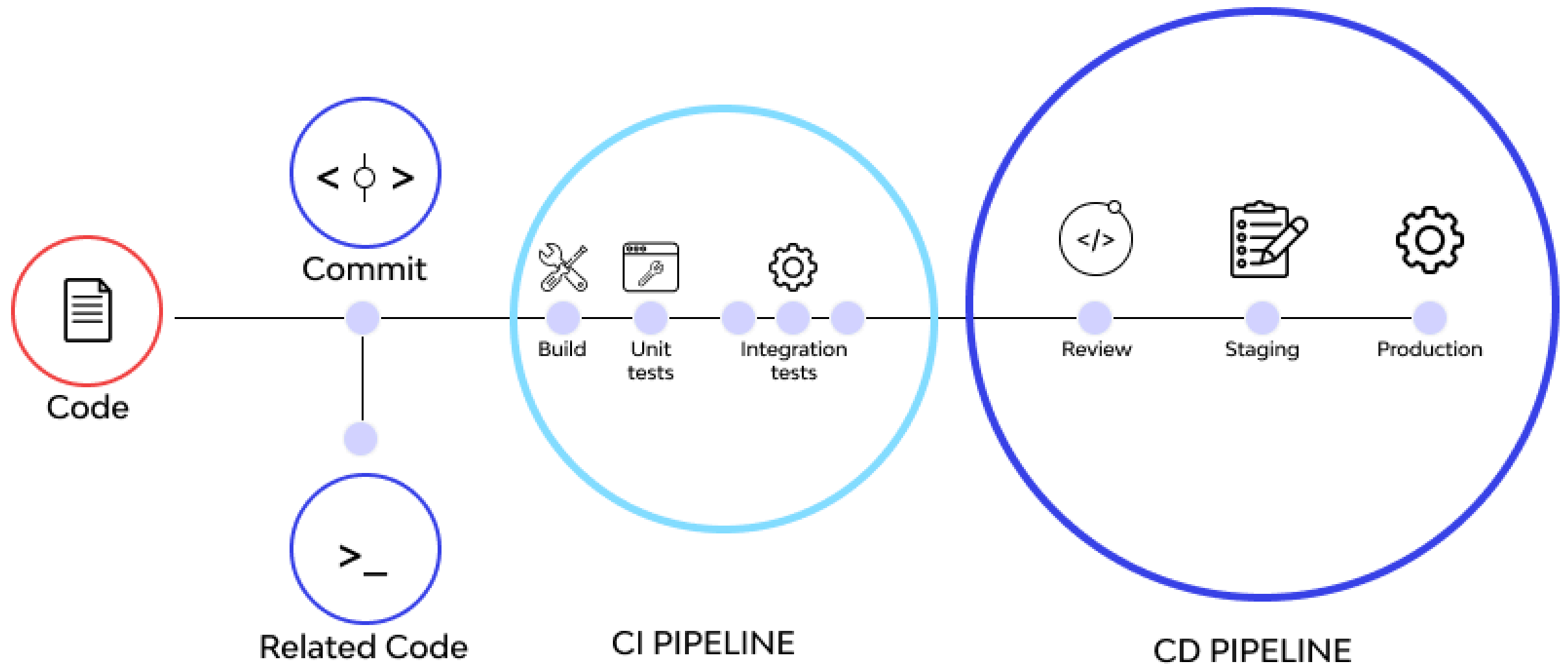


Тема: Внедрение CI/CD с использованием Gitlab

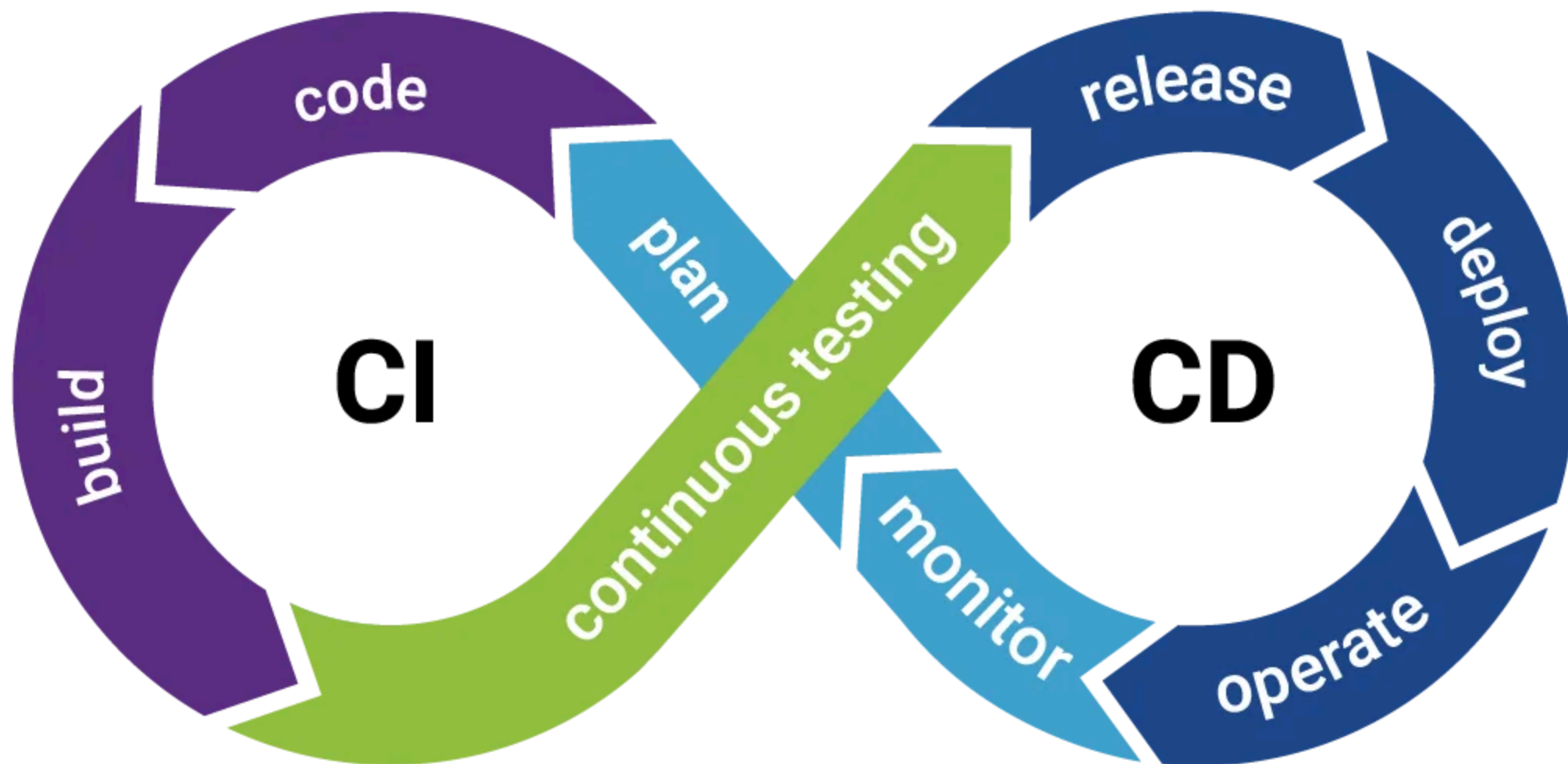
CI/CD pipeline



Основные этапы CI/CD Pipeline

- **Source stage:** Изменения в коде или расписание запускают пайплайн.
- **Build stage:** Сборка приложения и зависимостей.
- **Testing stage:** Автоматизированное тестирование качества и поведения кода.
- **Deploy:** Развертывание проверенного приложения в нужной среде.

Основы непрерывной интеграции и непрерывной доставки



Непрерывная интеграция (CI)

Основные шаги CI:

Коммит изменений : Разработчики коммитают изменения в общий репозиторий.

Автоматическая сборка : Система CI автоматически запускает сборку проекта.

Автоматическое тестирование : После сборки запускаются автоматические тесты для проверки корректности изменений.

Отчет о результатах : Система CI предоставляет отчет о результатах сборки и тестирования.

Непрерывная доставка (CD)

Основные шаги CD:

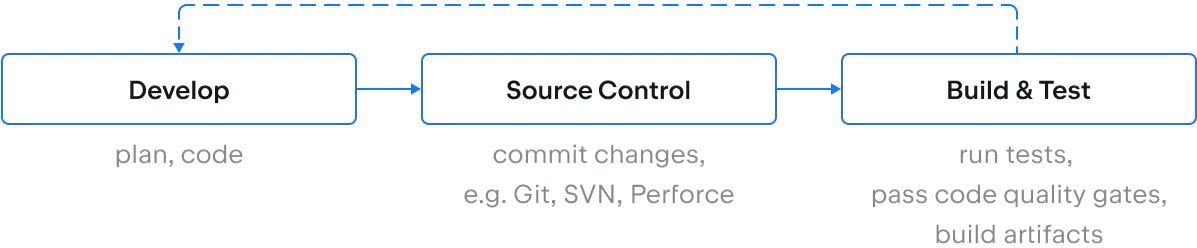
Подготовка артефактов : После успешного прохождения CI, артефакты (например, Docker-образы) подготавливаются для развертывания.

Развертывание в staging : Артефакты автоматически развертываются в staging-среде для дальнейшего тестирования.

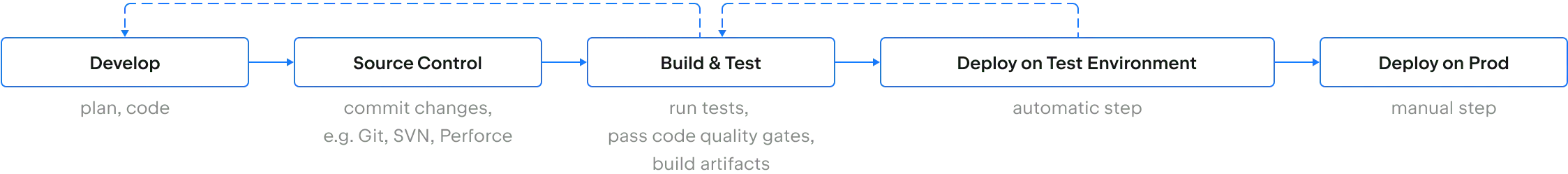
Автоматическое тестирование : В staging-среде запускаются дополнительные тесты (например, интеграционные тесты).

Ручное или автоматическое развертывание в production : После успешного тестирования в staging, изменения могут быть развернуты в production-среде.

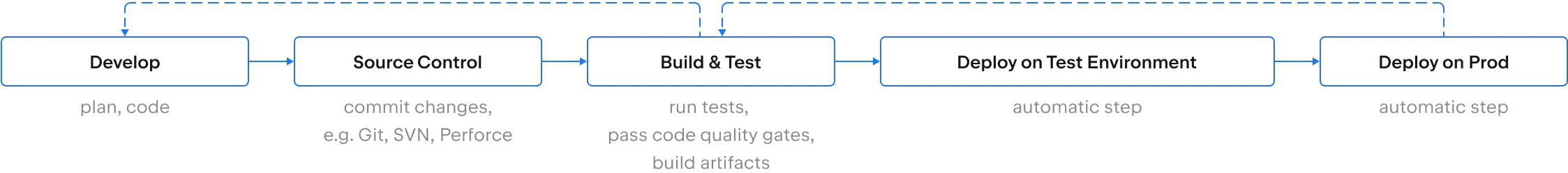
Continuous Integration:



Continuous Delivery:



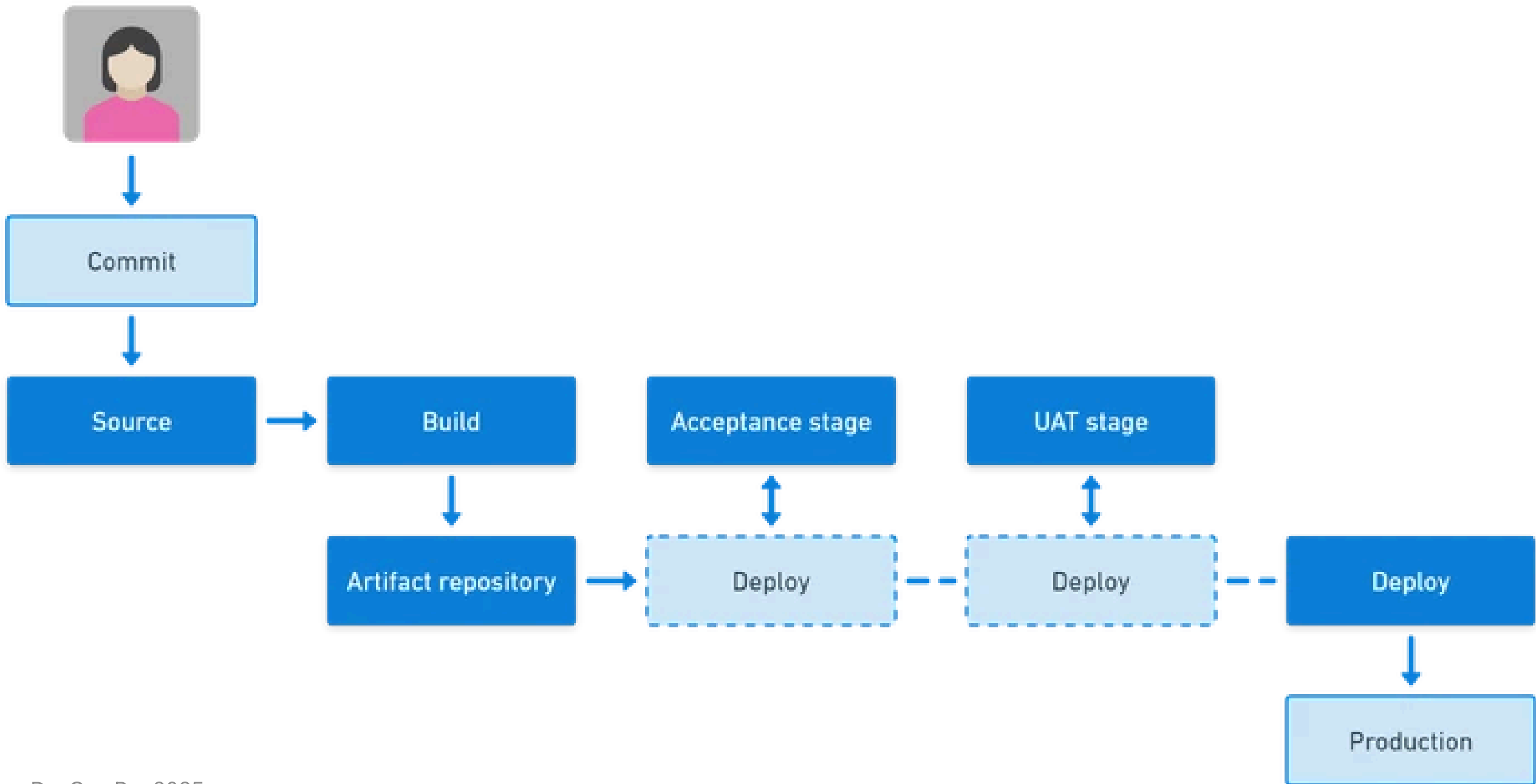
Continuous Deployment:



CI/CD Pipeline

The main high-level stages of a standard CI/CD pipeline:

- **Source stage**
- **Build stage**
- **Testing stage**
- **Deployment stage**



CI/CD pipeline



Stage One – Source

- Pipeline is triggered by changes in the source code repository.
- Can also be triggered by scheduled workflows or other pipelines.
- Initiates the automated CI/CD process.

Stage Two – Build

- Source code and dependencies are compiled or built.
- Failing this stage indicates fundamental configuration issues.
- Only successful builds proceed to testing.

Stage Three – Testing

- Automated tests validate code quality and behavior.
- Duration depends on project size and complexity.
- Problems found here must be resolved before deployment.

Stage Four – Deployment

- Successfully tested software is deployed to target environments.
- Multiple deployment environments (staging, production) are common.
- Enables rapid and reliable delivery for agile teams.

CI/CD Pipeline: Advanced Benefits




- **Efficient development:** Smaller, frequent changes make testing and bug fixing easier.
- **Rapid adaptation:** Quickly implement changes to meet evolving product requirements.
- **Supports experimentation:** Enables safe trials of new ideas and technologies.
- **Improved maintenance:** Continuous flow allows faster bug fixes and more stable releases.
- **Higher code quality:** Early error detection leads to more reliable products.

Potential Drawbacks of CI/CD Pipelines




- **High learning curve**
 - Requires established toolsets and mature environments
 - Teams may need significant training and process adjustments
- **Value depends on project goals**
 - Most beneficial for projects with frequent updates and deployments
 - May not justify the effort for projects with infrequent changes

Potential Drawbacks of CI/CD Pipelines

Tool compatibility and integration complexity

-  Управление несколькими инструментами может быть сложным
-  Выбирайте хорошо интегрированные решения
-  Обеспечьте знакомство команды с выбранными инструментами

Security risks in automated deployments

-  Автоматизация может раскрыть чувствительные данные или привести к уязвимостям
-  Используйте безопасные практики кодирования, шифруйте секреты, внедряйте контроль доступа
-  Регулярно запускайте автоматизированные проверки безопасности

ROI of CI/CD Pipelines

- **Reduces operational costs** by automating repetitive tasks
- **Accelerates time-to-market** with faster, more reliable releases
- **Enhances customer satisfaction** through frequent and stable updates

Best Practices for CI/CD Pipelines

- Commit code frequently
- Automate everything possible
- Use parallel testing
- Monitor and log

Security Considerations for CI/CD Pipelines

- Follow secure coding practices
- Manage secrets securely
- Automate security testing

CI/CD Pipeline Metrics to Track

- **Build Success Rate**
Percentage of successful builds vs. failures
- **Mean Time to Recovery (MTTR)**
Time to recover from failed builds or deployments
- **Deployment Frequency**
How often updates are deployed to production
- **Lead Time for Changes**
Time from code commit to deployment

Rollback Strategies in CI/CD

- **Automated rollback procedures** minimize risks and downtime during deployment errors.
- Tools: Git, Jenkins, Terraform, and others support rollback strategies.
- Common rollback approaches:
 - **Red-Black Deployment**
 - **Canary Releases**
 - **Rolling Back Database Migrations**
 - **Snapshot Rollback**

Red-Black Deployment

- Maintain two production environments: "Red" (current) and "Black" (new).
- Deploy and test the new version in "Black".
- Switch traffic to "Black" after successful testing.
- If issues occur, revert traffic back to "Red".

Canary Releases

- Gradually release the new version to a small subset of users or servers.
- Monitor for issues before full rollout.
- Stop or reverse deployment if problems are detected.

Rolling Back Database Migrations

- Prepare scripts to undo database schema changes.
- Essential for restoring previous application versions safely.

Snapshot Rollback

- Take system or component snapshots before deployment.
- Restore from snapshots to revert to a known good state if needed.

Continuous Testing and Security Integration

- **Continuous Testing:** Automated tests run throughout the development lifecycle
- **Security Integration:** Security checks embedded in every CI/CD phase
- **Key Tools:** Selenium, TestNG, Veracode, SonarQube

What Is GitLab CI/CD?

- **GitLab CI/CD** is a tool for implementing continuous methodologies:
 - Continuous Integration (CI)
 - Continuous Delivery (CD)
 - Continuous Deployment (CD)
- Enables early detection of code errors and bugs in the SDLC
- Ensures code deployed to production meets compliance and coding standards

Gitlab CI pipeline

Available specific runners

- #12706016 (rFy-Y8bD) Remove runner
gitlab-demo-runner-f44dbf46d-d9xjk
x86 openshift
- #12706009 (cXxQHE7K) Remove runner
gitlab-demo-runner-897c78446-clk8c
openshift ppc64le

gitlab.com - Runners connected to the project



GitLab



```

stages:
- step1
- step2

job1:
stage: step1
tags:
- openshift
- ppc64le
image:
name: myimage
script:
- echo "Job Running on Power"
- cd build && make
...

job2:
stage: step2
tags:
- openshift
- amd64
image:
name: myimage
script:
- echo "Job Running on x86"
- cd build && make
...

```

.gitlab-ci.yaml example



runner tags

openshift
x86

runner tags

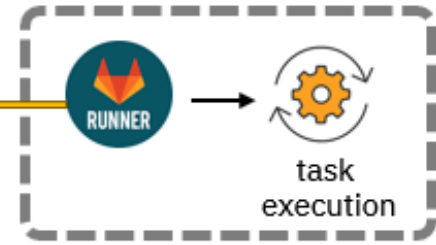
openshift
ppc64le

project



Openshift4 on x86

project



Openshift4 on ppc64le

Runners connect to Gitlab server

Image push

Image push

RED HAT Quay.io EXPLORE REPOSITORIES TUTORIAL

Repositories Account dpkshetty / demos

Repository Tags

TAG	LAST MODIFIED	SECURITY SCORE
gitlab-pytask-mulibarch	an hour ago	New (0 vulnerabilities)
gitlab-pytask-ubi	an hour ago	0 Medium
gitlab-pytask-ppc64le	an hour ago	0 Medium

Quay.io container registry

Пример CD-пайплайна с использованием GitLab CI/CD:

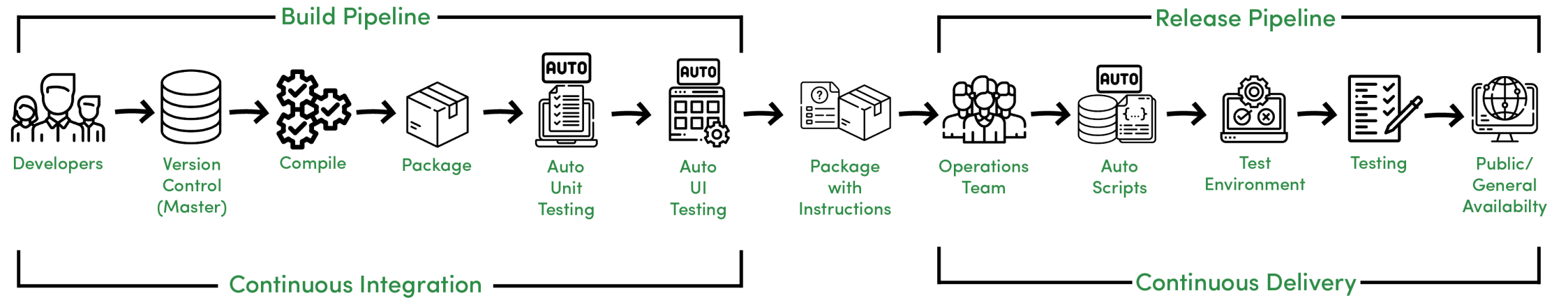
```
stages:
  - build
  - test
  - deploy

build:
  stage: build
  script:
    - docker build -t my-app:latest .
    - docker tag my-app:latest registry.example.com/my-app:latest
    - docker push registry.example.com/my-app:latest

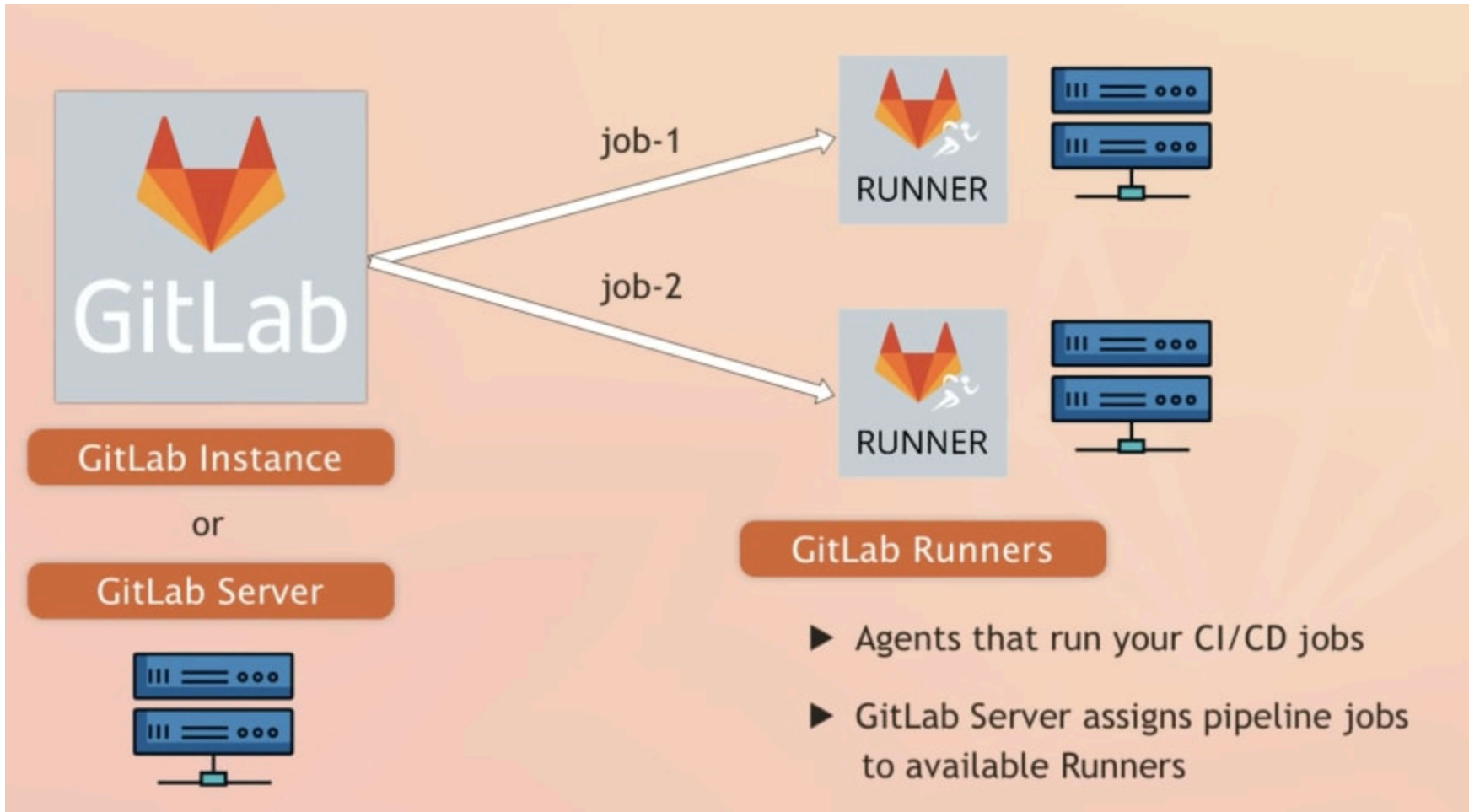
test:
  stage: test
  script:
    - docker run --rm my-app:latest pytest

deploy:
  stage: deploy
  script:
    - kubectl apply -f k8s/deployment.yaml
  environment:
    name: production
    url: https://example.com
  only:
    - main
```

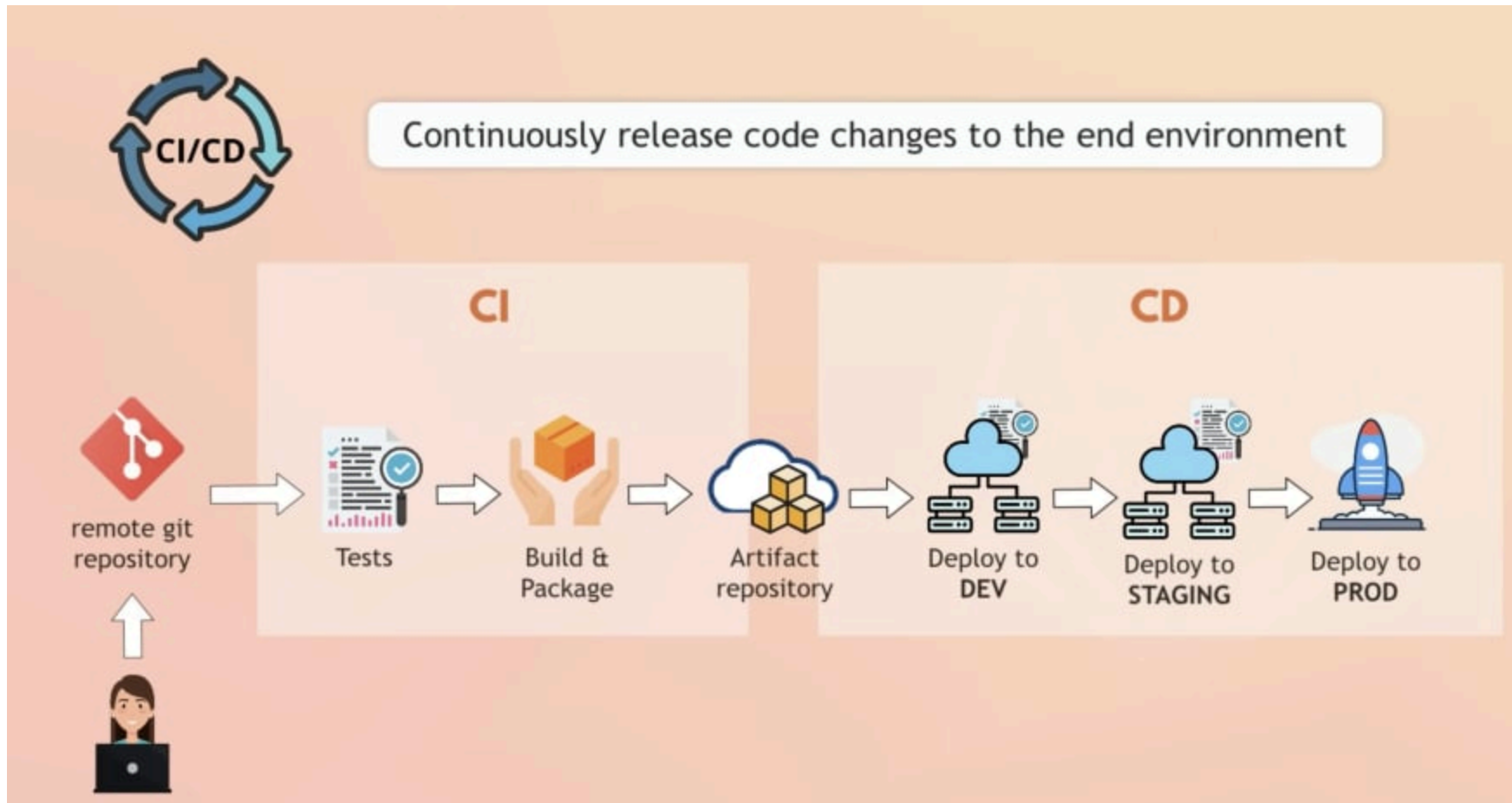
CI/CD

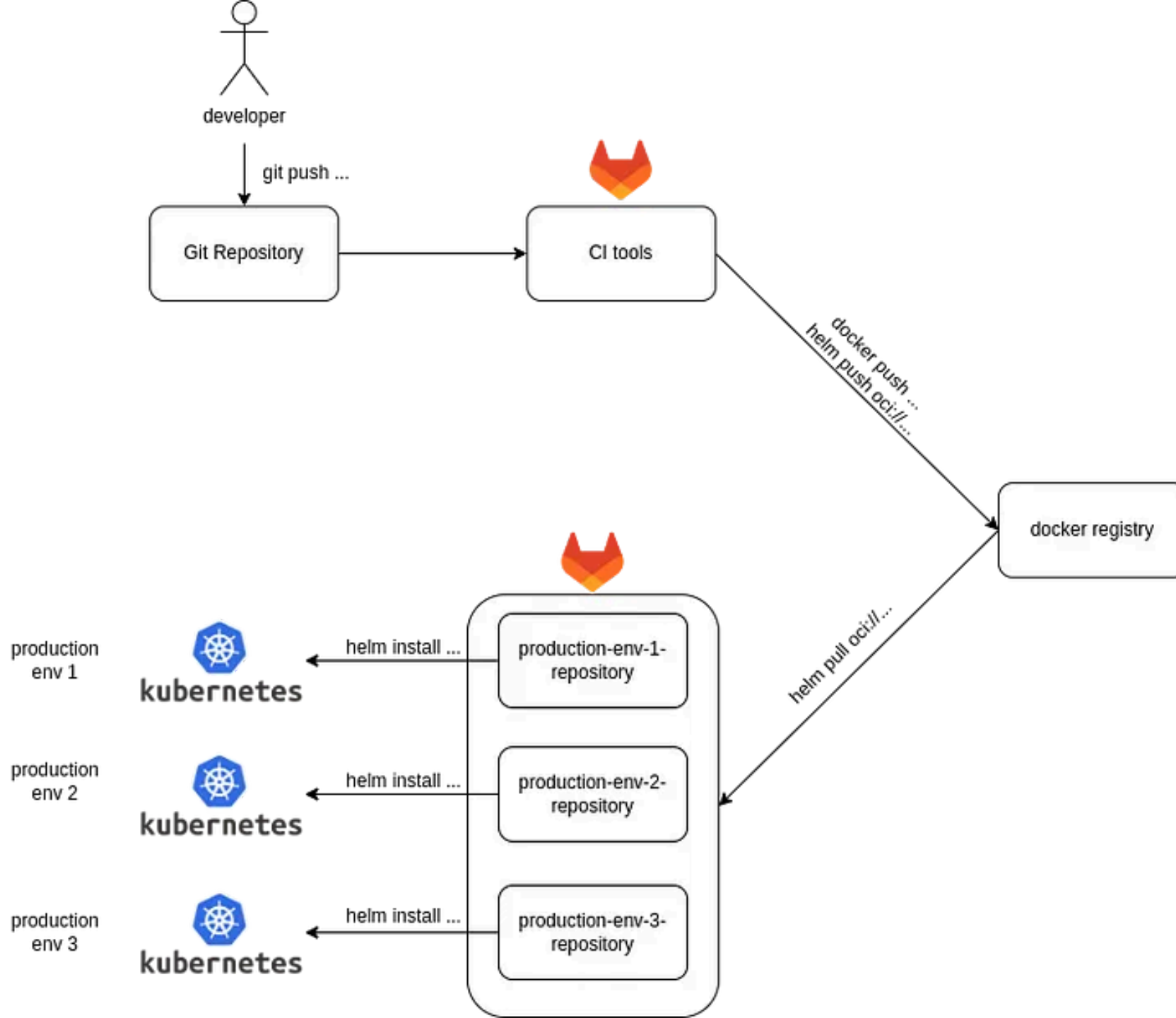


GitLab CI architecture



Giltab CI architecture





GitLab Pipeline Examples with rules

```
stages:
  - build
  - test
  - deploy

build:
  stage: build
  script:
    - echo "Building the project..."
    - ./build_script.sh
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
    - if: '$CI_COMMIT_BRANCH == "develop"'

test:
  stage: test
  script:
    - echo "Running tests..."
    - ./test_script.sh
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
    - if: '$CI_COMMIT_BRANCH == "develop"'

deploy:
  stage: deploy
  script:
    - echo "Deploying the project..."
    - ./deploy_script.sh
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
```

Example with rules and changes

```
stages:
- build
- test
- deploy

build:
  stage: build
  script:
    - echo "Building the project..."
    - ./build_script.sh
  rules:
    - changes:
      - src/**

test:
  stage: test
  script:
    - echo "Running tests..."
    - ./test_script.sh
  rules:
    - changes:
      - tests/**

deploy:
  stage: deploy
  script:
    - echo "Deploying the project..."
    - ./deploy_script.sh
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
    - changes:
      - src/**
```

Example with rules and pipeline Source

```
stages:
- build
- test
- deploy

build:
  stage: build
  script:
    - echo "Building the project..."
    - ./build_script.sh
  rules:
    - if: '$CI_PIPELINE_SOURCE == "push"'
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'

test:
  stage: test
  script:
    - echo "Running tests..."
    - ./test_script.sh
  rules:
    - if: '$CI_PIPELINE_SOURCE == "push"'
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'

deploy:
  stage: deploy
  script:
    - echo "Deploying the project..."
    - ./deploy_script.sh
  rules:
    - if: '$CI_PIPELINE_SOURCE == "schedule"'
```

Example with rules and exists

```
stages:
  - build
  - test
  - deploy

build:
  stage: build
  script:
    - echo "Building the project..."
    - ./build_script.sh
  rules:
    - exists:
      - Dockerfile

test:
  stage: test
  script:
    - echo "Running tests..."
    - ./test_script.sh
  rules:
    - exists:
      - tests/test_script.sh

deploy:
  stage: deploy
  script:
    - echo "Deploying the project..."
    - ./deploy_script.sh
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
    - exists:
      - deploy/deploy_script.sh
```

Создание файла `.gitlab-ci.yml`

```
stages:
  - build
  - test
  - deploy
variables:
  APP_NAME: "my-app"
  APP_VERSION: "1.0.0"
build:
  stage: build
  script:
    - echo "Building the project..."
    - ./build_script.sh
  artifacts:
    paths:
      - build/
test:
  stage: test
  script:
    - echo "Running tests..."
    - ./test_script.sh
  dependencies:
    - build
```


Стадии (Stages)

```
stages:  
  - build  
  - test  
  - deploy
```

Задачи (jobs)

Задача сборки (build)

```
build:  
  stage: build  
  script:  
    - echo "Building the project..."  
    - ./build_script.sh  
  artifacts:  
    paths:  
      - build/
```

Задача тестирования (test)

Задача тестирования выполняется на стадии `test` и включает команды для запуска тестов. Она также зависит от артефактов, созданных на стадии `build`.

```
test:  
  stage: test  
  script:  
    - echo "Running tests..."  
    - ./test_script.sh  
  dependencies:  
    - build
```

Зависимости

Зависимости (dependencies) позволяют задачам использовать артефакты, созданные в предыдущих задачах.

```
dependencies:  
  - build
```

Задача развертывания (deploy)

Задача развертывания выполняется на стадии `deploy` и включает команды для развертывания проекта. Она выполняется только на ветке `main`.

```
deploy:
  stage: deploy
  script:
    - echo "Deploying the project..."
    - ./deploy_script.sh
  only:
    - main
```

Триггеры (triggers)

```
deploy:  
  stage: deploy  
  script:  
    - echo "Deploying the project..."  
    - ./deploy_script.sh  
  only:  
    - main
```

Переменные окружения

Вы можете определить переменные окружения, которые будут использоваться в задачах.

```
variables:  
  APP_NAME: "my-app"  
  APP_VERSION: "1.0.0"
```

Артефакты

Артефакты (artifacts) позволяют сохранять файлы, созданные в одной задаче, для использования в последующих задачах.

```
artifacts:  
  paths:  
    - build/
```


YAML anchors

```
.job_template: &job_configuration # Hidden yaml configuration that defines an anchor named 'job_configuration'
  image: ruby:2.6
  services:
    - postgres
    - redis

test1:
  <<: *job_configuration          # Add the contents of the 'job_configuration' alias
  script:
    - test1 project

test2:
  <<: *job_configuration          # Add the contents of the 'job_configuration' alias
  script:
    - test2 project
```

extends

```
.tests:
  rules:
    - if: $CI_PIPELINE_SOURCE == "push"

.rspec:
  extends: .tests
  script: rake rspec

rspec 1:
  variables:
    RSPEC_SUITE: '1'
  extends: .rspec

rspec 2:
  variables:
    RSPEC_SUITE: '2'
  extends: .rspec

spinach:
  extends: .tests
  script: rake spinach
```

trigger:include

Example of trigger:include:

```
trigger-child-pipeline:  
  trigger:  
    include: path/to/child-pipeline.gitlab-ci.yml
```

trigger:project

Example of trigger:project:

```
trigger-multi-project-pipeline:  
  trigger:  
    project: my-group/my-project
```

Example of trigger:project for a different branch:

```
trigger-multi-project-pipeline:  
  trigger:  
    project: my-group/my-project  
    branch: development
```

trigger:strategy

Example of trigger:strategy:

```
trigger_job:  
  trigger:  
    include: path/to/child-pipeline.yml  
    strategy: depend
```

trigger:forward

```
variables: # default variables for each job
  VAR: value

# Default behavior:
# - VAR is passed to the child
# - MYVAR is not passed to the child
child1:
  trigger:
    include: .child-pipeline.yml

# Forward pipeline variables:
# - VAR is passed to the child
# - MYVAR is passed to the child
child2:
  trigger:
    include: .child-pipeline.yml
    forward:
      pipeline_variables: true

# Do not forward YAML variables:
# - VAR is not passed to the child
# - MYVAR is not passed to the child
child3:
  trigger:
    include: .child-pipeline.yml
    forward:
      yaml_variables: false
```

Parallel Matrix Jobs in GitLab CI/CD

```
stages:
  - test

variables:
  NODE_ENV: test

test:
  stage: test
  script:
    - echo "Running tests with Node.js version $NODE_VERSION"
    - nvm install $NODE_VERSION
    - nvm use $NODE_VERSION
    - npm install
    - npm test
  parallel:
    matrix:
      - NODE_VERSION: ["12", "14", "16"]
```

include:local

Example of include:local:

```
include:  
  - local: '/templates/.gitlab-ci-template.yml'
```

You can also use shorter syntax to define the path:

```
include: '.gitlab-ci-production.yml'
```


include:project

Example of include:project:

```
include:  
- project: 'my-group/my-project'  
  file: '/templates/.gitlab-ci-template.yml'  
- project: 'my-group/my-subgroup/my-project-2'  
  file:  
    - '/templates/.builds.yml'  
    - '/templates/.tests.yml'
```

include:project

You can also specify a ref:

```
include:
  - project: 'my-group/my-project'
    ref: main                                     # Git branch
    file: '/templates/.gitlab-ci-template.yml'
  - project: 'my-group/my-project'
    ref: v1.0.0                                   # Git Tag
    file: '/templates/.gitlab-ci-template.yml'
  - project: 'my-group/my-project'
    ref: 787123b47f14b552955ca2786bc9542ae66fee5b # Git SHA
    file: '/templates/.gitlab-ci-template.yml'
```

include:remote

Example of include:remote:

```
include:  
  - remote: 'https://gitlab.com/example-project/-/raw/main/.gitlab-ci.yml'
```

Content of a file named /templates/.before-script-template.yml:

```
default:
  before_script:
    - apt-get update -qq && apt-get install -y -qq sqlite3 libsqlite3-dev nodejs
    - gem install bundler --no-document
    - bundle install --jobs $(nproc) "${FLAGS[@]}"
```

Content of .gitlab-ci.yml:

```
include: 'templates/.before-script-template.yml'

rspec1:
  script:
    - bundle exec rspec

rspec2:
  script:
    - bundle exec rspec
```

GitLab Cache vs Artifacts

Cache

Key Characteristics:

- **Scope:** The cache is shared between different jobs and pipeline runs.
- **Expiration:** You can set an expiration time for the cache, after which it will be invalidated.
- **Key:** The cache can be keyed by a specific string, allowing you to create different caches for different branches or jobs.
- **Performance:** Using the cache can significantly reduce the time required for tasks like dependency installation.

GitLab Cache vs Artifacts

Example Usage:

```
cache:  
  key: ${CI_COMMIT_REF_SLUG}  
  paths:  
    - node_modules/  
    - .m2/repository/
```

GitLab Cache vs Artifacts

Artifacts

Key Characteristics:

- **Scope:** Artifacts are specific to a single pipeline run and are not shared between different pipeline runs.
- **Expiration:** You can set an expiration time for artifacts, after which they will be deleted.
- **Dependencies:** Artifacts can be used as dependencies for subsequent jobs, ensuring that the necessary files are available for those jobs.
- **Downloadable:** Artifacts can be downloaded from the GitLab web interface for inspection or debugging.

GitLab Cache vs Artifacts

Example Usage:

```
build:
  stage: build
  script:
    - npm install
    - npm run build
  artifacts:
    paths:
      - dist/
    expire_in: 1 week

test:
  stage: test
  script:
    - npm test
  dependencies:
    - build
```


Key Differences

1. Scope:

- **Cache:** Shared across multiple pipeline runs and jobs.
- **Artifacts:** Specific to a single pipeline run and used to pass files between jobs in the same pipeline.

2. Purpose:

- **Cache:** Used to speed up the build process by storing dependencies or other reusable files.
- **Artifacts:** Used to store and pass files generated by a job to subsequent jobs in the same pipeline.

Key Differences

3. Expiration:

- **Cache:** Can be configured to expire after a certain period, but is generally intended to persist across multiple pipeline runs.
- **Artifacts:** Can be configured to expire after a certain period, but are generally intended to be short-lived and specific to a single pipeline run.

4. Key:

- **Cache:** Can be keyed by a specific string to create different caches for different branches or jobs.
- **Artifacts:** Do not use keys; they are tied to the specific pipeline run and job that generated them.

When to Use Cache vs Artifacts

- **Use Cache:**

- When you have dependencies or files that are reused across multiple pipeline runs.
- To speed up tasks like dependency installation by caching the results.

- **Use Artifacts:**

- When you need to pass files generated by one job to subsequent jobs in the same pipeline.
- To store build outputs, test results, or other files that are specific to a single pipeline run.

GitLab CI/CD Best Practices

- Use caches and artifacts effectively
- Read and share documentation
- Optimize pipeline stages
- Leverage failures for improvement
- Test environments should mirror production

Use Caches and Artifacts Effectively

- **Cache:** Save/download files once, reuse in subsequent jobs for faster execution
- **Artifacts:** Pass files between jobs within the same pipeline
- **Tag runners** and use consistent tags for jobs sharing caches
- **Use unique cache keys** (e.g., per branch) for workflow isolation

Read the Documentation

- **GitLab documentation is extensive and regularly updated**
- **Encourage team members to review docs and FAQs**
- **Bookmark important sections for quick reference**

Optimize Pipeline Stages

- Organize jobs to make failures easy to identify and fix
- Run jobs as early as possible if safe
- Avoid grouping similar jobs into stages if not optimal

Use Failures to Improve Processes

- Foster a culture of learning, not blame
- Investigate root causes of failures
- Use `allow_failure` for non-critical jobs
- Frequent commits make issues easier to trace and fix

Test Environments Should Mirror Production

- Match test and production environments for reliable results
- Use the "test pyramid": more unit tests, fewer end-to-end tests
- Leverage GitLab Review Apps for production-like previews

Практика

LAB:

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/gitlab_ci_lab_helm.md

Практика

LAB:

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/gitlab_ci_lab_basic_trivy_scan.md

Практика

LAB:

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/gitlab_ci_lab_tf.md