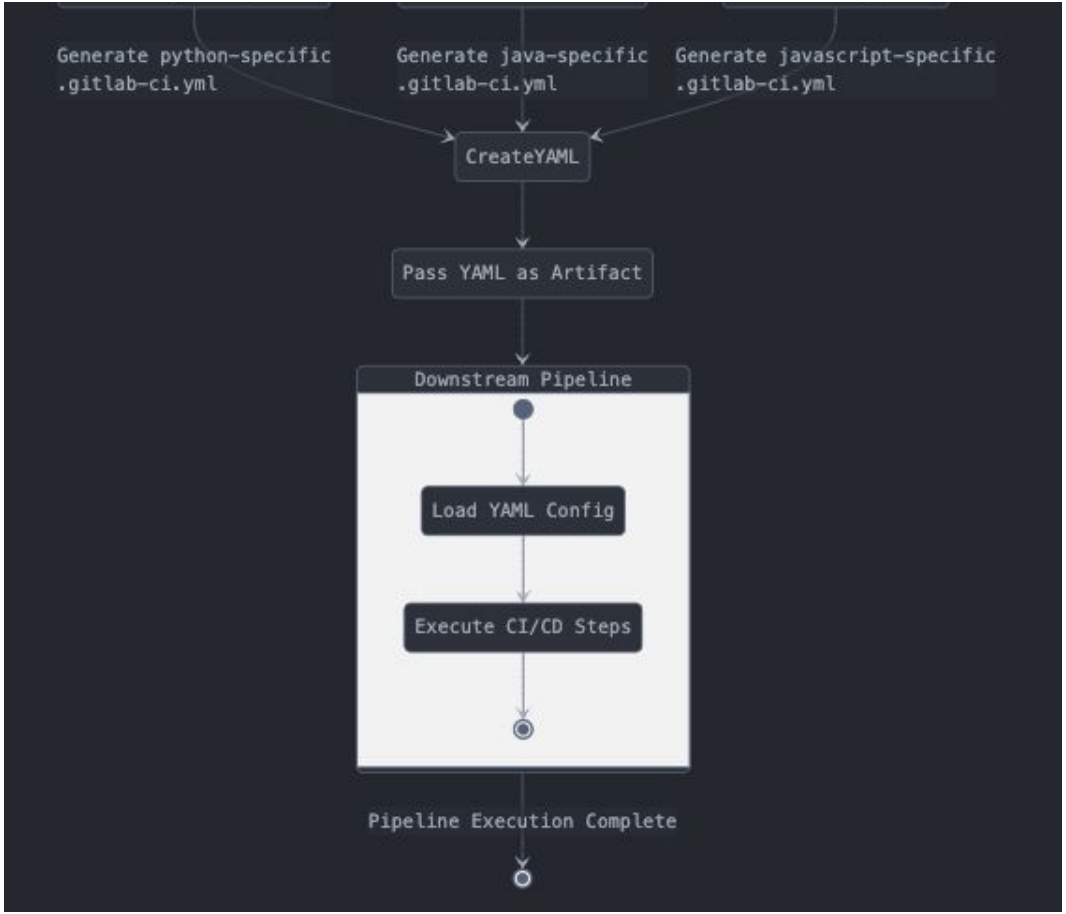
Pipeline generator workflow



Pipeline generator workflow

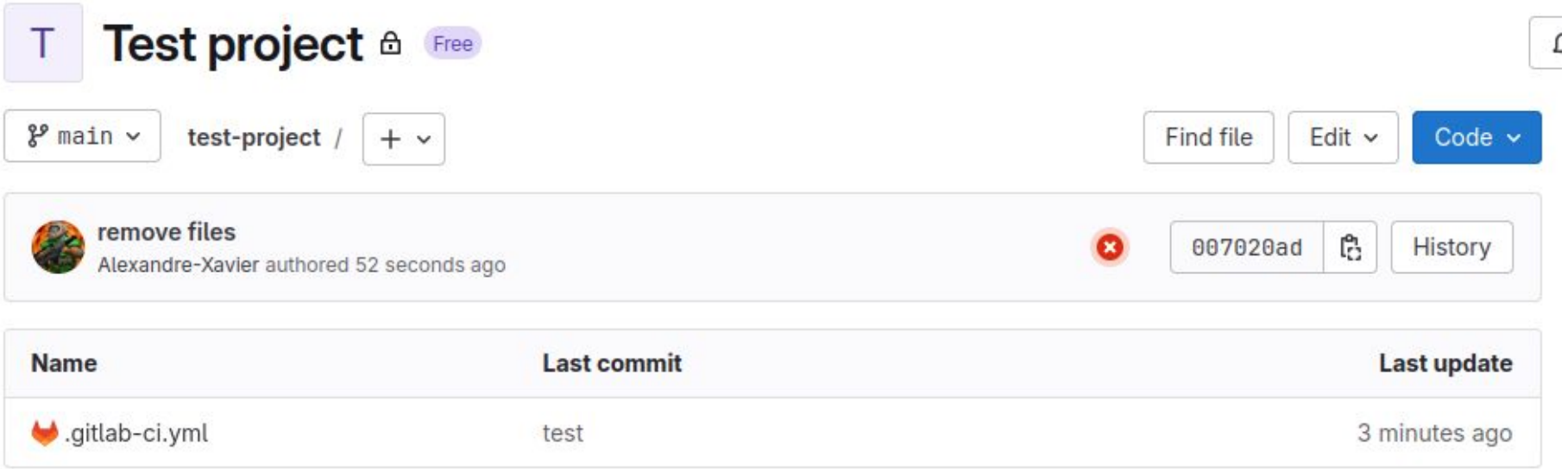


Pipeline generator workflow

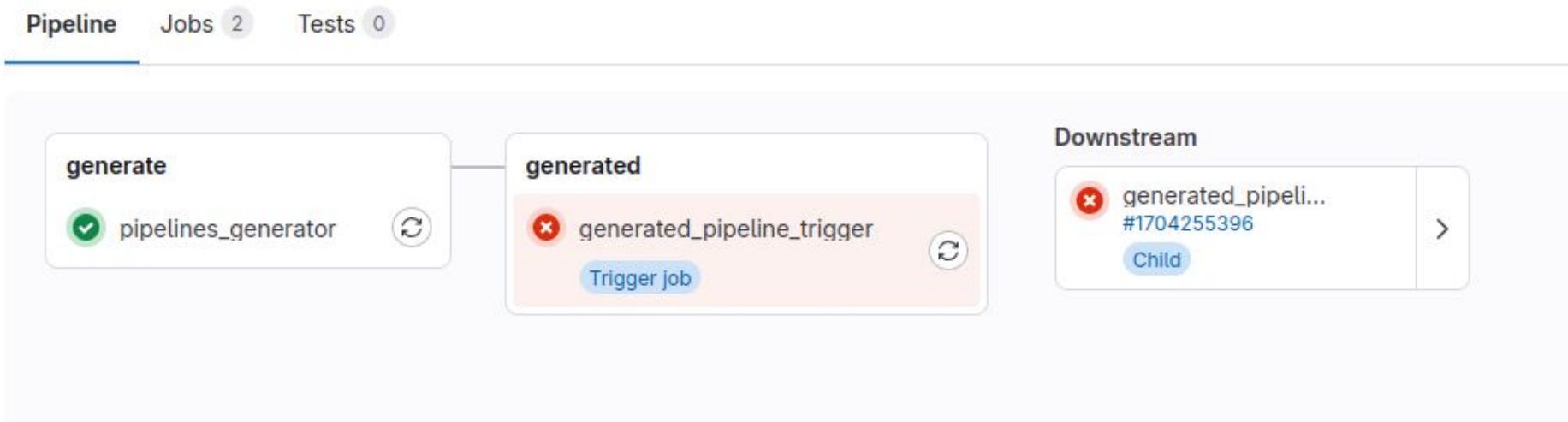


Demo

Demo - Empty test project



Demo - CI script in the empty project

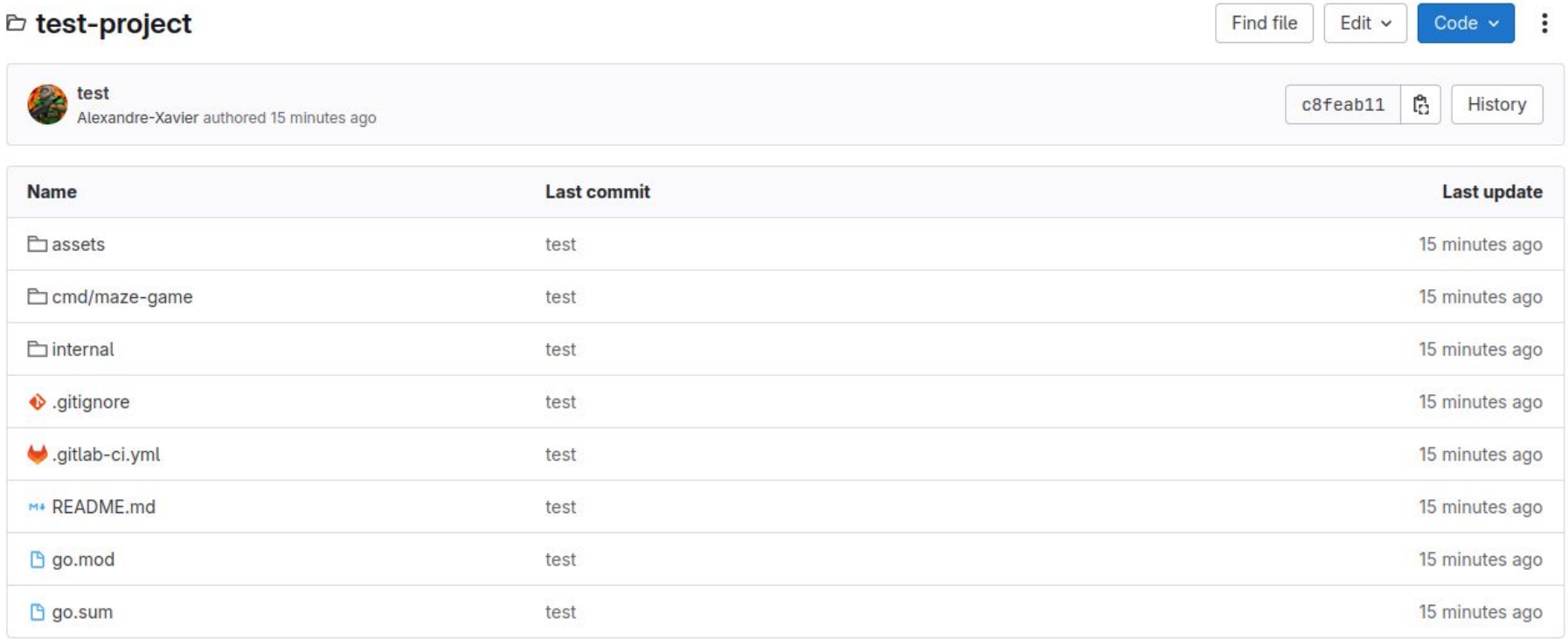


Demo - CI script in the empty project

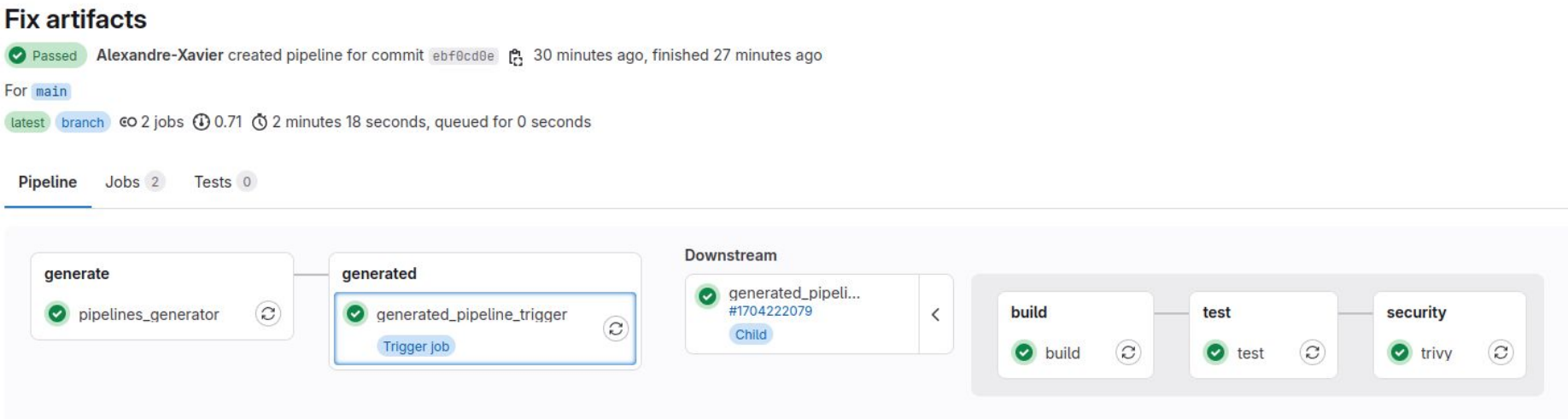
.gitlab-ci.yml 366 B

```yaml
stages:
  - generate
  - generated

pipelines_generator:
  stage: generate
  image: axdoomer/pipeline-generator:latest
  script:
  - echo "Ran"
  artifacts:
    paths:
    - generated-pipeline.yml

generated_pipeline_trigger:
  stage: generated
  trigger:
    include:
      - artifact: generated-pipeline.yml
        job: pipelines_generator
    strategy: depend

```

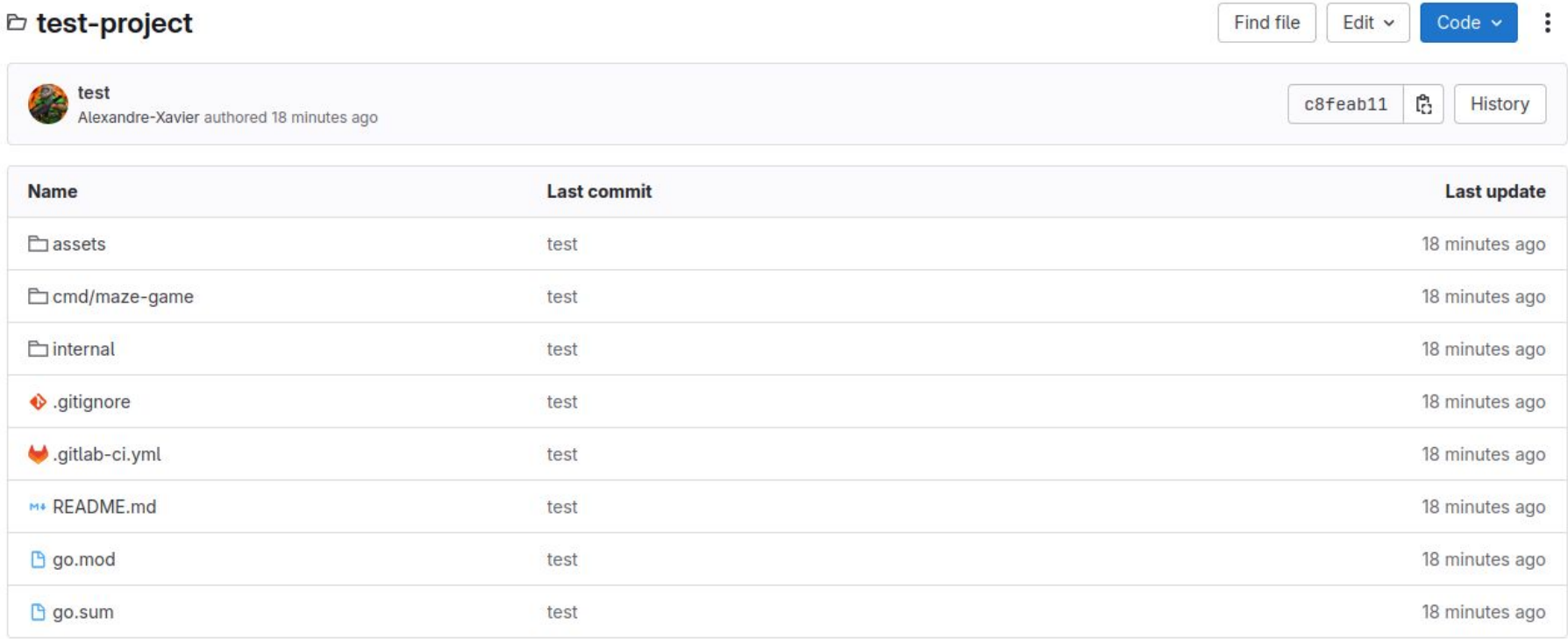
Demo - Go project has been committed



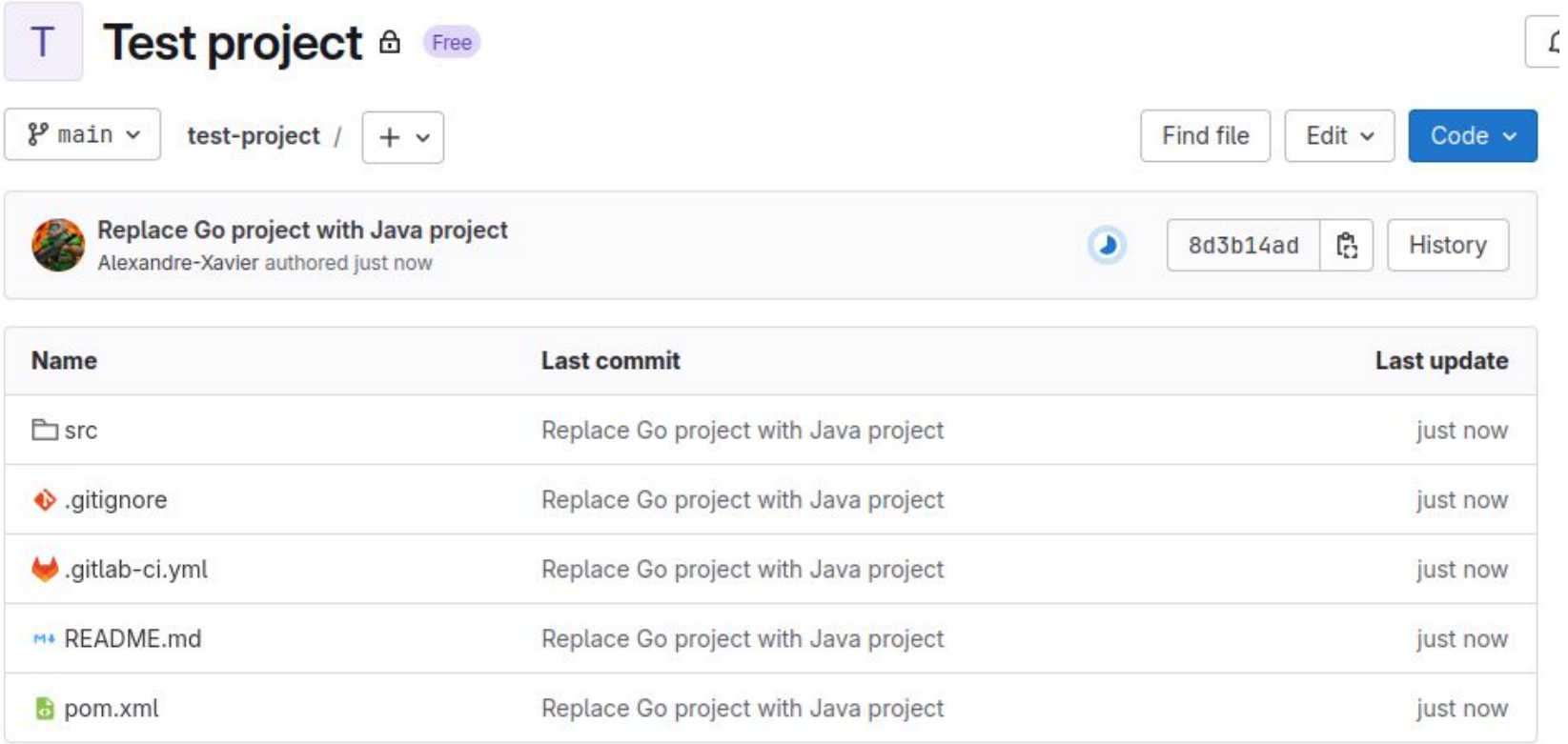
Demo - The CI pipeline has been changed



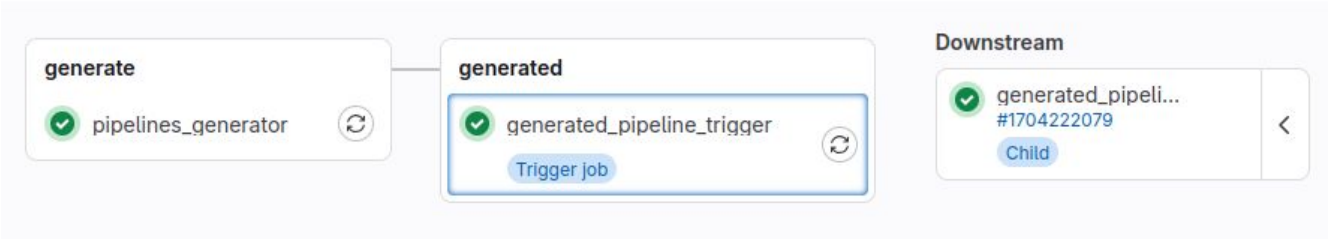
Demo - Replacing the project with another



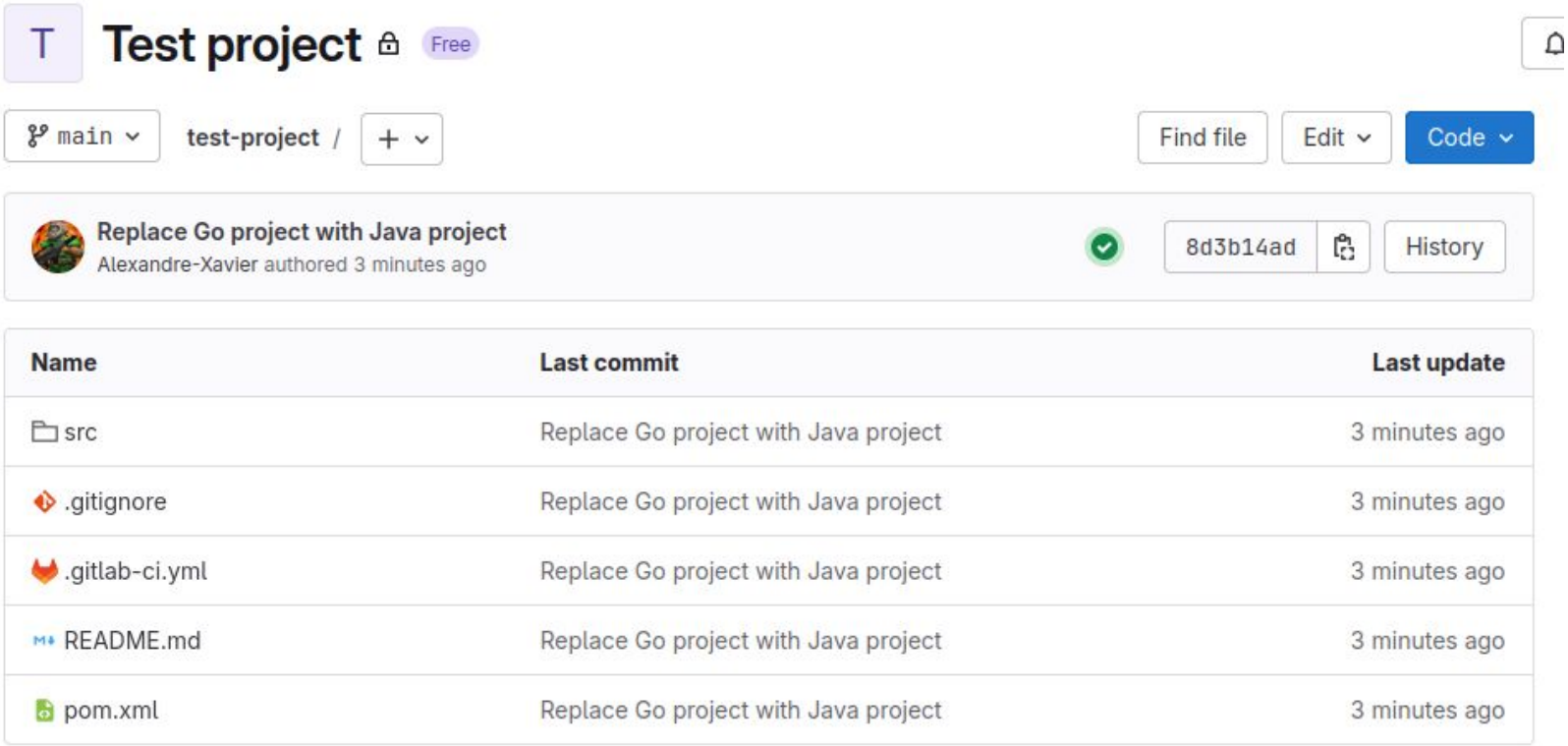
Demo - Replaced project with Java project



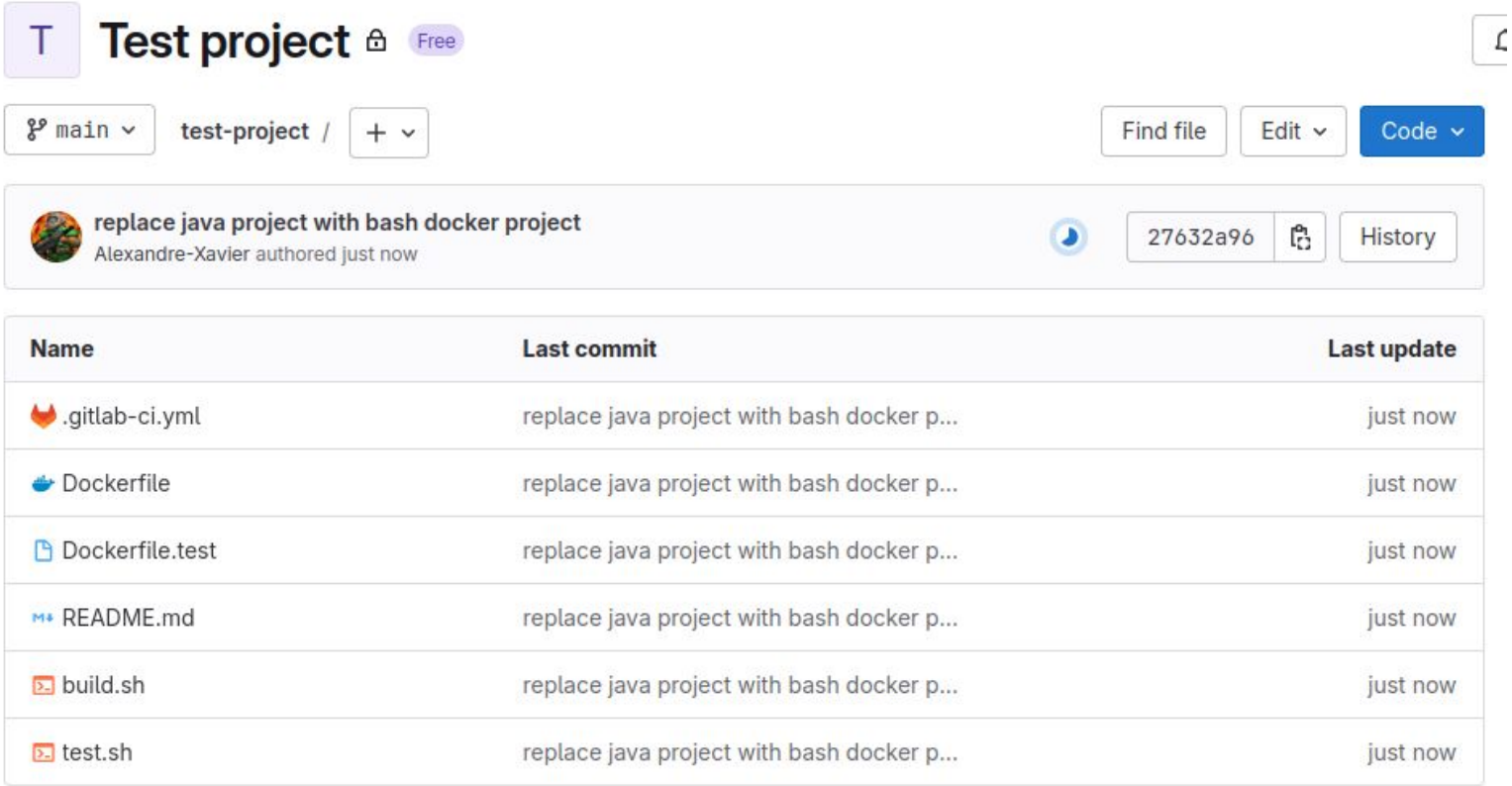
Demo - Replaced project with Java project



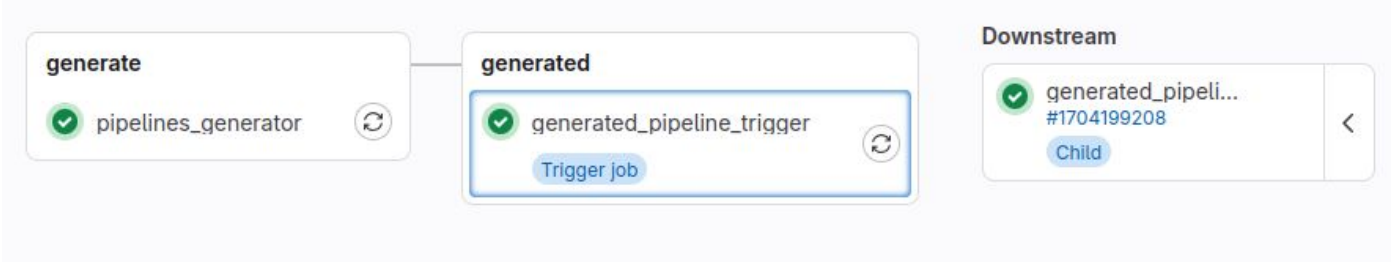
Demo - Replaced project with Java project

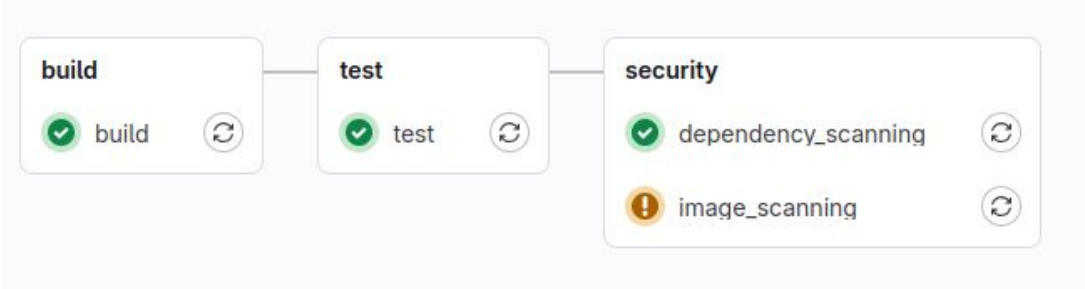


Demo - Replacing the project with a Docker project



Demo - Docker CI pipeline

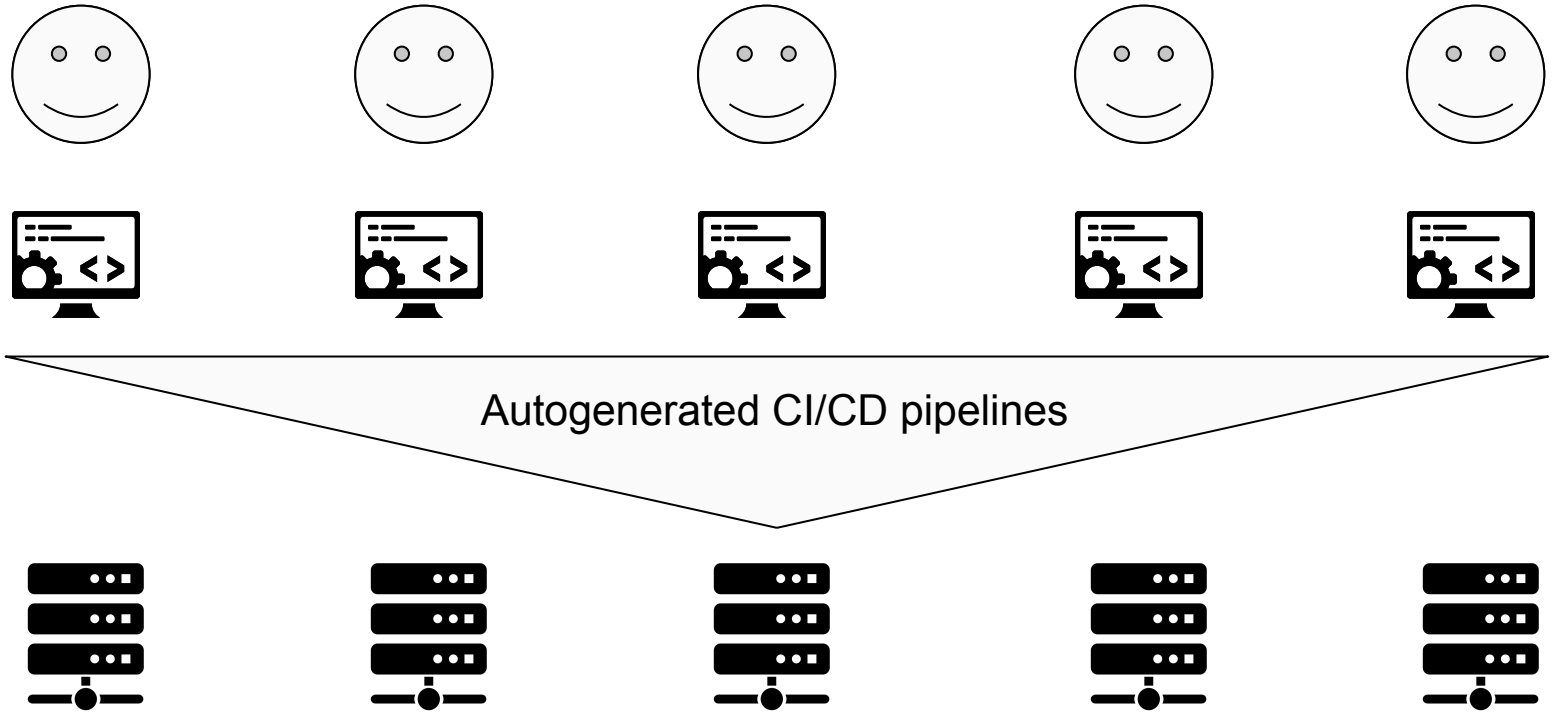




Advantages of this approach

- Contrary to a GitLab CI script, the generator is pure software
  - It can be unit tested
  - It can be integration tested
  - TDD (Test-driven development) can be used to develop pipelines
  - Less cognitive load for developers (better DevEx)
  - Pipelines are always up-to-date with the best practices

DevOps supported by PE team



Paper on Automatic Pipeline Provisioning

https://axdoomer.gitlab.io/Automatic_Pipeline_Provisioning.pdf

Send questions and feedback to my email