RISE Summer School 2025

Git and Docker

Eric Kerfoot, King's College London

July 18, 2025

- Talk will cover basic usage of Git and Docker, two very different technologies
- Git is used to manage versioned source code and other documents both locally and in remote locations
- Docker is a containerisation technology used to define working environments
- Both will be important for your projects so we'll focus on what you need now rather than get into deep details

Git

- Source (or version) control involves organising the code files for a project in a known place with tools to keep track of change
- Typically this means a set directory for code to live in, tools to track when changes are made and keep a history of those changes, and store the code with changes locally and remotely.
- Git does this with a repository system and the `git` tool which operates with a `.git` subdirectory within repository directories
- Github and other sites host repositories and interface with `git` to push and pull changes

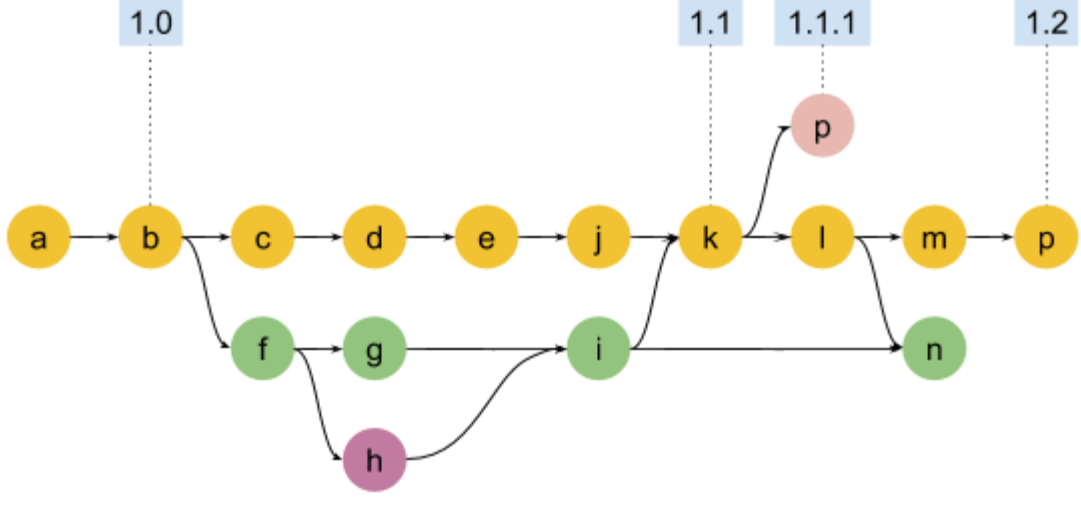
Installation

- Read here: https://git-scm.com/downloads
- GUI tools like GitAhead (https://gitahead.github.io/gitahead.com) can help
- For Ubuntu, `sudo apt install git`

Git Concepts

- Important to understand concepts well early on
- Git tracks the changes to files which have been committed (added) to a repo
- Any time files are change a commit is made to add those changes to the repository state
- The state of a repository thus represents a directed graph where each node is a state of the committed files
- Branches can be made from commits to allow diverging versions of the code, and can be merged back together later

Version control - an illustration



Starting a Repository

- Any directory can be made into a repository using `git init`
- A site like Github can be used to create a repository stored remotely which can then be "cloned" locally
- This is probably what you want to do, the local clone will track who the remote repo is and allow you to "push" changes to the net

Remote Repositories

- Repositories can be hosted on remote systems accessed by HTTPS or SSH
- Github is a common choice along with Gitlab and Bitbucket
- A remote repository can be copied locally with `git clone` and will keep track of who its remote is and its URL
- Commits and branches made locally in the clone can be pushed to the remote with `git push`, similarly `git pull` will bring down changes from the remote not in the clone

- The repo for these slides can be cloned with:

```
git clone https://github.com/ericspod/rise_summer_school_25.git
```

- If you have your Github account setup with an ssh key you can use:

```
git clone git@github.com:ericspod/rise_summer_school_25.git
```

- There is an authentication difference between these
- You won't be able to push changes to these as you don't have authorisation, but the clone is otherwise what one would use for collaborative work

- For this workshop, you'll likely want to make a Github account, create a new repository through its interface, then clone it
- Once cloned locally, you can copy your project into it, create a commit, and push it to the server
- Go here: https://docs.github.com/en/get-started
- First few tutorials will go through this process

- For now a local repository can be created to explore using git locally:

- For now a local repository can be created to explore using git locally:

In [1]:
```
%%bash
mkdir local_repo
cd local_repo
git init -b main
ls -alh
```

```
Initialised empty Git repository in /home/localek10/workspace/ri
se_summer_school_25/local_repo/.git/
total 12K
drwxrwxr-x 3 localek10 localek10 4.0K Jul 11 22:49 .
drwxrwxr-x 6 localek10 localek10 4.0K Jul 11 22:49 ..
drwxrwxr-x 7 localek10 localek10 4.0K Jul 11 22:49 .git
```

- This notebook can be run as the tutorial for this content so all cells can be executed
- The following cells use bash to run commands, either as `%%bash` magic cells or with the `!` magic line
- Commands are typically run within the repo `./local_repo` so the current working directory is changed here for convenience:

- This notebook can be run as the tutorial for this content so all cells can be executed
- The following cells use bash to run commands, either as `%%bash` magic cells or with the `!` magic line
- Commands are typically run within the repo `./local_repo` so the current working directory is changed here for convenience:

In [ ]:
```
%cd local_repo
```

- The `.git` hidden directory contains all the change information for the repo and much other data
- Next write a file and commit it:

- The `.git` hidden directory contains all the change information for the repo and much other data
- Next write a file and commit it:

In [3]:
```
%%bash
cat - > test.txt << _EOF_
a = 1
b = 2
c = 3
_EOF_
```

- The current status of a repo can be seen with `git status`:

- The current status of a repo can be seen with `git status`:

In [4]:
```
!git status
```

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        test.txt

nothing added to commit but untracked files present (use "git ad
d" to track)
```

- This file needs to be added to the repository, then the change committed to create a new state:

- This file needs to be added to the repository, then the change committed to create a new state:

In [5]:
```
%%bash
git add test.txt
git commit -m "Adding test.txt"
```

```
[main (root-commit) b4f80ad] Adding test.txt
 1 file changed, 3 insertions(+)
 create mode 100644 test.txt
```

- The repo now has a commit in its history:

- The repo now has a commit in its history:

In [6]:
```
!git log
```

```
commit b4f80ad155a8e600d75efbb8a37f6e56b7816c1e (HEAD -> main)
Author: Eric Kerfoot <eric.kerfoot@kcl.ac.uk>
Date:   Fri Jul 11 22:49:55 2025 +0100

    Adding test.txt
```

- If `test.txt` is changed, `git status` will report this fact:

- If `test.txt` is changed, `git status` will report this fact:

In [8]:
```
%%bash
cat - > test.txt << _EOF_
a = 1
b = 2
c = 3333
_EOF_
git diff
```

```
diff --git a/test.txt b/test.txt
index a9aeef0..27be8cc 100644
--- a/test.txt
+++ b/test.txt
@@ -1,3 +1,3 @@
 a = 1
 b = 2
-c = 3
+c = 3333
```

- This represents a change which isn't committed yet, so the history isn't recorded
- Changing the file again will show a difference but the first change's history won't be kept
- The file must be added and then committed to take a "snapshot" of its state:

- This represents a change which isn't committed yet, so the history isn't recorded
- Changing the file again will show a difference but the first change's history won't be kept
- The file must be added and then committed to take a "snapshot" of its state:

In [9]:
```
%%bash
git add test.txt
git commit -m "Updated test.txt"
```

```
[main 334b63a] Updated test.txt
 1 file changed, 1 insertion(+), 1 deletion(-)
```

- The history of changes which have been committed can now be seen for this branch:

- The history of changes which have been committed can now be seen for this branch:

In [10]:
```
!git log
```

```
commit 334b63af300e8b30c12c80f5ad034d65f85c9e6e (HEAD -> main)
Author: Eric Kerfoot <eric.kerfoot@kcl.ac.uk>
Date:   Fri Jul 11 22:52:04 2025 +0100

    Updated test.txt

commit b4f80ad155a8e600d75efbb8a37f6e56b7816c1e
Author: Eric Kerfoot <eric.kerfoot@kcl.ac.uk>
Date:   Fri Jul 11 22:49:55 2025 +0100

    Adding test.txt
```

Branches

- Branches are references to specific commits, in the above this is `main`
- Commits have parents, so each represents the cummulative changes its parents and itself store
- Branches thus point to a state of the repo, as updates are made branches change to point to new states, or can be created to track divergent changes
- A new branch can be made to make a new version of the codebase:

In [11]:
```
%%bash
git branch new_branch
git checkout new_branch
```

```
Switched to branch 'new_branch'
```

- Changes made to files now will be tracked by this branch while `main` (the initial branch) will continue to refer to the previous state:

- Changes made to files now will be tracked by this branch while `main` (the initial branch) will continue to refer to the previous state:

In [12]:
```
%%bash
cat - > test.txt << _EOF_
a = 1
b = 2222
c = 3333
_EOF_
git commit -am "Update test.txt again"
```

```
[new_branch bb54346] Update test.txt again
 1 file changed, 1 insertion(+), 1 deletion(-)
```

- `main` can be checked out again which restores the files to the state it represents
- Changes can be committed so that its state diverges from the new branch

- `main` can be checked out again which restores the files to the state it represents
- Changes can be committed so that its state diverges from the new branch

In [13]:
```
%%bash
git checkout main
echo 'd = 4' >> test.txt
git commit -am "Update test.txt in main"
```

```
Switched to branch 'main'
```

```
[main 17daa22] Update test.txt in main
 1 file changed, 1 insertion(+)
```

- `git` can be used to print a visual representation of this divergence:

- `git` can be used to print a visual representation of this divergence:

In [14]:
```
!git log --graph --oneline --all
```

```
* 17daa22 (HEAD -> main) Update test.txt in main
| * bb54346 (new_branch) Update test.txt again
|/
* 334b63a Updated test.txt
* b4f80ad Adding test.txt
```

- Changes in `new_branch` can be incorporated into the current branch (`main`) with merging:

- Changes in `new_branch` can be incorporated into the current branch (`main`) with merging:

In [15]:
```
%%bash
git merge new_branch
git log --graph --oneline --all
```

```
Auto-merging test.txt
Merge made by the 'ort' strategy.
 test.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
*   ee0391e Merge branch 'new_branch'
|\
| * bb54346 Update test.txt again
* | 17daa22 Update test.txt in main
|/
* 334b63a Updated test.txt
* b4f80ad Adding test.txt
```

- Able to merge because the changes made to the file weren't conflicting in any way
- Conflicts occur when merging branches if changes from merged branch overlap those in the merging branch, ie. a file was modified in both branches after they diverged
- Conflicts must be resolved, `git` can do a lot but often one has to manually resolve issues
- For this project avoid this by sticking to one branch for your changes

Not Covered

- Many topics are not covered here for brevity
- A detached HEAD is a state where the current repo points to a commit rather than a named branch, badly named and complex to deal with
- Rebasing to align changes between branches that isn't a merge commit
- `git push` wasn't used here but should be to push anything committed when using a remote repo
- Remote URLs and other configuration
- Tags, cherry picking, and many other git commands
- Pull Requests and many other collaborative features

Docker

- Set of tools and applications for containerisation, that is creating a running self-contained operating system environment
- Unlike virtual machines, a container runs using the host OS rather than sitting on top of virtualised hardware as the OS itself
- Other technologies like Apptainer or Podman provide similar facilities
- Used to package software systems into portable deployable units called images containing all needed libraries

- Advantage is that everything your app needs can be packaged into one place
- Any time this container is run again it won't rely on OS libraries, so long as the container layer doesn't change your container will run
- Dockers works by building an image from a script-like file specifying commands
- Images are stored services called registries or (less commonly) to archive files
- Containers are started from named images and often a command to run within the container
- File system is entirely what the image provides plus whatever directories are chosen to be mounted from the host

- Invoking Docker on the command line to run an image (first `cd` back to the root of this exercise):

- Invoking Docker on the command line to run an image (first `cd` back to the root of this exercise):

In [ ]:
```
%cd ..
```

- Invoking Docker on the command line to run an image (first `cd` back to the root of this exercise):

In [ ]:
```
%cd ..
```

In [23]:
```
%%bash
docker run grycap/cowsay /usr/games/cowsay "Hello Docker!"
```

```
 _________________
< Hello Docker! >
 -----------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

- Invoking Docker on the command line to run an image (first `cd` back to the root of this exercise):

In [ ]:
```
%cd ..
```

In [23]:
```
%%bash
docker run grycap/cowsay /usr/games/cowsay "Hello Docker!"
```

```
 _______________
< Hello Docker! >
 ---------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

- Command will look for the image named (or tagged) `grycap/cowsay` which is the "cowsay" image provided by user "grycap" on Docker Hub (the default registry)

- Containers that aren't explicitly cleaned up will persist with viewable logs, you can see these with `docker containers` command:

- Containers that aren't explicitly cleaned up will persist with viewable logs, you can see these with `docker containers` command:

In [24]:
```
!docker container ls -a
```

```
CONTAINER ID   IMAGE           COMMAND                  CREATED
STATUS                      PORTS     NAMES
7e109afc3d97   grycap/cowsay   "/usr/games/cowsay '…"   3 second
s ago   Exited (0) 2 seconds ago             adoring_bhabha
5cb07983cdda   grycap/cowsay   "/usr/games/cowsay '…"   9 second
s ago   Exited (0) 7 seconds ago             sweet_newton
```

- Containers that aren't explicitly cleaned up will persist with viewable logs, you can see these with `docker containers` command:

In [24]:
```
!docker container ls -a
```

```
CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS                     PORTS     NAMES
7e109afc3d97   grycap/cowsay   "/usr/games/cowsay '…"   3 second
s ago   Exited (0) 2 seconds ago             adoring_bhabha
5cb07983cdda   grycap/cowsay   "/usr/games/cowsay '…"   9 second
s ago   Exited (0) 7 seconds ago             sweet_newton
```

- Images that you have locally can be viewed with the `docker images` command:

- Containers that aren't explicitly cleaned up will persist with viewable logs, you can see these with `docker containers` command:

In [24]:
```
!docker container ls -a
```

```
CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS                     PORTS     NAMES
7e109afc3d97   grycap/cowsay   "/usr/games/cowsay '…"   3 second
s ago   Exited (0) 2 seconds ago             adoring_bhabha
5cb07983cdda   grycap/cowsay   "/usr/games/cowsay '…"   9 second
s ago   Exited (0) 7 seconds ago             sweet_newton
```

- Images that you have locally can be viewed with the `docker images` command:

In [26]:
```
!docker images
```

```
REPOSITORY      TAG          IMAGE ID       CREATED         SIZE
cowsaypy        latest       f4b3ce4527ac   22 hours ago    133MB
in_out          latest       3d8c31d4552a   11 days ago     11MB
monai_101       latest       b0c155add529   11 days ago     5.77GB
min_pt          latest       025ffba75b8e   11 days ago     5.82GB
ubuntu          24.04        bf16bdcff9c9   6 weeks ago     78.1MB
python          3.12-slim    3d1d8f39b99d   3 months ago    124MB
alpine          latest       aded1e1a5b37   4 months ago    7.83MB
ubuntu          latest       a04dc4851cbc   5 months ago    78.1MB
hello-world     latest       74cc54e27dc4   5 months ago    10.1kB
grycap/cowsay   latest       80cd02790675   8 years ago     166MB
```

- To create an image, define a `Dockerfile` file within a project's directory which will contain the commands to configure the image for the project
- This will select the base image to build on top of, copy files into the image from the project, install whatever is needed in the image, and define an entry point (or command)
- The Dockerfile has its own command language, eg. `COPY` and `RUN` for copying files into the image and running commands

- Write out a simple Dockerfile:

- Write out a simple Dockerfile:

In [16]:
```
%%writefile Dockerfile

FROM python:3.12-slim
RUN pip install cowsay
WORKDIR /app
COPY myapp.py /app
CMD ["python","myapp.py"]
```

```
Writing Dockerfile
```

- `FROM` defines the base image, in this case a simple Python image based on Debian
- `RUN` specifies a command to execute at build time, **not** runtime, which configure the environment
- `WORKDIR` specifies a working current directory
- `COPY` copies data from the host into the image, here only the script file
- `CMD` specifies the command to run when the container starts (this can be replaced unlike `ENTRYPOINT`)
- A few other important commands, there can be multiple `RUN` and `COPY` commands

- Need to write out the example script:

- Need to write out the example script:

In [17]:
```
%%writefile myapp.py
import cowsay
cowsay.cow("Hello Docker!")
```

```
Writing myapp.py
```

- Next the image is built:

- Next the image is built:

In [18]:
```
!docker build -q . -t cowsaypy
```

```
sha256:f4b3ce4527ac36ed3c8f2d454360e35ac6a8bacde641b40cc6b9aea8e
ff33722
```

- Next the image is built:

In [18]:
```
!docker build -q . -t cowsaypy
```

```
sha256:f4b3ce4527ac36ed3c8f2d454360e35ac6a8bacde641b40cc6b9aea8e
ff33722
```

- Uses `Dockerfile` to make the image we've tagged as `cowsaypy`
- `-q` makes the command quiet for this notebook, remove to see all the output of what happens

- The app can now be run

- The app can now be run

In [19]:
```
!docker run --rm cowsaypy
```

```
 _____________
| Hello Docker! |
  =============
                \
                 \
                   ^  ^
                    __
                   (oo)\_______
                   (__)\       )\/\
                       ||----w |
                       ||     ||
```

- The app can now be run

In [19]:
```
!docker run --rm cowsaypy
```

```
 _____________
| Hello Docker! |
 =============
            \
             \
               ^__^
               (oo)\_______
               (__)\       )\/\
                   ||----w |
                   ||     ||
```

- `docker run` can have a lot of arguments for many features
- Here `--rm` ensure the container is cleaned up once it finishes
- If `-ti` is given and `/bin/bash` added to the end, the command will start the container in interactive mode with a bash prompt rather than run `myapp.py`

- Data is typically transferred to and from running containers with mounted directories
- `-v` flag indicates a mount point and is specified with the absolute path of the host directory and where it will be mounted in the container
- Eg. mount the curret directory at `/curdir` in a running `cowsaypy` instance, instead of running the cow script run `ls` to inspect what was mounted:

- Data is typically transferred to and from running containers with mounted directories
- `-v` flag indicates a mount point and is specified with the absolute path of the host directory and where it will be mounted in the container
- Eg. mount the curret directory at `/curdir` in a running `cowsaypy` instance, instead of running the cow script run `ls` to inspect what was mounted:

In [20]:
```
!docker run --rm -v $(pwd):/curdir cowsaypy /bin/ls -lh /curdir
```

```
total 12K
-rw-rw-r-- 1 1001 1001 106 Jul 11 22:46 Dockerfile
-rw-rw-r-- 1 1001 1001  42 Jul 11 22:46 myapp.py
-rw-rw-r-- 1 1001 1001  30 Jul 11 21:54 test.txt
```

- Images can be exported with `docker save` but typically are shared by uploading to registries like Dockerhub
- `docker save cowsaypy | gzip > cowsaypy.tgz` will save the above image
- Big images can be very onerous to package this way, use a registry when you can
- Dockerhub has a huge number of images for many tasks, eg. Python of various flavours like the ones seen here, Pytorch and other big frameworks have their own, Nvidia provides many CUDA ready images through their own channels
- Images are often used together, so getting one for a database and another for a front end is common
- Use existing images where possible, don't needlessly reinvent the wheel

Not Covered

- Many more command line arguments to `docker run` for mounting locations, controlling network access, setting ulimit related things, selecting GPUs, etc. so read docs: https://docs.docker.com/reference/cli/docker/container/run/
- Other commands and techniques for working with Dockerfile
- Layers and optimisation in Dockerfile construction
- Any Docker Compose or other orchestration tools
- Other container frameworks like Apptainer which have their own properties
- Look to the example files for further techniques and explanation

Thanks!

Questions?