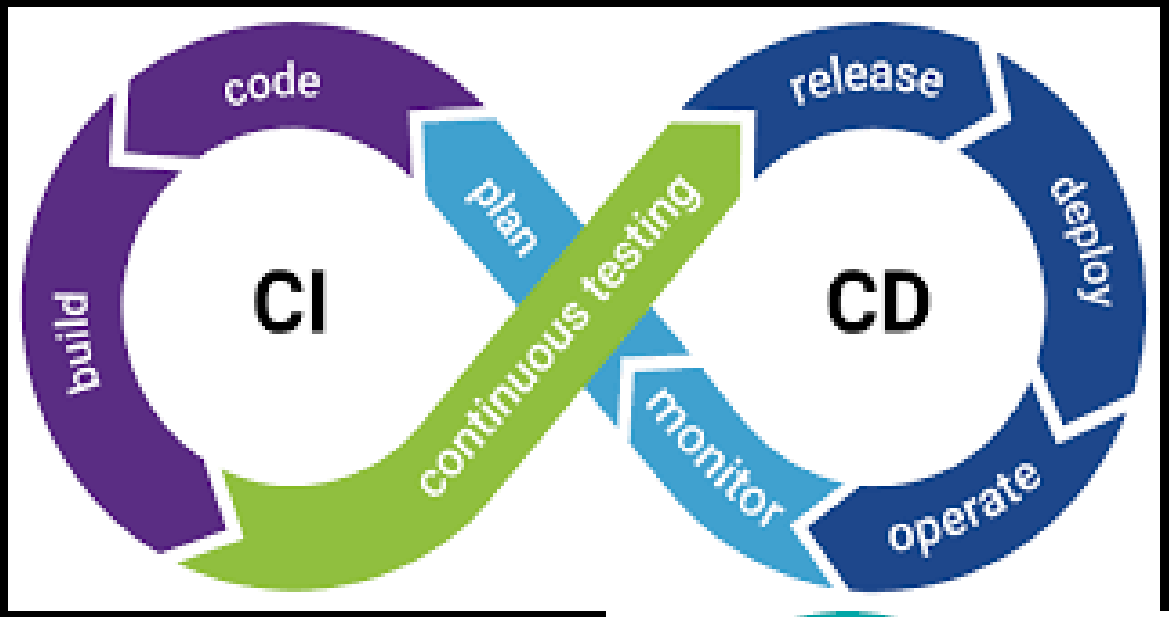
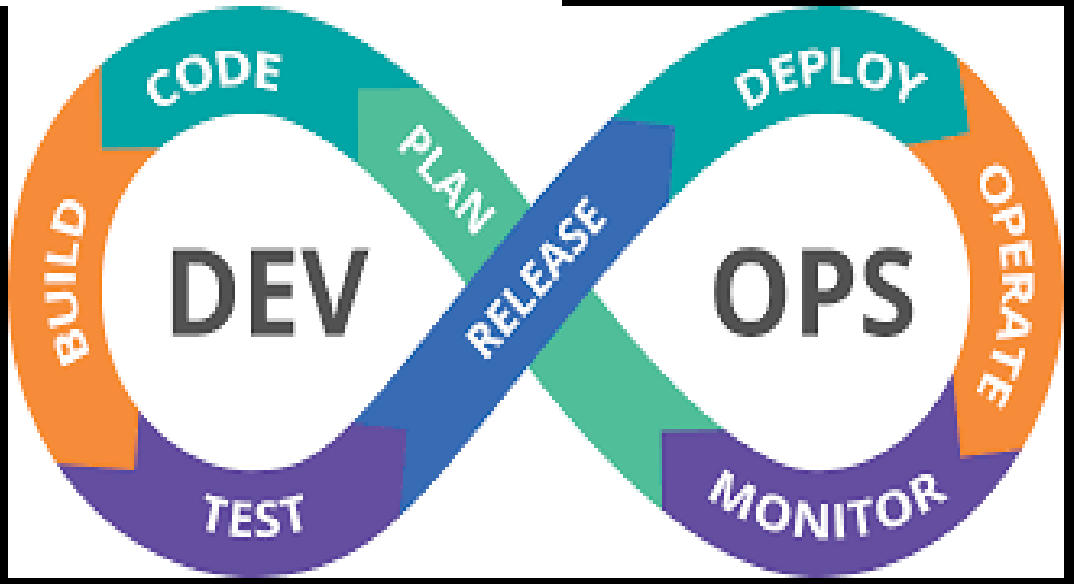
Continuous Integration/Continuous Deployment (CI/CD)


code
plan
build
CI
continuous testing
release
deploy
CD
monitor
operate

CI/CD → DevOps


CODE
PLAN
BUILD
DEV
TEST
RELEASE
DEPLOY
OPERATE
OPS
MONITOR

Continuous Integration (CI) - Definition

- CI is a software development practice where members of a team integrate their work frequently (daily) leading to multiple integrations per day. Each integration is verified by an automated build (including tests) to detect integration errors as quickly as possible.

  Martin Fowler

- In software engineering, CI implements continuous processes of applying quality control - small pieces of effort, applied frequently. CI aims to improve the quality of software, and to reduce the time taken to deliver it, by replacing the traditional practice of applying quality control after completing all development.

CI – How and Why.

- How:
  1. Maintain a single (central) source repository.
  2. Automate the build tasks.
  3. Everyone commits ‘every day’ to the central repository.
  4. Do the build on a dedicated CI machine.
  5. Keep the build fast, e.g. caching 3rd party libraries.
  6. Test the application on a clone of the production environment.
  7. Make it easy for everyone to get the latest executable.
  8. Make the process transparent for everyone.

CI – How and Why.

- Why:
  1. Detect development problems earlier.
  2. Find and remove bugs earlier.
  3. Reduce risks of cost, schedule and budget overrun.
  4. Increase project visibility.
  5. Deliver new features and get user feedback more rapidly.
  6. Improve team cohesion.
  7. Have greater availability of deployable software.

CI – Objective.

- Better:
  - Build better quality software …… because it's tested early and often …….. adheres to best practices & coding standards.
- Faster:
  - Test in parallel with development (Agile), not at the end (Waterfall).
  - Do not have 'integration days'.
  - Application builds should be a non-event.
- Cheaper:
  - Identify defects earlier.
  - Fix when it's least costly.
  - Testing should be repeatable.

Continuous Delivery/Deployment

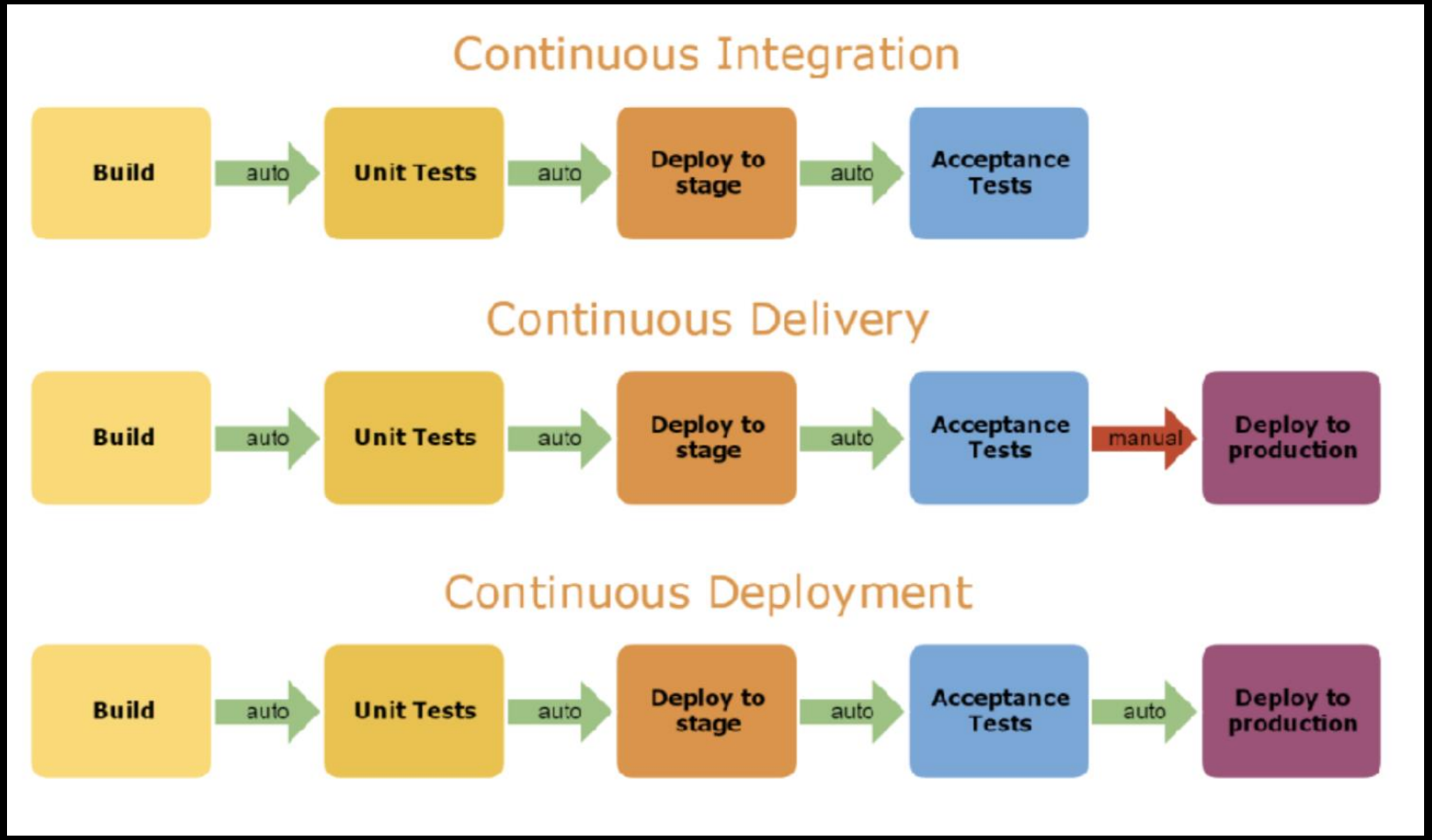
- Continuous Delivery is a development discipline where you build software in such a way that it can be released to production at any time.
  - A human decides to release into production or not

- Continuous Deployment means that every software change goes through the CI pipeline and automatically gets deployed into production, resulting in very frequentproduction deployments.

CD – How and Why.

- How:
  1. New features are continuously integrated into the software.
  2. Run automated tests.
  3. Automatically deploy the application to a production-like environment, termed a staging environment.
- Why:
  1. Reduce production deployment risks.
  2. Change the version in production more rapidly.
  3. Be able to release new version of the application at the ‘push of a button’.
  4. Get early user feedback.

CI/CD Pipelines



CI metrics

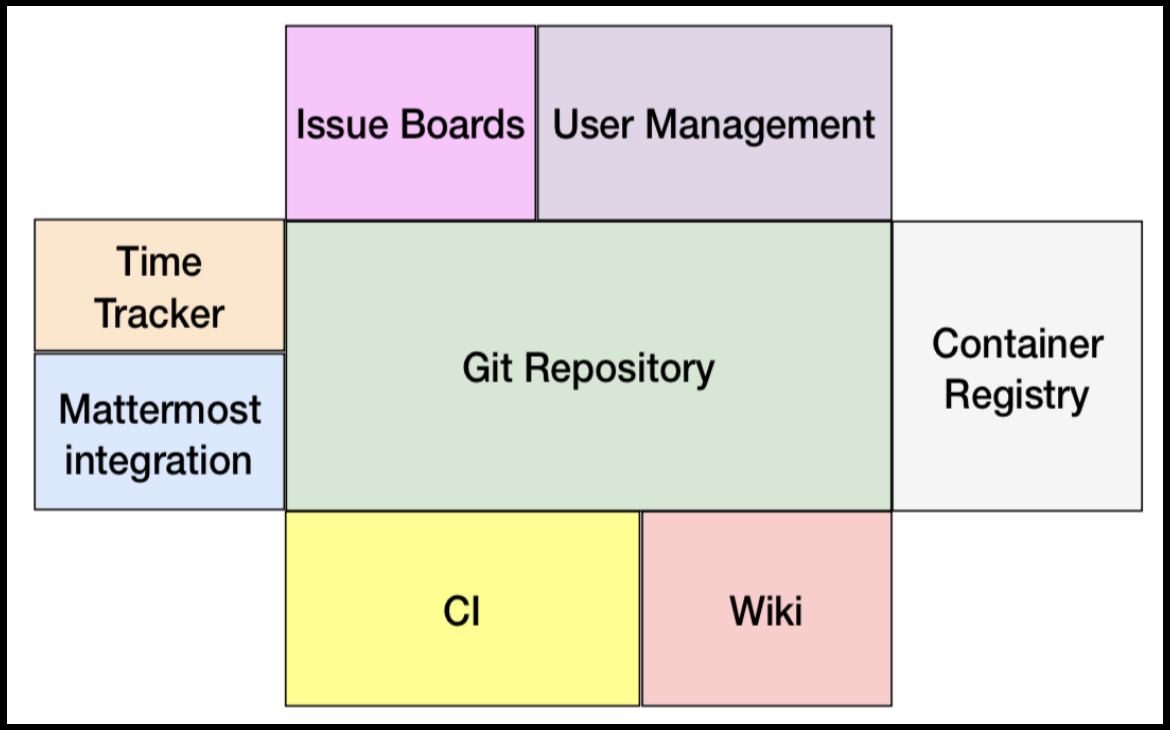
- Is our CI process successful?

- Possible metrics including:
  - Successful Build Rate.
  - Build Repair Rate.
  - Total Numbers of Static Tool Errors
    - Ex. Lint, Checkstyle, PMD, Findbugs
  - Unit Testing Line Coverage.
  - Functional Testing Line Coverage.
- CI Metrics Matter:
  - Identify key metrics and track them visually.
  - Act on them immediately.

CI/CD - Platforms

GitLab.

- A Git-based hosting and collaboration platform
- Open source.
- Hosted (free) or on premise.
- Actively maintained.



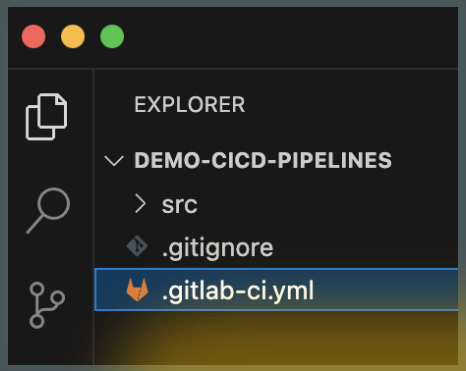
GitLab CI.

- What:
  - Fully integrated with Gitlab repository
  - Version control of CI/CD scripts.
  - Git hooks.
  - Built on the containerization model (optional).

- Why:
  - Application Code and build scripts in the same repo.
  - Easy to get started.
  - Scalable.
  - Isolated test environment.

GitLab CI - Concepts.

- Three constructs:
  1. Pipelines. 2. Stages. 3. Jobs.
- Declare a CI/CD process using these constructs.
  - Pipeline >> Stages >> Jobs.
- A pipeline is a group of jobs that get executed in stages (batches).
  - All jobs in a stage are executed in parallel.
  - If all jobs in a stage succeed, the pipeline moves to the next stage.
  - If a stage job fails, the next stage is not executed.
- Pipelines are defined in ***.gitlab-ci.yml*** (mandatory) in the project base *folder*.



Demo

- A pipeline with 4 jobs spread across 3 stages.

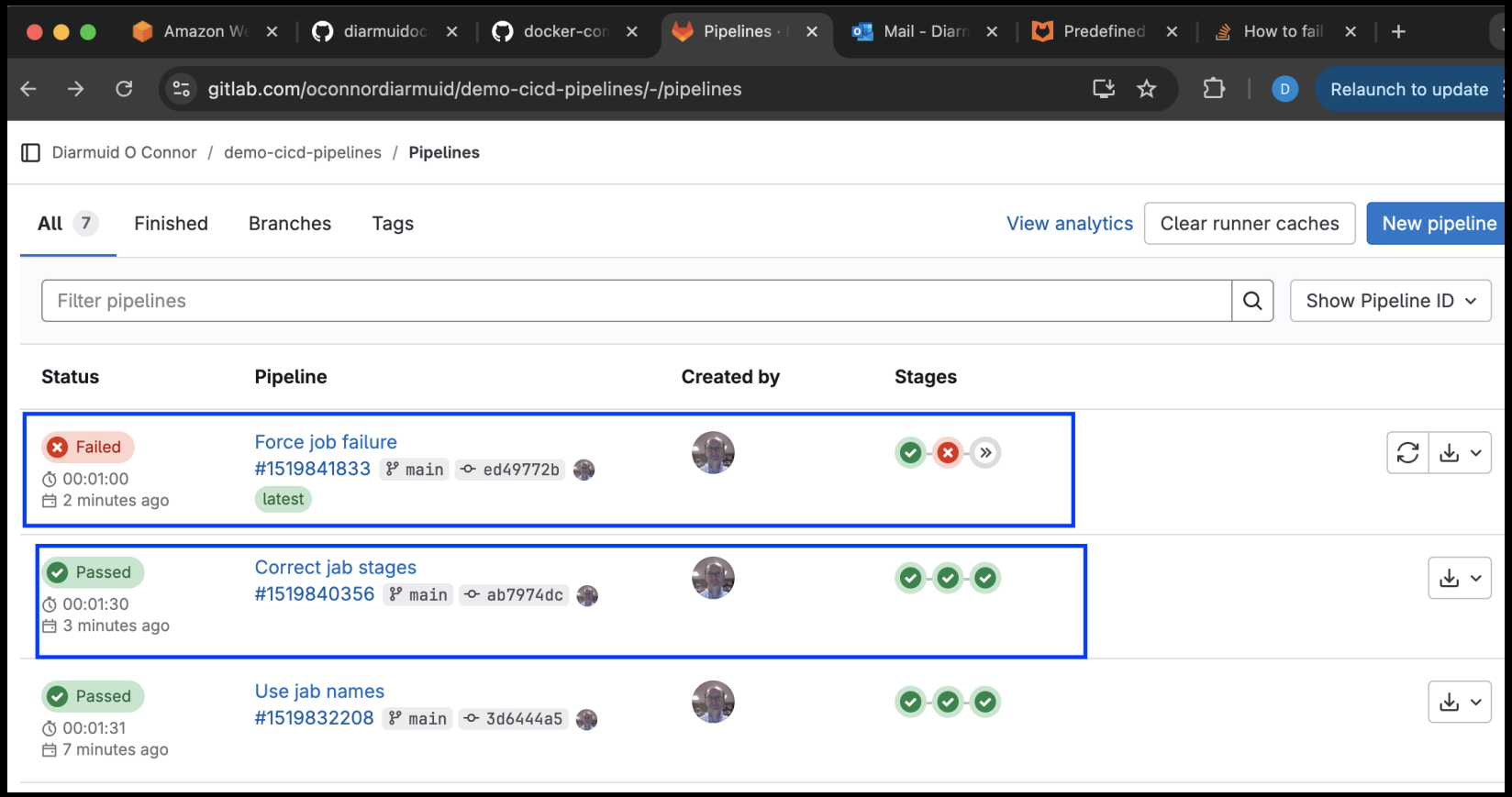
.gitlab-ci.yml M

.gitlab-ci.yml

```
1
2   stages:
3     - stage1
4     - stage2
5     - stage3
6
7   jobA:
8     stage: stage1
9     script:
10      - echo "Start of '$CI_PIPELINE_ID' pipeline "
11      - echo "This is job '$CI_JOB_NAME' in stage '$CI_JOB_STAGE'"
12
13  jobB:
14    stage: stage2
15    script:
16      - echo "This is job '$CI_JOB_NAME' in stage '$CI_JOB_STAGE'"
17      - exit 1
18
19  jobC:
20    stage: stage2
21    script:
22      - echo "This is job '$CI_JOB_NAME' in stage '$CI_JOB_STAGE'"
23
24  jobD:
25    stage: stage3
26    script:
27      - echo "This is job '$CI_JOB_Name' in stage '$CI_JOB_STAGE'"
28      - echo "End of '$CI_PIPELINE_ID' pipeline "
29
```

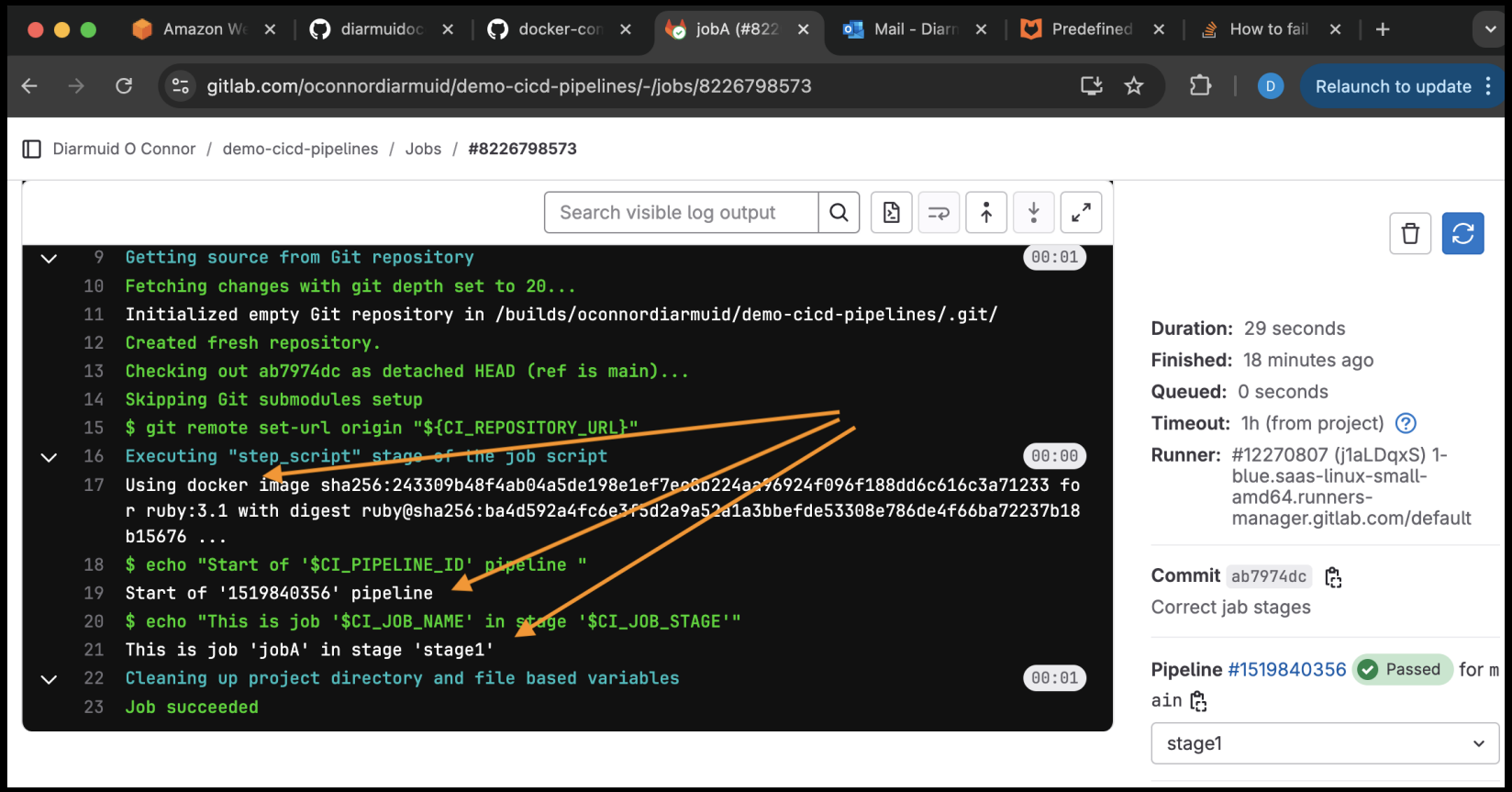
Demo – Pipeline executions

- The pipeline is executed when the repository is updated (git push)



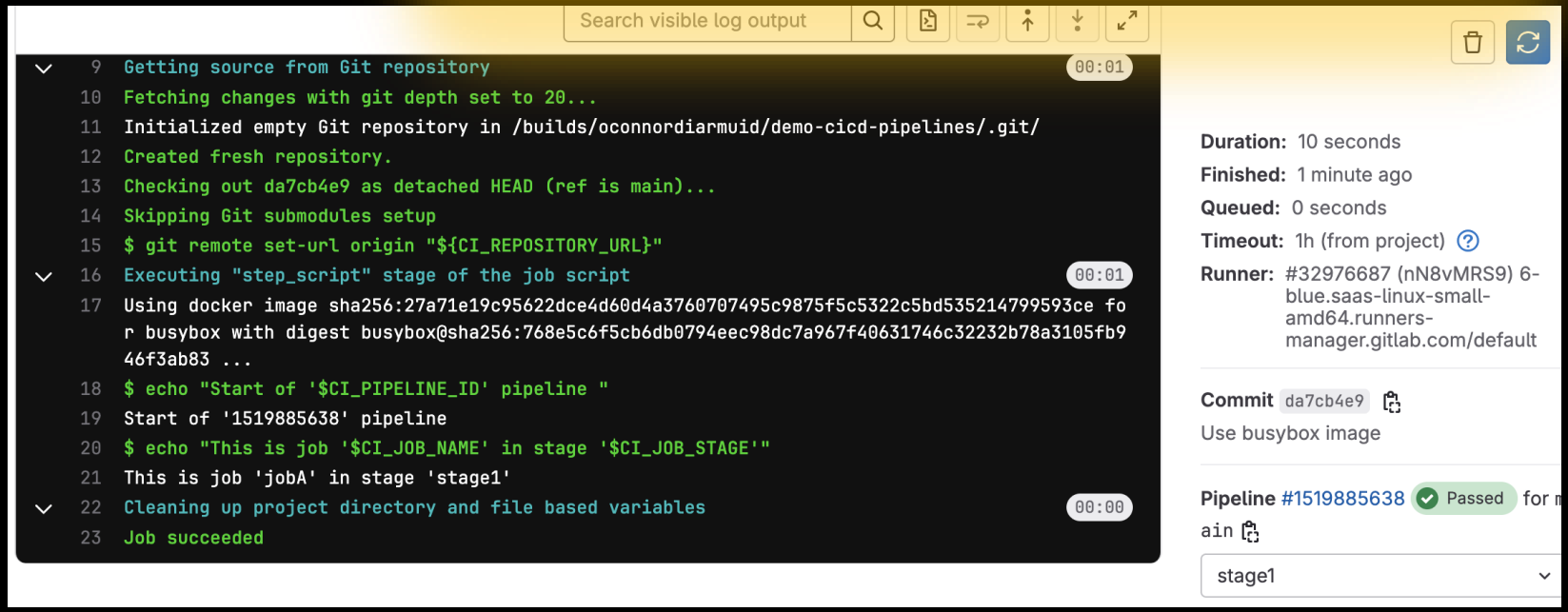
Demo – Job execution

- A job's script is executed inside a container (configurable)



Demo – Job container image

```yaml
jobA:
  stage: stage1
  image: busybox
  script:
    - echo "Start of '$CI_PIPELINE_ID' pipeline "
    - echo "This is job '$CI_JOB_NAME' in stage '$CI_JOB_STAGE'"
```



Variables in GitLab.

- Three sources:
  1. Predefined – see https://docs.gitlab.com/ee/ci/variables/predefined_variables
  2. Local – defined inside a job.
  3. Pipeline – defined at the pipeline level; available to all jobs
  4. Environment – for sensitive data; available to all jobs

Variables in GitLab.

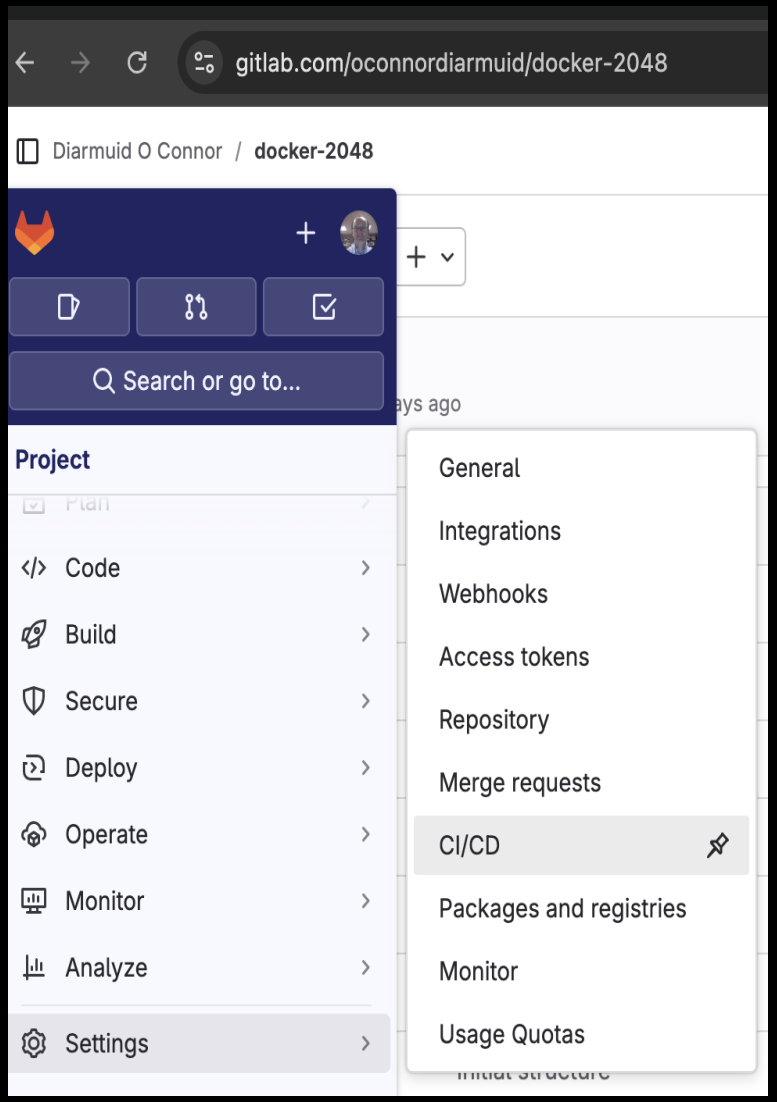
- Pipeline variables.

```
1
2   variables:
3     APP_NAME: user-profile
4     DOCKER_TAG: ${CI_COMMIT_SHORT_SHA}
5     DOCKER_HUB_USERNAME: doconnor
6
7   stages:
8     - stage1
9     - stage2
10    - stage3
11
```

- Reference these variables inside a job using ${var_name}, e.g. ${DOCKER_HUB_USERNAME}
- Use uppercase and _ in names (Convention)

Variables in GitLab.

- Environment variables.
- Variables have a key (name) and value.
- Same syntax for referencing inside a job.

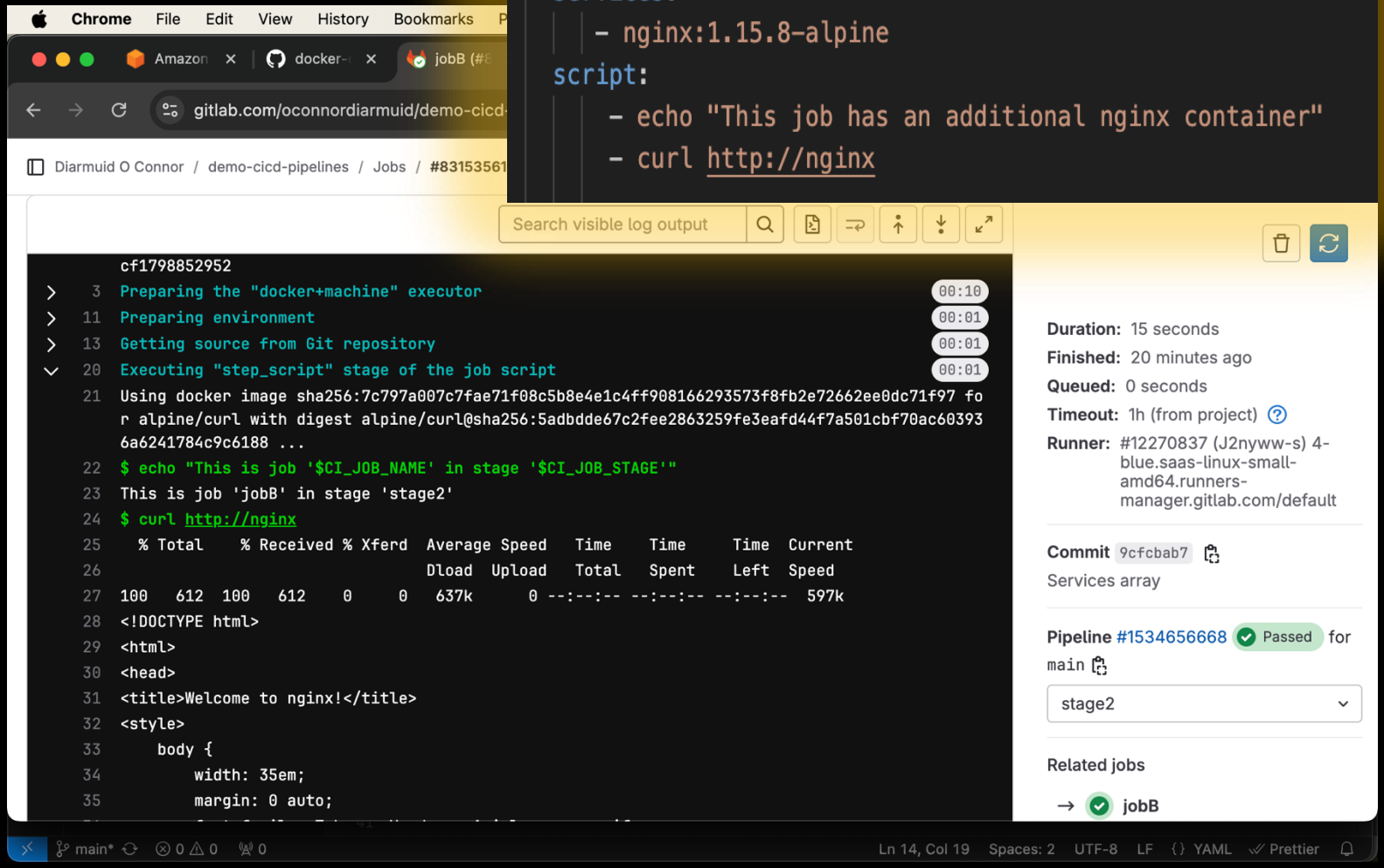


Services.

- A job's script (build script) executes in a container whose image is declared with the image property.
- Can have additional containers available to the build container, using the services property.
- The domain name of a service container is based on its image name, e.g.
  - imageX:2.10-alpine → imageX
  - userA/imageX:2.10-alpine → userA__imageX or userA-imageX

Services

```yaml
jobB:
  stage: stage2
  image: alpine/curl
  services:
    - nginx:1.15.8-alpine
  script:
    - echo "This job has an additional nginx container"
    - curl http://nginx
```



Services – Use case.

- The Docker in Docker (DinD) image contains the Docker daemon/engine.

```yaml
jobC:
  stage: stage2
  image: docker:27.0.2-cli-alpine3.20
  services:
    -  docker:27.0.2-dind-alpine3.20
  script:
    - docker pull doconnor/movies-api:1.0
    - docker images
```

```
90  Digest: sha256:90aebf2ca5179a96f0a23d44aa606856900985b066b7e8ba7abec0c4fd6da020
91  Status: Downloaded newer image for doconnor/movies-api:1.0
92  docker.io/doconnor/movies-api:1.0
93  $ docker images
94  REPOSITORY            TAG       IMAGE ID       CREATED       SIZE
95  doconnor/movies-api   1.0       536d4758124e   2 weeks ago   2.84GB
96  Cleaning up project directory and file based variables                00:00
97  Job succeeded
```

Job Structure

	`job1:` `  stage: build`
Variables	`variables:` `  DATABASE_URL: "test"`
Before script	`before_script:` `  - execute-before-script-for-job1`

Script	`script:`
	`  - execute-script-for-job1`
	`  - something else`

After Script	`after_script:` `  - execute-after-script-for-job1`

Artifacts	`artifacts:`
	`  paths:`
	`  - .variables`
	`  expire_in: 1 week`

Only/ Except	`only:`
	`  - master`
	`except:`
	`  - develop`

Tags	`tags:`
	`  - ruby`
	`  - postgres`

When	`when: manual`

Job rules

- Replaces the deprecated only/except option.
- Use rules to specify when jobs run.

```
jobB:
  stage: stage2
  image: alpine/curl
  services:
    - nginx:1.15.8-alpine
  script:
    - echo "This job has an additional nginx container"
    - curl http://nginx
  rules:
    - if: $CI_COMMIT_BRANCH == 'develop'
```

Job rules

- Examples: Run a job only when:
  - The commit is to a certain branch.
  - The commit is a merge request.
  - The commit tag is a certain value.
  - The commit tag matches a pattern

- Can have multiple rules.
  - If any rule is true then the job is executed

GitLab CI/CD Architecture

- GitLab CI/CD uses registered Runner nodes to execute pipeline jobs.
- GitLab Runner:
  - An application to run jobs and send results back to GitLab
  - Installed on cloud / on-premise machines.
  - Written in Go and can be run as a single binary.
  - Works with Linux, Windows and OSX.
  - Acts as an agent (worker) for GitLab CI/CD service
- GitLab Executor:
  - A Runner passes a job to an Executor.
  - Executor determines the environment a job runs in .
  - Lots of Executor types:
    - PowerShell, VirtualBox, Docker, Kubernetes, Custom, etc,