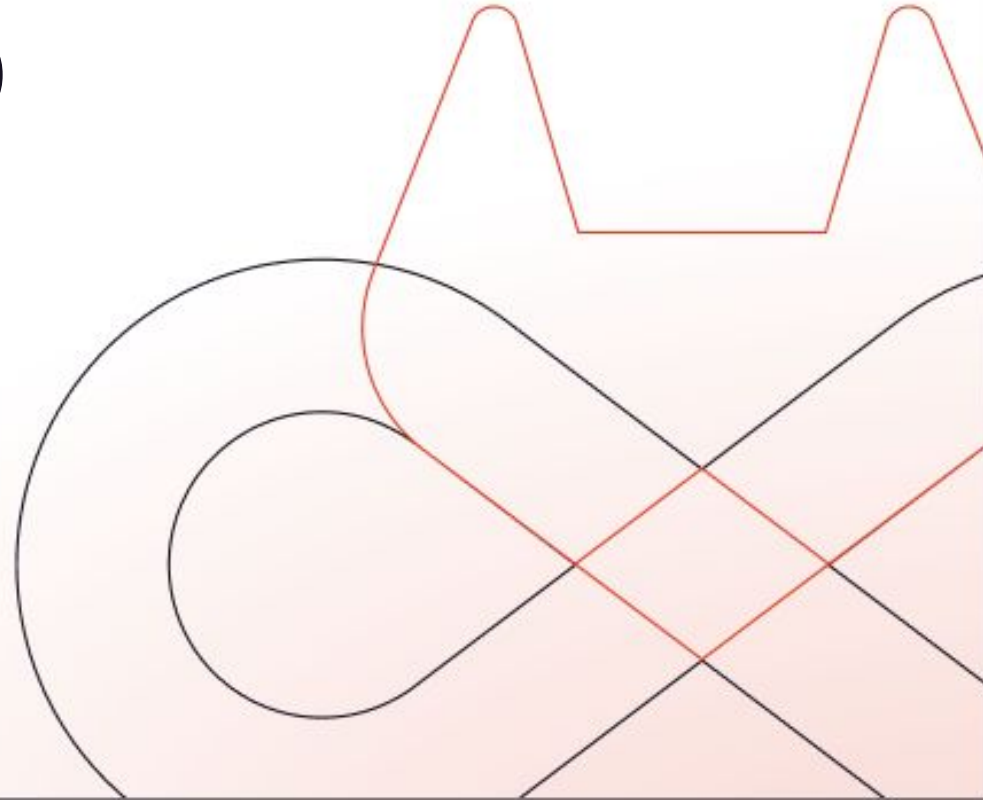
GitLab

Advanced GitLab CI/CD



Agenda

1. **CI/CD Basics: a quick review**
2. **Pipeline structure and ways to hack it**
   Pipeline Architectures
3. **Variables: a foundation before using rules**
4. **Controlling when your jobs run**
   The “rules” label
5. **Holding, securing, and sharing results**
   Job Artifacts
6. **Assembling pipelines from components**
   Using “includes” and “extends” labels

This workshop assumes that participants are familiar with basic GitLab CI/CD.

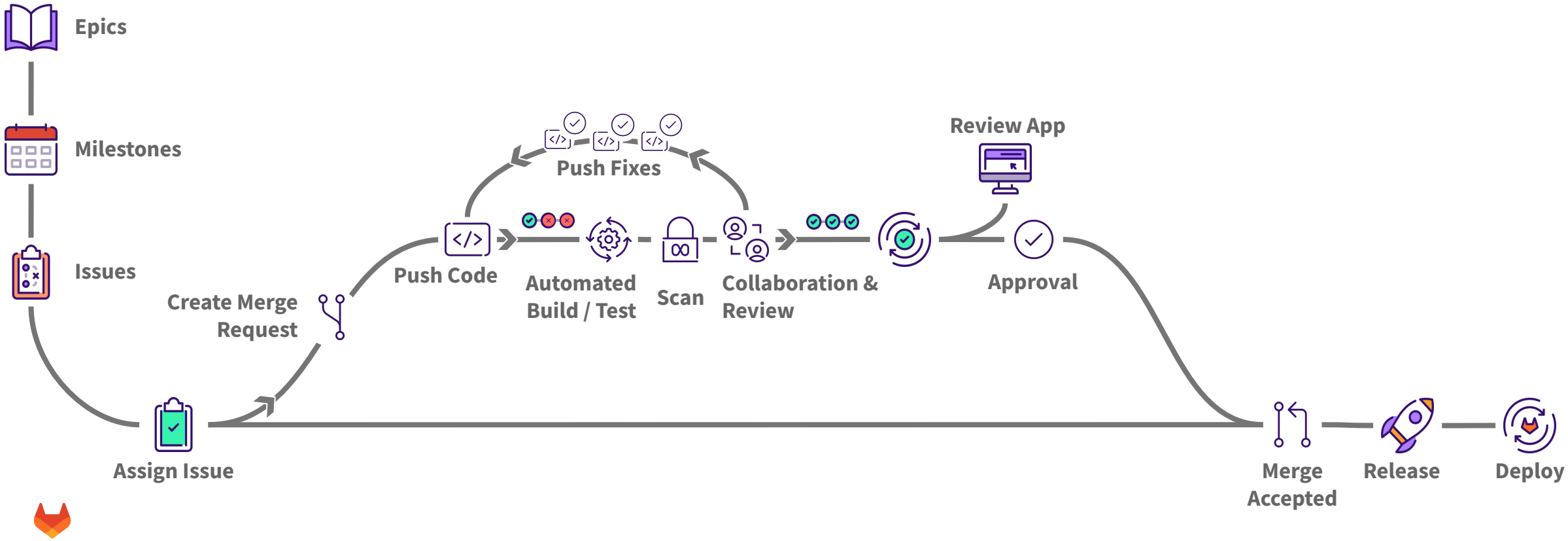
It presents advanced concepts and techniques for people who are active users of GitLab CI.

We do not cover Runner configuration and any of the CD-specific features such as Environments and Deployments.

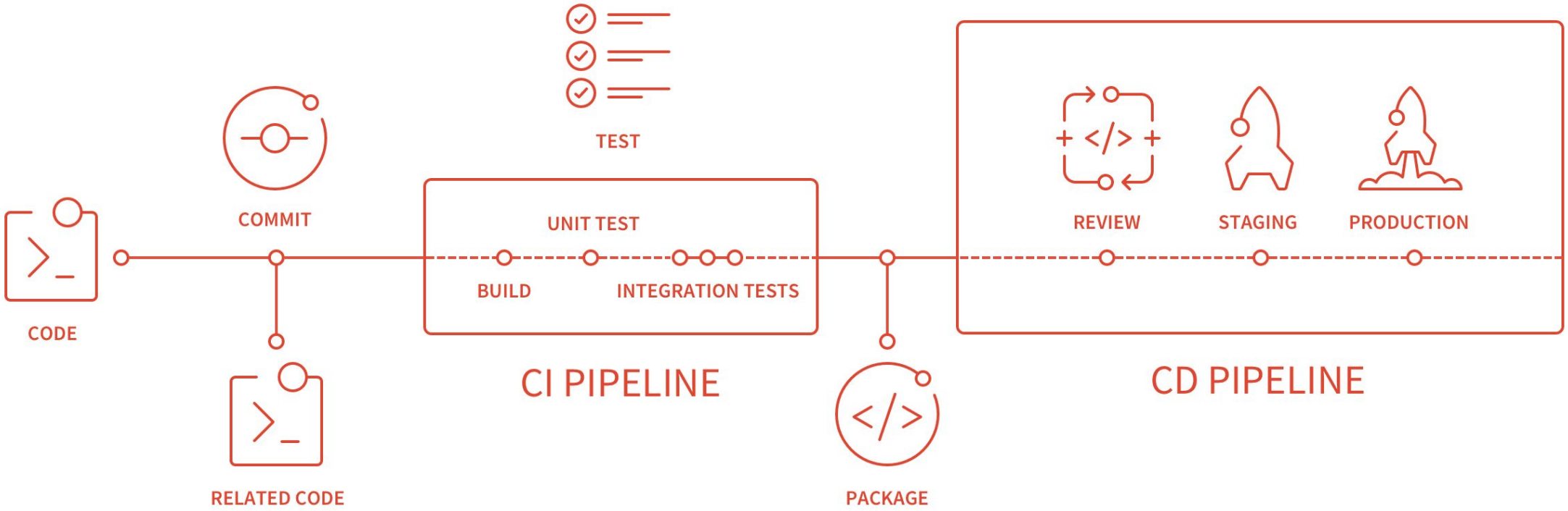
GitLab

GitLab CI/CD Basics Review

GitLab Recommended Process



GitLab CI/CD


TEST
COMMIT
UNIT TEST
REVIEW
STAGING
PRODUCTION
CODE
BUILD
INTEGRATION TESTS
CI PIPELINE
CD PIPELINE
RELATED CODE
PACKAGE

Anatomy of a GitLab CI/CD build

Pipeline

- Set of one or more jobs. Optionally organized into stages

Stages

- Collection of jobs to be run in parallel
- e.g. Build, Test, Deploy

Jobs

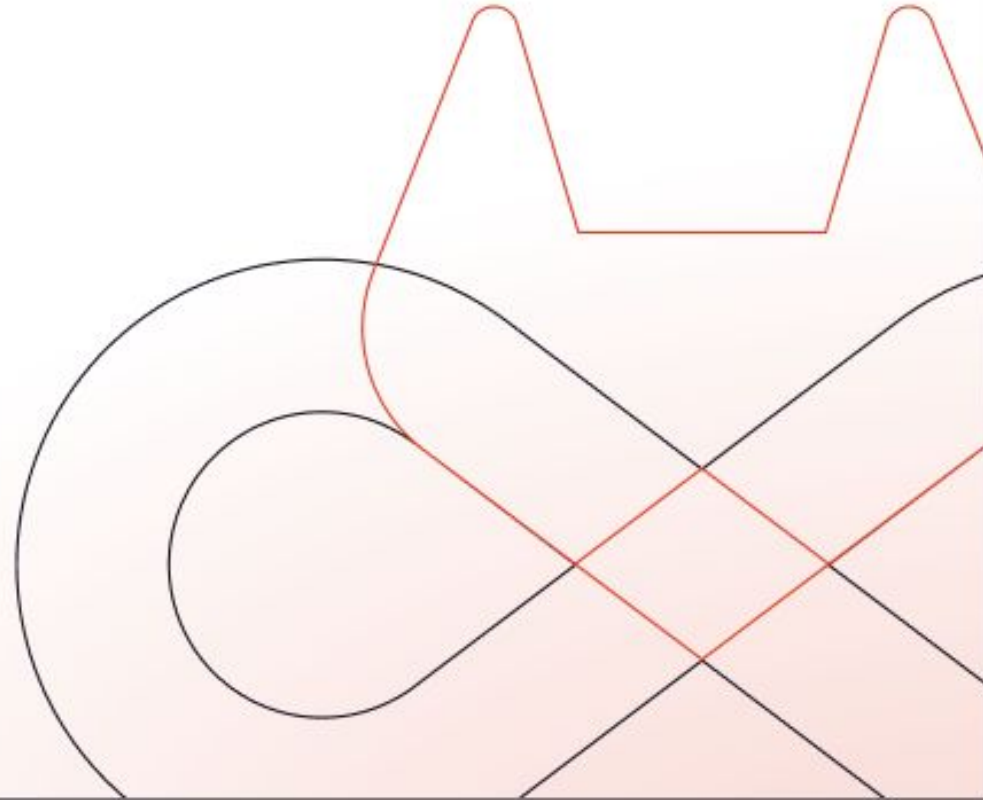
- Scripts that perform tasks
- e.g. npm test; mvn install; etc.

Environments

- Where we deploy (Test, Review, Staging, Canary, Prod)

GitLab

Pipeline Architectures



Pipeline Architectures

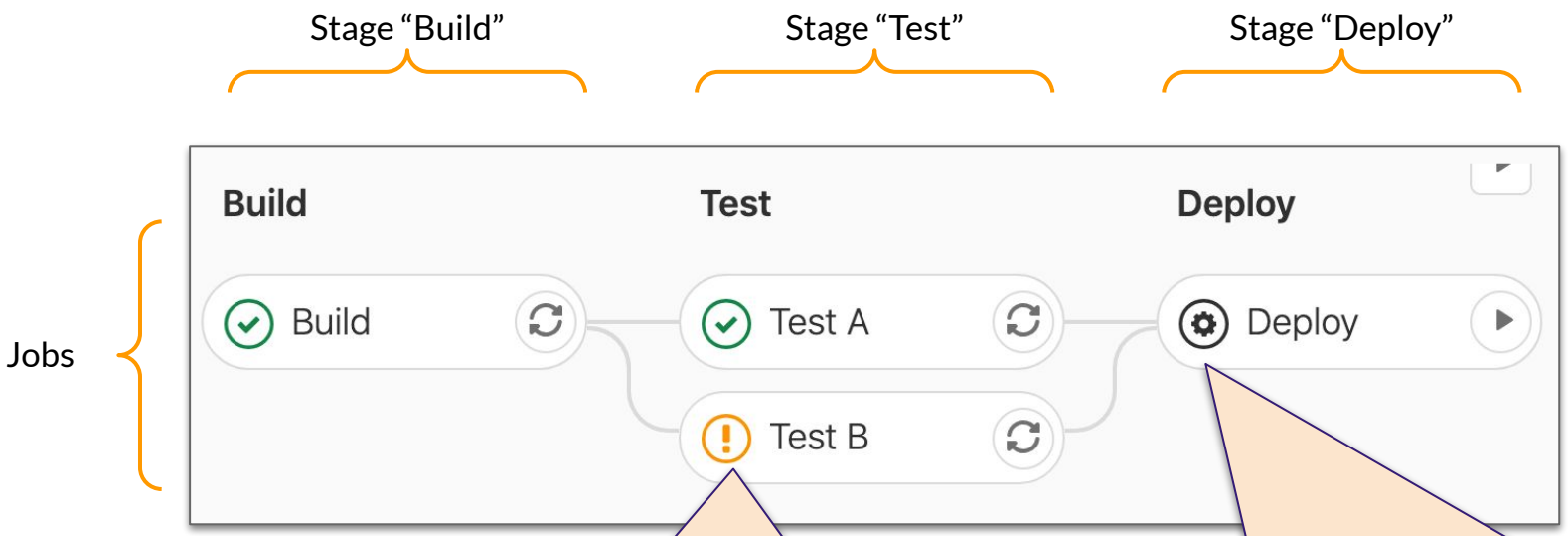
Docs

- The building-blocks of your pipeline.
  - Basic
  - Directed Acyclic Graph
  - Parent/Child Pipelines
  - Dynamic Child Pipelines
  - Multi-Project Pipelines

Basic pipelines

- Simplest pipeline in GitLab
- Jobs run independently, sometimes on different runners
- All Jobs in a Stage must complete successfully before proceeding to the next stage
- Control pipeline with pipeline options

Pipeline Architectures - Basic pipeline with options



Other options:

```
when: delayed
when: on_failure
when: always
```

This job has

```
allow_failure: true
```

...so the pipeline proceeds even though it failed

This job has

```
when: manual
```

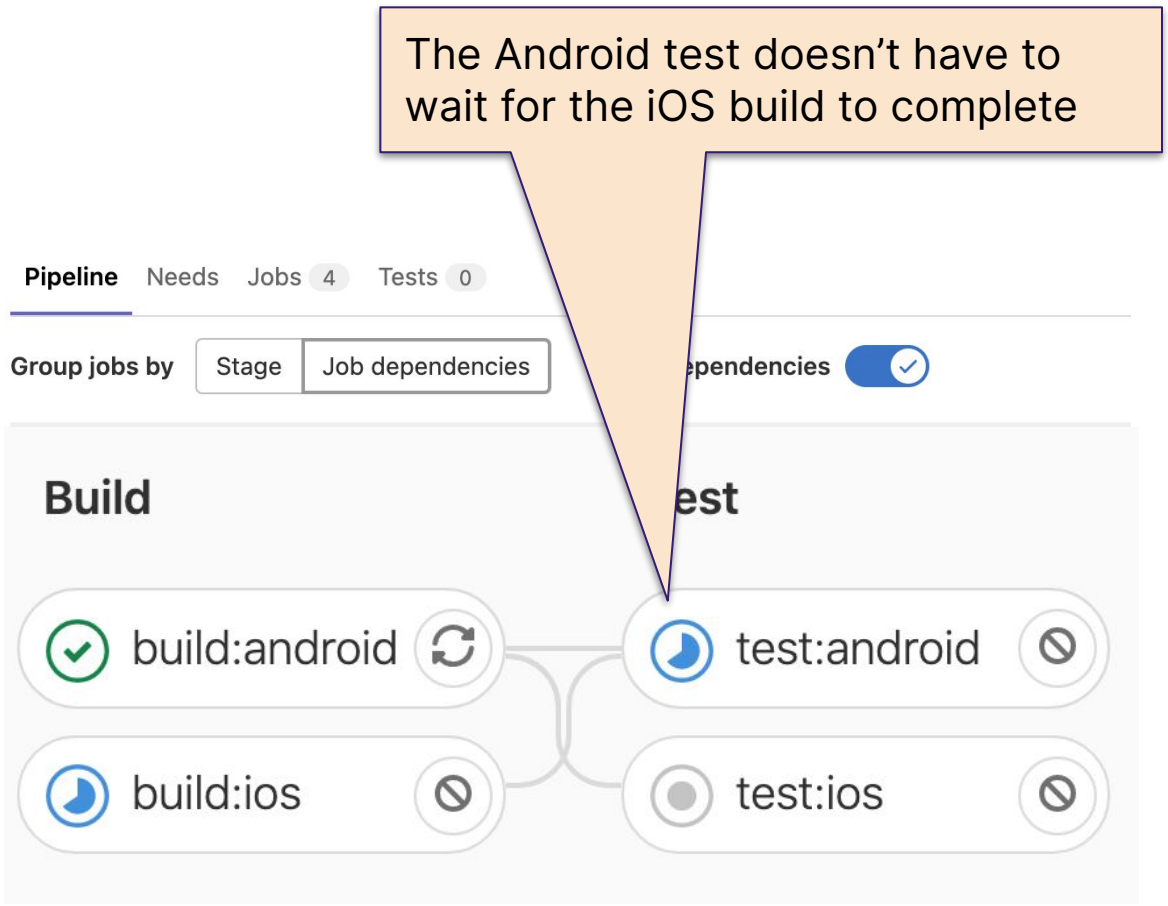
...so it waits for someone (with the right permissions) to click a “Play” button

Needs

or Directed Acyclic Graph (DAG)

- Define job dependencies to optimize pipeline flow
- Jobs still run independently, sometimes on different runners
- Dependent jobs can proceed to next stage without waiting for other jobs in stage to finish

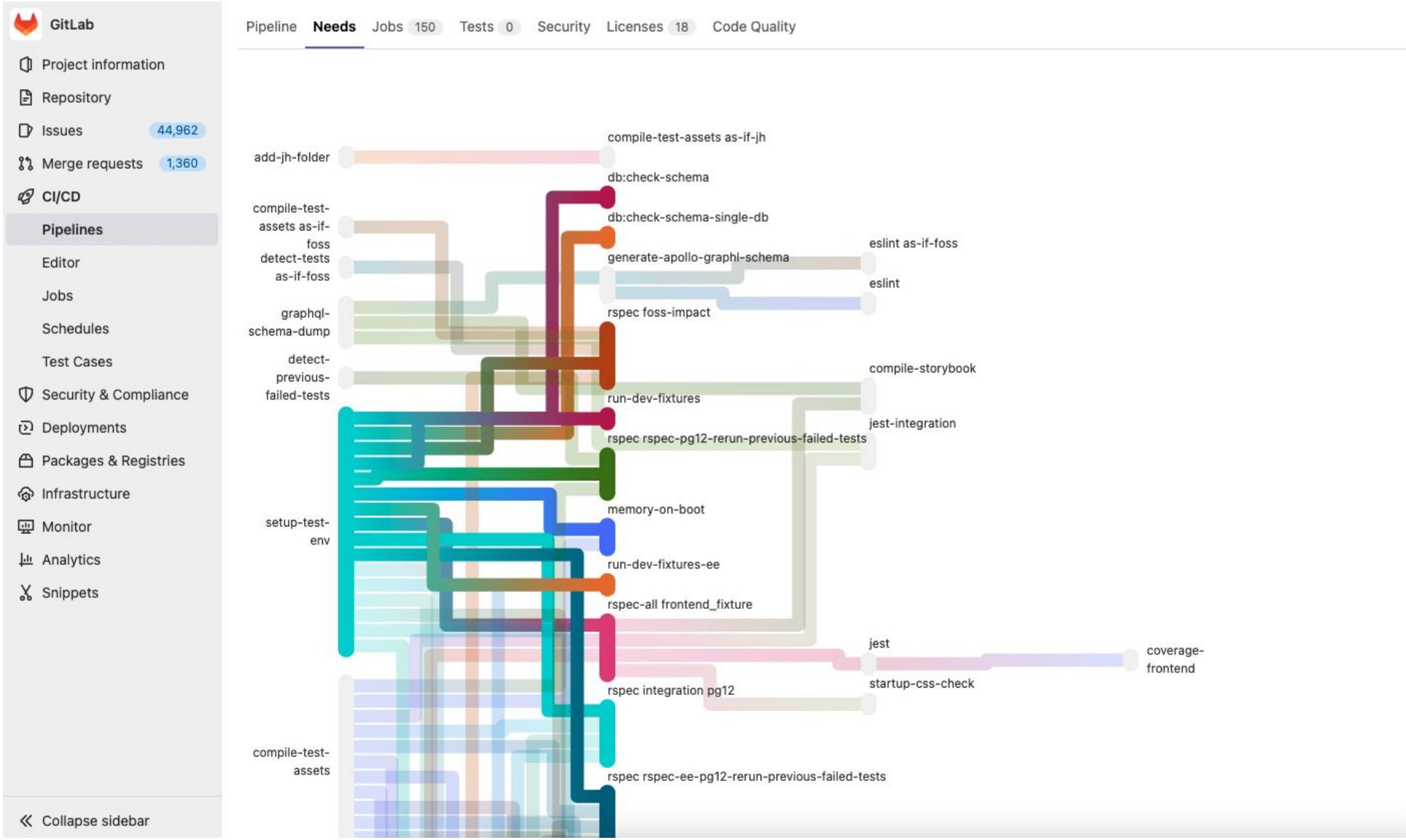
Pipeline Architectures - Directed Acyclic Graph (DAG)




```yaml
build:ios:
  stage: build
  script:
    - echo "Building for iOS"
    - sleep 10

build:android:
  stage: build
  script:
    - echo "Building for Android"

test:ios:
  stage: test
  needs: ["build:ios"]
  script:
    - echo "Testing for iOS"

test:android:
  stage: test
  needs: ["build:android"]
  script:
    - echo "Testing for Android"
```

Jobs still runs in stages, but the "needs" keyword overrides the need for the previous stage to complete entirely.

Directed Acyclic Graph Pipelines



Example: needs

```yaml
linux:build:
  stage: build
  script: echo "Building linux..."

mac:build:
  stage: build
  script: echo "Building mac..."

lint:
  stage: test
  needs: []
  script: echo "Linting..."

linux:rspec:
  stage: test
  needs: ["linux:build"]
  script: echo "Running rspec on linux..."

mac:rspec:
  stage: test
  needs: ["mac:build"]
  script: echo "Running rspec on mac..."

production:
  stage: deploy
  script: echo "Running production..."
```

Linux path: the `linux:rspec` job will be run as soon as the `linux:build` job finishes without waiting for `mac:build` to finish.

macOS path: the `mac:rspec` job will be run as soon as the `mac:build` job finishes without waiting for `linux:build` to finish.

The `lint` job will run immediately without waiting for `build` jobs.

The `production` job runs as soon as all previous jobs finish.

Source: https://docs.gitlab.com/ee/ci/yaml/#needs

Additional example: needs with artifacts

```yaml
test-job1:
  stage: test
  needs:
    - job: build_job1
      artifacts: true

test-job2:
  stage: test
  needs:
    - job: build_job2
      artifacts: false

test-job3:
  needs:
    - job: build_job1
      artifacts: true
    - job: build_job2
    - build_job3
```

The `test-job1` job downloads the `build_job1` artifacts

The `test-job2` job does not download the `build_job2` artifacts.

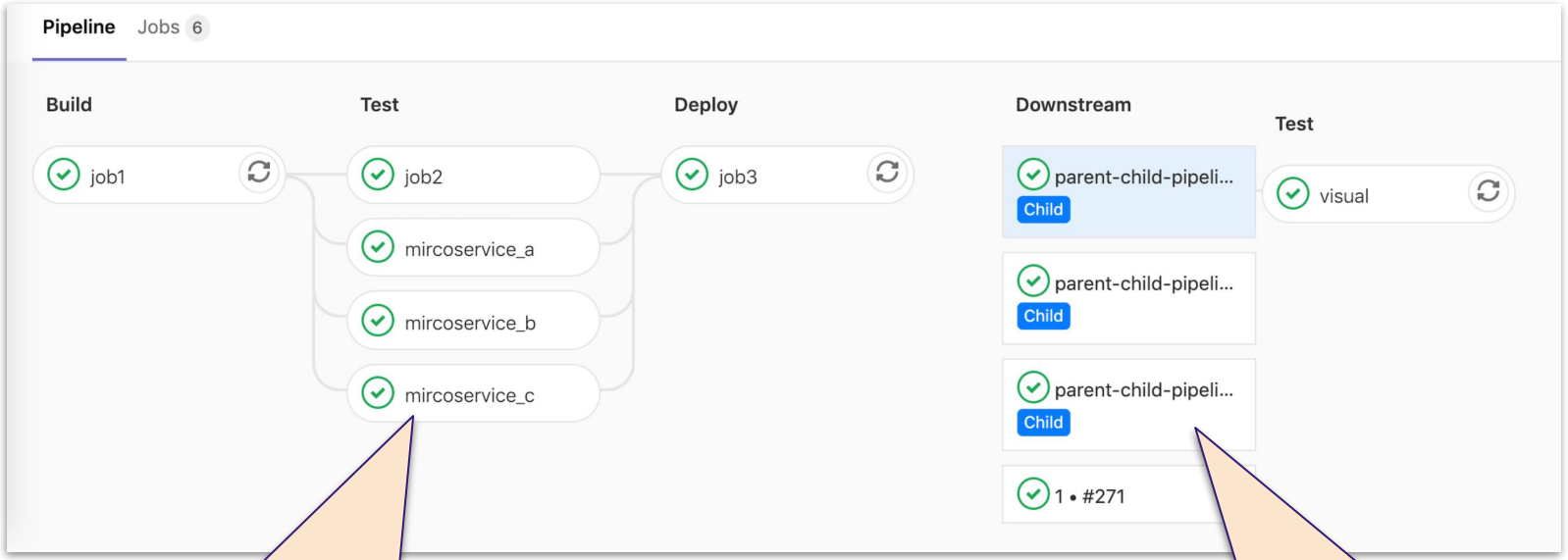
The `test-job3` job downloads the artifacts from all three `build_jobs`, because `artifacts` is `true`, or defaults to `true`, for all three needed jobs.

Source: https://docs.gitlab.com/ee/ci/yaml/#needsartifacts

Parent/Child Pipelines

- Run child pipelines independent from each other.
- Separates entire pipeline configuration in multiple files to keep things simple.
- Combine with DAG pipelines to achieve benefits of both.
- Useful to branch out long running tasks into separate pipelines.

Pipeline Architectures - Parent/Child Pipelines



The pipeline you're viewing

Downstream pipelines that are triggered from within this one.

Pipeline Architectures - Parent/Child Pipelines for Monorepos

```yaml
stages:
  - build
  - finalize

build-1:
  stage: build
  only:
    changes:
      - project-1/*
  trigger:
    include: project-1/.gitlab-ci.yml
    strategy: depend
```

Run the job if there are changes in those files

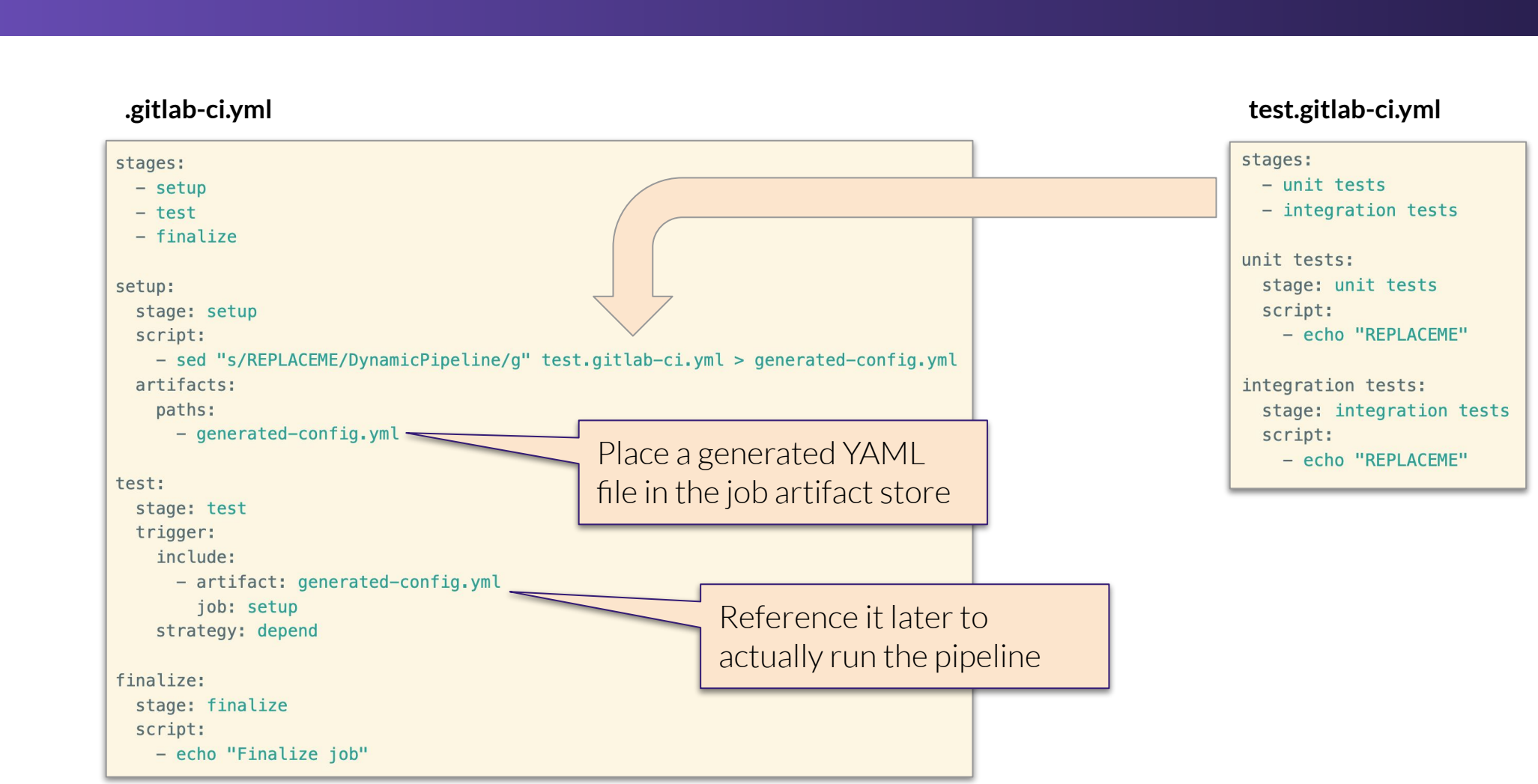
Include files from elsewhere in the project - must reference a YAML file

`strategy: depend` means to hold this pipeline until the other pipeline finishes

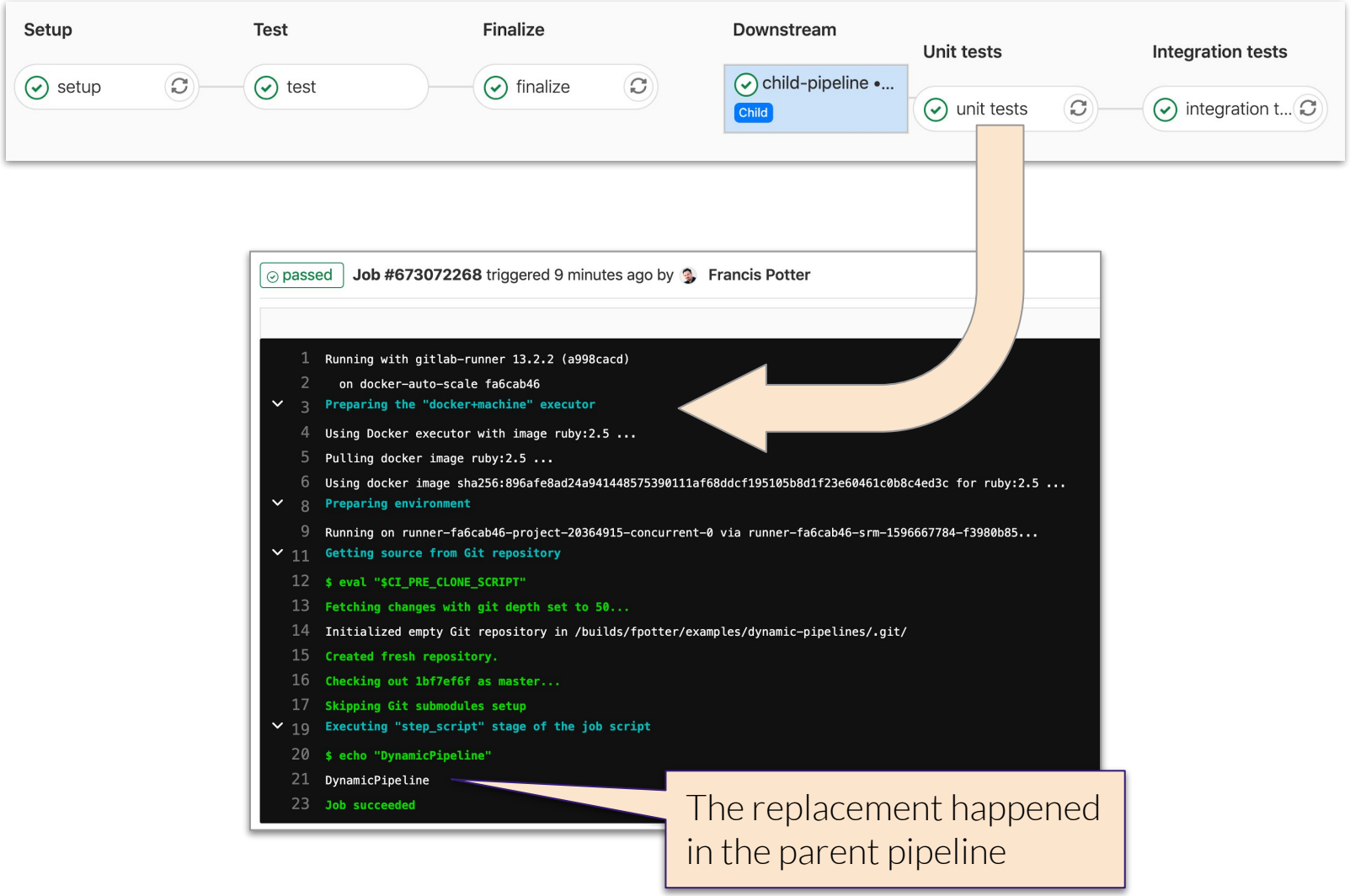
Dynamic Pipelines

- Generate pipeline configuration at build time
- Use the generated configuration at a later stage to run as a child pipeline.
- Useful to use a single pipeline configuration with different settings to support a matrix of targets and architectures

Pipeline Architectures - Dynamic Pipelines



Pipeline Architectures - Dynamic Pipelines

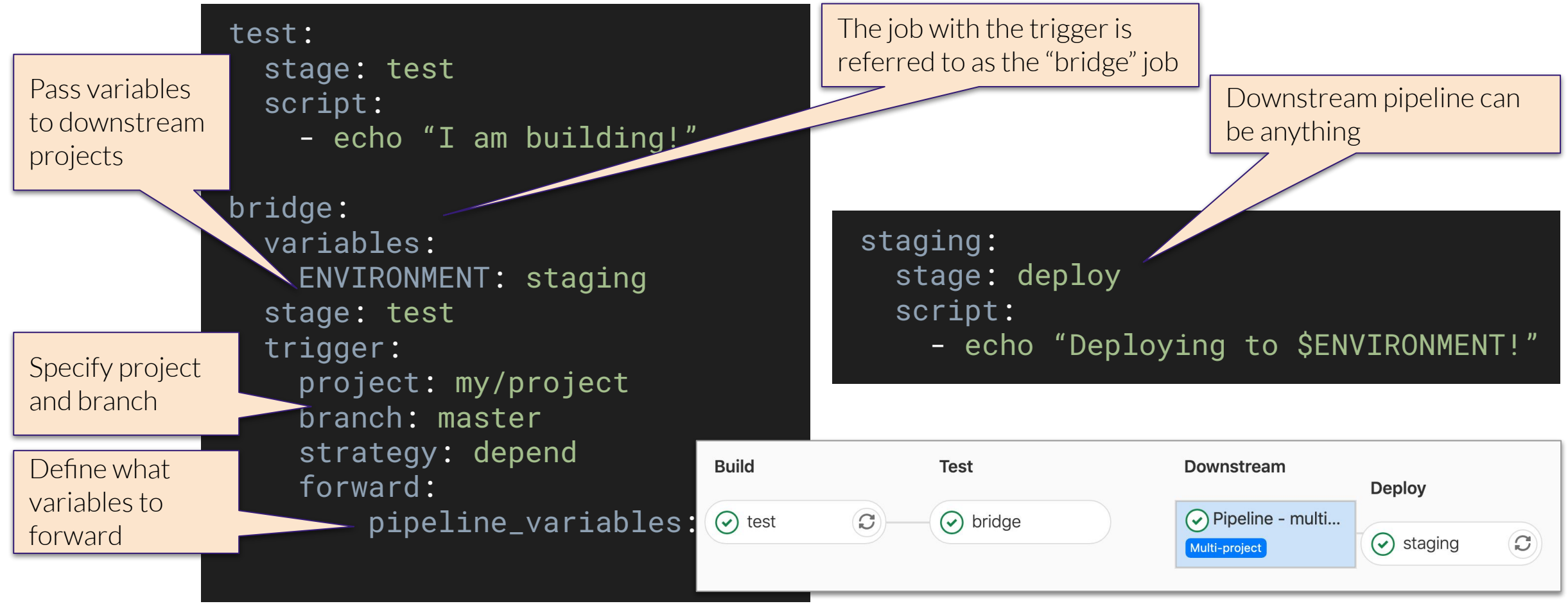


Multi-project Pipelines

- A pipeline in one project can trigger a pipeline in another project
- You can specify a specific branch
- You can pass variables to downstream pipelines
- If downstream pipeline fails it will not fail upstream pipeline
- Useful when building / deploying large applications that are made up of different components that have their own project and build pipeline

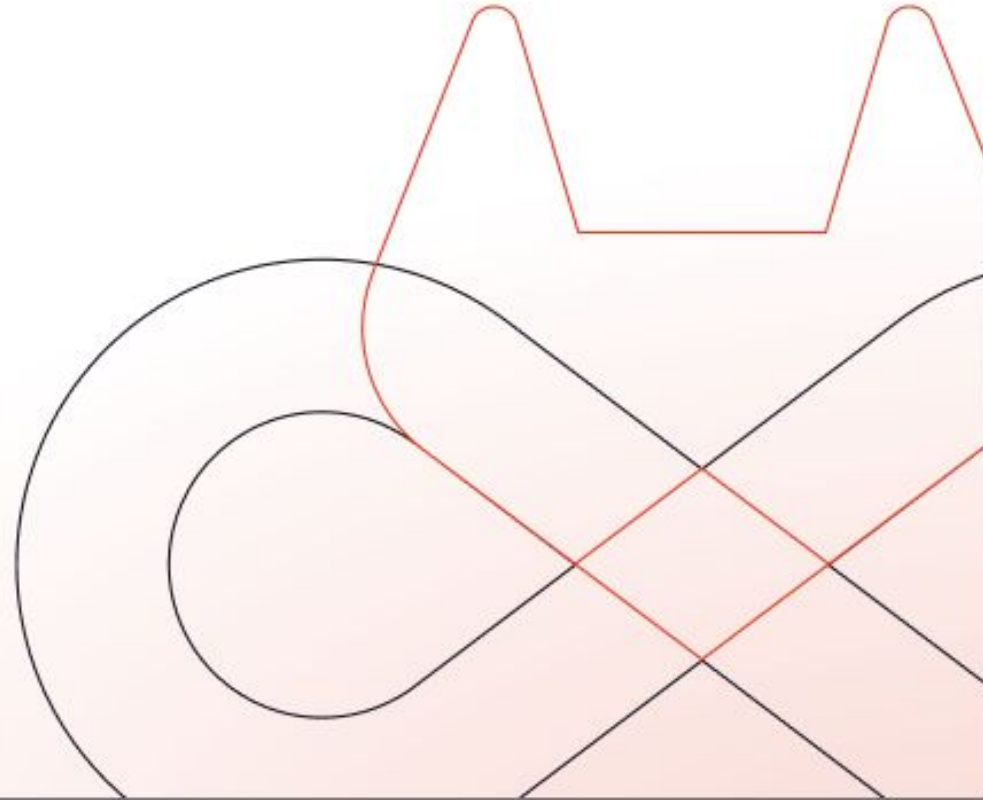
Documentation link:
https://docs.gitlab.com/ee/ci/multi_project_pipelines.html

Pipeline Architectures - Multi-Project Pipelines

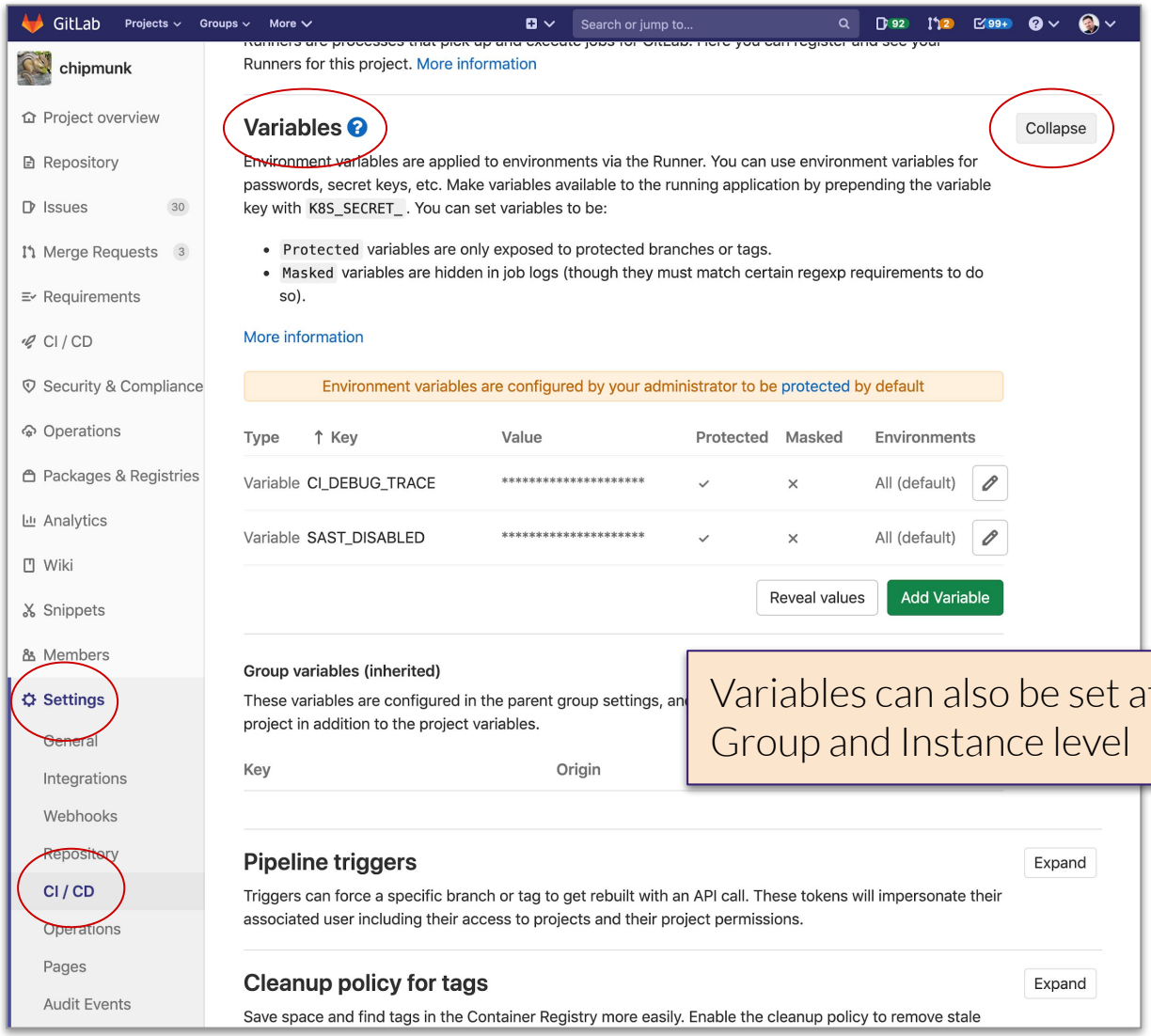


GitLab

Variables

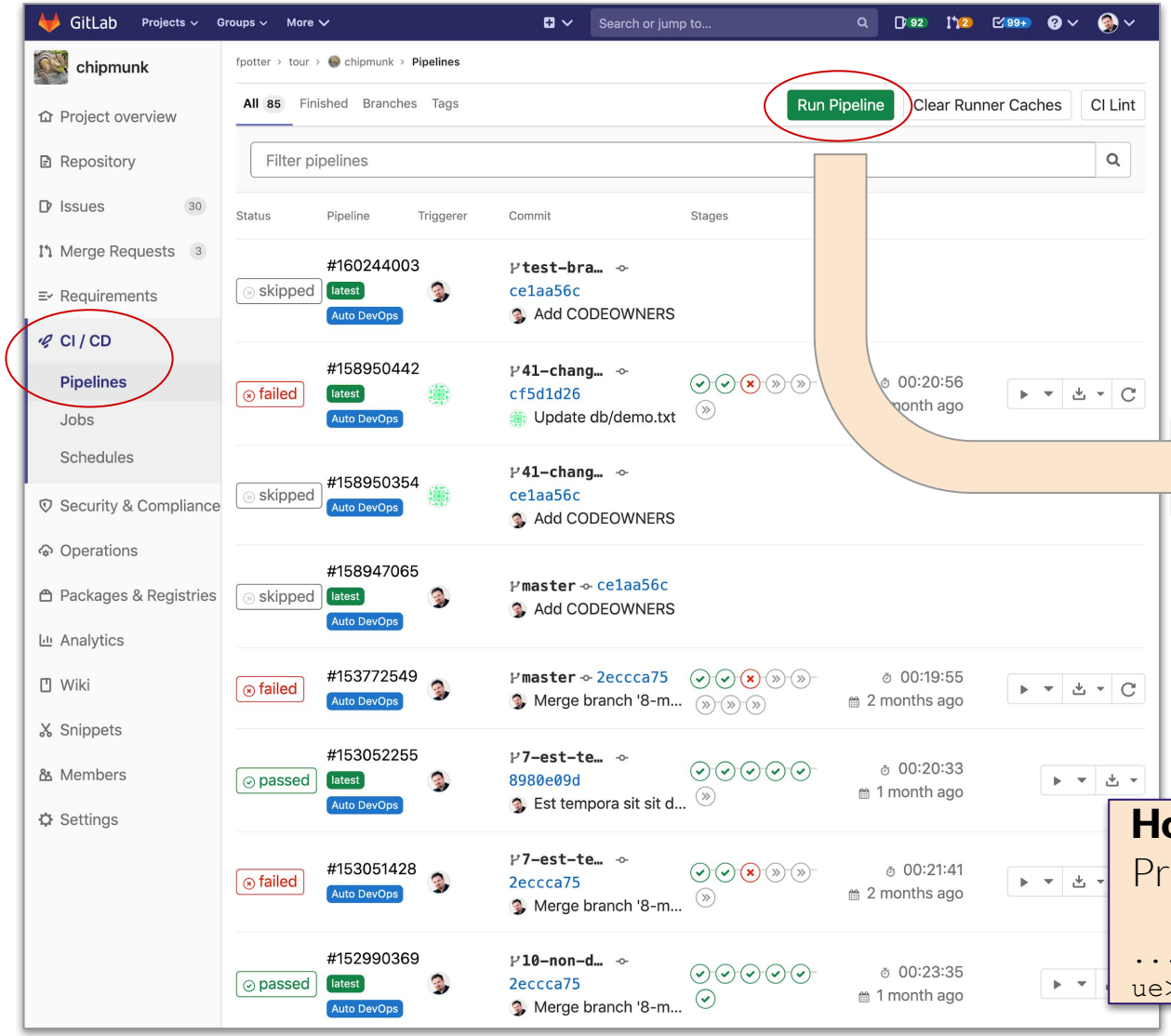


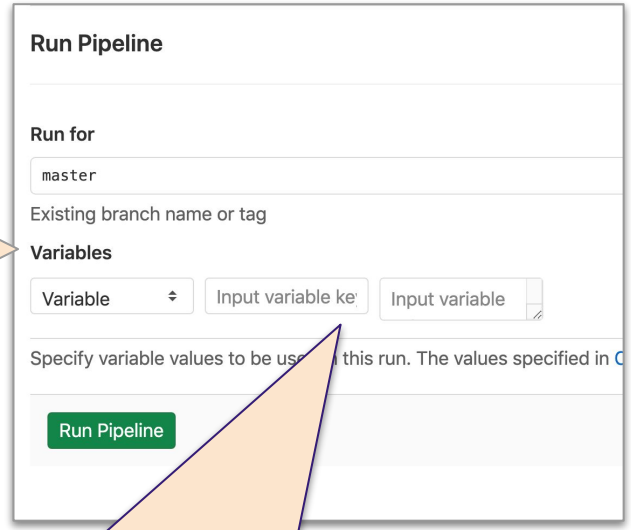
Variables in UI at project level



Variables can also be set at the Group and Instance level

Variables entered for the pipeline run





Hot tip!

Prepopulate the keys and values using URL parameters

```
.../pipelines/new?ref=<branch>&var[<variable_key>]=<value>
```

Variables defined in the CI configuration

```
variables:
  GLOBAL_VAR: test

build:
  variables:
    ENVIRONMENT: staging
  script:
    - echo “Trying $GLOBAL_VAR on $ENVIRONMENT”
```

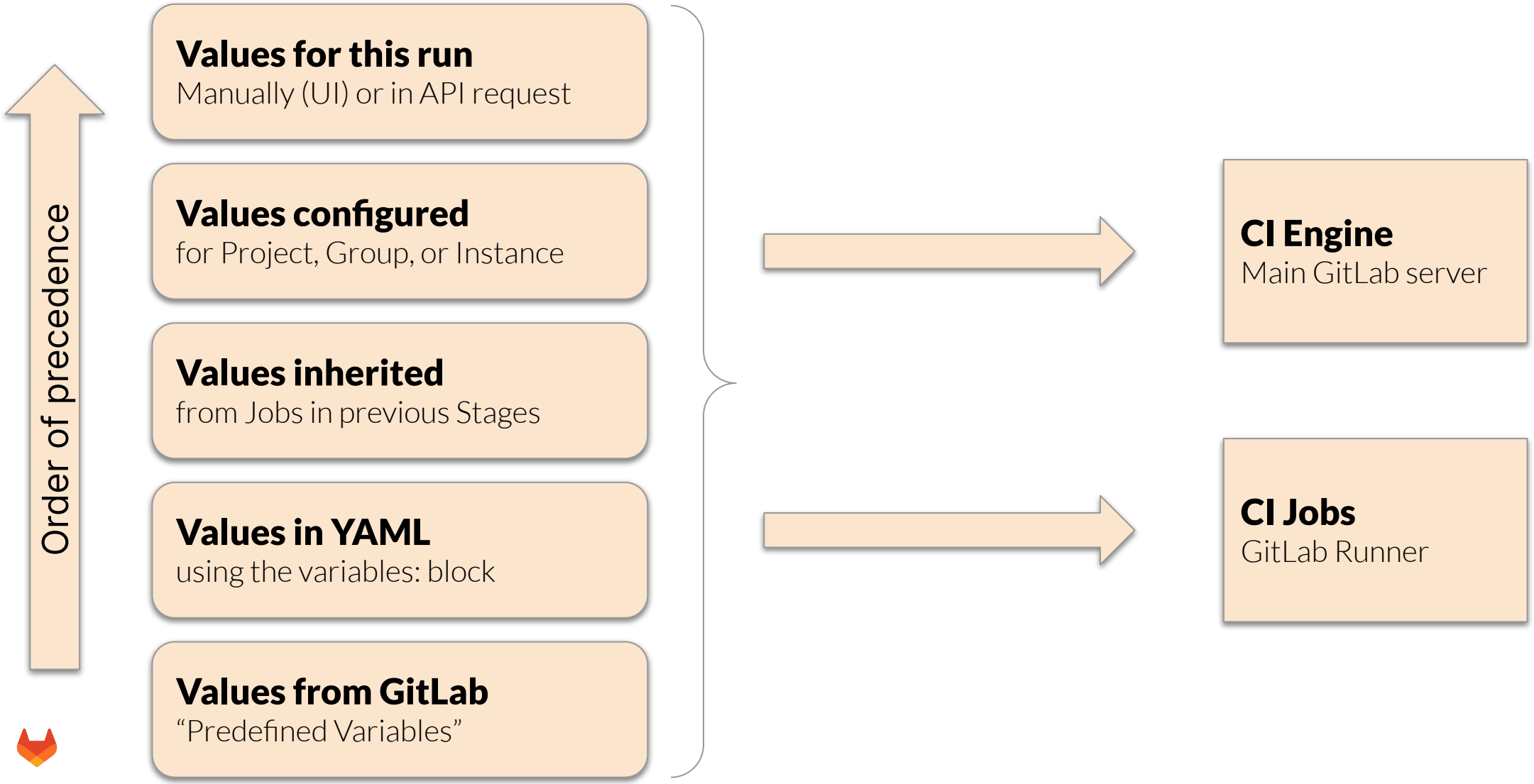
Secrets

```yaml
deploy:
  stage: deploy
  secrets:
    DATABASE_PASSWORD:
      vault:  # Translates to secret: `ops/data/production/db`, field: `password`
        engine:
          name: kv-v2
          path: ops
        path: production/db
        field: password
      file: false

# Shortened version:
# vault: production/db/password@ops
```

Inherited variables

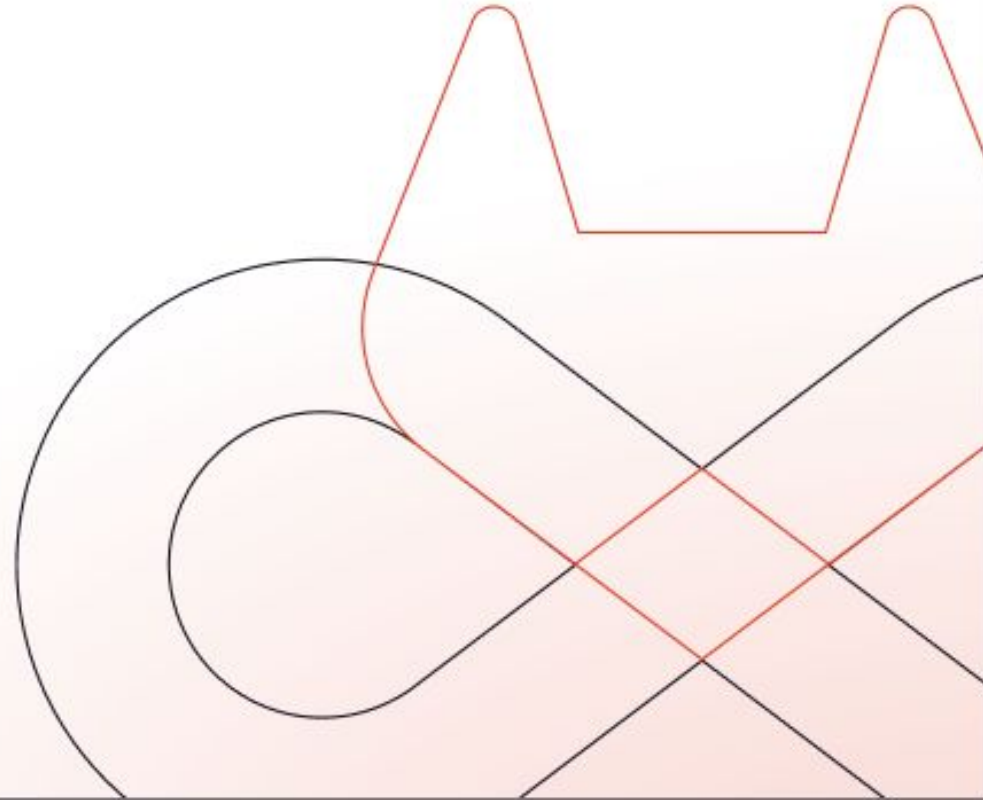
```yaml
build:
  stage: build
  script:
    - echo "BUILD_VARIABLE=value_from_build_job" >> build.env
  artifacts:
    reports:
      dotenv: build.env

deploy:
  stage: deploy
  variables:
    BUILD_VARIABLE: value_from_deploy_job
  script:
    - echo "$BUILD_VARIABLE"  # Output is: 'value_from_build_job' due to precedence
```

How variables are processed

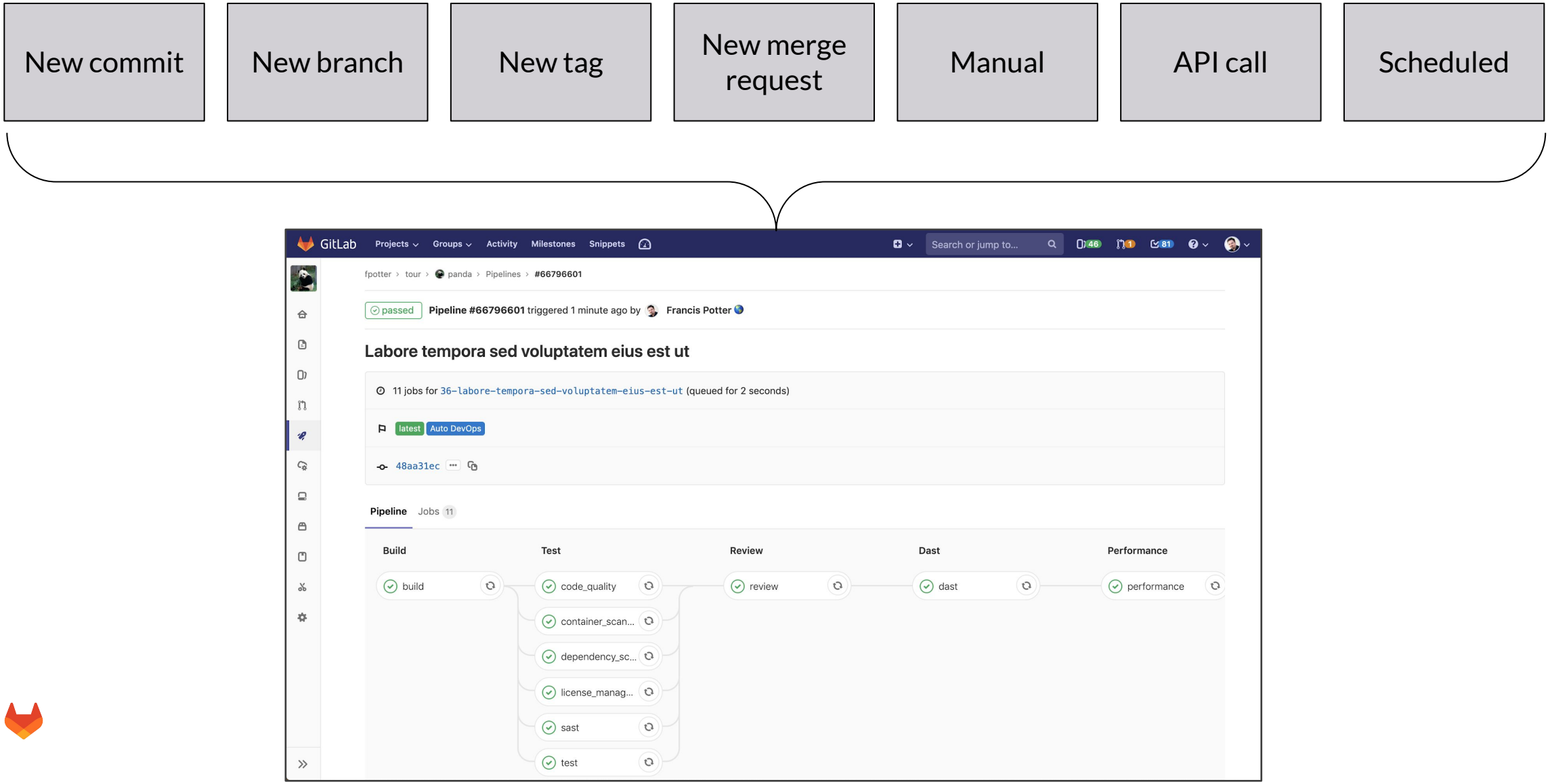


GitLab

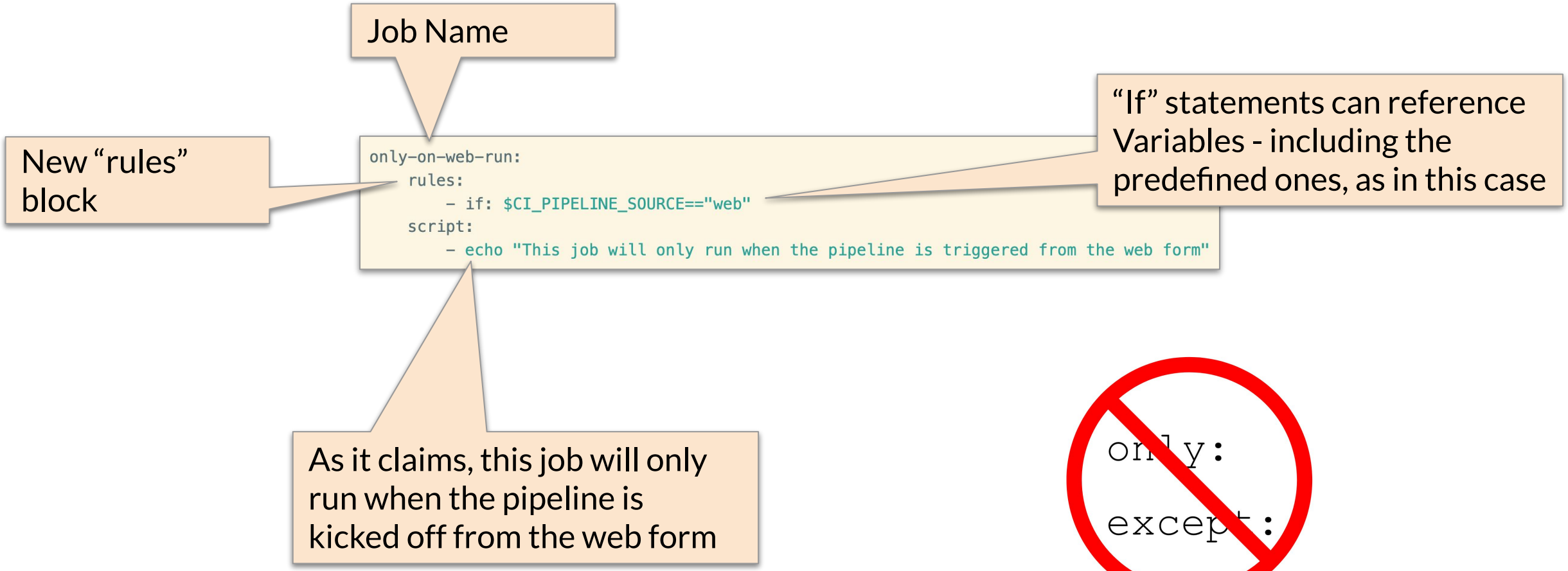
Rules



When are pipelines run?



Rules: The basics



Rules Quick Reference

Clauses

`if`
`changes`
`exists`
(... or none)

Operators

(just the variable)
`==`
`!=`
`=~`
`!~`
`&&`
`||`

Results

`when`
`allow_failure`
`start_in`

“when” options

`always`
`never`
`on_success`
`on_failure`
`manual`
`Delayed`
(... or none)

Rules Quick Reference

Default values

```
when: on_success
allow_failure: false
```

Job is <u>added</u> when:

A rule matches and has either:

- `when: on_success`
- `when: delayed`
- `when: always`

No match, but last clause is:
(As standalone attribute)

- `when: on_success`
- `when: delayed`
- `when: always`

Job is **not** <u>added</u> when:

No rules matches and no standalone:

- `when: on_success`
- `when: delayed`
- `when: always`

A rule matches with:

- `when: never`

Rules Example 01

```
job:
  Script: ”echo Hello, Rules!“
  rules:
    - if: ‘$CI_PIPELINE_SOURCE == “merge_request_event”’
    - if: ‘$CI_PIPELINE_SOURCE == “push”’
```

- Control when merge request pipelines run.
  - `if: '$CI_PIPELINE_SOURCE == "merge_request_event"'`
- Control when both branch pipelines and tag pipelines run.
  - `if: '$CI_PIPELINE_SOURCE == "push"'`

Rules Example 02

```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
```

- This job will not run if this was triggered from a merge request
  - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
- This job will not run if pipeline was scheduled
  - if: '$CI_PIPELINE_SOURCE == "schedule"'
- Otherwise, this job will run if the previous stage was successful

Rules Example 03

```yaml
job:
  script: "echo This creates double pipelines!"
  rules:
    - if: '$CUSTOM_VARIABLE == "false"'
      when: never
    - when: always
```

- This job will not run if CUSTOM_VARIABLE is equal to false
  - if: '$CUSTOM_VARIABLE == "false"'
- Otherwise, this job will run even if the previous stage had a failure

Rules Example 04

```
docker build:
  script: docker build -t my-image:$CI_COMMIT_REF_SLUG .
  rules:
    - if: '$VAR == "string value"'
      changes: # Will include the job and set to when:manual if any of the follow paths match a modified fi
        - Dockerfile
        - docker/scripts/*
      when: manual
  # - when: never would be redundant here, this is implied any time rules are listed.
```

- This job will be in the pipeline and will be a manual job
  - if: '$VAR == "string value"' and there was a change to Dockerfile or any file in docker/scripts
- Otherwise, this job will not run

Rules Example 05

```yaml
docker build:
  script: docker build -t my-image:$CI_COMMIT_REF_SLUG .
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: delayed
      start_in: '3 hours'
      allow_failure: true
```

- This job will run 3 hours after triggered and will be allowed to fail (will not prevent further stages from firing)
  - if: '$CI_COMMIT_BRANCH == "main"'
- Otherwise, this job will not run

Workflow Rules

```
workflow:
  rules:
   - if: $CI_COMMIT_MESSAGE =~ /-wip$/
     when: never
   - if: $CI_COMMIT_TAG
     when: never
   - when: always
```

- This pipeline will not run if the commit message ends with “-wip”
  - if: '$CI_COMMIT_MESSAGE =~ /-wip$/
- This pipeline will not run if this was triggered by a TAG being applied
  - if: '$CI_COMMIT_TAG
- Otherwise, this pipeline will run

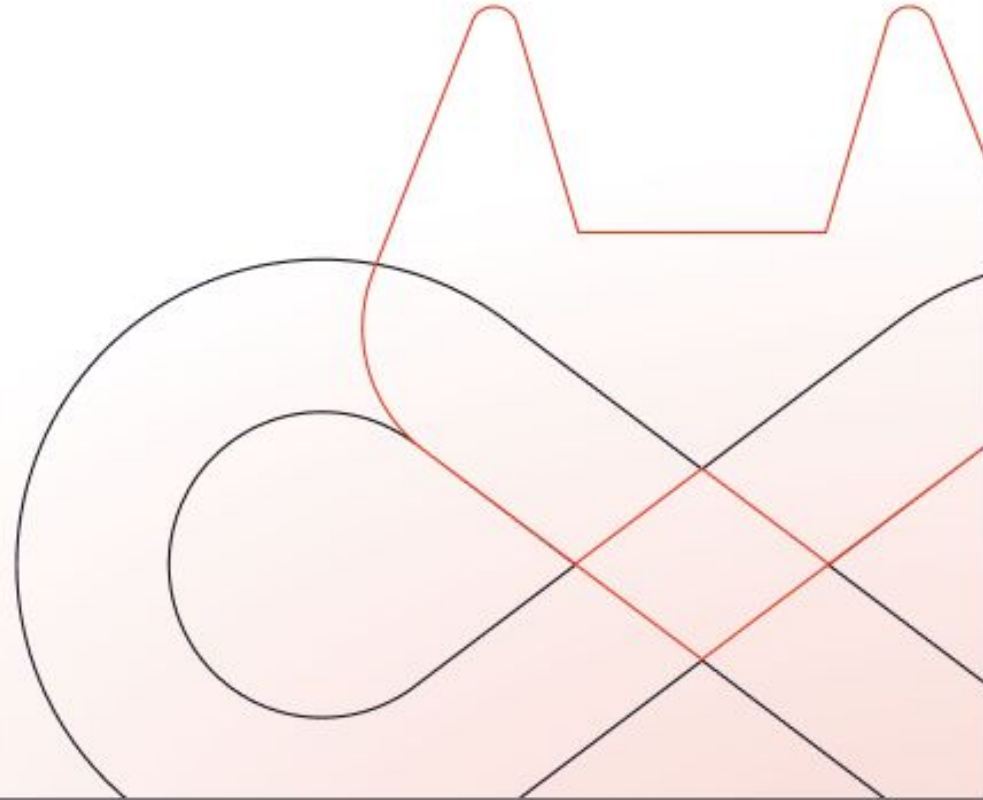
Workflow - Variables depending on

```
workflow:
  rules:
   - if: $CI_COMMIT_REF_NAME =~ /-wip$/
     variables:
       DOCKER_FILE: test.dockerfile
   - if: $CI_COMMIT_REF_NAME =- $CI_DEFAULT_BRANCH
     variables:
       DOCKER_FILE: main.dockerfile
   - when: always
```

- The variable `DOCKER_FILE` will be set differently based on the rules.
- This pipeline will always run

GitLab

Artifacts



CI configuration

- Allows for **saving of build artifacts** and/or the output of any job
- Are available for use by **subsequent jobs**
- Can pull in **any combination of paths and/or files**
- Use exclude to **limit what is added**
- Use depends to **limit what gets downloaded** on subsequent jobs
- Use when to determine **if artifacts will be stored** or not
- Use expire_in to determine **when artifacts will be destroyed**

```
artifacts:
  when: on_success
  paths:
    - bin/target
    - .exe
  exclude:
    - throw-away.exe
  expire_in:  3mos
```

https://docs.gitlab.com/ce/ci/yaml/README.html#artifacts

Artifact download by URL

Three ways of accessing Artifacts via secure HTTPS

Build product for a specific CI pipeline run:

https://gitlab.com/fpotter/examples/c-with-artifact/-/jobs/207917559/artifacts/file/helloworld

Build product for the latest CI pipeline run on a specific branch (main):

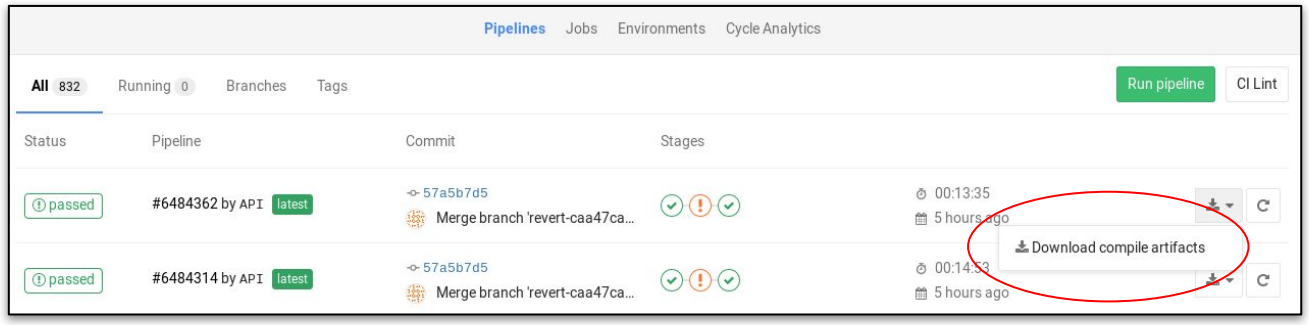
https://gitlab.com/fpotter/examples/c-with-artifact/-/jobs/artifacts/main/raw/helloworld?job=build

All build products (as a Zip) for the latest CI pipeline run on a specific branch:

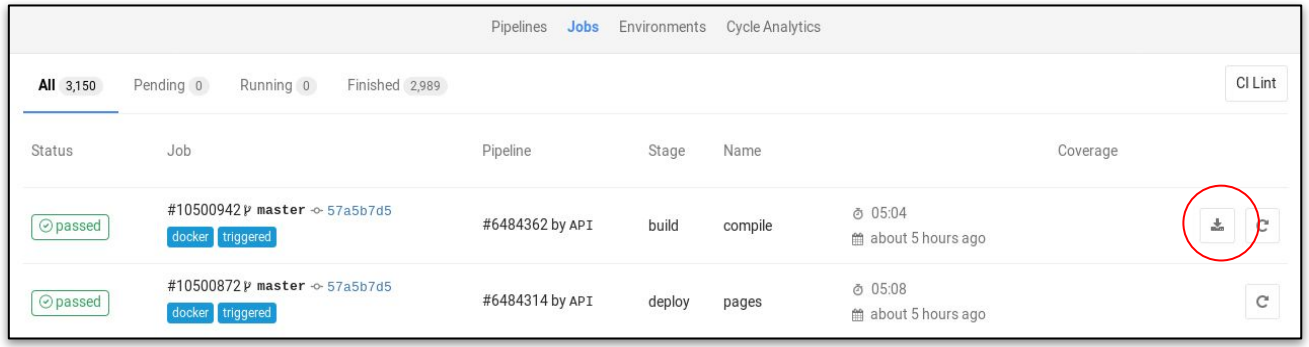
https://gitlab.com/fpotter/examples/c-with-artifact/-/jobs/artifacts/main/download?job=build

Artifact download in GitLab UI

On the Pipelines page



On the Jobs page



On a specific Job

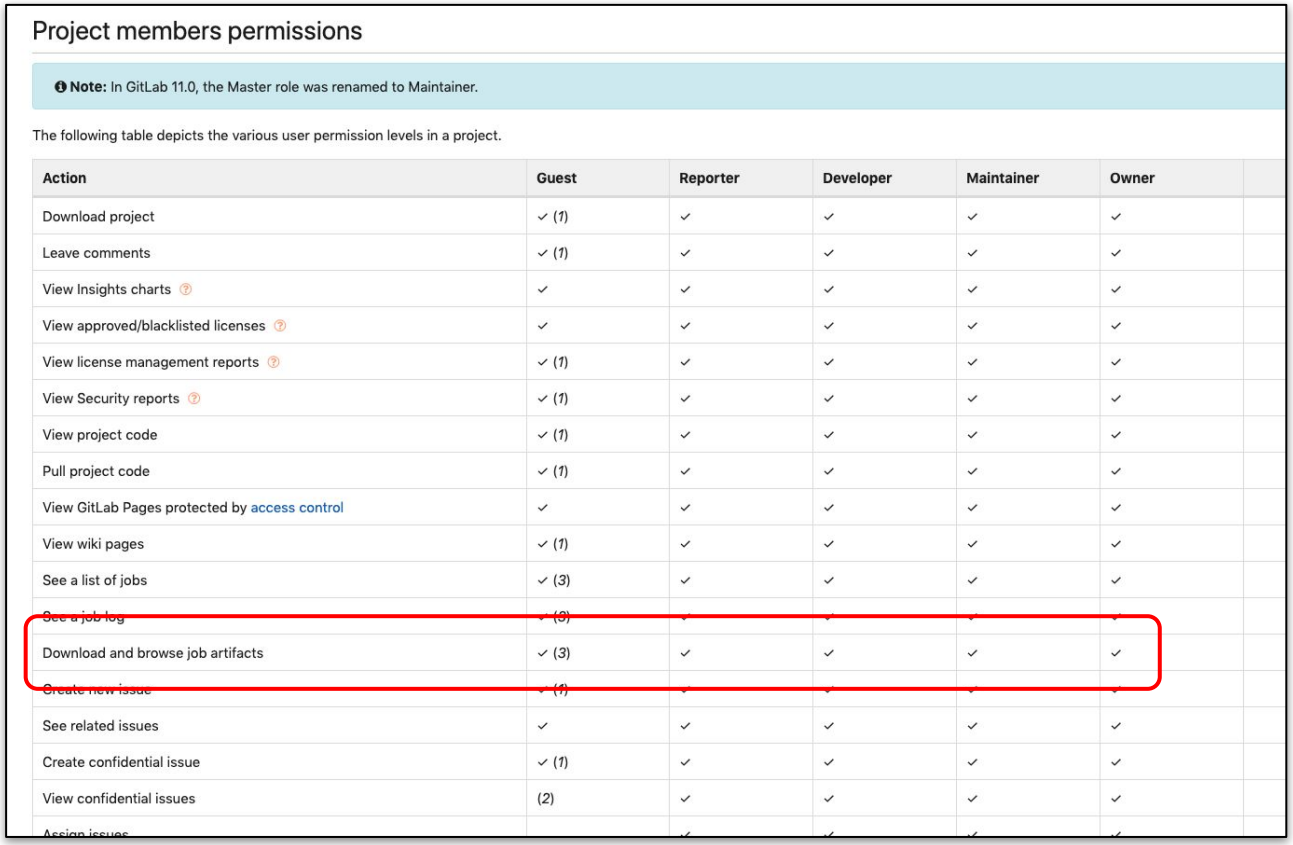
Artifact browser (for a Job)

Artifact administration

In a self-managed GitLab instance, job artifacts may be stored in **local storage** or **object storage**

Artifact **expiration times** can be configured at the instance level

Artifact downloads fall under GitLab's access control



Project members permissions

Note: In GitLab 11.0, the Master role was renamed to Maintainer.

The following table depicts the various user permission levels in a project.

Action	Guest	Reporter	Developer	Maintainer	Owner
Download project	✓ (1)	✓	✓	✓	✓
Leave comments	✓ (1)	✓	✓	✓	✓
View Insights charts	✓	✓	✓	✓	✓
View approved/blacklisted licenses	✓	✓	✓	✓	✓
View license management reports	✓ (1)	✓	✓	✓	✓
View Security reports	✓ (1)	✓	✓	✓	✓
View project code	✓ (1)	✓	✓	✓	✓
Pull project code	✓ (1)	✓	✓	✓	✓
View GitLab Pages protected by access control	✓	✓	✓	✓	✓
View wiki pages	✓ (1)	✓	✓	✓	✓
See a list of jobs	✓ (3)	✓	✓	✓	✓
See a job log	✓ (3)	✓	✓	✓	✓
Download and browse job artifacts	✓ (3)	✓	✓	✓	✓
Create new issue	✓ (1)	✓	✓	✓	✓
See related issues	✓	✓	✓	✓	✓
Create confidential issue	✓ (1)	✓	✓	✓	✓
View confidential issues	(2)	✓	✓	✓	✓
Assign issues		✓	✓	✓	✓

Container and language-specific package registries

Docker registry

Software repository	Description	Available in GitLab version
Container Registry	The GitLab Container Registry enables every project in GitLab to have its own space to store Docker images.	8.8+
Dependency Proxy	The GitLab Dependency Proxy sets up a local proxy for frequently used upstream images/packages.	11.11+
Conan Repository	The GitLab Conan Repository enables every project in GitLab to have its own space to store Conan packages.	12.6+
Maven Repository	The GitLab Maven Repository enables every project in GitLab to have its own space to store Maven packages.	11.3+
NPM Registry	The GitLab NPM Registry enables every project in GitLab to have its own space to store NPM packages.	11.7+
NuGet Repository	The GitLab NuGet Repository will enable every project in GitLab to have its own space to store NuGet packages.	12.8+
PyPi Repository	The GitLab PyPi Repository will enable every project in GitLab to have its own space to store PyPi packages.	12.10+
Go Proxy	The Go proxy for GitLab enables every project in GitLab to be fetched with the Go proxy protocol.	13.1+
Composer Repository	The GitLab Composer Repository will enable every project in GitLab to have its own space to store Composer packages.	13.2+

Dependency Proxy

Language-specific package registries

dependencies

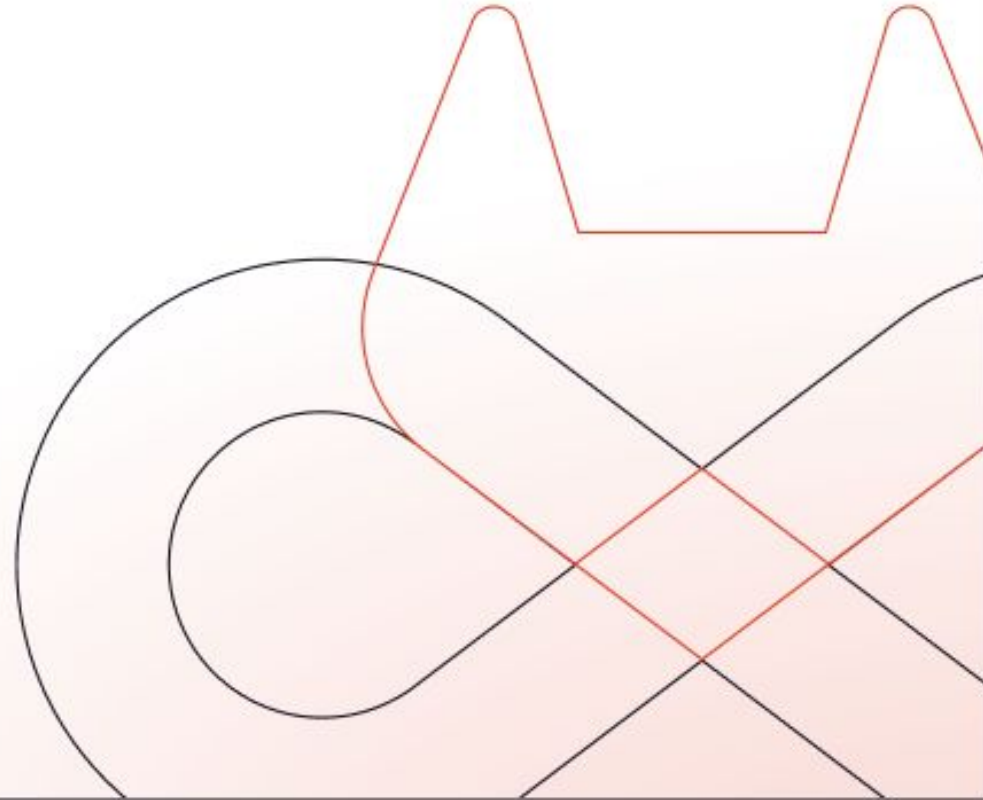
```yaml
build:osx:
  stage: build
  script: make build:osx
  artifacts:
    paths:
      - binaries/

build:linux:
  stage: build
  script: make build:linux
  artifacts:
    paths:
      - binaries/

test:osx:
  stage: test
  script: make test:osx
  dependencies:
    - build:osx

test:linux:
  stage: test
  script: make test:linux
  dependencies:
    - build:linux

deploy:
  stage: deploy
  script: make deploy
```

- Reduce number of artifacts passed from previous jobs
- Improve job performance by preventing large artifacts to download for each job
- Empty array will skip downloading artifacts
- Only from jobs from previous stages

Source: https://docs.gitlab.com/ee/ci/yaml/#dependencies

GitLab

Include and Extends



Include

- Allows the inclusion of external YAML files
- Helps to break down the CI/CD configuration into multiple files and increases readability for long configuration files
- Possible to have template files stored in a central repository and projects include their configuration files
- Helps avoid duplicated configuration, for example, global default variables for all projects
- Can be nested (up to 100 includes)

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

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

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

```
include:
  - remote: 'https://gitlab.com/awesome-project/raw/master/.gitlab-ci-template.yml'
```

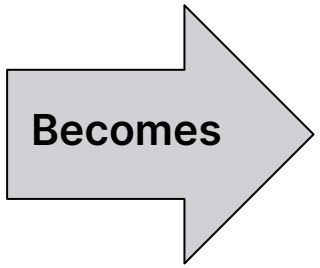
```
include:
  - template: Android-Fastlane.gitlab-ci.yml
  - template: Auto-DevOps.gitlab-ci.yml
```

Method	Description
`local`	Include a file from the local project repository.
`file`	Include a file from a different project repository.
`remote`	Include a file from a remote URL. Must be publicly accessible.
`template`	Include templates which are provided by GitLab.

Extends

- Defines entry names that a job uses extends is going to inherit from
- Alternative to YML anchors, and is more flexible and readable
- Performs a merge
- Supports multilevel inheritance (but more than 3 is not recommended)
- Can merge hidden jobs (.tests) or from regular jobs

```
.tests:
  script: rake test
  stage: test
  only:
    refs:
      - branches

rspec:
  extends: .tests
  script: rake rspec
  only:
    variables:
      - $RSPEC
```




```
rspec:
  script: rake rspec
  stage: test
  only:
    refs:
      - branches
    variables:
      - $RSPEC
```

Using Includes and Extends together

- Extends works across configuration files with include

If you have a local included.yml:

```
.template:
  script:
    - echo Hello!
```

You can do the following:

```
include: included.yml

useTemplate:
  image: alpine
  extends: .template
```

This will run a job called `useTemplate` that runs `echo Hello!` as defined in the `.template` job, and uses the `alpine` Docker image as defined in the local job.

Q&A