Workshop on the use of the NEA GitLab

Michael Fleming

OECD Nuclear Energy Agency
Division of Nuclear Science

27 November 2019, WPEC Subgroup 49 Meeting, NEA,
Boulogne-Billancourt, France

Preliminary

- Before we begin, please ensure you have git and docker installed
- Information can be found on the agenda and may vary by OS
- If you can install an SSH key it will be very helpful:

`https://git.oecd-nea.org/profile/keys`

Objectives

- *Familiarise* you with the main features of
  - the `git` version control system
  - the GitLab repository manager / hosting system
  - Docker containerisation ~~[time permitting]~~
- Make you confident that you can find answers to technical questions when the need arises
  - Google, Stack Exchange and/or the git+GitLab documentation will typically answer any question within seconds/minutes
- Help you find how best to realise the value of these tools for your software and nuclear data evaluation projects

Agenda

1. Brief introduction and background on VCSs
2. Introduction to `git`
3. Making a project
4. Tracking changes and sharing versions
5. Branches, merging, tags and models

   Planned coffee break
6. Submodules and dependency
7. GitLab features for managing projects
8. Continuous integration
9. Docker, making images and running containers

   Discussion and hands-on Q&A

'Local version control'

- Common experience informs us to retain a history of versions
- The oldest and simplest system is a set of (hopefully backed up, tested, timestamped) copies

```
code/
code-2019-11-27/
code-2019-11-01/
...
```

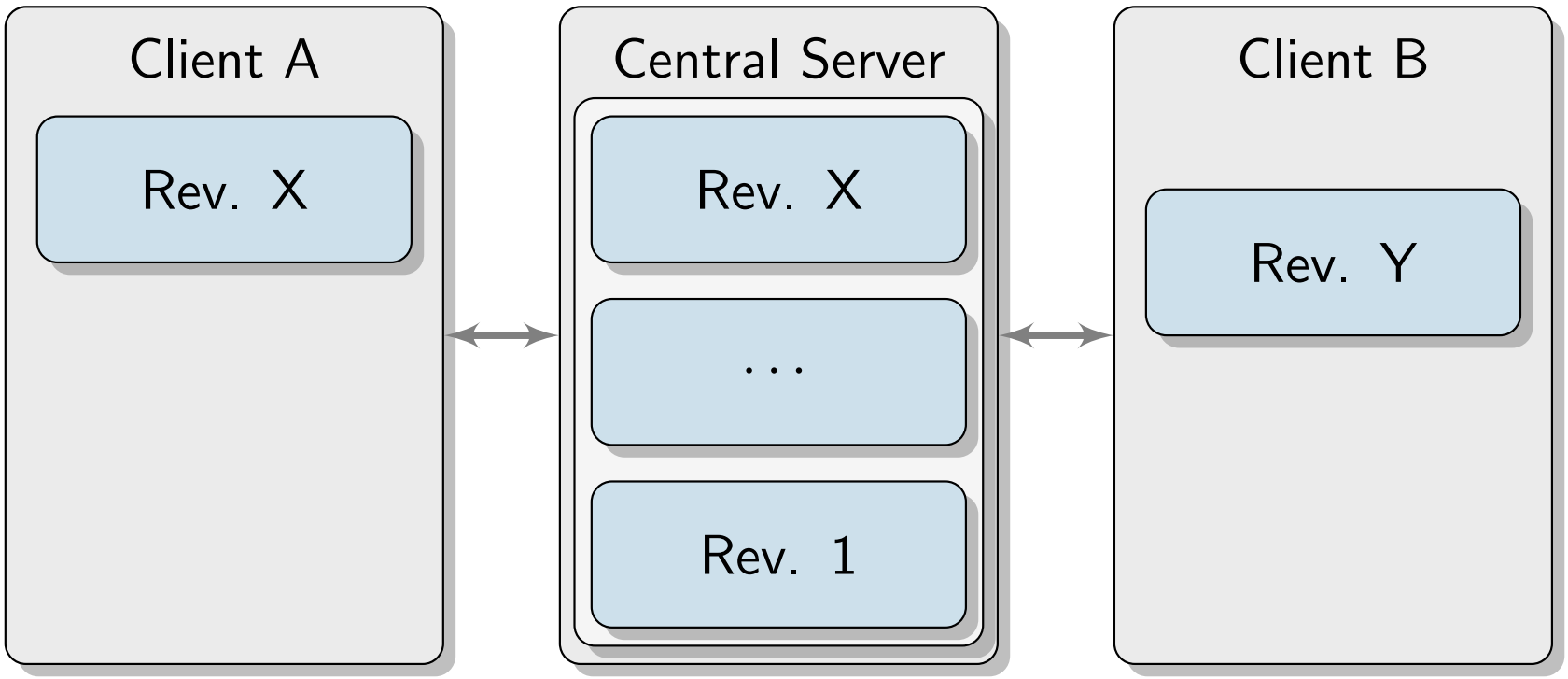
- While this *can store* the required information, it is very inefficient, easy to corrupt, doesn't distribute itself – it doesn't provide more than an archival system

Partial history of version control systems

- **RCS** (1982): Early UNIX 'First Generation Tool'
- **CVS** (1986): 'Second Generation' centralised 'client-server' system
- **SVN** (2000): Centralised 'client-server' system
- **BitKeeper** (2000): Originally non-open distributed system used for Linux dev – licence issues caused the creation of Git *[open 2016]*
- **Git** (2005): Distributed system originally created by Torvalds but with mass modern use on all OS

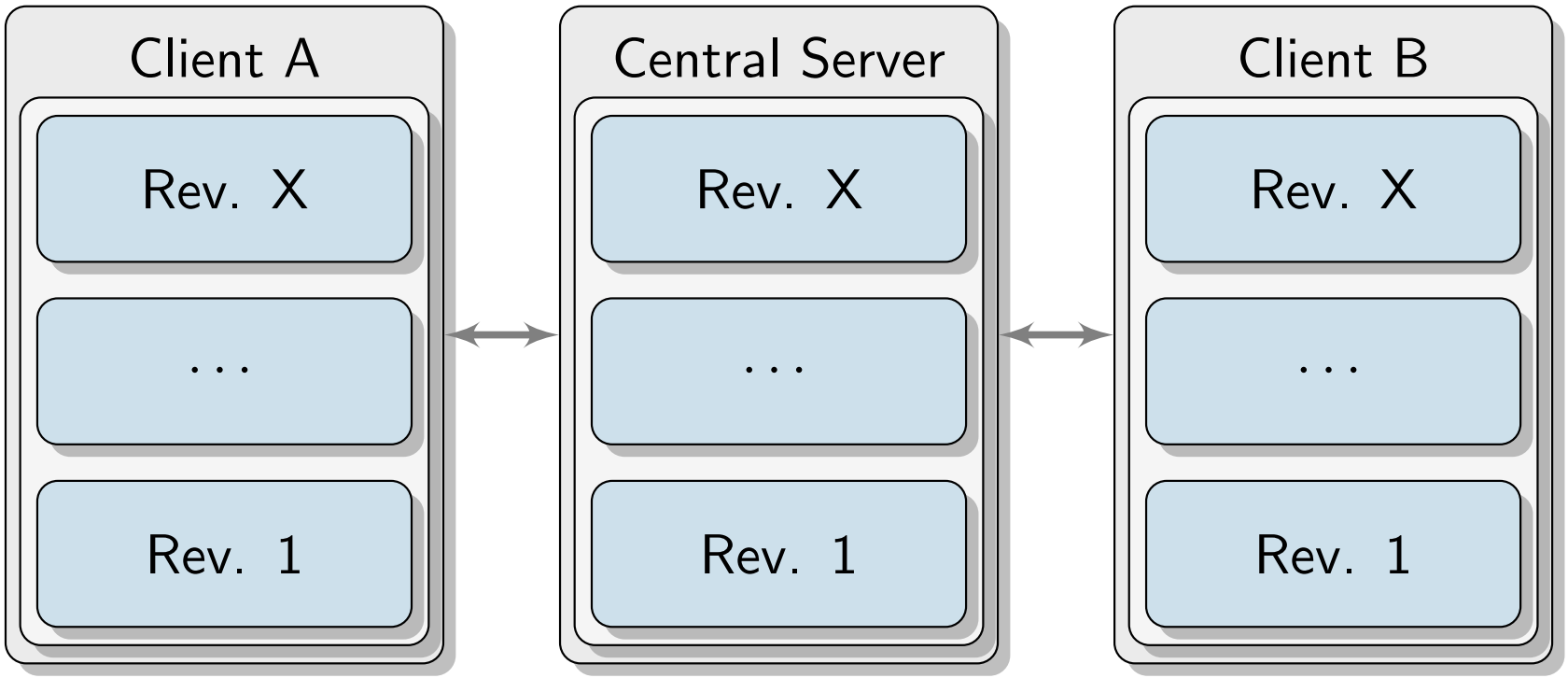
Client-server version control

- *À la mode* since the 80s with systems such as `CVS` and `SVN`
- Clients connect to master server and take versions from the full history stored there
- Branching problematic since we don't interact with all content



Distributed version control

- Implemented earlier, but most popular systems created since 2000
- Every user has full history and all branches storied intelligently
- Allows us to move through all of the repo within the local machine



Git and GitLab

- **Git** is an open-source (GPLv2), distributed, version-control system
  - Git provides all the staging, branching, merging, tags, etc
  - Git contains meta data and all the history in every working copy
- **GitLab** is an open source (MIT) platform that manages Git repositories, users and provides many other functionalities
  - GitLab provides a sophisticated and easy-to-use interface to explore repositories
  - GitLab contains project management tools including merge interfaces, issue boards, wikis, milestone trackers, and much more
  - One major GitLab addition is an integrated 'continuous integration/deployment' functionality to automate simple or complex jobs on any change

Support time

- People sometimes say:
  - "I started with VCS-1 (e.g. CVS) in the 90s"
  - "Then I started using VCS-2 (e.g. SVN) in the 2000s"
  - "Now this VCS-3 (Git) seems to be very popular, but for how long?"
- SVN and Git are from the same era, but they operate differently and have had different systems established with different user-bases
- GitHub (2008) and GitLab (2014) have revolutionised this area
- GitLab recently had 268M$ series E funding = 2.7B$ valuation
- Ultimately, GitLab manages Git repositories

Let's get started!

- Navigate to `https://git.oecd-nea.org/` on your browser and log into the website with your username or institutional email
- Click on the project:
  'Nuclear Science/WPEC/WPEC Subgroup 49/Workshop/Example'
- 'Fork' and select the space with your username **[top right]**

- Now you have made your first project! Explore what is inside it.

We will resume when all have created their own project spaces

Making and tracking changes

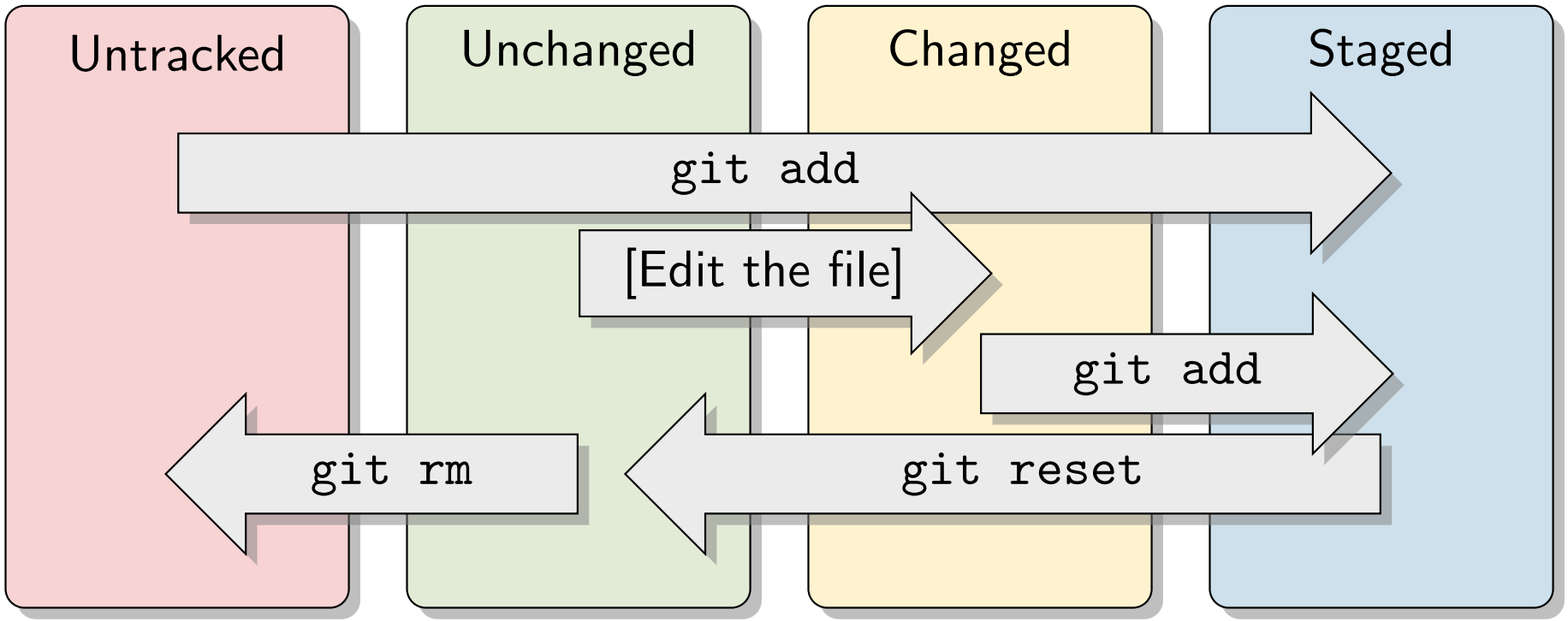
- First, let's download a copy of the repository or 'clone' it. Those without a terminal may need to ask for help.
  - With a standard terminal obtain the `REPO_URL` from the blue drop-down 'Clone' button next to 'Fork':

```
> git clone REPO_URL
> cd example
```

  - With `git gui` set the source directory as `REPO_URL` and target at some folder on your machine
- Now make a file `[yourname].txt` – edit with a text editor and write that this is your personal copy
- Now let's see if Git has noticed the change
  - In the terminal within the repository folder, run:

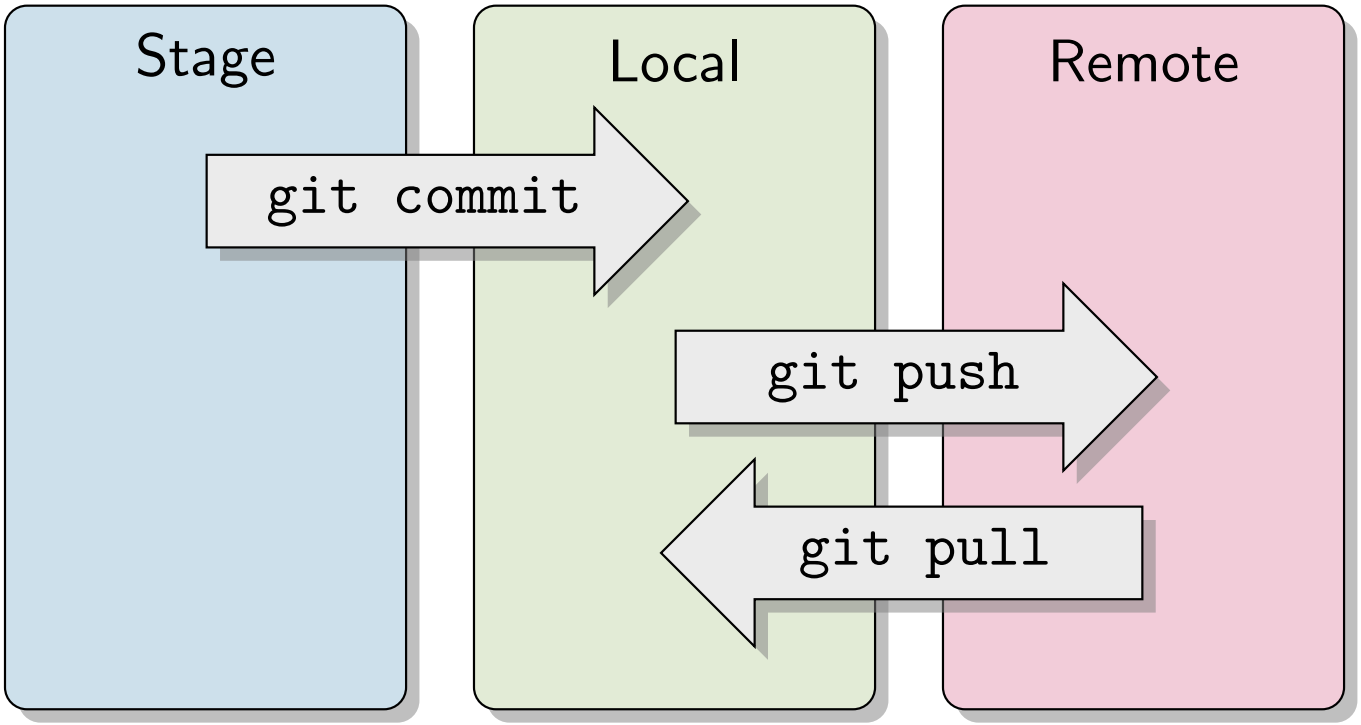
```
> git status
> git diff
```

Git staging



- `> git status` shows what is in the above categories
- We are free to keep changing and checking until the stage is set

Commits and pushing



- When staged content is ready, `> git commit` locks the current stage and produces a SHA-1 cryptographic hash identifier
- We can then share this with the server by `> git push` and take the most recent changes from others with `> git pull`

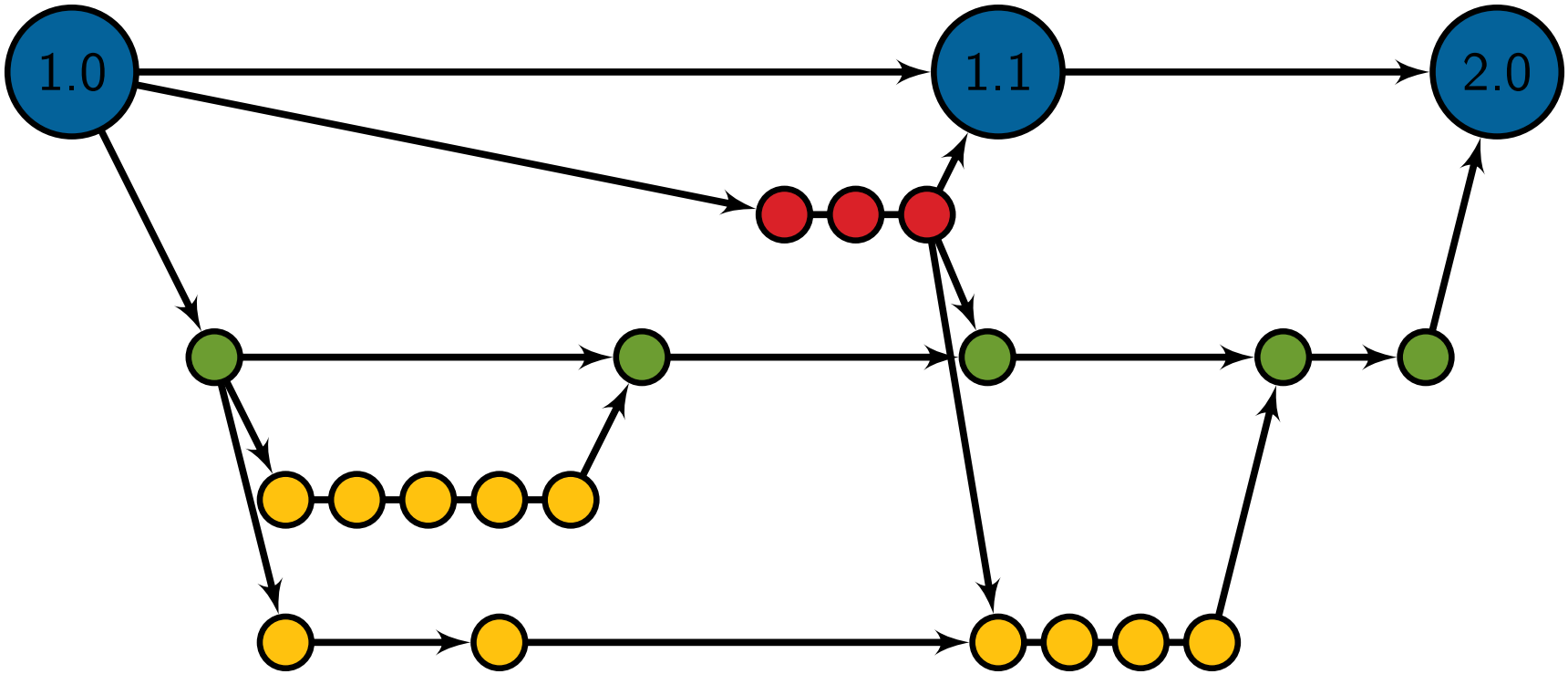
Make changes to your repository

- Use git to see what local changes have been made
- Stage these changes (`git add file1 file2 directory...`)
- Verify what is staged! (`git status`)
- Commit these changes to your local repository:
  `> git commit -m "An indicative message"`
- Push these changes to the remote: `> git push origin master`
- Visit the repository on the website and see your changes
- Try editing `README.md` with the online interface

What do you have to do now on your local copy?
Bonus: Check out a markdown 'cheatsheet'

Branching

- Branching allows us to intelligently structure our work
- Even 1-person/simple projects should use branches
- master and dev are essential, although feature-X, hotfix-Y and others are common



Making branches

- Make a new `dev` branch for your project
  ```
  > git branch dev
  ```
- You can see what branches are available with `git branch`
- You aren't in the branch yet - let's 'check it out'
  ```
  > git checkout dev
  ```
- Add a line in the `yourname.txt` to say that you made a branch
- Commit this and then push to the remote **dev**
  ```
  > git push origin dev
  ```

Go online and see what happened – can you find your change?
On the left menu checkout the tabs under Repository

Making merge requests

- Two slides ago you saw arrows meeting – these are merge requests (pull requests if you are in GitHub)
- The process should always include a review and the ability to merge is governed by the user permissions on the system
- These can be completed via the terminal and pushed to the remote or done online
  - Complex merges may require direct control via the terminal where conflicts arise

Go online and select the Merge Requests tab on the left
Make a merge request from dev into master and complete it
Check out the Repository Graph!

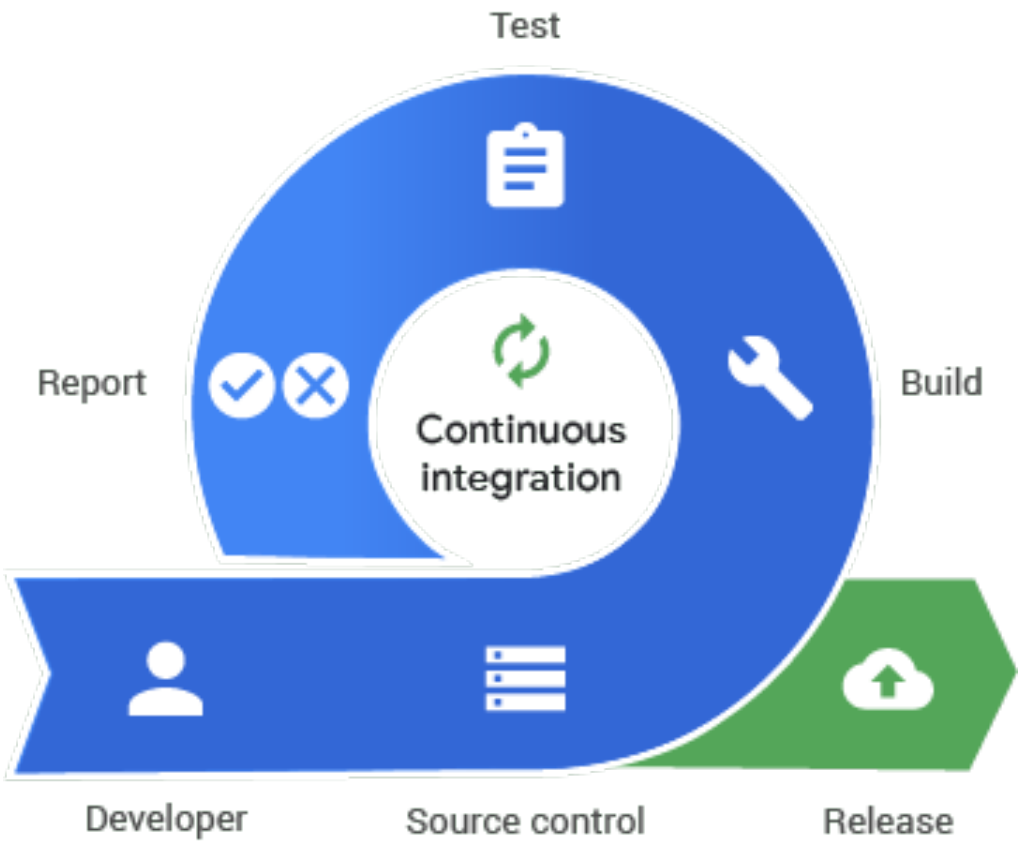
Managing projects: users and permissions

- Now we are going to introduce collaborators – on the online system go to Settings → Members
- Here we can add permissions to users with accounts
  - Add 1-2 of your neighbours with 'Developer' roles
- Check your email account – did you receive any notification?
- Make a change in one of your neighbors' [theirname].txt files either online or through clone/add/commit/push
- Could you commit and push these changes?
  - HINT: which branch are you on?
- Make a merge request of your new commit into their master branch

Check out the options on the MR page!
The master branch is protected by default – control this in settings

Continuous integration

- GitLab has an integrated system for running jobs with repositories
- There is a complex theory and practice, but at the simplest:



Source: cloud.google.com

- CI can include whatever processes you wish, for example:
  - Compile code
  - Run unit/integration/regression tests
  - Package content for release
- CI can be specified to run on multiple (virtual) machines, testing different environments, OSs, compilers, etc.
  - Tools like Docker make this very straightforward
- Logs are created based on the terminal outputs
- **Artifacts** are outputs that are stored on the GitLab server

Configuring CI

- As an example, check out the `test/tares` repository of the WPEC SG49 space
  - On the left tabs look for CI / CD (CD = Continuous Deployment)
  - Explore the pipelines and the logs
- This process is governed by the **.gitlab-ci.yml** file in the repository – review its contents
- Now return to your own repository and let's move the template **.gitlab-ci.yml** that is in the `sources/` folder
  - How can we move this file with git and not have the repository have two copies? **Use Google/SE/documentation**

Launching CI

- So at this point you should have a big red X and GitLab should have notified you that the build has failed!
- This is why we:
  1. **never commit to master**
  2. **test code before committing**
     - *"Don't shoot from the hip"*
- CI tests are often more rigorous than our local machine test, so failure does indeed happen without 'shooting from the hip'

Can you find the error? Look at the pipeline log
Open the file on GitLab and click on 'Blame' – then fix it

Hunting after bugs

- Did you have another failure?
- Did you test your code before changing the Fortran?
- The world has not yet ended, but still we should avoid this...
  - *This isn't quite fair for those who don't have the required compilers on their local machine – apologies to you!*

The CI should never be the first time the repository is tested

1. Once you have a passing build, add another `stage` to compile the second Fortran code
2. Figure out how to store the text tile output as an artifact and have this expire after 1 hour

Tags and licences

- Before we make a release, let's add a licence for the repository
  - You could write one from scratch and add it to the repository but
  - GitLab provides templates for a dozen standard licences
  - Some example info: `https://choosealicense.com/licenses/`
  - **Go ahead an select one for your repository**
- One you have a tested code passing in the master branch, let's tag a version that would be a specific release:
  1. `git tag -a vX -m "my version X"`
  2. Visit the repository and see the tag on the system
  3. See what you can do within the GitLab system with a new tag
- Git also supports lightweight tags that are a static branch commit

Dependencies

- For nuclear data evaluation, we have several types of dependency:
  - We rely on another complete code system (e.g. PREPRO)
  - Source/library is integrated into another system (e.g. ECIS)
  - Some data is shared between projects (e.g. RIPL, EXFOR)
  - Perhaps you want to integrate multiple things into an evaluation?
- In the past we may duplicate these, but:
  - Now everyone must maintain their local PREPRO
  - Local changes aren't passed back to authors/community
  - Repositories become bloated monstrosities
  - **Sometimes licences are not respected**

Submodules

- Git handles this naturally with `git submodule`, which links to another repository
- Submodule **always** points to a specific version
  - You can choose to update as required
  - You can propose a edit/branch on the linked repository and link it
- Now fork the example-data repository into your space and then submodule it to your example repository:

```
git submodule add [git repo] [location in current repo]
```

- Check everything and then push to remote

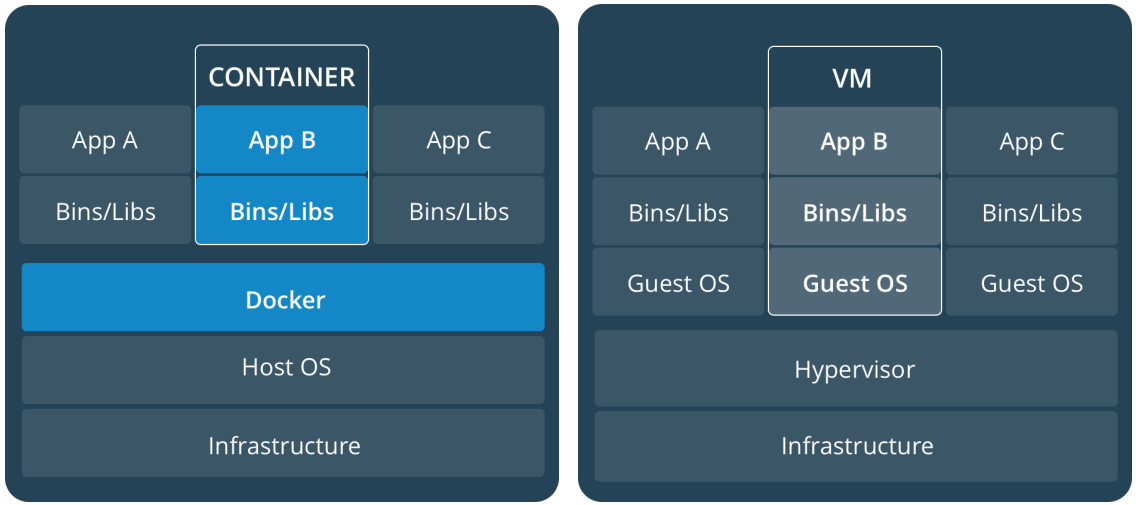
Working with submodules

- Check the GitLab page for your repository and see how the submodule works
- Try cloning a fresh copy of your example repository
- Look up how you download the submodules and look at the terminal output when you do it
  - Recursive downloads will follow submodules of submodules
- Make a change in the data repository **on GitLab**

Did anything happen on your example repository?
How do you update it?

Introduction to Docker

- Docker is a platform for packaging, running and deploying applications via **containers**
- A container has the code, libraries, environment and configuration that is required for your application in a compact, encapsulated and portable (via images) manner
- Similar to (but much better for many applications) VMs:



Source: docs.docker.com/

Containers and images

- First let's use the hello-world to get us started:
  `> docker run hello-world`
- Review the terminal outputs to see what happened
- Look at the images that we have:
  `> docker images`
- Separately, let's look at the containers:
  `> docker ps -a`
- The hello-world image was run in a container and completed its job
  `docker ps` shows only the active containers

Grabbing images

- Let's consider something more interesting – first pull an image: `docker pull ubuntu`
- You should have been told that it took a default tag 'latest'
- Check out the Docker hub: `https://hub.docker.com/_/ubuntu`
- Look under the 'tags' tab – there are several pages!

Check out your favourite OSs and look at:

`https://hub.docker.com/search/`

Going inside a container

- `docker run [image]` takes an image and makes a container
- This can be setup to run some specific job *if defined on build*
- We can interactively go inside the container with (for example):
  `> docker run -it ubuntu:latest bash`
- Explore the container `which X` – compilers, standard tools?
- See what processes are running `ps aux`

This is a clean OS from which we can build an exact system

Building an image from within a container

- Start updating/creating the system that you want to be able to compile the example:
  - `exit` will leave the container
  - Make the container again but mount the repository fortran files:
    `docker run -v /host/directory:/container/directory ...`
  - Update and upgrade the OS and then install `gfortran`
    (look at the Dockerfile on your repository for a hint)
  - Compile as a test inside the container
  - Exit the container again
- Now look at your containers with `> docker ps -a`
- You *could* save this container with `docker commit` (RTM)

Building from a Dockerfile

- Alternatively, we can specify the build instructions and automatically create the image
- Within a terminal, enter the folder with `Dockerfile` in the repository
- `> docker build -t [name:tag] .` will execute the build of the image based on that file and give it a repository name and tag
- Now interactively run this image and check if you can compile

Distributing images

- We can freeze that image as a file to share with:
  `docker save -o [path for generated file] [image name]`
- You *could* share these directly but there are two great methods:
  1. **Docker Hub** – public repositories for images
     - You can create an account today, login via the terminal and `docker push` images to your space
  2. **GitLab Docker Registry** – associated with repositories
     - We can distribute compiled versions alongside the repository
     - Exact environments can be used for testing and reproducing outputs
- Currently the registry is disabled for the NEA GitLab, as is docker as a CI executor
- This is an active WIP area and we expect updates in early 2020

Thank you for your attention

Source: xkcd.com/1597/