What is Gitlab?

Gitlab is a devops platform that provides you with so many features including version control, CI/CD, security, container registry etc.

Pipelines in GitLab CI/CD offer a variety of flexible configurations to tailor your software delivery process:

Basic Pipelines: Execute jobs within a stage concurrently, moving to the next stage only after all jobs in the current stage are complete.

DAG Pipelines: Optimize efficiency by running jobs based on dependencies, potentially reducing execution time compared to basic pipelines.

Merge Request Pipelines: Trigger specifically for merge requests, ensuring code quality and consistency before merging branches.

Merged Results Pipelines: Simulate the merged state of a merge request, providing early insights into potential conflicts or issues.

Merge Trains: Queue merge requests for sequential execution, ensuring a controlled and orderly integration process.

Parent-Child Pipelines: Manage complexity by breaking down extensive pipelines into a parent pipeline that orchestrates multiple child pipelines, often used in monorepo environments.

Multi-Project Pipelines: Coordinate pipelines across different projects, fostering collaboration and streamlining cross-project dependencies.

Gitlab Architecture:

The key Components of the GitLab CI/CD Architecture which were responsible for defining and executing a pipeline are Gitlab Server, Runner, and Executor.

GitLab Server

The GitLab server is the core component of the GitLab platform. It hosts your Git repositories and provides the web interface for managing projects, issues, merge requests, and CI/CD pipelines. It comes in 2 flavors - Gitlab SaaS vs GitLab Self-Managed

	GitLab Saas	GitLab self-managed
Pros	1. Easy to use 2. Regular updates 3. Managed security 4. Scalable 5. Highly available	1. Full control to customize as per needs 2. No vendor lock-in, easy to migrate 3. Total control over code and data 4. Cost-effective for large teams
Cons	1. Less control over platform configuration 2. Migrating to another platform can be complex and costly 3. Potential data privacy because of shared infrastructure 4. Can become expensive for large teams	1. Requires technical expertise to manage infrastructure, applying security updates, managing vulnerabilities, scaling and updates

GitLab Runner:

GitLab Runner is an application that works with GitLab CI/CD to run jobs in a pipeline. It's the agent that executes the CI/CD jobs and reports the results back to the GitLab server.

	Shared runners	Self-managed runners
Pros	1. Included with GitLab subscription 2. Scalable	1. Can be faster and more reliable than shared 2. Full control over the runner configuration, environment and security 3. More secure

Cons	1. No control over the runner configuration or environment. 2. May be slower due to competition for resources. 3. Potential security risk because of shared resources	1. Setup and maintenance complexity 2. Cost 3. Scalability

GitLab Executor

The executor is a part of the GitLab Runner that determines the environment in which your jobs run. It's responsible for preparing the environment, cloning the repository, and executing the job scripts. Choosing the right executor is crucial for ensuring efficient, secure, and reliable pipeline execution. We can think of executors as containers within the runner.

Runner Executor	Definition	Cons
Shell	Runs jobs on the local machine where the runner is installed. Simple and lightweight, it's suitable for quick scripts or testing basic functionality.	1. Shares runner's OS and resources, risking conflicts 2. Sharing resources might pose security risks with different access levels 3. Results may vary depending on runner's environment and installed tools
Docker	Runs each job in a separate and isolated Docker container. We can Choose from pre-built Docker images with specific tools and environments readily available.	1. Running multiple containers can consume more system resources on the runner. 2. Docker containers often have restricted access to the runner's native resources. 3. Requires additional configuration and understanding of Docker concepts.

Kubernetes	Runs jobs as pods in a Kubernetes cluster. This leverages the cluster's resource management and scaling capabilities for efficient execution and parallel processing.	1. Complex setup and maintenance 2. Need to know Kubernetes cluster configuration and management 3. Understanding Kubernetes best practices is crucial
Virtual Machine	Runs jobs on a VM (supports VirtualBox, Parallels, VMware). The VM executor creates a new virtual machine for each job execution, guaranteeing isolation and eliminating potential contamination from previous runs.	1. Requirea significant disk space, memory, and CPU resources 2. Setup and management is not easy
SSH	Runs jobs on a remote machine over SSH. It's useful for utilizing specific resources available on a different machine or integrating with pre-existing infrastructure.	1. Requires reliable network connectivity between runner and remote machine 2. Requires secure SSH access and configuration 3. Monitoring jobs on remote machine can be challenging
Custom	GitLab offers additional specialized executors like Parallels, Docker Machine, and even custom ones you can develop yourself. Each caters to specific needs and environments, ensuring flexibility and catering to diverse workflows.	

Do checkout official docs to help selecting the executor:
https://docs.gitlab.com/runner/executors/#selecting-the-executor

How They Work Together

Pipeline Trigger:

- A developer pushes code to the GitLab server
- The server detects the push and checks for a .gitlab-ci.yml file

Job Assignment:

- GitLab server looks for available runners
- An available runner picks up the job

Job Execution:

- The runner uses the specified executor to prepare the environment
- The executor clones the repository and runs the job scripts

Result Reporting:

- The runner reports job status and logs back to the GitLab server
- The server updates the pipeline status and notifies relevant parties

Artifact Handling:

- If configured, the runner uploads job artifacts to the GitLab server

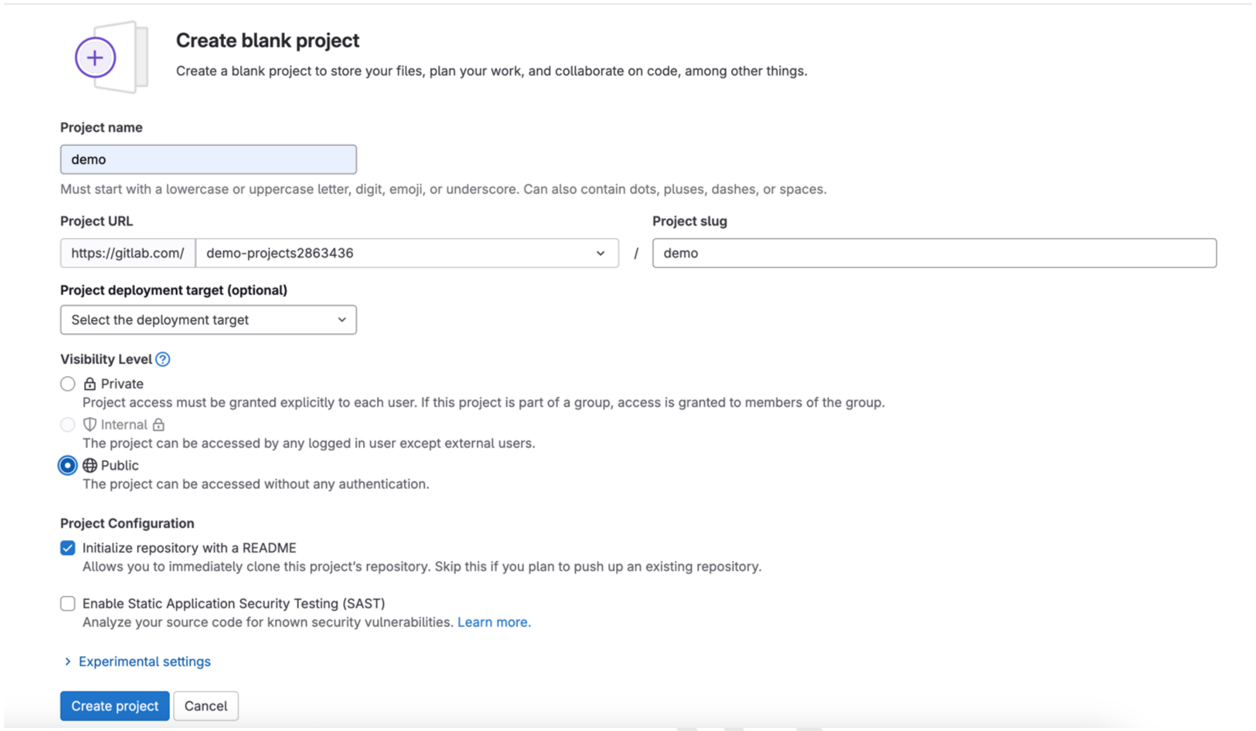
This process repeats for each job in the pipeline, potentially using different runners and executors based on the configuration and availability.

Pre-requisites: Git, CI/CD

Create a Gitlab account .

First CI/CD pipeline

1. From the homepage, create a project -> Blank project
2. Enter the project name and visibility details -> create project



Create blank project

Create a blank project to store your files, plan your work, and collaborate on code, among other things.

Project name: demo

Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.

Project URL: https://gitlab.com/ demo-projects2863436 / Project slug: demo

Project deployment target (optional): Select the deployment target

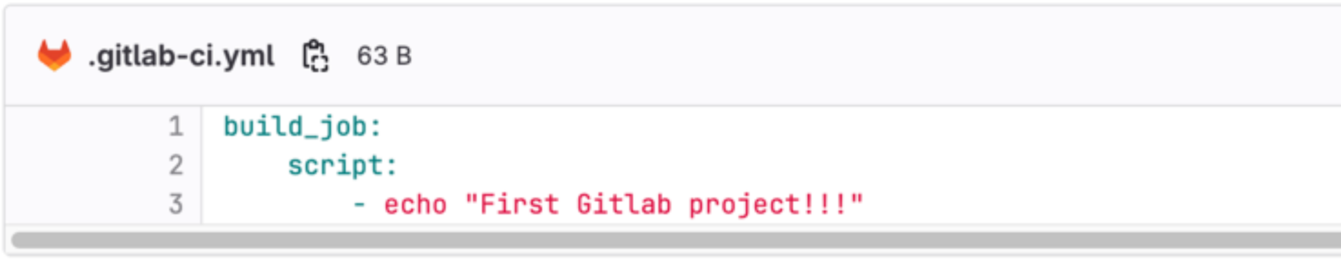
Visibility Level

- Private
  Project access must be granted explicitly to each user. If this project is part of a group, access is granted to members of the group.
- Internal
  The project can be accessed by any logged in user except external users.
- [✓ Public]
  The project can be accessed without any authentication.

Project Configuration

- [✓] Initialize repository with a README
  Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.
- [ ] Enable Static Application Security Testing (SAST)
  Analyze your source code for known security vulnerabilities. Learn more.

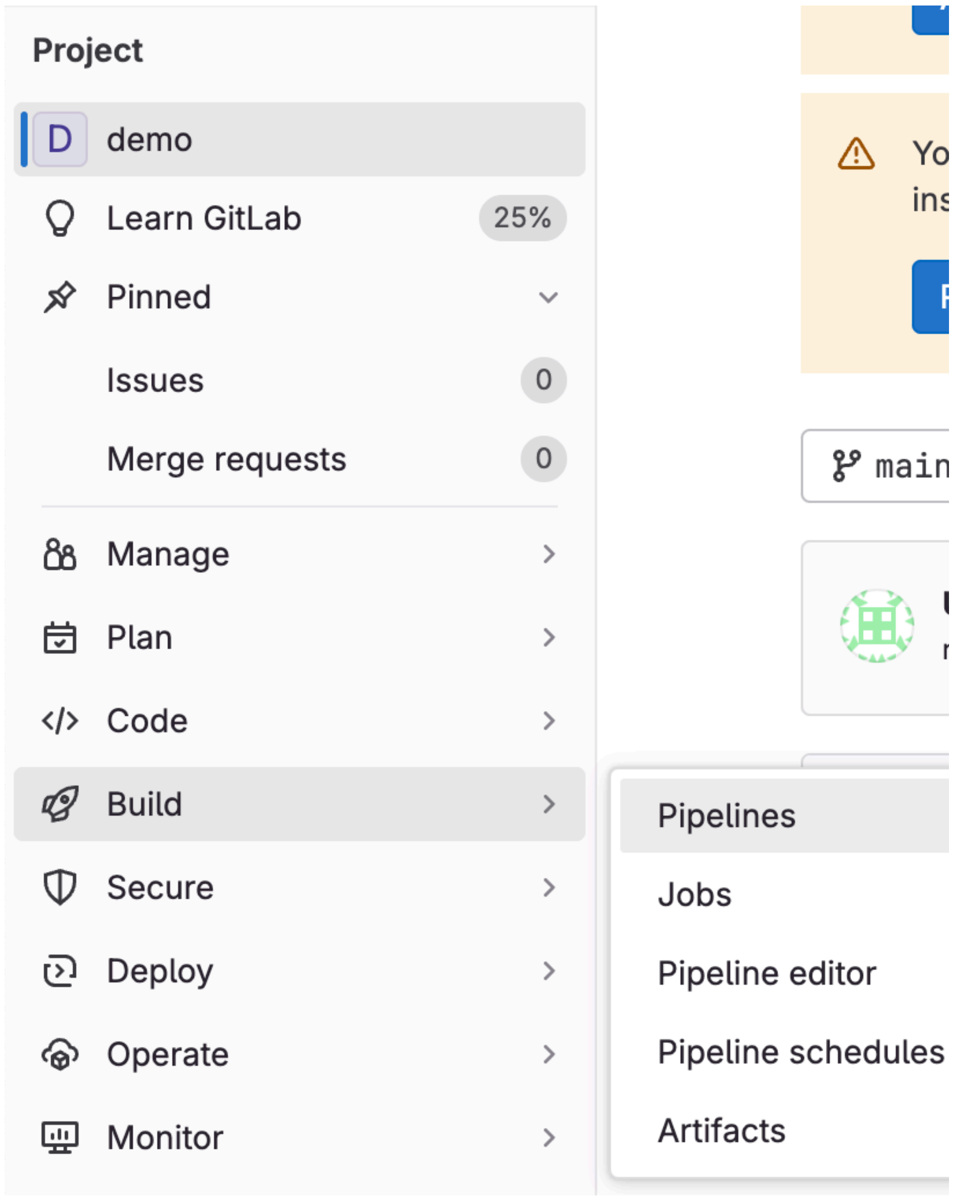
3. Create a new file at the root of the demo project. In Gitlab, to create CI/CD pipeline file should be names as .gitlab-ci.yml unless you change the name in settings
4. Inside this file, create jobs. Jobs are fundamental elements of a GitLab CI/CD pipeline. Jobs are configured in the .gitlab-ci.yml file with a list of commands to run to accomplish tasks like building, testing, or deploying code.
5. We will create a job to execute a script to print a message.



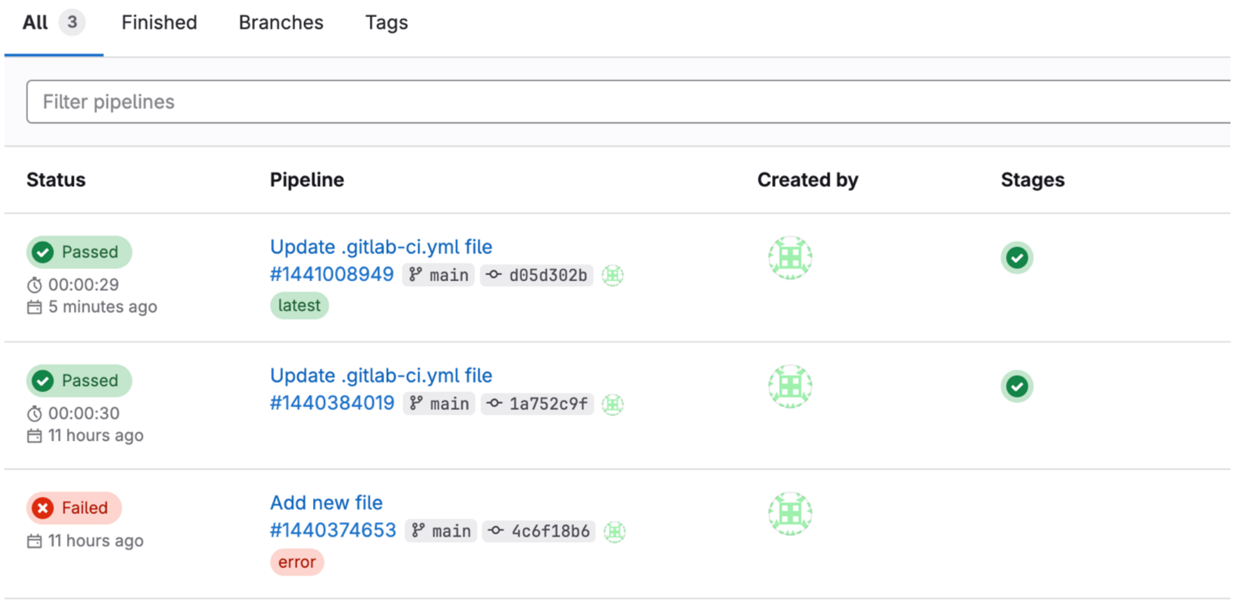
.gitlab-ci.yml 63 B

```yaml
build_job:
    script:
        - echo "First Gitlab project!!!"
```

6. Commit changes
7. Once the file is committed, click on the Build->pipelines to see the running pipelines



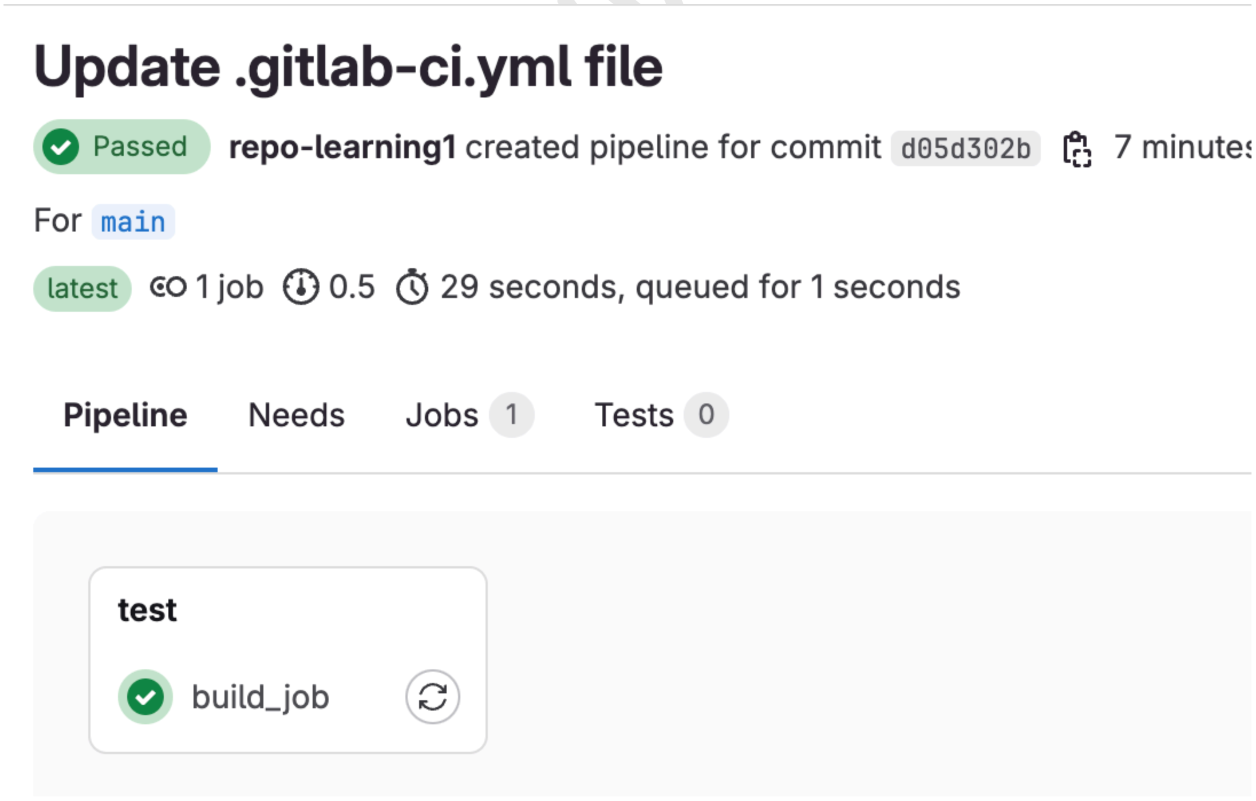
8. We see the Running, Passed or Failed status of the pipeline.


All 3 | Finished | Branches | Tags

Filter pipelines

Status	Pipeline	Created by	Stages
Passed 00:00:29 5 minutes ago	Update .gitlab-ci.yml file #1441008949 main d05d302b latest		✓
Passed 00:00:30 11 hours ago	Update .gitlab-ci.yml file #1440384019 main 1a752c9f		✓
Failed 11 hours ago	Add new file #1440374653 main 4c6f18b6 error		

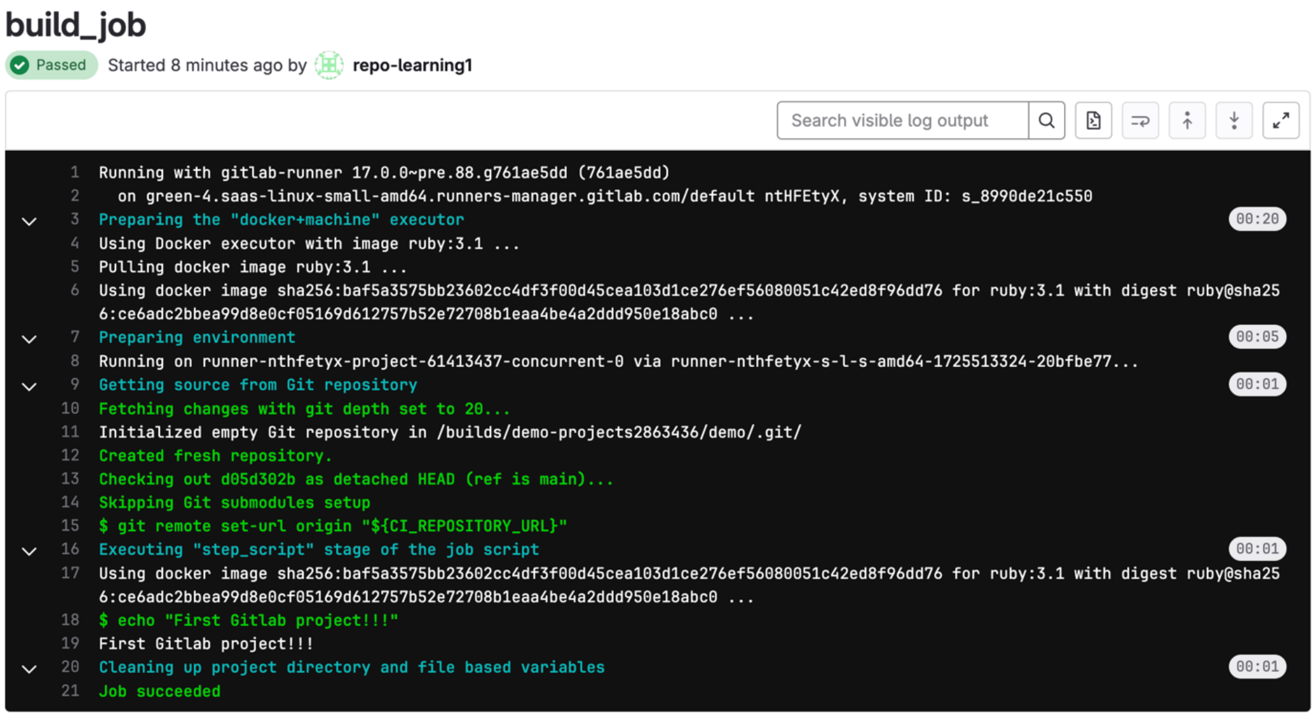
9. Click on the Passed status and the job name to see the results


Update .gitlab-ci.yml file

Passed repo-learning1 created pipeline for commit d05d302b 7 minutes

For main

latest 1 job 0.5 29 seconds, queued for 1 seconds


build_job

Passed Started 8 minutes ago by repo-learning1

Search visible log output

```
Running with gitlab-runner 17.0.0~pre.88.g761ae5dd (761ae5dd)
  on green-4.saas-linux-small-amd64.runners-manager.gitlab.com/default ntHFEtyX, system ID: s_8990de21c550
Preparing the "docker+machine" executor                                                            00:20
Using Docker executor with image ruby:3.1 ...
Pulling docker image ruby:3.1 ...
Using docker image sha256:baf5a3575bb23602cc4df3f00d45cea103d1ce276ef56080051c42ed8f96dd76 for ruby:3.1 with digest ruby@sha25
6:ce6adc2bbea99d8e0cf05169d612757b52e72708b1eaa4be4a2ddd950e18abc0 ...
Preparing environment                                                                              00:05
Running on runner-nthfetyx-project-61413437-concurrent-0 via runner-nthfetyx-s-l-s-amd64-1725513324-20bfbe77...
Getting source from Git repository                                                                 00:01
Fetching changes with git depth set to 20...
Initialized empty Git repository in /builds/demo-projects2863436/demo/.git/
Created fresh repository.
Checking out d05d302b as detached HEAD (ref is main)...
Skipping Git submodules setup
$ git remote set-url origin "${CI_REPOSITORY_URL}"
Executing "step_script" stage of the job script                                                    00:01
Using docker image sha256:baf5a3575bb23602cc4df3f00d45cea103d1ce276ef56080051c42ed8f96dd76 for ruby:3.1 with digest ruby@sha25
6:ce6adc2bbea99d8e0cf05169d612757b52e72708b1eaa4be4a2ddd950e18abc0 ...
$ echo "First Gitlab project!!!"
First Gitlab project!!!
Cleaning up project directory and file based variables                                             00:01
Job succeeded
```

From the logs, we can see that

1. It is running with gitlab-runner. Gitlab provides you with few shared runners that you can use. Runner is a machine which executes the script which you define in the job
2. Once the runners spin up, by default, GitLab Runner perform a clone of the repository to fetch the latest commit

3. Build is using docker Ruby image

4. Executed script echo statement

5. Job is succeeded, and pipeline is successful.

6. Click on Retry to re-run the job

We have successfully created simple CI/CD pipeline