Rapporti Tecnici INAF
INAF Technical Reports

Number	358
Publication Year	2026-01-07
Acceptance in OA@INAF	2026-01-15T11:42:09Z
Title	A small-scale infrastructure for GitLab CI/CD
Authors	URBAN, Cristiano, TAVAGNACCO, Daniele, SPONZA, Massimo, KNAPIC, Cristina, LANDONI, Marco
Affiliation of first author	O.A. Trieste
Publisher's version (DOI)	https://doi.org/10.20371/INAF/TechRep/358
Handle	http://hdl.handle.net/20.500.12386/46133

A small-scale infrastructure for GitLab CI/CD

Issue/Rev. No.: 1.0
Date: January 7, 2026
Authors: Cristiano Urban[a], Daniele Tavagnacco[a], Massimo Sponza[a], Cristina Knapic[a] and Marco Landoni[b]
Affiliations: [a]INAF - Istituto Nazionale di Astrofisica - Osservatorio Astronomico di Trieste, [b]INAF - Istituto Nazionale di Astrofisica - Osservatorio Astronomico di Brera
Approved by: Andrea Bignamini

Contents

1 Introduction 3

2 Test platform 3

3 Runner configuration 4

4 Exploiting dynamic child pipelines for repetitive tasks 5

5 Pull policy and concurrent pull of Docker images 7

6 Restrictions on Docker images 7

7 Monitoring hints 7

8 Final remarks and future developments 8

References 9

1 Introduction

Continuous Integration and Continuous Delivery/Deployment (CI/CD)[1] constitute a pillar of the present-day software development. Whether you need to compile, test, distribute your code, or simply automatically generate documentation, there is always a pipeline involved in one way or another. GitLab allows us to define a CI/CD pipeline in a relatively simple way, by filling in a file called `.gitlab-ci.yml` and saving it in the root of our repository. Runners[2] are the applications that are responsible for executing CI/CD jobs. More precisely, jobs are taken in charge by the runners and each runner is associated to a well-defined executor type (i.e. Shell, Docker, etc...). The IA2 group, in collaboration with other working groups under the parish of USC-C, has started to build up a simple test infrastructure whose purpose, once in production, would be to guarantee an adequate processing power in order to support the execution of several simultaneous CI/CD jobs.This kind of approach could help a lot our developers and researchers.

The methodology used to pursue our goal was to gain a deeper understanding of the configuration and operation of the runners, set up the necessary hardware and software infrastructure and finally create a repository on GitLab to run jobs in parallel and test various use cases.

Section 2 will describe the hardware and software specifications of the test platform. Section 3 will describe the installation and configuration of the runners in detail. Section 4 will show a simple example of using pipelines to dynamically generate a certain number of similar jobs. Sections 5 and 6 will describe the Docker image pull policies and some security aspects, respectively. Section 7 will briefly discuss some basic concepts concerning the monitoring part.

2 Test platform

In order to perform our tests, we set up four physical machines with a Intel(R) Xeon(R) CPU E5645 @ 2.40GHz (1 thread/core, 6 cores/socket and 2 sockets) and 96 GB of RAM (12 banks of 8 GB each one). The operating system installed is Rocky Linux stuck to version 8.9, because we also need compatibility with Lustre file system, since all the tests have been done at IA2 and Lustre is a parallel file system widely used for storage at IA2. On each of these nodes we configured two runners with a Docker executor. These runners are "Instance runners", namely globally visible from all the repositories present on the INAF GitLab installation. Each runner was configured to be able to handle four parallel jobs. In this way, with four nodes, we obtained a total amount of 32 parallel CI/CD jobs (4 jobs/runner, 2 runners/node and 4 nodes).

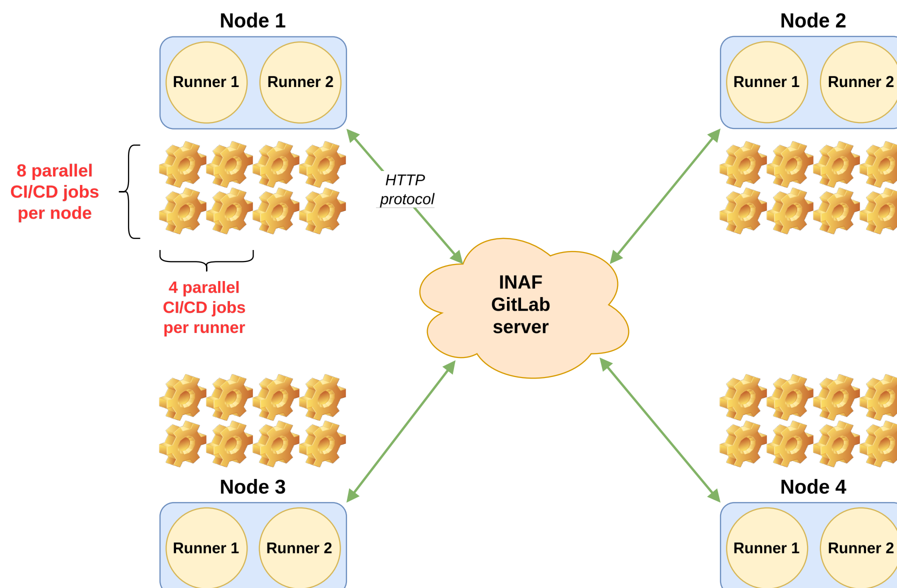

Figure 1: The test platform.

The choice to have multiple runners (two for each node) each responsible for a small number of jobs, instead of having a single runner on a single node, is due to the fact that in this way we can divide the workload across multiple nodes, but at the same time the single runner will be more reliable with a small number of jobs to manage. Anyway, testing different configurations goes beyond the purpose of this work.
The runners communicate with the INAF GitLab server through HTTP requests. The communication is always started by the runner. The runner constantly polls the GitLab server for jobs. When a CI/CD job is triggered, the GitLab server queues that job and the runner picks up the job on the next poll. The runner instantiates the executor and sends back to the GitLab server results, logs and status once the job is done. During the job execution phase, the executor downloads the source code and any artifacts (i.e. produced by a job and used by another job in a subsequent pipeline stage).

3 Runner configuration

First of all, we need the GitLab runner service up and running on each node. There are several ways to install that service, in this case we opted for the classical installation using the official GitLab repositories:

```
# Add the official GitLab repository
curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.rpm.sh"
    ↪ | sudo bash
# Install the gitlab-runner package
dnf install gitlab-runner
# Enable the systemd unit to run on startup and start the service immediately
systemctl enable --now gitlab-runner
```

We can identify three types of runners[3]:

- **Project**: can be registered by the owners or maintainers of a project and can be used within the single project (advantage: no need to ask the admin anything, you can install the runner directly on your laptop, this means maximum control)
- **Group**: can be registered and managed by group owners and are visible to all projects and subgroups
- **Instance**: can be registered and managed by GitLab admins and are visible to all projects and groups.

Our purpose is to share the runners to all the INAF users, so we opted for registering a bunch of Instance runners.
We can configure an Instance runner in two steps:

1. Create a new runner using the GitLab user interface (from the admin area: CI/CD→Runners→New instance runner)
2. Install/register the runner using the token obtained at point 1.

The registration command is the following (as root):

```
gitlab-runner register --url https://www.ict.inaf.it/gitlab --token YOUR_GITLAB_RUNNER_TOKEN
```

An interactive procedure starts and asks you to choose an executor. We choose Docker in our specific case, because it guarantees an isolated and reproducible environment. For Docker executor, you also need to specify a default image (e.g. `python:3.11`) which can be overridden with a different one using the `image` directive within the pipeline configuration in GitLab. At the end, the runner is registered and the configuration is stored under `/etc/gitlab-runner/config.toml`. The following snippet shows you a sample config file.

```
##### Global section #####
# limits how many jobs can run concurrently, across all registered runners
concurrent = 8
# interval length, in seconds, between the runner checking for new jobs
check_interval = 3

[session_server]
session_timeout = 1800
```

```
##### Specialized section for this specific runner #####
[[runners]]
  # runner name (specified during the registration)
  name = "MyRunner"
  # limits how many jobs can be handled concurrently by this registered runner
  limit = 4
  # limits number of concurrent requests for new jobs from the GitLab server
  request_concurrency = 4
  # GitLab server URL (specified during the registration)
  url = "https://www.ict.inaf.it/gitlab"
  # runner ID
  id = DDDD
  # runner authentication token (specified during the registration)
  token = "glrt-XXXXX"
  token_obtained_at = 2024-10-17T10:32:44Z
  token_expires_at = 0001-01-01T00:00:00Z
  # runner executor type (specified during the registration)
  executor = "docker"
  [runners.custom_build_dir]
  [runners.cache]
    MaxUploadedArchiveSize = 0
    [runners.cache.s3]
    [runners.cache.gcs]
    [runners.cache.azure]
  [runners.docker]
    tls_verify = false
    # restricted list of allowed images for the executor
    allowed_images = ["python:3.11","postgres:16"]
    # default image (specified during the registration)
    image = "python:3.11"
    # run the container in unprivileged mode
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    # pulls the executor Docker image only if it is not present locally (two tries)
    pull_policy = ["if-not-present", "if-not-present"]
    shm_size = 0
    network_mtu = 0
```

We usually want to place runners on nodes other than the one hosting our GitLab server. Potentially, we could install a runner even on our laptop or on an embedded device, such as a RaspberryPi.

4 Exploiting dynamic child pipelines for repetitive tasks

When we are dealing with the generation of a pipeline containing many identical jobs, it can be particularly useful to take advantage of dynamic child pipelines. Roughly speaking, we can state that these pipelines consist of two main jobs. The first one produces an artifact representing a pipeline definition: this pipeline contains our jobs configuration and is generated through instructions in any programming language (e.g. a Bash script). The second job builds a trigger to launch our jobs.

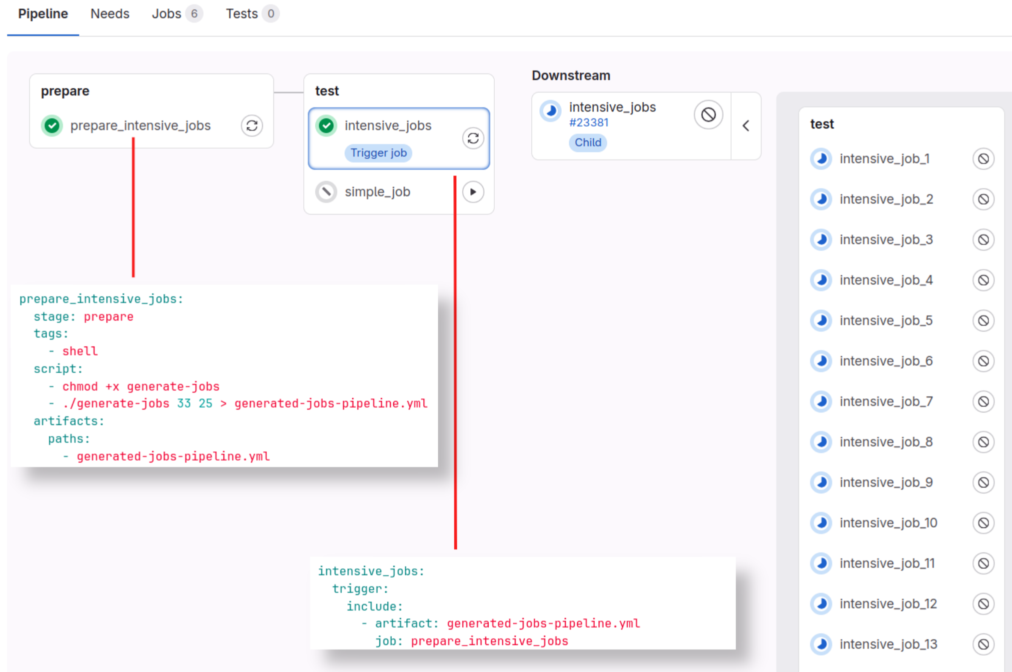

Figure 2: Dynamic child pipelines.

The following Bash script is called to generate the pipeline definition dynamically. It requires two arguments:

- NUM_JOBS: the number of repetitive jobs definitions to generate
- JOB_EXEC_TIME: the desired job execution time in seconds.

In the above picture we have generated 33 identical job definitions and each one of those "intensive" job runs for 25 seconds. What happens is that 32 jobs are taken in charge almost immediately and the remaining one waits until one of these 32 jobs completes. Moreover, you must take into account that there is a fair algorithm[4] that assigns jobs based on the projects having the lowest number of jobs already running on Instance runners. This is important because prevents one or more projects to occupy all the runners resources.

```
#!/bin/bash

# Number of jobs
NUM_JOBS=$1

# Desired job execution time in seconds
JOB_EXEC_TIME=$2

cat <<EOF
.test_job:
stage: test
tags:
- git-run-ia2
rules:
- if: '\$CI_COMMIT_REF_NAME == "main"'
when: manual

EOF

for job in $(seq 1 $NUM_JOBS); do
cat <<EOF
intensive_job_${job}:
extends: .test_job
```

```
script:
- apt-get update && apt-get install -y psmisc
- yes > /dev/null &
- sleep ${JOB_EXEC_TIME}
- killall yes
- echo "Job done."

EOF
done
```

5 Pull policy and concurrent pull of Docker images

The runners with a Docker executor allow us to specify the so-called "pull policy"[5] for the images we want to use. The default policy is to always pull images (i.e. `pull_policy = "always"`). Of course, this is not the best choice in terms of efficiency and, in fact, there is the possibility to choose another policy that allows to pull an image only if that image is not already stored locally (i.e. `pull_policy = "if-not-present"`). However, we have proven that in the case of concurrent pull, namely automatically launching all the 32 jobs, the first-time pull is attempted by all the jobs at the same time. It is possible to set more than one policy or repeat the same policy, in the eventuality the pull operation fails (e.g. `pull_policy = ["if-not-present", "if-not-present"]`).

6 Restrictions on Docker images

Giving unrestricted access to all the public available Docker images could pose a potential security risk. Within the runner configuration file we can define a list containing the names of the allowed images[5] (e.g. `allowed_images = ["python:3.11", "postgres:16"]`). Next, if we override the `image` setting in our pipeline configuration file with an image that is not in that list, the pipeline will fail.

7 Monitoring hints

GitLab Runner comes with a ready-to-go Prometheus[6] exporter, an endpoint that exposes some metrics in a human-readable text-based format. A Prometheus metric[7] entry has two attributes: HELP and TYPE.
HELP provides a description of the metric.
TYPE defines the metric type:

- counter: for metrics that describe the number of times a given event has occurred (e.g. number of requests served by a web application)
- gauge: for metrics whose values can fluctuate (e.g. CPU usage)
- histogram: for metrics where values describe time durations or dimensions.

Here is an example of a metric entry:

```
# HELP gitlab_runner_jobs_total Total number of handled jobs
# TYPE gitlab_runner_jobs_total counter
gitlab_runner_jobs_total{runner="XXXXXXXX",system_id="XXXXXXXXXXXXXX"} 122
```

The metrics endpoint can be enabled by adding the following line in the global section of the GitLab runner configuration file (`/etc/gitlab-runner/config.toml`):

```
listen_address = "0.0.0.0:9252"
```

At this point the metrics are exposed at `http://RUNNER_NODE_IP_ADDRESS:9252/metrics` and can be scraped by a properly configured Prometheus server.

8 Final remarks and future developments

In this document we have described the implementation of a very simple support infrastructure for GitLab CI/CD. The results obtained from this experiment are satisfactory. We can say that we have achieved our goal, even if we are only in a prototype stage for now. Possible future developments may include automatic runner registration via GitLab APIs and Kubernetes integration.

References

[1] *Gitlab CI/CD*. URL: https://docs.gitlab.com/ee/ci/.

[2] *GitLab runners*. URL: https://docs.gitlab.com/runner/.

[3] *GitLab runner types*. URL: https://docs.gitlab.com/runner/#who-has-access-to-runners-in-the-gitlab-ui.

[4] *Instance runners fair algorithm*. URL: https://docs.gitlab.com/ci/runners/runners_scope/#how-instance-runners-pick-jobs.

[5] *GitLab runner advanced configuration*. URL: https://docs.gitlab.com/runner/configuration/advanced-configuration/.

[6] *Prometheus*. URL: https://prometheus.io/.

[7] *Prometheus Metrics*. URL: https://prometheus.io/docs/concepts/metric_types/.