Introduction to CI@NERSC

Automating code integrations through Cori

Aditya Kavalur
UEG – NERSC & SCIC - ECP
July 7, 2021

Goals: What would count as a successful tutorial

If you all walk away with

- Broad understanding of GitLab and Continuous Integration (CI)
- Grasp of GitLab CI fundamentals
- Successful execution of basic hands-on-examples
- Familiarization with NERSC GitLab instance
- Knowledge about useful CI resources

Overview

- GitLab, CI and ECP-CI
- NERSC GitLab instance
- Fundamentals of CI in GitLab
- Examples

GitLab, CI and ECP-CI

NERSC

BERKELEY LAB

U.S. DEPARTMENT OF ENERGY | Office of Science

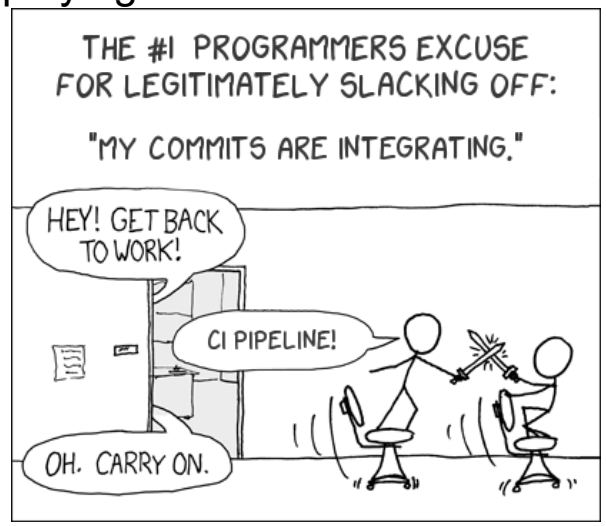
Gitlab = git + more features

- Gitlab is primarily a git-based code hosting service.
- Aims to provide a platform for the entire DevOps lifecycle.
- It offers a lot of useful features beyond git viz:
  - Project wiki
  - Issue tracking
  - Project/team management
  - Continuous Integration

Other platforms viz. Github, Bitbucket also offer similar features.

Continuous Integration: Automating the busy work

- CI is a best practice in DevOps.
  - An event trigger (commit/pull-request/time periodicity etc) sets off a range of jobs that can encompass building, testing, packaging and deploying the source code that git is managing.
- GitLab enables CI workflows through its `runner` utility.
- A runner is an active process running on a target machine, that is continuously polling the GitLab server for jobs.
- Users write CI configuration file in YAML.
- The CI YAML file specifies one or more acceptable runners for its execution through tags. The first acceptable runner that polls the server receives the job.



Source: https://xkcd-excuse.com

Continuous Integration: Automating the busy work

- The runner executes the script and sends the results back to the Gitlab server.
- CI workflows in Gitlab are highly configurable, they can be customized for different activities using its rules functionalities viz:
  - Branch
  - Event trigger
  - Target machine

HPC customization for CI: ECP-CI

- CI is offered for free in a lot of web servers viz. gitlab.com, so why HPC?
  - These CI jobs typically run in a container on a cloud
  - Developers may want to test their codes on facility infrastructure because that's where the users run the bulk of their jobs.
  - Developers may want to leverage the compute hours they have on the HPC centers to run expansive tests.
- Runners can perform certain common and recurring actions for typical workflows before executing the users scripts. These common actions can be leveraged through using a particular executor type of the runner.

HPC customization for CI: ECP-CI

- Standard GitLab runner has a few executor options
  - SSH, Shell, Docker, Kubernetes etc.
  - None of these perfectly fit the HPC workflows and security requirements

  Hold on! Shell executor seems like a good fit for HPC environments.
  - In this executor GitLab runner, spawns a shell where user scripts will be executed.
  - However! The executor falls short on 2 counts:
    - It does not consider a multi-tenant environment. Therefore, during the initial git fetch, it passes tokens through the command line. Anyone running `ps` can therefore view your job token, which allows them to read all your repositories.
    - It does not automate the interaction with the scheduler. Users have to write their own scripts to submit jobs to the scheduler, poll them till completion and pass return codes.

HPC customization for CI: ECP-CI

- ECP-CI developed its custom executor as part of the ECP project (https://ecp-ci.gitlab.io/)
- This ECP customization allows users to
  - Submit CI jobs to the scheduler.
    - Users provide scheduler parameters as a CI variable instead of in a shell file that you run sbatch with. The runner submits the user script as a job and polls the scheduler.
  - Run securely on multi-tenant systems.
    - The runner uses the environment variable GIT_ASKPASS to provide tokens for git commands.
  - Run CI pipelines as themselves through a common runner.
    - A common runner picks up jobs from the server. Views the GitLab username and downscopes to the same user on the target machine before executing the user provided scripts.

NERSC Gitlab instance

The server: software.nersc.gov

- NERSC hosts an instance at software.nersc.gov
  - It is a development service.
  - Users are also added to a google group for urgent/important notifications.
  - Server maintenance coincides with monthly Cori maintenance to minimize downtime
- Login with NERSC credentials
  - No forms for access!!!
- Documentation is in the banner at the top of each page (https://software.nersc.gov/NERSC/nersc-ci-docs).
- It include instructions for when you first login -> add your email to your profile.
  - This will alert you if a pipeline fails. You can develop nightly testing and never have to check job status till you get a failure alert.

The server: software.nersc.gov

- Useful CI workflows including examples of this tutorial are at https://software.nersc.gov/ci-resources
- The server has CORE level of subscription.
  - Features may still be available in the API, they are just not automated viz. mirroring in code and posting job status to GitHub via integrations. (It's like owning a car with manual transmission)
- Backups are created every Sunday and stored in HPSS.
  - However, it is recommended you also maintain a mirror of your repository.

Best practices: security (and common sense)

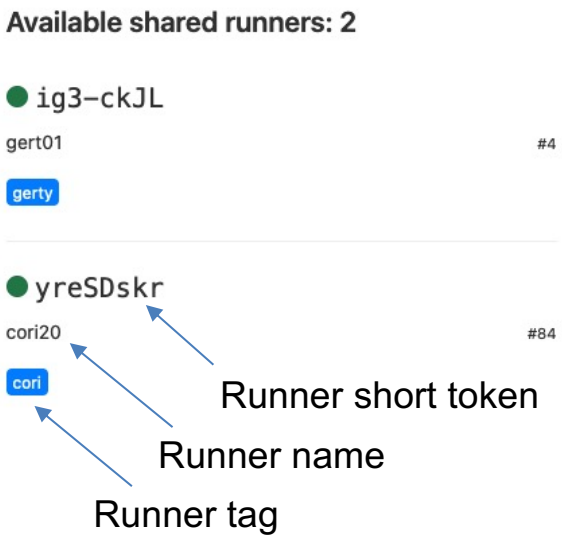
- Mirroring code into the server is more consequential than doing so on your local filesystem.
  - Think of it as (git clone + cron job)*all branches.
    - It can quickly eat away your allocation.
    - Malicious code, if somehow injected, gets executed faster.
    - Mistakes have a larger effect. Build processes routinely involve a `make clean` step. Review the workflow before mirroring. A misplaced `rm -rf` could wipe out a lot of things.
  - Users are responsible for codes they mirror in + jobs they trigger
  - Do not automate mirroring of code you do not own.
  - Only clone protected branches of repositories you do not own.
    - Most development teams have a review process for merging into protected branches, so these are likely to be safest.

Best practices: security (and common sense)

- The runner shell is the same as the login shell i.e. all the filesystems are real and user permissions are the same etc.
- Broader NERSC user policy agreements still apply e.g. do not pull in export control material.
- Do not share tokens or put them in a location where others can access it. This is the equivalent of password sharing.
- Tokens should be of as limited a scope as permissible for the workflow and of limited time validity (e.g. 30 days)
    - If you suspect a token has been compromised, revoke it and inform help.nersc.gov. Tokens may also be compromised through an external utility viz. the CodeCov incident.

Runner: Cori (and Gerty)

- Deployed runners:
  - The server has 2 ready-to-use runners:
    - cori20
    - gert01
  - The runners run on their namesake machines.
    - If you don't have access to Gerty, don't use the runner, your jobs will fail.
    - Gerty is a test cluster and access is mostly restricted to NERSC staff.
  - User CI jobs are submitted to the slurm scheduler.



Runner

- Self hosted runners:
  - Users can add their own runners, if their use case does not fit with the provided runner. However, they are responsible for securely deploying it. If you need help with this open a ticket with NERSC.
  - Do not make your self hosted runners available beyond your project.
  - If you have other members in the project, make sure to appropriately name and tag the runner.
    - A good nomenclature for runner identification is to incorporate the target hostname/type of host and the target username ${username}-${hostname}

Other DOE facilities have a similar combination of server + official runner.
Developments in one location are portable with minimal work.

Support policy

In case of any issues please file a ticket through the NERSC help portal
help.nersc.gov

Fundamentals of GitLab CI

NERSC

BERKELEY LAB

U.S. DEPARTMENT OF ENERGY | Office of Science

YAML/CI configuration: Basics and tips

- YAML: Yet Another Markup Language
- Default CI configuration file is `.gitlab-ci.yml` in the top directory of the git repository
    - Can be customized to point to a different directory and filename. This is useful if the repository needs to target different machines with different CI configurations. For instance Spack has a nersc pipeline under spack/share/spack/gitlab/nersc_pipeline.yml

YAML/CI configuration: Basics and tips

- The workflow is categorized in stages, with each stage having a distinct functionality viz. compile, test, package, deploy.
- Each stage can have multiple jobs within it. These jobs can run in parallel if there is no dependency e.g. tests in a testsuite
- Tags are used to identify the runner that will pickup a job.

.gitlab-ci.yml 376 Bytes

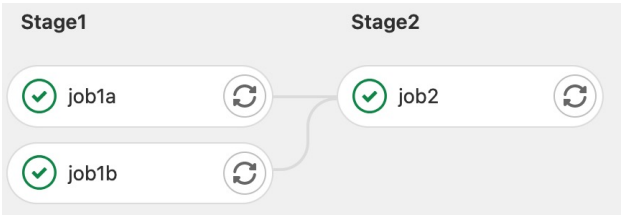
```yaml
stages:
  - stage1
  - stage2

job1a:
  stage: stage1
  tags: [cori]
  script:
    - echo "Hello from Job 1a of Stage 1"

job1b:
  stage: stage1
  tags: [cori]
  script:
    - echo "Hello from Job 1b of Stage 1"

job2:
  stage: stage2
  tags: [cori]
  script:
    - echo "Hello from Job 2 of Stage 2"

variables:
  SCHEDULER_PARAMETERS: "-M escori -q compile -N1 -t 00:05:00"
```



YAML/CI configuration: Basics and tips

- Hidden jobs (identified through a dot before the job name) are not executed and need not have a stage. They can be used to inject common features in multiple jobs viz. runner tags

```
.ci-runner:
  tags:
    - cori
```

- Labels can be used used to inject common script into the job without overriding that section of the job.

```
.common_script1: &label1
  - pwd
  - ls
```

```
level1:
  stage: stage1
  extends: .ci-runner
  script:
    - *label1
    - echo "level1 end"
```

- Workflow section of the CI file can be used to control when and if the job is executed by checking conditionalities.
- Users can introduce environment variables in the shell through the variables section.

YAML/CI configuration: Basics and tips

- Sections such as variables specified globally i.e. outside a job are applied to all jobs (they can overridden locally in each job).
- Adding [ci_skip] to commits stops the CI pipelines from being triggered, even if they are setup to trigger on each commit.
- Use `Web IDE` instead of `Edit` if you plan to edit multiple files. This ensure changes to multiple files can be done in a single commit.

Examples

Logging in

- Login to software.nersc.gov
- If it is your first time, go to the wiki documentation in the banner (https://software.nersc.gov/NERSC/nersc-ci-docs/-/wikis/Home) and follow the instructions for first time login.
    - You get an email if your pipeline fails. This is useful for scheduled workflows.
- The wiki page is a good starting point and shows how to target different slurm queues, create a PAT(personal access token) and other useful tips.
- All of todays examples and other useful workflows can be found at https://software.nersc.gov/ci-resources

Beginner example: Hello Environment

- Repository: https://software.nersc.gov/ci-resources/hello-environment
- Objective: Execute a simple CI job that scopes the environment.
- Steps:
  1. Fork the repository into your userspace, using the fork button in the top right quadrant of the page.
  2. Modify the empty GitLab CI file to create the output as discussed in the Readme.
  3. Run the CI job
- An example CI file is provided in the branch `nersc-answer`
- There is a reservation for todays training session that users can use between 09:30-13:30 Pacific
  - SCHEDULER_PARAMETERS: "-C haswell --reservation=citutorial1 --qos=shared -N1 -n2 -A nintern -t 00:05:00"

Beginner example: Simple workflow

- Repository: https://software.nersc.gov/ci-resources/simple_workflow
- Objective: Execute a typical CI job that builds and runs a couple of executables.
- Steps:
  1. Fork the repository into your userspace, using the fork button in the top right quadrant of the page.
  2. Modify a GitLab CI file to create the output as discussed in the Readme.
  3. Run the CI job
- An example CI file is provided in the branch `nersc-answer`
- There is a reservation for todays training session that users can use between 09:30-13:30 Pacific
  - SCHEDULER_PARAMETERS: "-C haswell --reservation=citutorial1 --qos=shared -N1 -n2 -A nintern -t 00:05:00"

Intermediate examples: Mirroring in code

- Repository: https://software.nersc.gov/ci-resources/mirroring
- Objective: Understand how to securely mirror in code from external sources using tokens.
- Overview:
  1. Create the necessary personal access token (PAT). One for the NERSC instance. If your source repository is private you will need one for that instance as well
  2. Create a mirroring repository that performs the mirroring and an empty target repository that will host the mirrored project.
  3. In the mirroring repository create a shell file that performs the fetch and push operations. Then create a CI file that can trigger the execution of the above shell file.

Intermediate examples: Reporting CI status to GitHub

- Repository: https://software.nersc.gov/ci-resources/report-status
- Objective: Understand how to securely report the CI job status to another repository.
- Overview:
  1. Create personal access tokens (PAT), one for the NERSC instance and the other for an external GitHub instance
  2. Create a report status repository and store the PAT there.
  3. In the repository create a python file that examines the source repository for CI jobs and pushes the results to the same commit on GitHub. Then create a CI file that can trigger the execution of the above python file.

Questions?

https://tinyurl.com/4n344rx7