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

build_job:
  stage: build
  script:
    - echo "Compiling the code..."

unit_tests:
  stage: test
  script:
    - echo "Running unit tests..."

integration_tests:
  stage: test
  script:
    - echo "Running integration tests..."

deploy_job:
  stage: deploy
  script:
    - echo "Deploying to production..."
```

Jobs inside the same stage run in parallel (concurrently), **as long as enough GitLab Runners are available**.

In the above example:

- `unit_tests` and `integration_tests` belong to the **test stage**.
- They run **at the same time** on different runners.
- This reduces overall pipeline time.

Stage Execution Order

- GitLab runs **stages sequentially**, but jobs within a stage **concurrently**.
- In the example:

1. All jobs in `build` must finish before `test` starts.

2. All jobs in `test` must finish before `deploy` starts.

- This ensures order where needed but allows concurrency to maximize speed.

How Runners Enable Concurrency

- When jobs are triggered, the **GitLab server schedules them** and assigns them to **available runners**.

- If:

  - You have **1 runner** → jobs will run one after another (even in the same stage).

  - You have **multiple runners (or multiple parallel slots in a runner)** → jobs can run **simultaneously**.

What are `only` and `except`?

In GitLab CI/CD, every **job** can define **conditions** that decide **when the job should run**.
Two common keywords for this are:

- **`only`** → Run the job **only if certain conditions are true**.

- **`except`** → Run the job **except if certain conditions are true**.

These help control **job execution** depending on branch, tag, merge request, or pipeline type.

---

2. `only`

You use `only` when you want to **restrict a job to specific cases**.
Examples:

```
deploy_job:
  stage: deploy
  script:
```

```
    - echo "Deploying..."
  only:
    - main         # Run only on main branch

tag_build:
  stage: build
  script:
    - echo "Building for release..."
  only:
    - tags         # Run only when a Git tag is pushed

mr_check:
  stage: test
  script:
    - echo "Running checks for MR..."
  only:
    - merge_requests   # Run only for merge requests
```

---

3. except

You use `except` when you want to **exclude some cases**.
Examples:

```
test_job:
  stage: test
  script:
    - echo "Running tests..."
  except:
    - main         # Run everywhere except main branch

dev_build:
  stage: build
  script:
    - echo "Building dev version..."
  except:
    - tags         # Do not run for Git tags
```

---

4. Combining `only` and `except`

- GitLab will first check `only`.
- If `only` matches, the job is considered for execution.
- Then `except` is checked. If it matches, the job is **excluded**.

Example:

```
special_job:
  stage: test
  script:
    - echo "Special job"
  only:
    - branches
  except:
    - main
```

👉 Meaning: run this job for **all branches except `main`**.

---

5. ⚠️ Deprecation Notice

- `only` and `except` are **older syntax**.
- GitLab recommends using **`rules:`** instead, because rules are **more powerful and flexible**.
  Example with `rules`:

```
deploy_job:
  stage: deploy
  script:
    - echo "Deploying..."
  rules:
```

```
- if: '$CI_COMMIT_BRANCH == "main"'
```

With `rules:` you can:

- Use conditions (`if:` with variables like `$CI_COMMIT_BRANCH`)
- Control when to run (`when:` → `always`, `manual`, `never`, `on_success`, `on_failure`)
- Combine multiple conditions.

```
# New
test_job:
  stage: test
  script:
    - echo "Running tests..."
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: never   # explicitly skip job
    - when: always  # otherwise run


# New
tag_build:
  stage: build
  script:
    - echo "Build for release"
  rules:
    - if: '$CI_COMMIT_TAG'   # variable exists if commit is a tag


# New
mr_check:
  stage: test
  script:
    - echo "MR checks"
  rules:
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
```

. What are variables in GitLab CI/CD?

- Variables are **key–value pairs** you can use in your `.gitlab-ci.yml` or in your jobs.

Example:

```
job1:
  stage: test
  script:
    - echo "Project name is $CI_PROJECT_NAME"
```

-
- GitLab already provides many **predefined variables** (e.g., `$CI_COMMIT_BRANCH`, `$CI_PROJECT_DIR`).

But you can also **define your own variables**.

---

2. Defining Variables in `.gitlab-ci.yml`

You can set variables globally or per job.

Global variables (apply to all jobs)

```
variables:
  APP_ENV: "development"
  TIMEOUT: "30"
```

Job-specific variables

```
deploy_job:
  stage: deploy
  variables:
    APP_ENV: "production"
  script:
    - echo "Deploying in $APP_ENV"
```

---

3. Setting Variables in GitLab UI (project settings)

This is where **secrets** usually go (like passwords, tokens, API keys).

1. Go to your project in GitLab.
2. Navigate to:
   Settings → CI/CD → Variables → Expand.
3. Click **“Add variable”**.
4. Fill in:
   - **Key** → the variable name (e.g., `AWS_ACCESS_KEY_ID`)
   - **Value** → the secret (e.g., your AWS key)
   - Options:
     - **Masked** → hides the value in job logs (recommended for secrets).
     - **Protected** → only available in protected branches/tags (like `main` or release tags).

After this, the variable is available inside jobs as an **environment variable**.

Example usage in a job:

```
deploy_job:
  stage: deploy
  script:
    - echo "Using AWS key $AWS_ACCESS_KEY_ID"
```

---

4. Group-level and Instance-level Variables

- **Group variables** → defined once, used by all projects in a group.
- **Instance variables** → set by GitLab admin, available for all projects in an instance.

This avoids duplicating secrets across projects.

---

5. Secret Files

Sometimes secrets are too large for simple variables (like JSON service accounts, SSH keys, etc.).
You can store them as **file variables**:

- In GitLab UI, check **"Type: File"** when adding a variable.
- In your job, GitLab will **create a temporary file** with that content, and set the variable to the file path.

Example:

```
deploy_job:
  stage: deploy
  script:
    - cat "$GOOGLE_AUTH_JSON"   # prints contents of secret file
```

---

6. Best Practices

- **Never hardcode secrets** in `.gitlab-ci.yml`.
- Use **masked + protected variables** for sensitive data.
- Use **file variables** for large configs.
- Restrict variable scope if only needed for certain environments (e.g., production).

✅ **In summary:**

- Use `variables:` in `.gitlab-ci.yml` for non-sensitive configs.
- Use GitLab **Settings** → **CI/CD** → **Variables** for secrets.
- Mark them as **masked + protected**.

- Access them in jobs with `$VAR_NAME`.

What is `when:` in GitLab CI/CD?

In your `.gitlab-ci.yml`, each job can have a **`when:`** instruction.
It tells GitLab **when the job should run** (depending on pipeline conditions).

🔹 Common `when:` options

1. **`when: on_success` (default)**
   👉 Run the job only if all previous jobs succeed.

```
test_job:

  script: echo "Running tests"

  when: on_success
```

💡 If you don't write `when: ...`, this is the default.

---

2. **`when: on_failure`**
   👉 Run the job only if a previous job fails.

```
notify_failure:

  script: echo "Tests failed, sending alert"

  when: on_failure
```

---

3. **`when: always`**
   👉 Run the job whether previous jobs succeed or fail.

```
cleanup:

  script: echo "Cleaning up temp files"

  when: always
```

---

4. **`when: manual`**
   👉 Job will **not run automatically**. A person must click "Play" in the GitLab UI.

```
deploy_to_staging:

  script: echo "Deploying to staging"

  when: manual
```

💡 Useful when you want control over deployment.

---

5. **`when: delayed`**
   👉 Run the job after a set time delay.

```
delayed_job:

  script: echo "Running after 30 seconds"

  when: delayed

  start_in: 30 seconds
```

🔹 Example putting it all together

```
stages:
```

```yaml
  - build
  - test
  - deploy
  - notify

build_job:
  stage: build
  script: echo "Building project"

test_job:
  stage: test
  script: echo "Running tests"

deploy_job:
  stage: deploy
  script: echo "Deploying..."
  when: manual   # run only when you click "Play"

notify_job:
  stage: notify
  script: echo "Notifying team"
  when: on_failure  # only runs if something failed
```

✅ **Summary for beginners**

- `on_success` → run if previous jobs succeed (default).
- `on_failure` → run if something failed.
- `always` → run no matter what.
- `manual` → run only if you click it.
- `delayed` → run later after some delay.

🔹 `allow_failure`

Normally, if a job **fails**, the pipeline also **fails** ❌.
But if you set `allow_failure: true`, the job may fail, and the pipeline will still continue ✅.

👉 Example:

```
lint:

  script: echo "Running linting... but may fail"

  allow_failure: true
```

- If `lint` fails, the pipeline **does not stop**.
- Useful for **optional jobs** (e.g., experimental checks, non-blocking tests).

🔹 Other important control flags/options

1. `retry`

Sometimes jobs fail randomly (e.g., network issues). You can tell GitLab to retry automatically.

```yaml
integration_test:

  script: run-integration-tests.sh

  retry: 2   # retry up to 2 times if it fails
```

---

2. `timeout`

You can set how long a job should run before being killed.

```yaml
long_task:

  script: do-heavy-processing.sh

  timeout: 30m   # 30 minutes
```

---

3. `interruptible`

Cancels running jobs if a new pipeline is triggered (saves resources).

```yaml
build:

  script: make build

  interruptible: true
```

💡 Example: if you push a new commit, the old build is canceled automatically.

---

4. `parallel`

Runs multiple instances of the same job in parallel (for faster pipelines).

```yaml
test:
```

```
  script: run-tests.sh
  parallel: 5   # split into 5 parallel jobs
```

---

5. `resource_group`

Prevents multiple jobs from the same group running at the same time (avoids conflicts).

```
deploy:
  script: deploy.sh
  resource_group: production
```

💡 Useful to ensure **only one deploy** to production happens at a time.

🔹 Summary (easy view)

- **`allow_failure: true`** → job may fail, pipeline continues.
- **`retry: N`** → auto retry job if it fails.
- **`timeout: 30m`** → stop job if it runs too long.
- **`interruptible: true`** → cancel old jobs when new ones start.
- **`parallel: N`** → run multiple copies of a job.
- **`resource_group:`** → make jobs run one at a time in a group.

Example: tests in parallel

```
test:
  script: run-tests.sh
```

```
  parallel: 3
```

When the pipeline runs:

- GitLab creates **3 jobs**: `test 1/3`, `test 2/3`, `test 3/3`.
- They all run the same script (`run-tests.sh`) **in parallel**.
- Useful for splitting heavy work.

🔹 Basic: `parallel` + `rules`

You just define `rules:` normally, and inside the job you can still say `parallel: N`.

Example: run tests only on `main` branch, and split into 3 jobs.

```
test:
  script: run-tests.sh
  parallel: 3
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: always
    - when: never   # don't run on other branches
```

👉 On `main` → creates 3 parallel jobs.
👉 On other branches → no job at all.

🔹 Smarter: `parallel:matrix` with `rules`

You can also make **matrix jobs** with `rules`.

Example: run tests on multiple Python versions, but only if changes touch `backend/` folder.

```
test:
  script: pytest
  parallel:
    matrix:
      - PYTHON: ["3.8", "3.9", "3.10"]
  rules:
    - changes:
        - backend/**/*   # only if files in backend changed
```

👉 If backend code changes → GitLab spawns 3 jobs (`PYTHON=3.8, 3.9, 3.10`).
👉 If only frontend changes → no jobs created.

🔹 Advanced: conditional matrix

You can even control **matrix values with rules**.

Example:

- On `main` branch → run tests in 3 Python versions.
- On feature branches → only latest version.

```
test:
  script: pytest
  parallel:
    matrix:
```

```
    - PYTHON: ["3.10"]
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      variables:
        PYTHON: "3.8,3.9,3.10"   # override matrix values
```

1. What is Cache in GitLab CI/CD?

- In GitLab pipelines, each job usually runs in a **fresh environment** (new container, VM, or runner).
- That means any files created in one job are **lost** after the job finishes.
- **Cache** lets you **store files** so they can be **reused by later jobs or future pipeline runs**.

👉 Think of it like a **temporary storage** that speeds things up by avoiding re-downloading or re-installing things every time.

---

2. What is Cache Used For?

Common use cases:

- Dependencies (e.g., node_modules/, .m2/ for Maven, vendor/ in PHP).
- Build tools downloads (e.g., Gradle wrapper, pip cache).

- Large libraries or assets that don’t change often.

It helps pipelines run **faster** and **use fewer resources**.

---

3. Basic Example

Here’s a Node.js example:

```
stages:

  - install

  - test


install_dependencies:

  stage: install

  script:

    - npm install

  cache:

    paths:

      - node_modules/


run_tests:

  stage: test

  script:

    - npm test
```

```
  cache:
    paths:
      - node_modules/
```

Explanation:

- First job (install_dependencies) runs npm install and caches node_modules/.
- Second job (run_tests) **reuses the cached node_modules/**, so it doesn't reinstall packages.

---

4. Cache vs Artifacts (Important!)

- **Cache** → for speeding up jobs across stages or pipelines (dependencies, downloads).
- **Artifacts** → for saving **job outputs** (build results, test reports, binaries) that you want to **access later or download**.

👉 Cache is about **reusing**; artifacts are about **keeping results**.

---

5. Cache Key

You can define a **cache key** to control when a cache should be reused.

Static key (always same cache)

```
cache:
```

```
  key: common-cache
  paths:
    - node_modules/
```

Per-branch cache

```
cache:
  key: "$CI_COMMIT_REF_NAME"
  paths:
    - node_modules/
```

👉 Different cache for each branch.

Automatic key (based on files)

```
cache:
  key:
    files:
      - package-lock.json
  paths:
    - node_modules/
```

👉 If package-lock.json changes, a new cache is created.

---

6. Cache Policy

You can control cache download/upload behavior with policy:

- pull-push (default) → download cache at start, update at end.
- pull → only download cache.
- push → only upload cache.

Example:

```
cache:
  key: my-cache
  paths:
    - node_modules/
  policy: pull   # don't overwrite cache, only use existing
```

---

7. Cache Limitations

- Cache is stored on the **runner's cache storage** (shared runners may not always share cache between projects).
- Max cache size is **5 GB per project** by default.
- Cache is not guaranteed to persist forever (GitLab may clean old caches).

✅ **In summary:**

- `Cache in GitLab CI/CD = temporary storage for speeding up jobs.`
- `Useful for dependencies and downloads.`
- `Defined with cache: paths: [ ... ].`
- `Controlled with key and policy.`
- `Different from` **`artifacts`**`, which are for saving outputs.`

1. The Challenge

By default, GitLab CI/CD jobs run **inside containers** (if using Docker executor).
But:

- You **can't run Docker-in-Docker (dind)** commands directly unless configured.
- So we need a way to let the pipeline **use Docker** to build images.

---

2. Two Main Approaches

A. Docker-in-Docker (DinD) (most common)

You use a **Docker service container** (`docker:dind`) along with the job.

- `docker:dind` runs the Docker daemon.
- Your job uses the Docker CLI to talk to that daemon.

B. Docker socket binding (faster, but less secure)

You mount the host's `/var/run/docker.sock` into the job container.

- The job uses the **host's Docker daemon** directly.

- Faster, but jobs get host-level privileges.

---

3. Example: Docker-in-Docker (Recommended by GitLab)

Here's a `.gitlab-ci.yml` that builds and pushes a Docker image:

```
stages:
  - build

build_image:
  stage: build
  image: docker:24.0           # Docker CLI client
  services:
    - docker:24.0-dind         # Docker daemon (DinD)
  variables:
    DOCKER_HOST: tcp://docker:2375/
    DOCKER_TLS_CERTDIR: ""     # Disable TLS (simplifies setup)
  script:
    - docker version
    - docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA .
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
$CI_REGISTRY
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA
```

Explanation:

- `image: docker:24.0` → uses Docker CLI.
- `services: docker:24.0-dind` → starts a Docker daemon container.
- `DOCKER_HOST: tcp://docker:2375/` → tells CLI where to find daemon.
- `$CI_REGISTRY_*` → predefined GitLab variables for container registry.
- `docker build ...` → builds image.

- `docker push ...` → pushes to GitLab Container Registry.

---

4. Example: Docker Socket Binding (Faster but Risky)

This works only with **GitLab Runner on your own server** (not shared runners).

```
build_image:
  stage: build
  image: docker:24.0
  variables:
    DOCKER_HOST: unix:///var/run/docker.sock
  script:
    - docker build -t myapp:$CI_COMMIT_SHORT_SHA .
    - docker push myapp:$CI_COMMIT_SHORT_SHA
```

Here you mount `/var/run/docker.sock` into the job in runner config. The job talks directly to the host's Docker daemon.

---

5. Pushing to GitLab Container Registry

Each GitLab project has its own registry.
Registry URL:

```
$CI_REGISTRY/$CI_PROJECT_PATH
```

For example:

```
registry.gitlab.com/mygroup/myproject
```

Login & push:

```
docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
$CI_REGISTRY
docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA .
```

```
docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA
```

✅ **Summary:**

- Use **`docker:dind`** service in CI jobs to build Docker images.
- Or use **socket binding** if you control the runner.
- Push images to **GitLab Container Registry** using built-in `$CI_REGISTRY_*` variables.

A. Shared Runner

- Provided by GitLab itself (on GitLab.com SaaS) or by an admin in self-hosted GitLab.
- Available to **all projects** in the GitLab instance.
- Good for small projects or when you don't want to manage infrastructure.
- Downside → limited concurrency, slower, no customization (everyone shares the same pool).

👉 Example: If you create a new project on GitLab.com and run a pipeline without installing anything, it uses a **shared runner**.

B. Group Runner

- Installed and registered by you (or your admin) at the **group level**.
- Shared among **all projects in a GitLab group**.
- Useful if you have multiple related projects and want consistent CI/CD setup (same tools, same environment).

👉 Example: You create a group called `my-company/` with multiple repos (`frontend`, `backend`, `infra`) and register one runner for all.

C. Specific Runner

- Registered for **one project only**.
- Fully under your control (tools, resources, permissions).
- Best for projects that need special setup (e.g., CUDA/GPU builds, custom dependencies, access to internal infra).

👉 Example: You have a project that needs **NVIDIA GPU drivers** for ML pipelines, so you install a specific runner on a GPU server only for that project.

D. Kubernetes Executor

- One of the **executor types** GitLab Runner can use.
- Instead of running jobs on a VM or Docker container directly, it spins up a **temporary Pod in Kubernetes** for each CI/CD job.
- Benefits:
    - Scales automatically (each job = new pod).
    - Uses Kubernetes resources efficiently.
    - Isolated and secure (jobs run in separate pods).
- Downsides: needs a working Kubernetes cluster + more setup.

👉 Example: In a large team, you integrate GitLab Runner with your company's Kubernetes cluster → each pipeline job runs in a fresh pod.

A. Standard Variables

- Basic **key–value pairs** you define.

- Can be set:
    - In `.gitlab-ci.yml` (non-secret config)
    - Or in **Project → Settings → CI/CD → Variables**
- Available in **all jobs** by default.

👉 Example (in `.gitlab-ci.yml`):

```
variables:
  APP_ENV: "dev"
```

Usage:

```
script:
  - echo "Running in $APP_ENV"
```

B. Group Variables

- Defined at the **group level** (instead of project).
- Automatically inherited by **all projects** in that group.
- Useful when multiple projects need the same config (like cloud keys or repo URLs).

👉 Example:

- You have a GitLab group `mycompany/` with projects (`frontend`, `backend`).
- You set `AWS_REGION=ap-south-1` as a **group variable**.
- Both projects can now use `$AWS_REGION` without redefining it.

C. Protected Variables

- Variables marked as **protected** can only be accessed in pipelines that run on **protected branches/tags** (like `main`, `production`).
- Prevents secrets (like prod DB password) from leaking in dev branches or forks.

👉 Example:

- `DB_PASSWORD=supersecret` is **protected**.
- A pipeline on `main` can access it.
- A pipeline on `feature-xyz` **cannot**.

This keeps production secrets safe.

D. Masked Variables

- Secrets that should **never appear in job logs**.
- If a masked variable is echoed accidentally, it shows as `****`.
- Must follow certain rules:
  - At least 8 characters long.
  - Only `a-z A-Z 0-9 _ = - /` + allowed.

👉 Example:

- `API_KEY=ABC123XYZ` is masked.

If a script does `echo $API_KEY`, logs show:

```
****
```

-

✅ **Quick Comparison**

Type	Where Used	Purpose
Standard Variables	Project or `.gitlab-ci.yml`	General configs (env, version, timeout)
Group Variables	Group level	Share configs across projects
Protected Variables	Project/Group	Only available in protected branches/tags
Masked Variables	Project/Group	Hide secrets from logs

1. Problem Before `needs:`

- By default, GitLab pipelines run:
  - **Stages sequentially** (build → test → deploy).
  - **Jobs inside a stage concurrently**.

This means a job in stage **B** can only start after **all jobs** in stage **A** finish, even if it only depends on one of them.
👉 That can waste time.

---

2. What is `needs:`?

- The **`needs:` keyword** lets you define **direct job-to-job dependencies**, instead of being forced to wait for the whole previous stage.
- This allows **parallel execution across stages** and speeds up pipelines.

👉 Think of it as:
*"This job **needs only X**, not everything before it."*

---

3. Example Without needs

```
stages:
  - build
  - test
  - deploy

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

test_job1:
  stage: test
  script: echo "Unit tests"

test_job2:
  stage: test
  script: echo "Integration tests"

deploy_job:
  stage: deploy
  script: echo "Deploying..."
```

Execution order:

1. `build_job` runs.
2. After it finishes, **both `test_job1` and `test_job2`** run in parallel.
3. Only after both tests finish → `deploy_job` starts.

Even if `deploy_job` only needs `test_job1`, it still waits for `test_job2`.

---

4. Example With needs

```
stages:
  - build
```

```
  - test
  - deploy

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

test_job1:
  stage: test
  script: echo "Unit tests"

test_job2:
  stage: test
  script: echo "Integration tests"

deploy_job:
  stage: deploy
  needs: ["test_job1"]   # depend only on test_job1
  script: echo "Deploying..."
```

Execution order:

- `build_job` runs.
- Then `test_job1` and `test_job2` run in parallel.
- As soon as `test_job1` finishes, **`deploy_job` starts** (no need to wait for `test_job2`).

👉 This shortens pipeline time.

---

5. Passing Artifacts with needs

- Normally, jobs in later stages can access artifacts from earlier stages.
- With `needs:`, you can also **download artifacts from specific jobs immediately**.

Example:

```
build_job:
  stage: build
  script: echo "Build artifact" > build.txt
  artifacts:
    paths: [build.txt]

deploy_job:
  stage: deploy
  needs: ["build_job"]    # also fetches build.txt
  script:
    - cat build.txt
    - echo "Deploying..."
```

---

6. Special Notes

- If you use `needs: []` (empty array), the job starts **immediately** without waiting for its stage.
- You can mix normal stage order and `needs:` overrides.
- Be careful: if you misuse it, you can create jobs that start too early and fail (because dependencies aren't ready).

✅ **Summary:**

- `needs:` lets you declare **job dependencies** instead of relying only on stage order.
- Jobs can start as soon as their dependencies finish, **even if other jobs in earlier stages are still running**.
- This makes pipelines **faster** and more **efficient**.