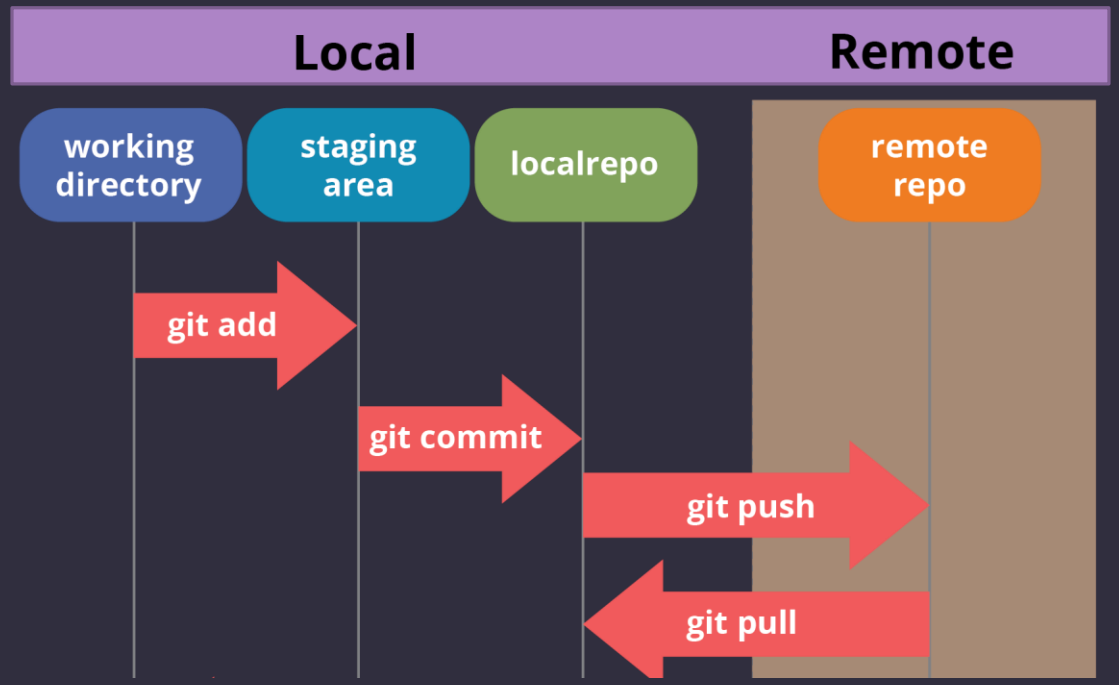
Code Collaboration

MODERN TEAM-BASED SOFTWARE DEVELOPMENT

How does git collaboration work?

- git has the concept of a ***remote***



- A remote is another instance of the repository in a "remote" place (e.g. online)
  - Several remotes can exist at the same time
  - They can be "forks" of the "original" remote

- Communication between a remote and the local copy is possible via ***push or pull.***
  - Before that you must decide to which online repository/branch to push and pull via the:
  - `git checkout -b <branch-name> --track <remote-name>/<branch-name>`
  - `git push -u <remote-name> <branch-name>`
  - and after you have established a connection, you can `pull` to bring advanced remote changes into the local branches or use `push` just like above to push your advanced changes to remote

GitHub

- It is a popular online service that hosts git repositories (i.e. the ***remotes***)

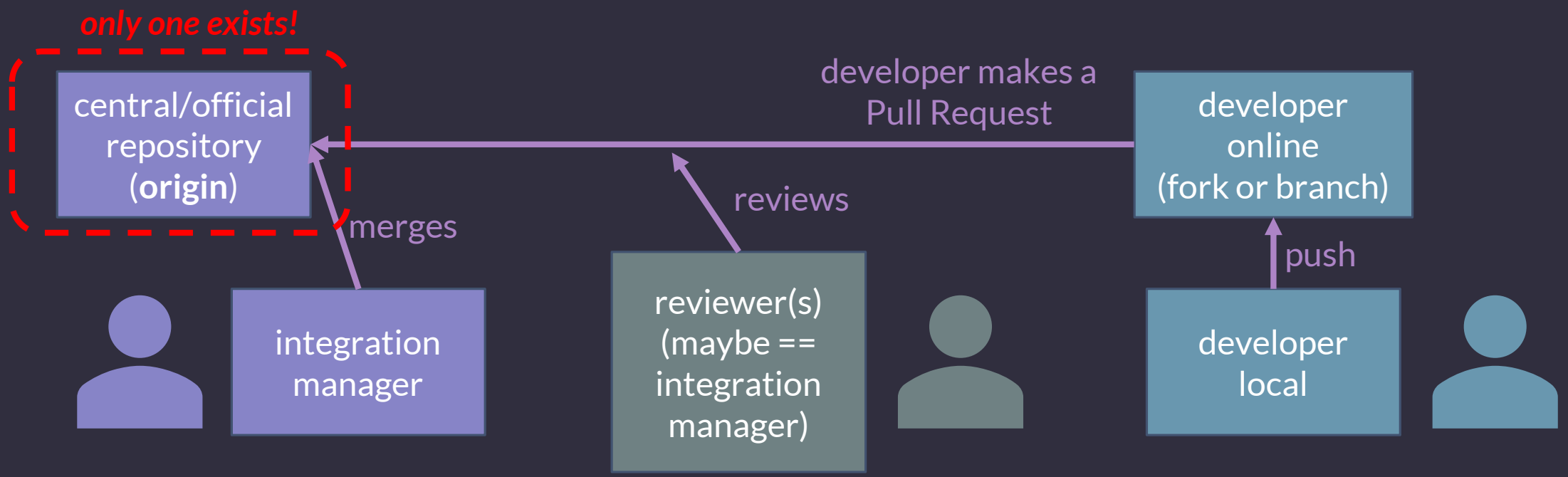
- Has many features to facilitate code collaboration
  - Issue tracker, comment functionality
  - Project management tools
  - Labels for issues and pull requests
  - Interface for integrating applications
  - Automatic testing and documentation hosting

- Long story short, it is absolutely amazing and completely free!

- ***GitHub is where modern code development takes place***
  - Python, Julia, VSCode are all developed on GitHub

But how does code collaboration ***really*** work?

- If everyone can push at any time, all hell breaks loose…
- All code is contained in the ***origin*** repository: everyone can read it and ***fork*** it
- Developers ***propose*** changes, by opening a Pull Request (PR)
- A few people have ***write access*** and can ***merge*** the PR, once the review is finished



Example of a Pull Request

- https://github.com/JuliaDynamics/ChaosTools.jl/pull/219

- Notice that if you update your own fork/branch via pushing from your local, your Pull Request is automatically updated!
  - After all, a pull request is about the ***branch***, not the specific commit it was opened at

Issues/bug reports the good way

- Do you want the developer to fix the bug you found?
- You should open a ***Good Issue Report*** on GitHub:
  1. ***Minimal Working Example:*** reproducible code with no unnecessary code commands that when run creates the bug. Having such will also clarify if you f***ed up, or there is an actual bug!
  2. Clear description of the bug and expected behavior
  3. Your environment status (package versions of relevant dependencies)
  4. Actually correct syntax formatting (code in backticks!)
  5. [Extra Kudos] Attempts to understand and locate the problem
  6. [Extra Kudos] Simply saying that you appreciate the package makes a lot of difference
- ***DO NOT*** paste links to code files or web pages that the developer ***has to read***
  - Summarize the content necessary for an answer, provide the link for further info
  - Make your code ***minimal*** and add it directly in the issue post with code formatting
- Always try your MWE in a clean environment with updated packages

When making a pull request...

- Fulfill the standard Pull Request checklist:
  - The PR has a good summary of proposed changes
  - If new features are added: tests and documentation for those are in the PR
  - If bugs are fixed: new tests are added to capture the previously unfixed bug...
  - If the repo has a Pull Request guideline, be sure you have read it first!
  - ***Make sure the tests pass!*** Unlikely to get a review to your PR without passing tests!
- Your PR is your creation and you will likely have feelings of attachment to it
  - Review comments will come. Some may point out mistakes, or other deficits of your PR.
  - Lower your defensiveness and welcome these comments
  - View them as a chance to improve the PR, not as a negative quality of your own self
- Throughout the lifetime of the PR, updated the DESCRIPTION (top-most comment in the PR discussion) with current status quo
- If a PR is taking too long (e.g., a lot of review stages), don't be sad. That's what it takes for good design! Feel free to regularly (1/week) ping developers.

When reviewing a pull request...

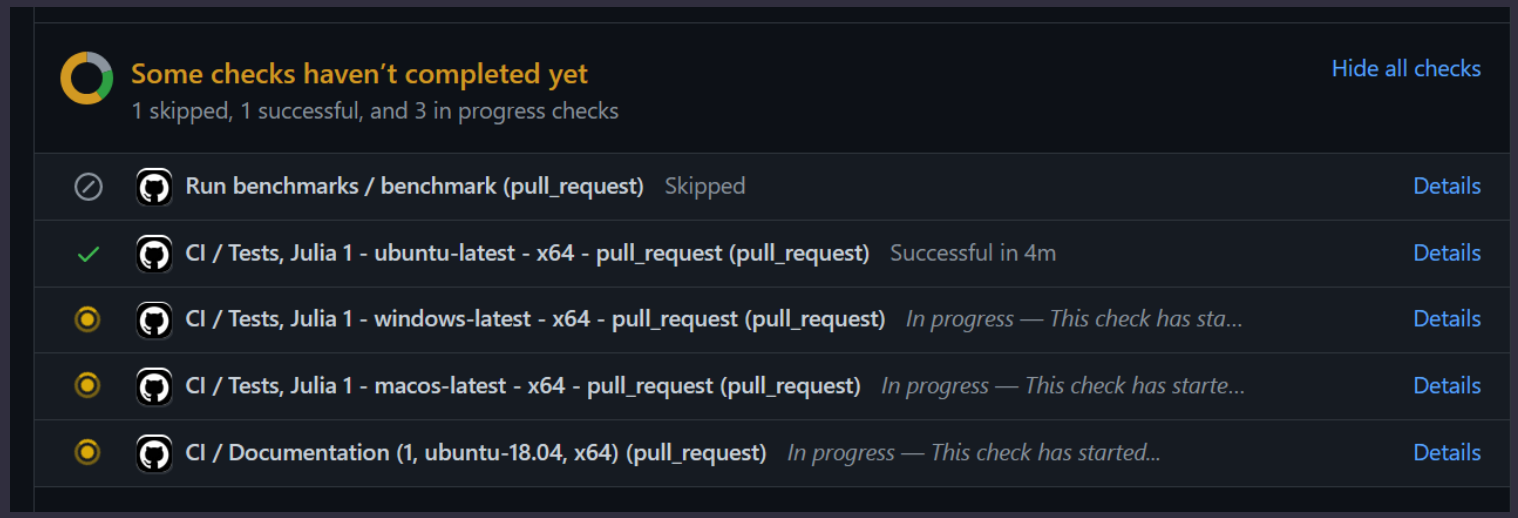
- Depersonalize code ownership, kinda the spirit of open source
- Raise confidence of PR author by appreciating the contribution and pointing out positive things in the review summary
- Use “we” instead of “you”, or third person, to communicate changes
  - “You should re-use X function here” → “We can re-use X function here.”
  - “Why did you do it X way? Y way is better” → “Y way is better than X for doing this, see ref. Z”
- If your expertise clearly deems something wrong, state this clearly as well
  - ***NO:*** Hm, maybe this part isn’t the best because it does type piracy, and some people don’t like it! I wonder if we can do it differently? For example some people do X.
  - ***YES:*** These lines commit type piracy and hence can’t be merged. Change them by doing X.

Exercise: a Pull Request

- Make a GitHub account if you don't have one already
  - Strongly recommended to actually upload a profile picture and description of you
- Fork the public target repository: https://github.com/JuliaDynamics/ScienceProjectTemplate
- Create a new local branch "update_contribs" and add two commits:
  - Make a file `contributors/firstname_lastname.txt` that describes you
  - Add your name in a bullet point under a section `## Contributors`
- Push the local branch to your fork
- Open a pull request from your fork to the origin
- ***After merging, clean up your fork***
  - Pull your main branch to update it with the origin's main branch
  - Delete the development branch
  - These steps are useful when you want to continue contributing to the original repo, which obviously isn't true for this exercise, but good practice to know

Continuous Integration

- Your test suite is run ***automatically*** by an online service (e.g. GitHub)
  - It is extremely simple to set up (at least for Julia/Python packages) and is free
- CI is triggered every time a new commit is pushed to the repo
  - This includes opened Pull Requests!



- CI ensures that nothing breaks over time and gives confidence
- CI also generates documentation (more on that on the documentation part)

Continuous Integration via GitHub actions

- In the repository you forked in the previous exercise, there is a special file:
- .github/workflows/ci.yml
  - (Some operating systems hide folders and files starting with . by default. Change this!)

- Let's go over this file to see how it allows us to test our code automatically

Open source licensing...

- Code on GitHub without a license is untouchable: all rights to the source code are reserved by the owner, and no one may reproduce, distribute, or create derivative works from your work... ***(kinda defeats the purpose of having it open!)***

- So, put a license there! Which one? Use https://choosealicense.com/

- My recommendation: ***MIT license***
  - Most permissive license, allowing absolutely anything to be done with your code, while giving no warranty or reliability claims (no burden to you)
  - Used by 44.69% of projects on GitHub
  - The Julia programming language, and almost all its packages, use this license

Your code → uploaded and tested on GitHub

- Put your codebase on GitHub. Strongly recommended to use MIT license.
- If you are too scared that people will “steal your research”, then…
  - …remember: science is about openness, and acknowledging and using other’s work
  - …consider: someone that knows nothing about your work can really publish it faster than you?
  - Anyways, make a private repository if you don’t want people to “steal your research”
- Other hosting services exist, second most common is GitLab. Comparison with GitHub:
  - GitLab is not “owned by Microsoft”, can be self-hosted, continuous integration can access your computing sources
  - GitHub has better community support, is more well known, more accessible
- Once the repo is on GitHub, add continuous integration (CI) by following the slide on CI or by looking at the provided sample folders for making your code CI. Add the files for CI by making a local branch, adding the files, pushing the branch to the remote, and then making a Pull Request (PR) from the branch to the `main` branch. The PR should already show the tests being run. Wait until they pass, and then merge the PR. Add another tag to your code at this point called “online”.

Your code → collaborative project

- Invite as collaborator your exercise partner via the GitHub interface.
- You each need to do pull requests to each other, and review the pull requests. The pull requests need to add an actual project description in each project's README file! So, explain, in 2-5 minutes, what your codebase is about, so that your exercise partner can write what they think into the README.md.
- Your partner thus must make a fork of your project, and then set up a clone of that remote fork on their local machine. After they make a branch on their local copy, they add the README.md info, commit, and push to the remote. Then, they open a Pull Request.
- Keep your partner's repo forked, because you will be attempting to do full science reproducibility later on!