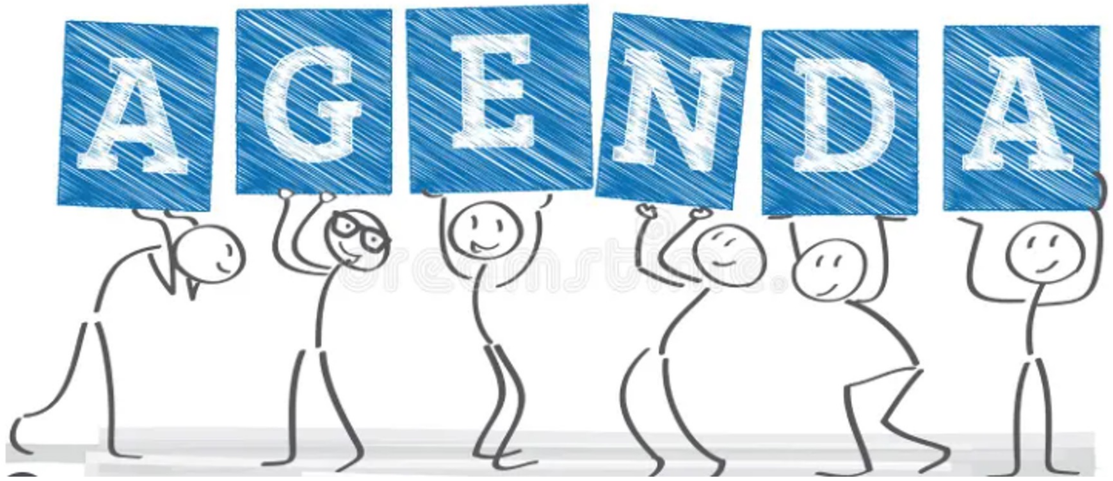
Operating Systems

Lecture 1.5

Workflow of git, GitHub and CI/CD Pipeline

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda

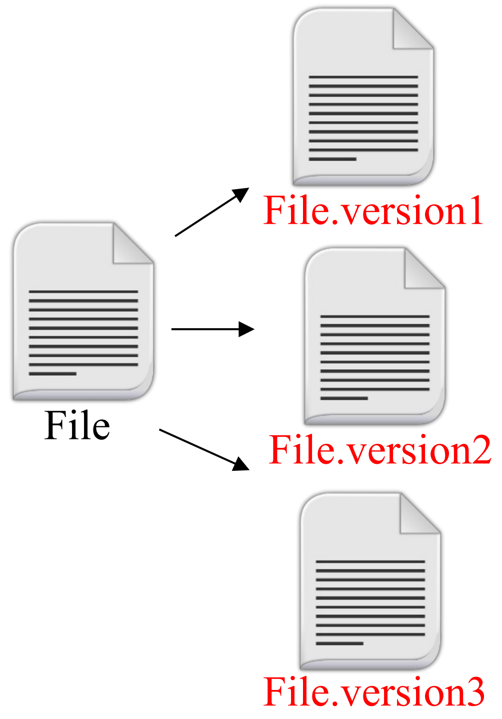


- Types of Version Control Systems
- Downloading, Installing and Configuring **`git`**
- Basic Workflow of **`git`** on your local repo
- Working with branches in **`git`**
- Web Portals and Cloud Hosting Services for **`git`**
- Basic Workflow of **GitHub**
- Creating a new **GitHub** repo and *pushing* your local repo on **GitHub**
- Adding a new Feature in your local repo and push it to GitHub (Adding **tags** and **releases**)
- *Fork*, *Clone* and Contribute to a Friend's repo using *Pull Request*
- Overview of **CI/CD Pipeline**
- CI/CD workflow with **GitHub Actions**

What are Version Control Systems?

Version Control System

A Version Control System is a software tool that records changes to a file or a set of files over time, so that you can recall specific versions later.



VCS allows to maintain history of different versions of a file

To move back and forth between these versions

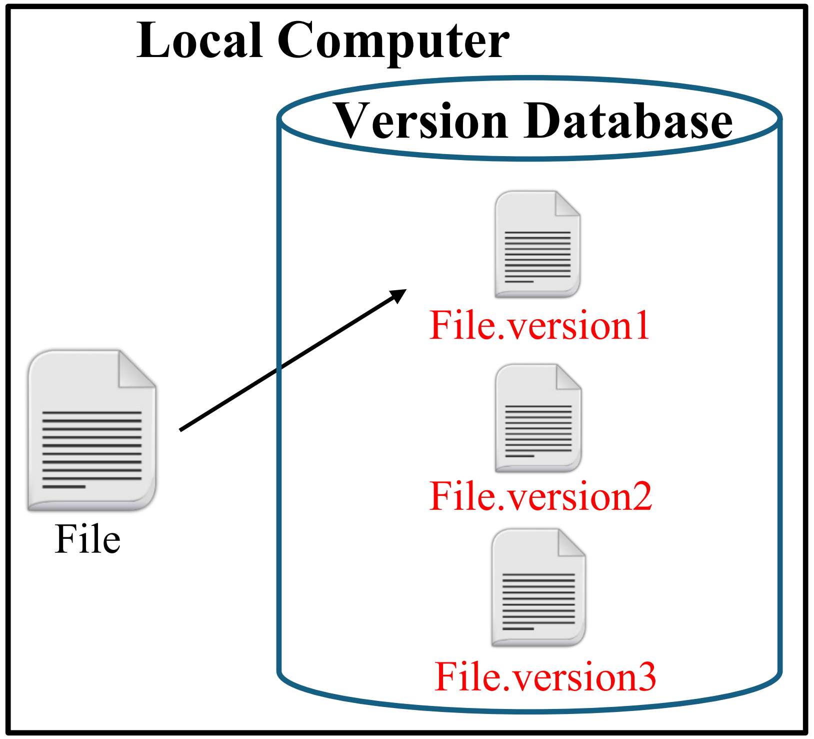
Compare different versions

Merge multiple versions of same file

Lock other users when one user is altering a file

Local Data Model

A local VCSs maintains a version database on your local system that keep track of all the changes made to file(s). By applying the change sets you can move from one file version to the other



Source Code Control System (SCCS-1972)

- It was written in C, developed by AT&T and was for UNIX only
- It just save the snapshot of the changes, If you want ver.3 of a file, you take ver.1 of the file and apply two set of changes to it to get to ver.3

Revision Control System (RCS-1982)

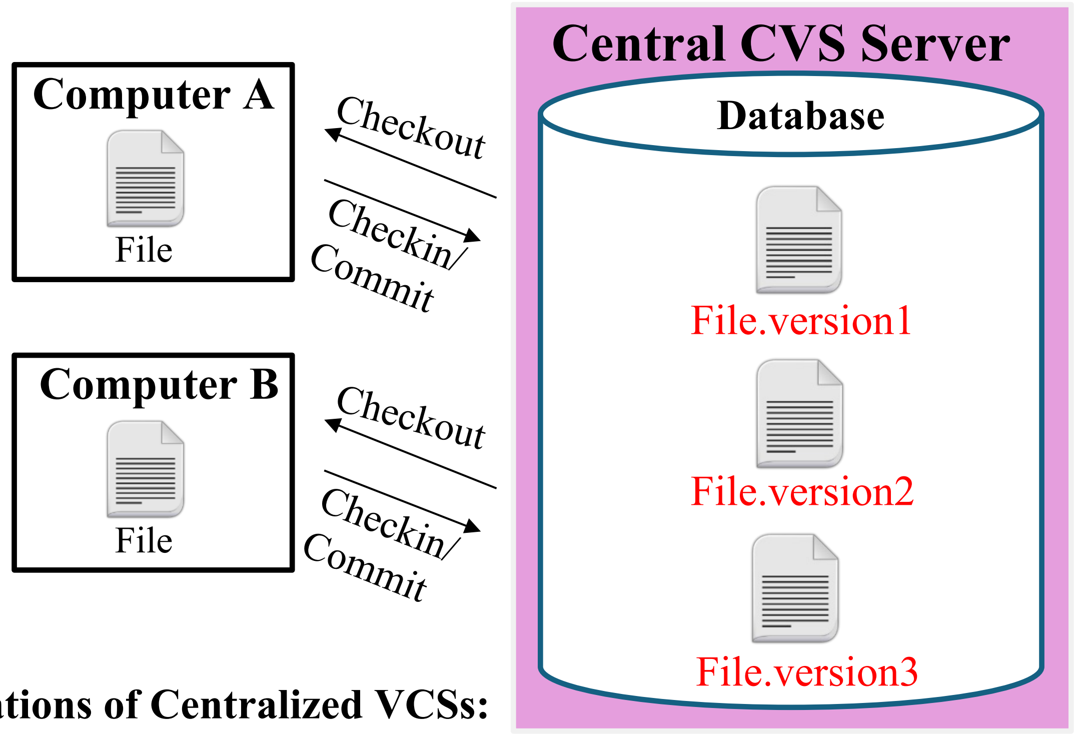
- It was written in C, developed at Purdue University, and other than UNIX works on PCs as well
- RCS keeps the most recent version of a file in its whole form and if you want a previous version, you make changes to the latest version to re-create the older version

Limitations of Local VCSs:

- You can track changes in a single file
- Only one user can work with a file at a single time, team members cannot collaborate and work on the same project

Centralized Data Model

In central VCSs, there is a server machine that contains the version database (repository) which keeps track of number of clients working on those file(s)



Limitations of Centralized VCSs:

- Single point of failure as the server containing the version database may crash
- Developers do not have history of project on their local machines
- No collaboration if server is down
- No file renaming as cannot track directories

Concurrent Version System (CVS-1990)

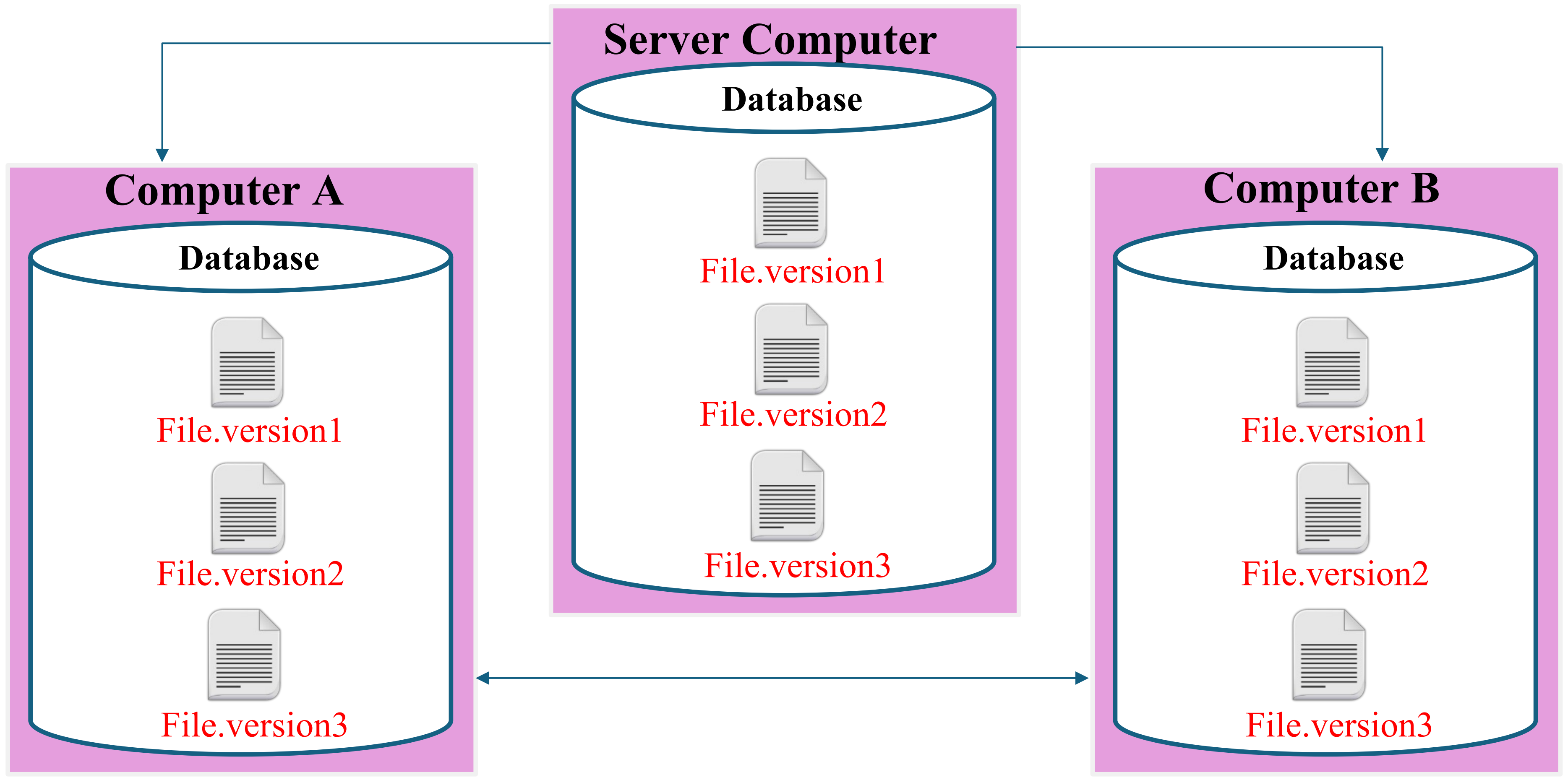
- Written in C, is open source, and available for UNIX and MS OSs
- Introduced the idea of branching
- CVS lack atomic operations
- File renaming not possible as CVS cannot track directories

Apache Subversion System (SVN-2000)

- Written in C, is open source, is cross platform and is faster than CVS
- Supports atomic commits
- Can track directories, so you can rename files within directories
- It can also track non-text files like images

Distributed Data Model

- In a DVCS, clients don't just check out the latest snapshot of the files; they fully mirror the entire repository (version database).
- Each developer works with his own local repository and changes are finally pushed or committed on the remote repository as a separate step.



Distributed Data Model (cont…)

Bitkeeper -2000
It was written in C, and is proprietary and closed source

BitKeeper
Scalable Version Control

Bitkeeper with limited functionalities was free and used to manage Linux Kernel

In 2005, the "community version of bitkeeper" stopped being free and it was then git was born

git -2005
Developed by Linus Torvald in 2005, is free and open source

git

It is compatible with all UNIX-like systems & MS Windows, written in C, TCL, Perl & python

Pros:

- Faster speed
- No risk of loosing history, as every user has complete mirror of repository

Cons:

- More space occupied on local disk of user
- More load on network while checking out project in local repository and committing project in remote repository

Downloading, Installing & Configuring `git`

Downloading and Installation

On Linux

```
https://git-scm.com
sudo apt-get install git
which git
git version
git help <git/tutorial/everyday>
```

You can Download git from official website

Or Download & install git using this command

Confirm the installation

To get help about any command or any concept

On Windows

```
https://gitforwindows.org/
git --version
git help
```

You can Download git GUI, CMD & bash interfaces

Confirm the installation

To get help about any command or any concept

GIT: GUI-Clients

Atlassian SourceTree	Platforms: Mac, Windows Price: Free License: Proprietary
Desktop	Platforms: Mac, Windows Price: Free License: MIT
GitKraken	Platforms: Mac, Windows, Linux Price: Free/Paid License: Proprietary
Git	Platforms: Windows Price: Free License: GNU GPL
cola	Platforms: Mac, Windows, Linux Price: Free License: GNU GPL

Git Configuration

User Configuration

~/.gitconfig

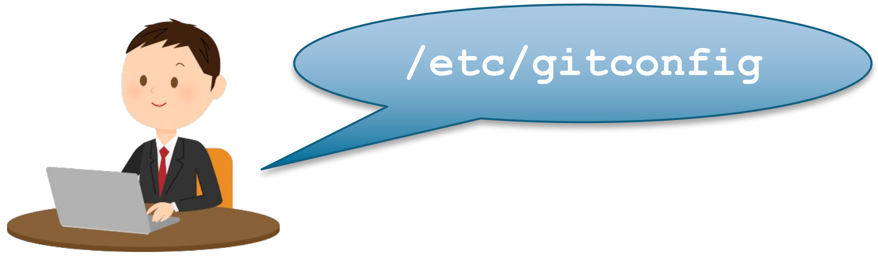
https://git-scm.com

```
$ git config --global user.name “Arif Butt”
$ git config --global user.email “arif@pucit.edu.pk”
$ git config --global core.editor “vim”
$ git config --global --list
$ cat ~/.gitconfig
```

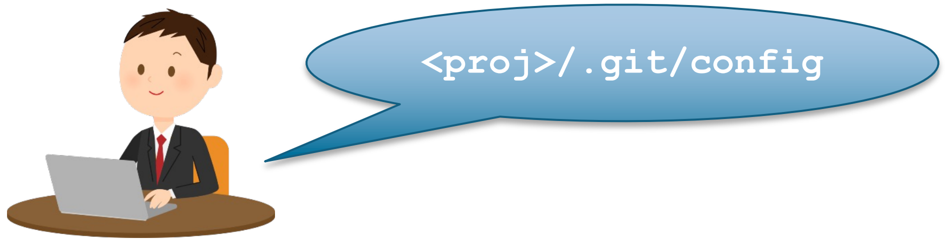
User Configuration Attributes

You can check values of these configurations using these commands

System Configuration

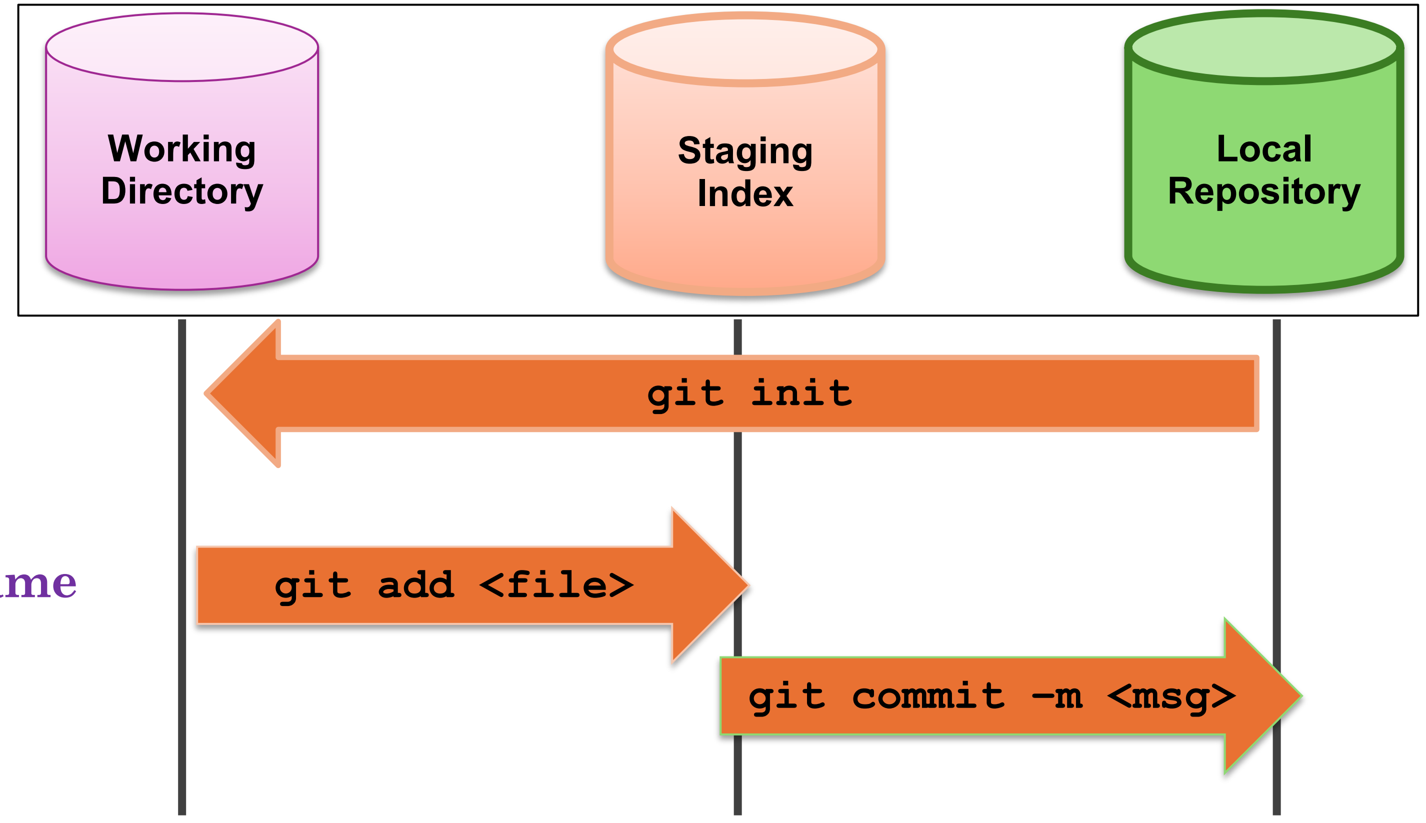


Project Configuration



Basic Workflow of `git`

Basic Workflow of git



- File creation
- Modification
- Deletion/Rename
- Ignore files

Working directory is any directory on your file system that has a a subdirectory named .git inside it

Staging Index is an intermediate area, changes doesn't commit directly from the working tree to repository. Instead changes are first made in the staging index

Repository or object store holds the changes in your source code over time as you perform commit ops

Instructor: Muhammad Arif Butt, PhD

Initialization and Life Cycle of file in `git`

```
Initializing git
$ git init
```

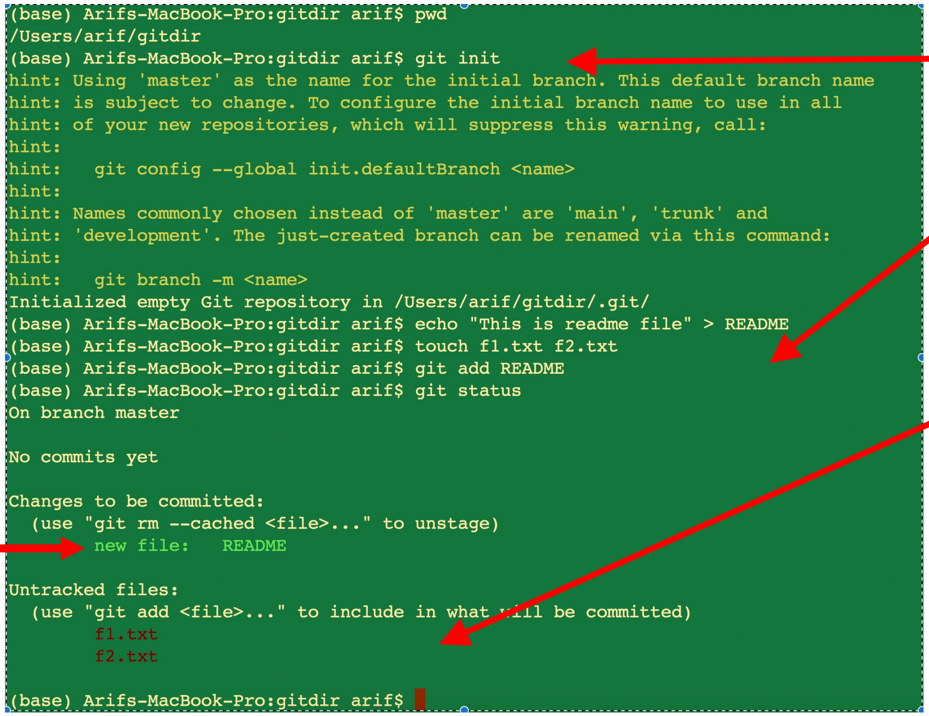
After configuration, next step is to initialize repository. It will make a hidden folder named .git in this directory. This is your local versioning database that track all the files/ inside the root directory of your project folder

```
$ git status
$ git add <filename>
```

Add file(s) or every change inside the staging area, where you put changes you want in the next commit

Tracked files: All the files which have been added at least once, or the files that were there in the last snapshot

- Unmodified
- Modified
- Staged



```
(base) Arifs-MacBook-Pro:gitdir arif$ pwd
/Users/arif/gitdir
(base) Arifs-MacBook-Pro:gitdir arif$ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/arif/gitdir/.git/
(base) Arifs-MacBook-Pro:gitdir arif$ echo "This is readme file" > README
(base) Arifs-MacBook-Pro:gitdir arif$ touch f1.txt f2.txt
(base) Arifs-MacBook-Pro:gitdir arif$ git add README
(base) Arifs-MacBook-Pro:gitdir arif$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        f1.txt
        f2.txt

(base) Arifs-MacBook-Pro:gitdir arif$ 
```

Initialize Repository

Create some files and add one to Staging Index

Untracked files: All the files in the working directory that have never been part of repository and are not even in the staging area

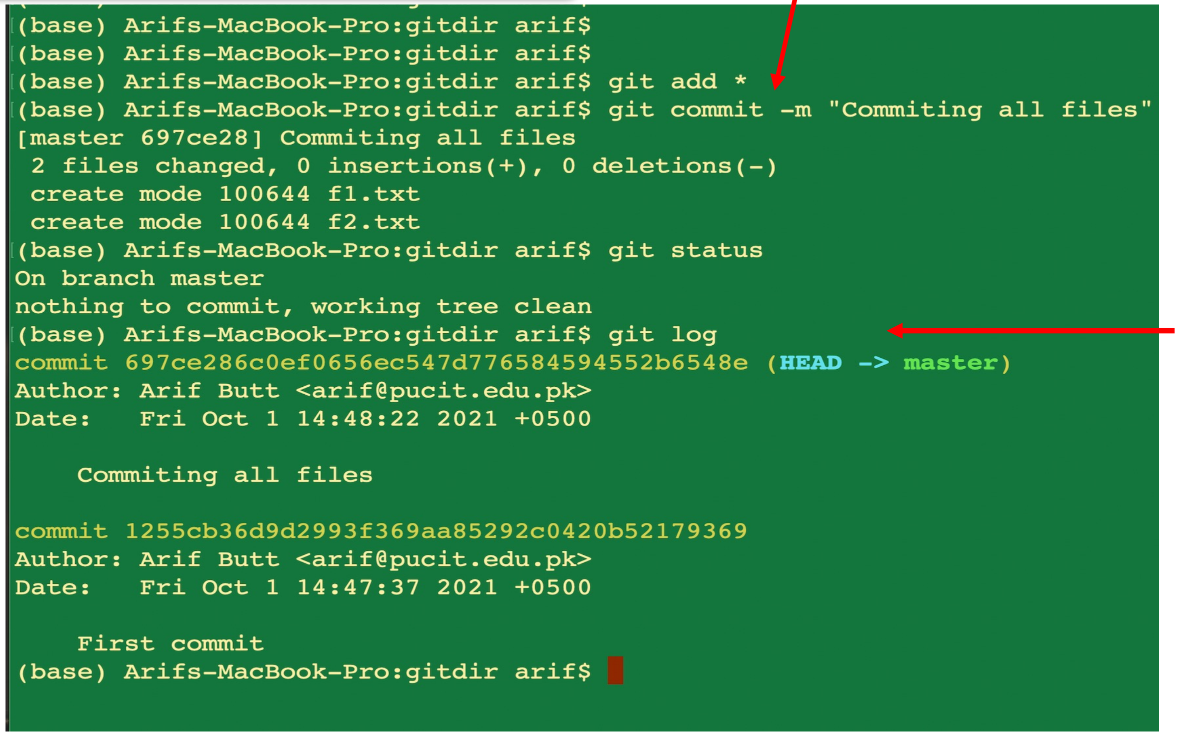
Life cycle of a file in git

Untracked	All the files which have been added at least once, or the files that were there in the last snapshot
Tracked	All the files which have been added at least once, or the files that were there in the last snapshot
Unmodified	If a file is unmodified, that means the copy of the file in the working directory, staging area and repo are same
Modified	If a file is unmodified, that means the copy of the file in the working directory is different than the copy of the file in staging area and repo
Staged	If a file is staged, that means the copy of the file in the working directory and staging area are same, but it is yet to be committed

Commit File and View Commit Log

```
$ git commit -m "message"
```

After adding all files to staging area now they are ready to commit



```
(base) Arifs-MacBook-Pro:gitdir arif$
(base) Arifs-MacBook-Pro:gitdir arif$
(base) Arifs-MacBook-Pro:gitdir arif$ git add *
(base) Arifs-MacBook-Pro:gitdir arif$ git commit -m "Commiting all files"
[master 697ce28] Commiting all files
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 f1.txt
 create mode 100644 f2.txt
(base) Arifs-MacBook-Pro:gitdir arif$ git status
On branch master
nothing to commit, working tree clean
(base) Arifs-MacBook-Pro:gitdir arif$ git log
commit 697ce286c0ef0656ec547d776584594552b6548e (HEAD -> master)
Author: Arif Butt <arif@pucit.edu.pk>
Date:   Fri Oct 1 14:48:22 2021 +0500

    Commiting all files

commit 1255cb36d9d2993f369aa85292c0420b52179369
Author: Arif Butt <arif@pucit.edu.pk>
Date:   Fri Oct 1 14:47:37 2021 +0500

    First commit
(base) Arifs-MacBook-Pro:gitdir arif$
```

Check log

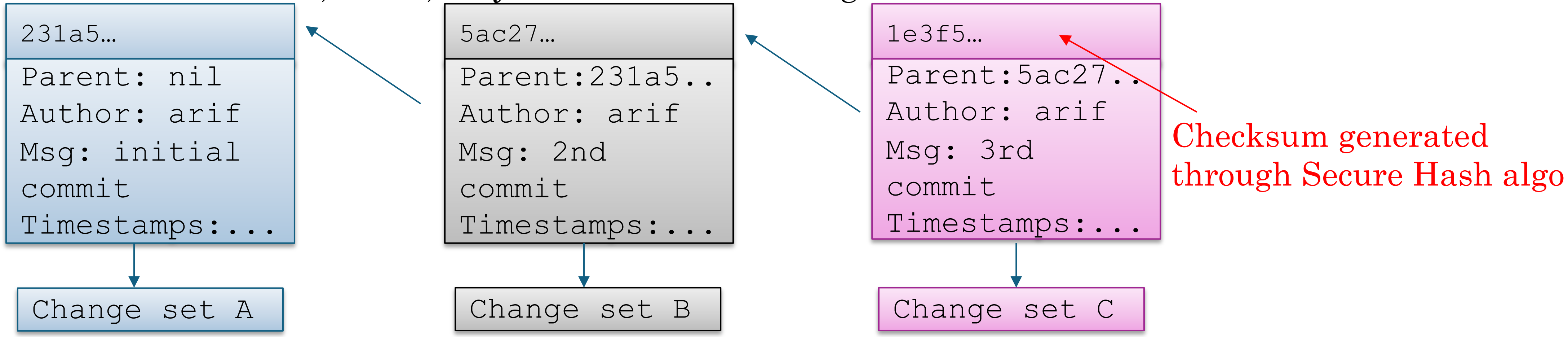
```
$ git log [--oneline][--author="name"]
commit <sha of commit o/p as 40 hex digits>
Author: username <email>
Date: <date and time>
<commit message>
```

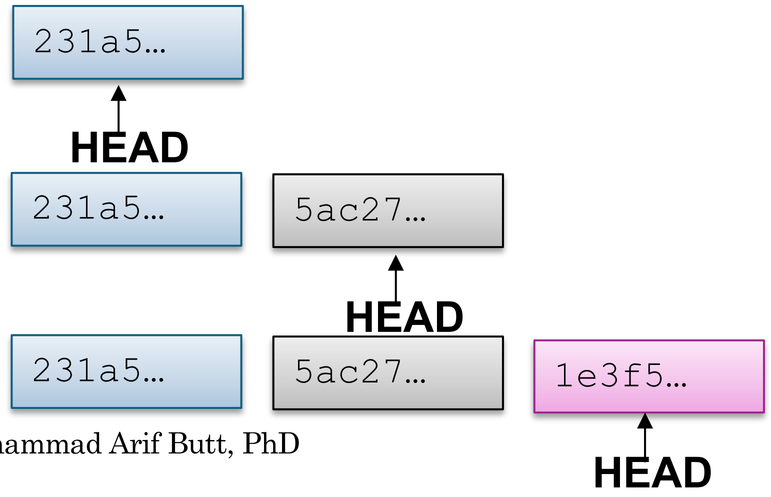
You can check log of commits and by whom it is committed

It will show you list of all commits in a specific format:

Commit Objects and Head Pointer in git

- Suppose you have made three commits in your project, that means there are three change sets.
- Each commit object refers to a change set.
- Note that the parent of each refers to a previous commit.
- We can see who has committed, when, why and with what change





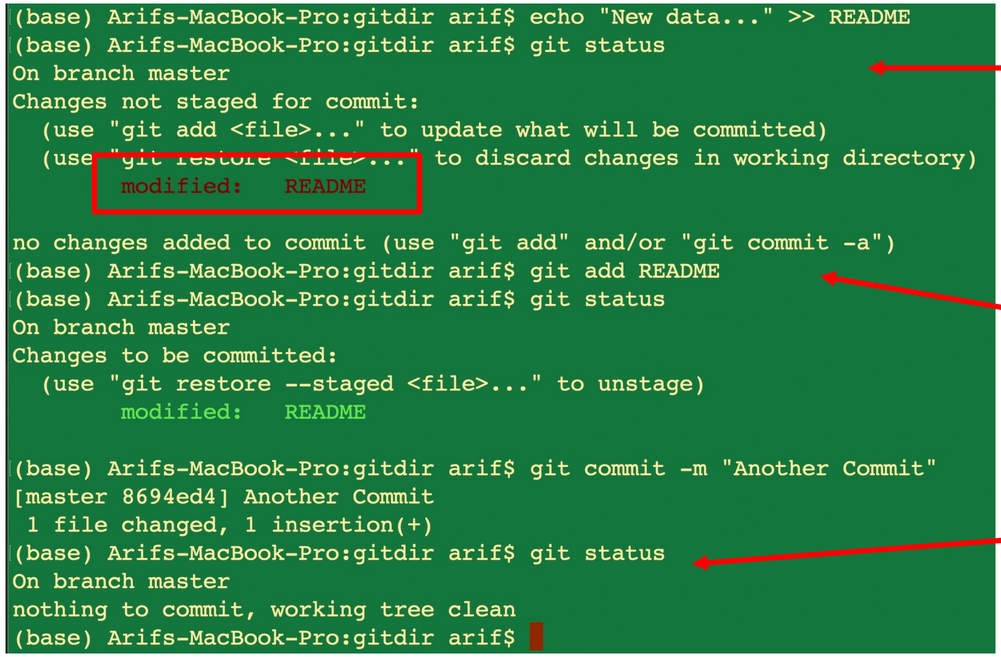
git maintains a reference variable called HEAD, which points to a specific commit in repo

As we make a new commit the HEAD moves to point the next commit

```
$ cat .git/HEAD
refs/heads/master
$ cat .git/refs/heads/master
5ac27..
```

Edit and Delete a File

➢ Edit File



```
(base) Arifs-MacBook-Pro:gitdir arif$ echo "New data..." >> README
(base) Arifs-MacBook-Pro:gitdir arif$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README

no changes added to commit (use "git add" and/or "git commit -a")
(base) Arifs-MacBook-Pro:gitdir arif$ git add README
(base) Arifs-MacBook-Pro:gitdir arif$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README

(base) Arifs-MacBook-Pro:gitdir arif$ git commit -m "Another Commit"
[master 8694ed4] Another Commit
 1 file changed, 1 insertion(+)
(base) Arifs-MacBook-Pro:gitdir arif$ git status
On branch master
nothing to commit, working tree clean
(base) Arifs-MacBook-Pro:gitdir arif$
```

We have already created a file README, added in staging index and then committed it to the repo. Make changes in the file and check status.

You again need to add and commit the file

Check status

➢ Delete File

```
$ rm f1.txt
$ git add f1.txt
$ git commit -m “deleted”
```

Option 1: Move the file out from the working dir into trash and then tell git about it

```
$ git rm f1.txt
$ git commit -m “deleted”
```

Option 2: Tell git to remove the file and add it to staging index in a single command

Rename a File

- Rename file

```
$ mv f1.txt newf1.txt
$ git add newf1.txt
$ git rm f1.txt
$ git commit -m "rename"
```

Option 1: Move or rename files using the GUI file browser or file system commands.

Then come back and tell git about those changes

```
$ git mv f1.txt newf1.tx
$ git commit -m "renamed"
```

Option 2: Move/rename file from git command line.

Ignoring Files

- Write files/directories names to be ignored in a text file.
- Git normally checks `gitignore` patterns from multiple sources, with the following order of precedence:
  - The patterns read from a file named `.gitignore` in the same directory or in any parent directory up to the top level of the working tree.
  - The patterns read from `.git/info/exclude` file in the project directory.
  - The patterns read from file specified by the configuration variable `core.excludesFile`

```
*.o
*.tar.gz
*.log
*.[oa]
*.exe
myexe
logs/**
dir1/
```

Moving to a Previous Commit

At its core, the **`git reset`** command moves the current branch pointer (HEAD) to a different commit.

- Soft Reset

```
$ git reset --soft <Commit ID>
```

- Head is moved to specific commit ID
- Keeps staging area and working directory unchanged

- Mixed Reset

```
$ git reset --mixed <Commit ID>
```

- Head is moved to specific commit ID
- Staging area is also changed to match the local repository
- No changes are made in the working directory

- Hard Reset

```
$ git reset --hard <Commit ID>
```

- Head is moved to specific commit ID
- Staging area and working directory both match the local repo

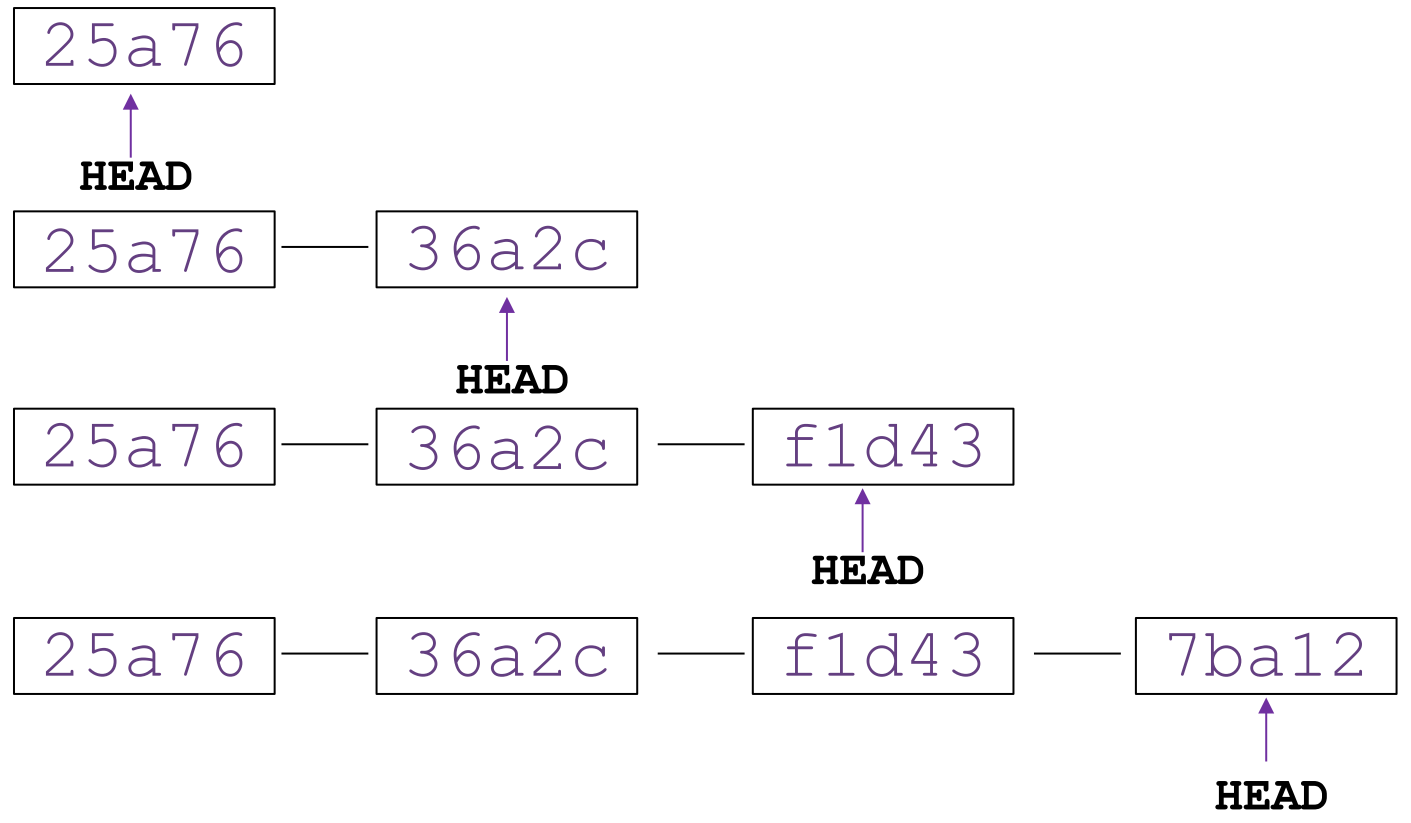
You can use the **`git reset`** command to un-stage a file, but keep the changes (not mess with commits)

```
$ git reset <file-name>
```

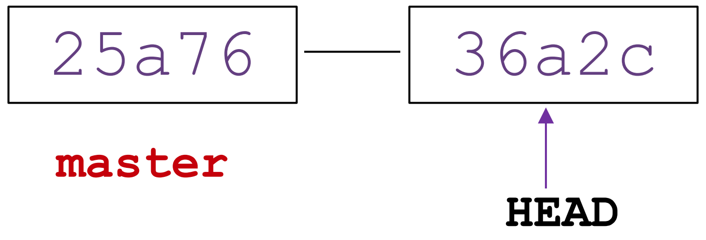
`git`
Branches

Overview of git Branches

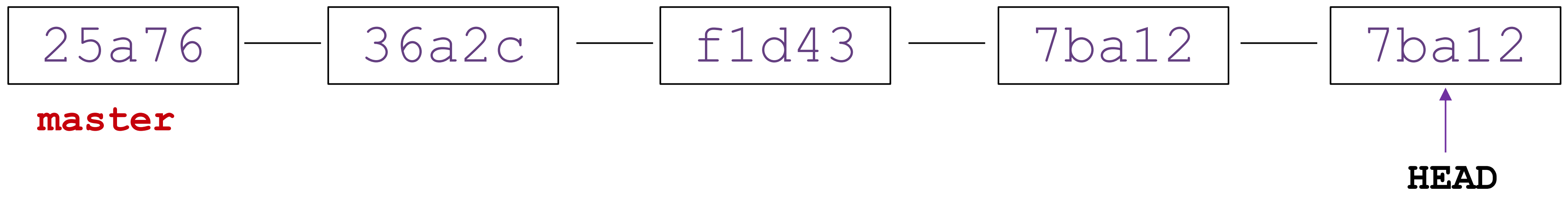
- A `git` branch represents an independent line of development.
- Every `git` repository has at least one branch called the master/main branch.
- An illustration of master branch is shown below:



Overview of git Branches (Cont.)

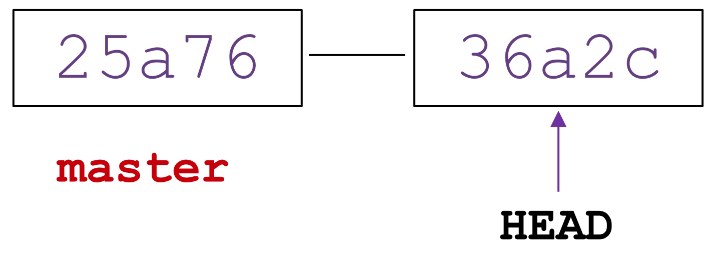


- Suppose you are working on a project and have done some commits on the master branch which is the main line of your project development as shown above. You think of adding a new feature to your project but you are not sure whether it will work or not
  - **OPTION 1:** You continue working on the same branch. If the new feature is a success, its **`GR8`** and the development continues as shown below:

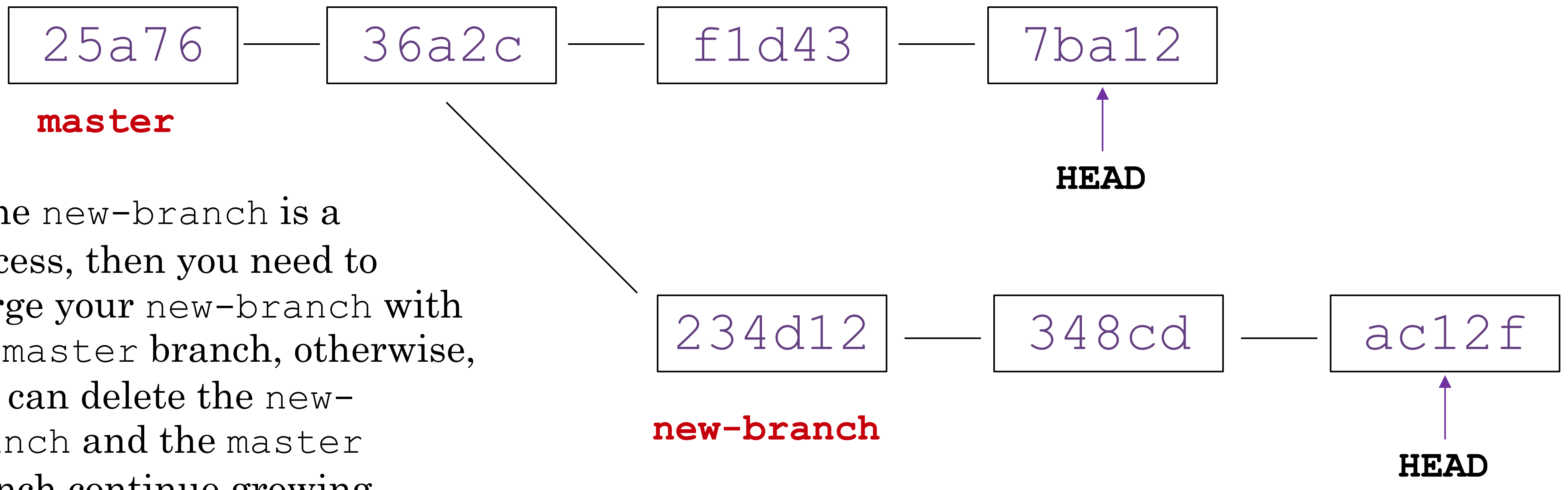


- However, if the new feature is a failure you roll back to commit with **`SHA 36a2c`** using a **`git reset`**, and your master branch again becomes similar to the one shown at the top

Overview of git Branches (Cont.)



OPTION 2: Create a new branch and try your new ideas there and if those ideas do not work you just throw away that branch and your master branch continues moving ahead without any issues



If the `new-branch` is a success, then you need to merge your `new-branch` with the `master` branch, otherwise, you can delete the `new-branch` and the `master` branch continue growing

Why & How to use Branches?

Why?

- Work on a new feature, so new work does not mess up the tested and runnable "master" code. Code on one branch won't affect other branches (until you merge).
- Several people can work on features at the same time, without conflicts. Each person works on his own "feature branch".
- When a bug is reported, create a new bugfix branch to work on a fix. Once the fix is thoroughly tested you merge it into the main/master branch. An extra benefit is the "bugfix" branch will contain a history of what you changed to fix the bug.
- Try new PoC code that may or may not be added to your project using a separate branch.

How?

- Create a new branch named `foo`:

```
$ git branch foo
```

- Show all branches (* shows current branch):

```
$ git branch -a
```

- To rename a branch:

```
$ git branch -m <old> new>
```

- To delete a merged/unmerged branch:

```
$ git branch -d/-D <branch>
```

- To switch to another branch , after this any commits will be added to the `foo` branch:

```
$ git checkout foo
```

- Compare two branches:

```
$ git dif master foo
```

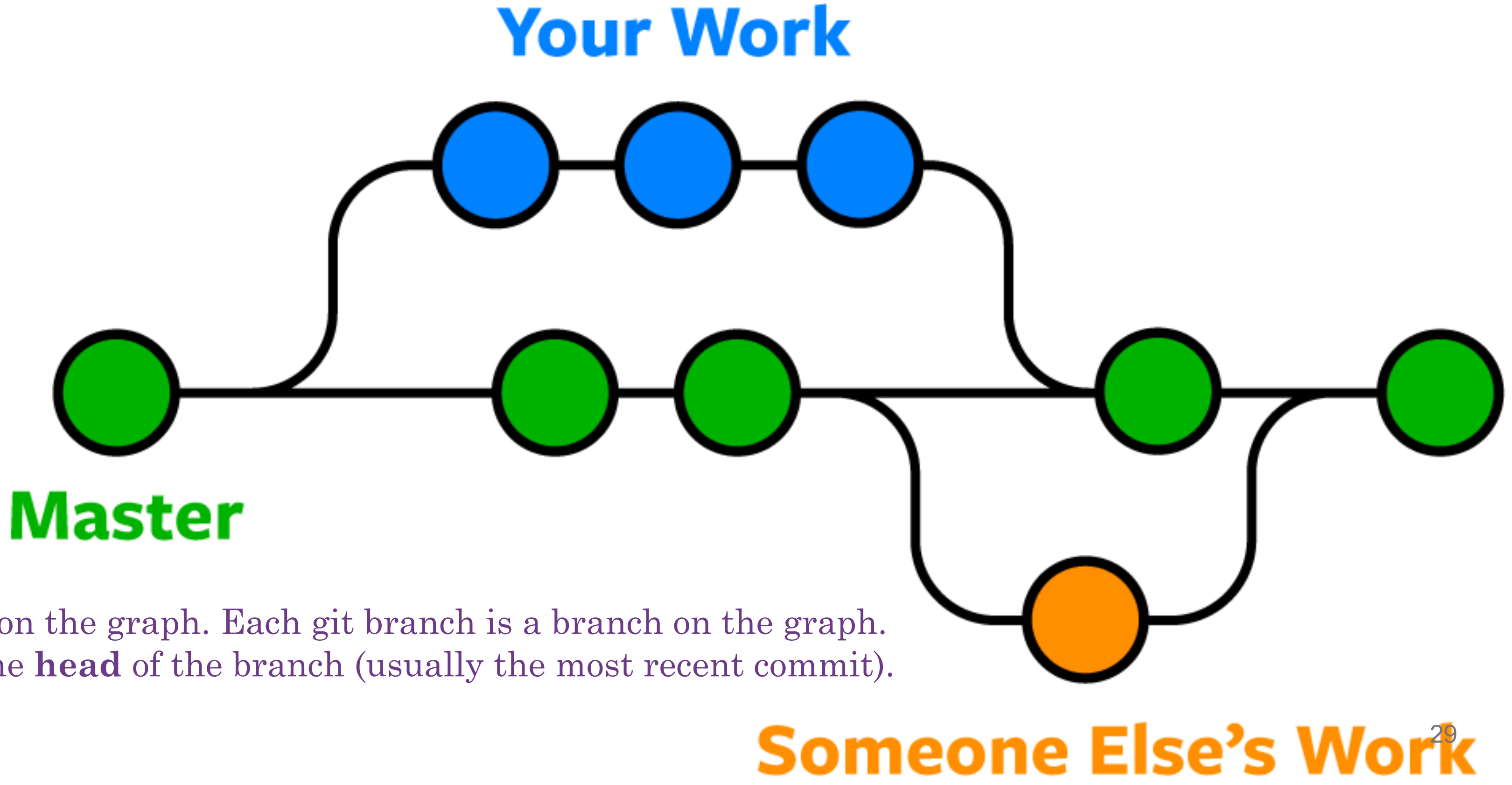
Your working directory should almost be clean in order to switch, otherwise, git will not let you switch to another branch

`git`
Merging the Branches

Merging Branches in git

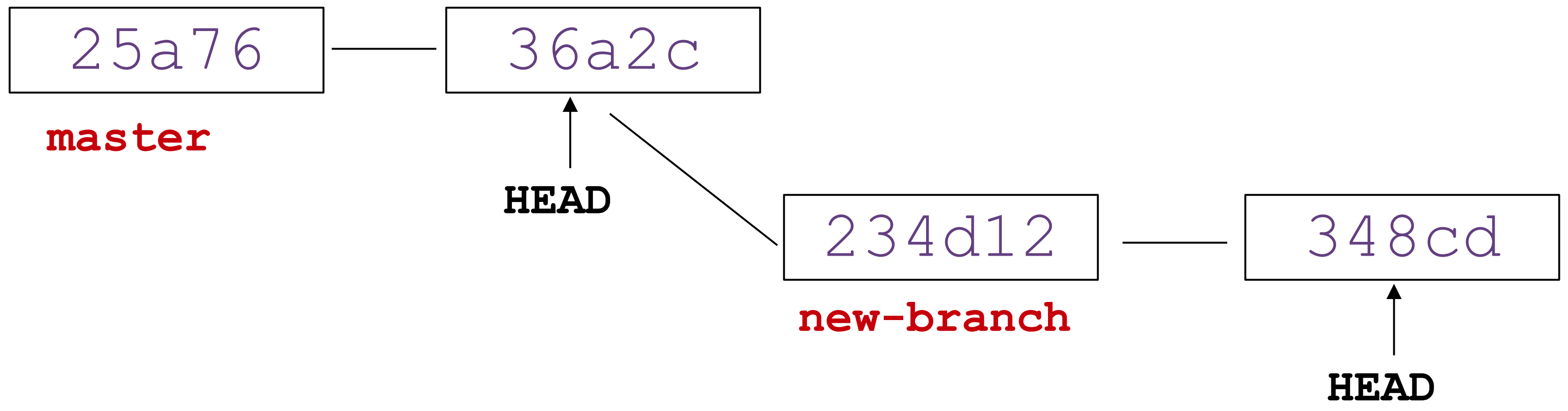
- Now we know how to create a new branch and how to perform development on that branch. After we are done developing and testing the new feature, it is time to bring those changes back to the master branch. For this we need to do a merge.
- A **merge** takes the changes (commits) from one branch and **integrates them** into another branch.
- Usually, you merge a **feature branch** into the **main branch** when the feature is finished.
- There can be two types of merges
  - Fast Forward Merge
  - Real Merge

A **merge** in `git` is the process of bringing together changes from one branch into another. If branches are diverged, `git` creates a merge commit; if not, it fast-forwards.

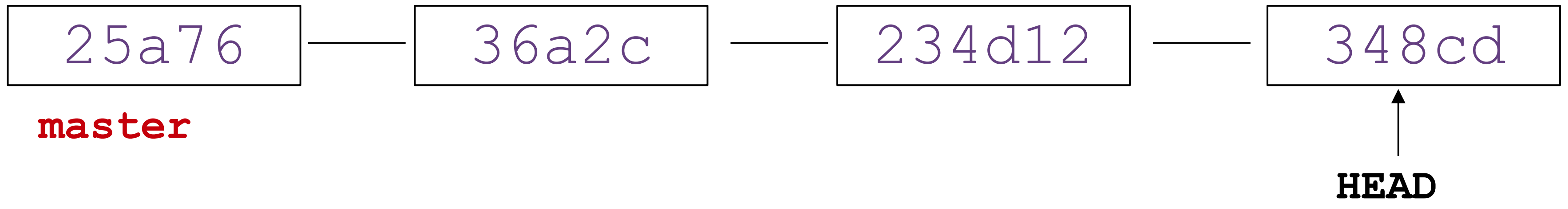


A git repository is a **graph**, with commits as nodes on the graph. Each git branch is a branch on the graph. The branch name is a **label** that always points to the **head** of the branch (usually the most recent commit).

Fast Forward Merge



After fast forward merge:

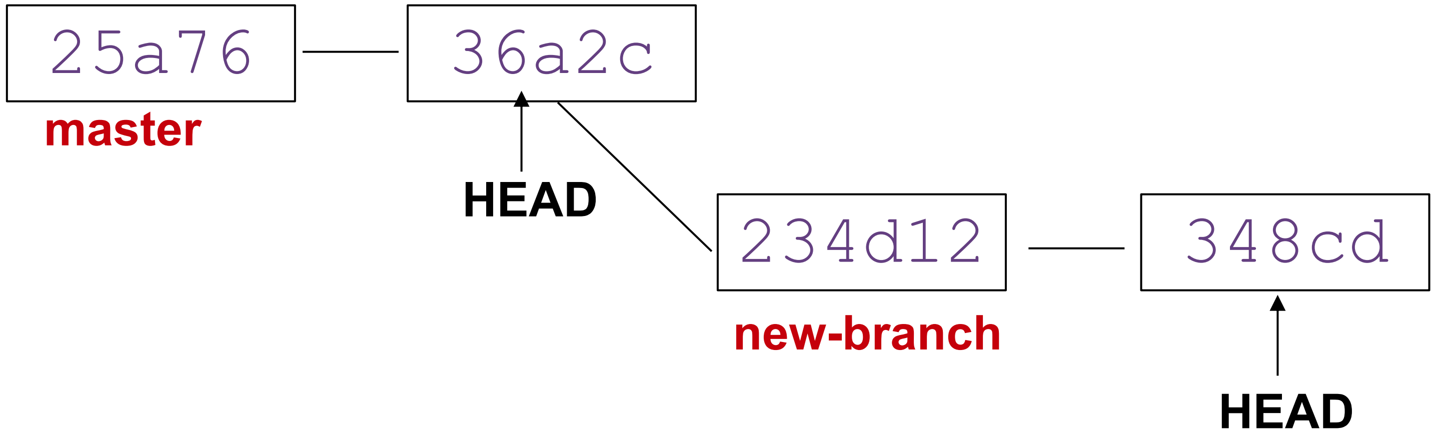


Before you give `git merge` command, your current branch should be the receiving branch

```
$ git checkout master
$ git merge new-branch
```

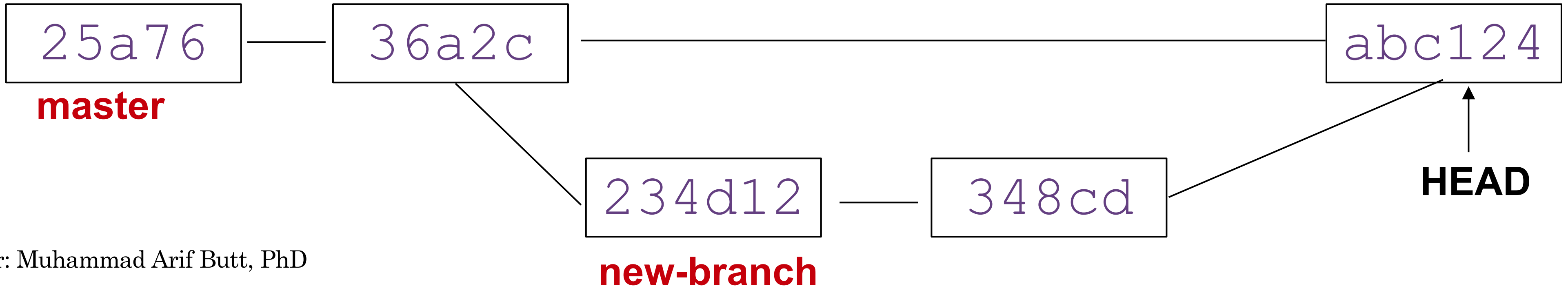
Real Merge

Suppose you made a new branch and no further commits have been done on the master branch after the creation of new-branch as shown:



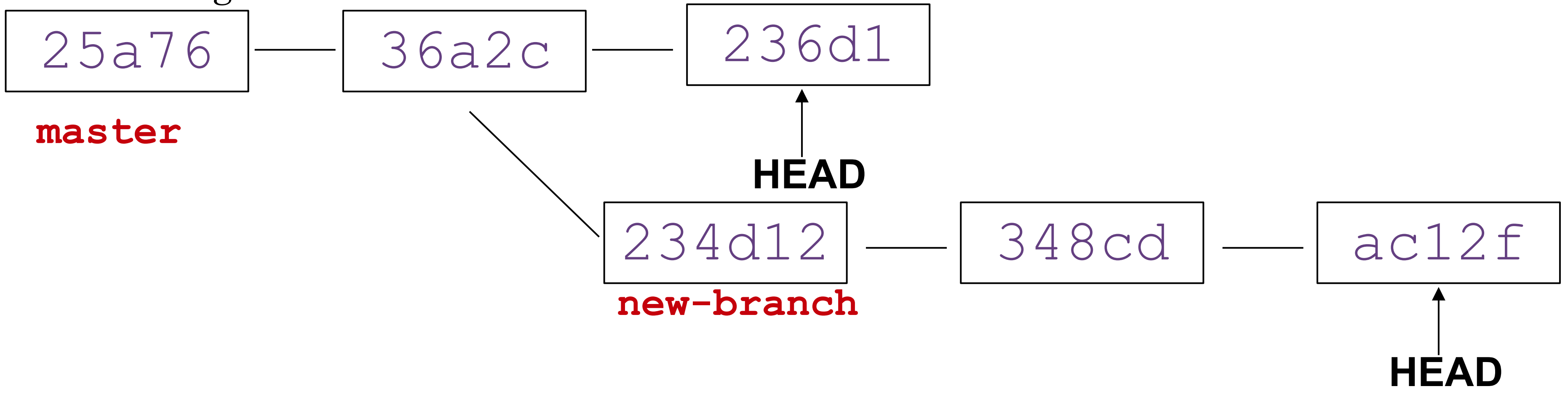
You can always force git not to do a fast forward merge, rather do an additional commit merge

```
$ git checkout master
$ git merge --no-ff new-branch
```

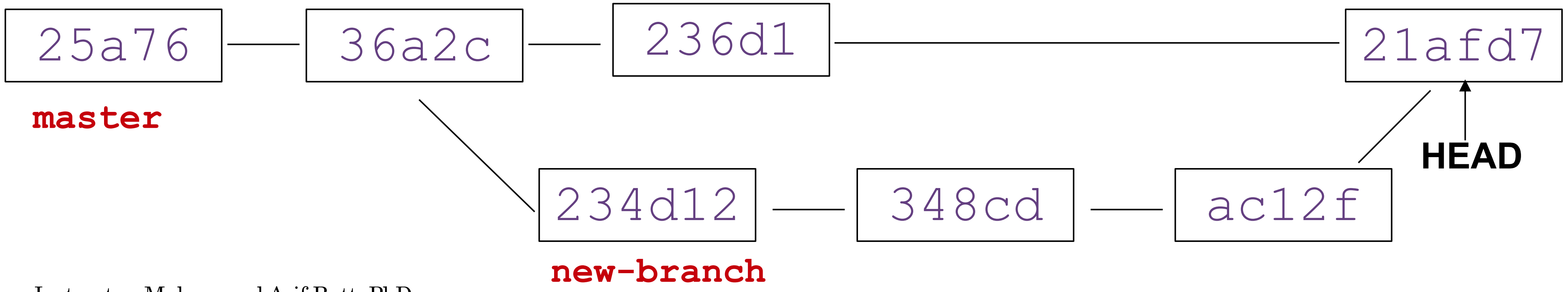


Real Merge (Cont...)

In the following scenario a fast forward merge is not possible. So once you do a merge, git will perform a real merge.



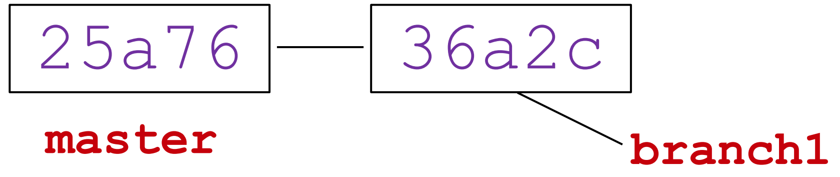
```
$ git checkout master
$ git merge new-branch
```



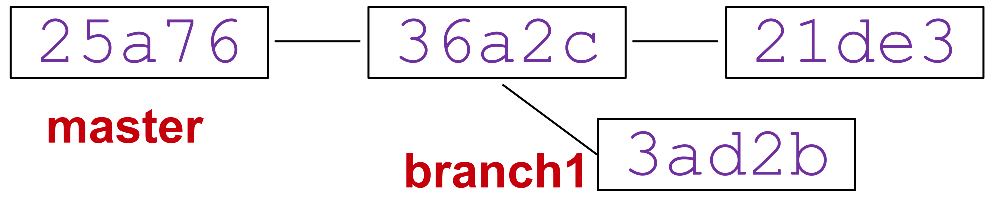
`git`
Handling Merge Conflicts

What is a Merge Conflict?

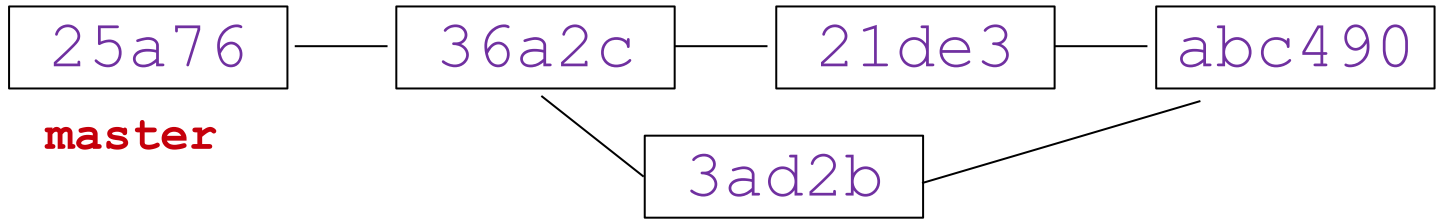
Suppose there are two branches **master** and **branch1**, as shown:



Both have a file suppose `file1.txt`, which is of course similar in both. A developer on **master** branch edit line#25 of `file1.txt` and do a commit. Another developer on **branch1** edit line#50 of `file1.txt` and do a commit



Now if you merge, it will be a success, because both have made changes to same file, but to different lines



Handling Merge Conflicts

However, if both the developers have made changes to same line or set of lines a conflict will occur, which git cannot handle and it will give a message that auto-merging failed. In case of a merge conflict we have three choices to resolve the conflict:

- **Abort merge:**
  ```
  $ git merge –abort
  ```

- **Resolve manually:** Open the file in some editor and perform the changes manually, add, commit, and finally perform merge
  ```
  $ git merge <branchname>
  ```

- **Use merge tools:** You can use different tools to automate this process like `araxis, diffuse, kdiff3, xxdiff, diffmerge`
  ```
  $ git mergetool --tool=diffuse
  ```

Semantic Conflicts

Scenario:

- Two developers edit the same file. Both add the same method to a class, but in different locations. The `git` will see:
  - Person 1's change: new lines at top.
  - Person 2's change: new lines at bottom.
- Since line ranges don't overlap, `git` concludes *"no conflict"* → merge succeeds.
- Result = duplicate method (logically incorrect, but syntactically fine).

Why does this happen?

- Git's merge algorithm is line-based, not syntax/semantics-aware.
- It only checks if two changes touched the same lines.
- If changes are in different places, `git` assumes they can coexist.
- Git has no understanding of code semantics (e.g., "duplicate method in a class is wrong").

Why does this happen?

- Humans must perform code review, to catch logical errors like duplicated methods.
- Use automated tools like linters, static analyzers, or compiler errors to catch duplicates.
- Use CI/CD Pipeline checks (run tests automatically after merges).

Web Portals and Cloud Hosting Services for `git`

Hosting Services for git Repositories

The way there are different web hosting services available on the Internet cloud, similarly there are hosting services available for repositories of distributed versioning systems as well

ATLASSIAN Bitbucket
https://bitbucket.org

GitHub
https://github.com

GitLab
https://gitlab.com

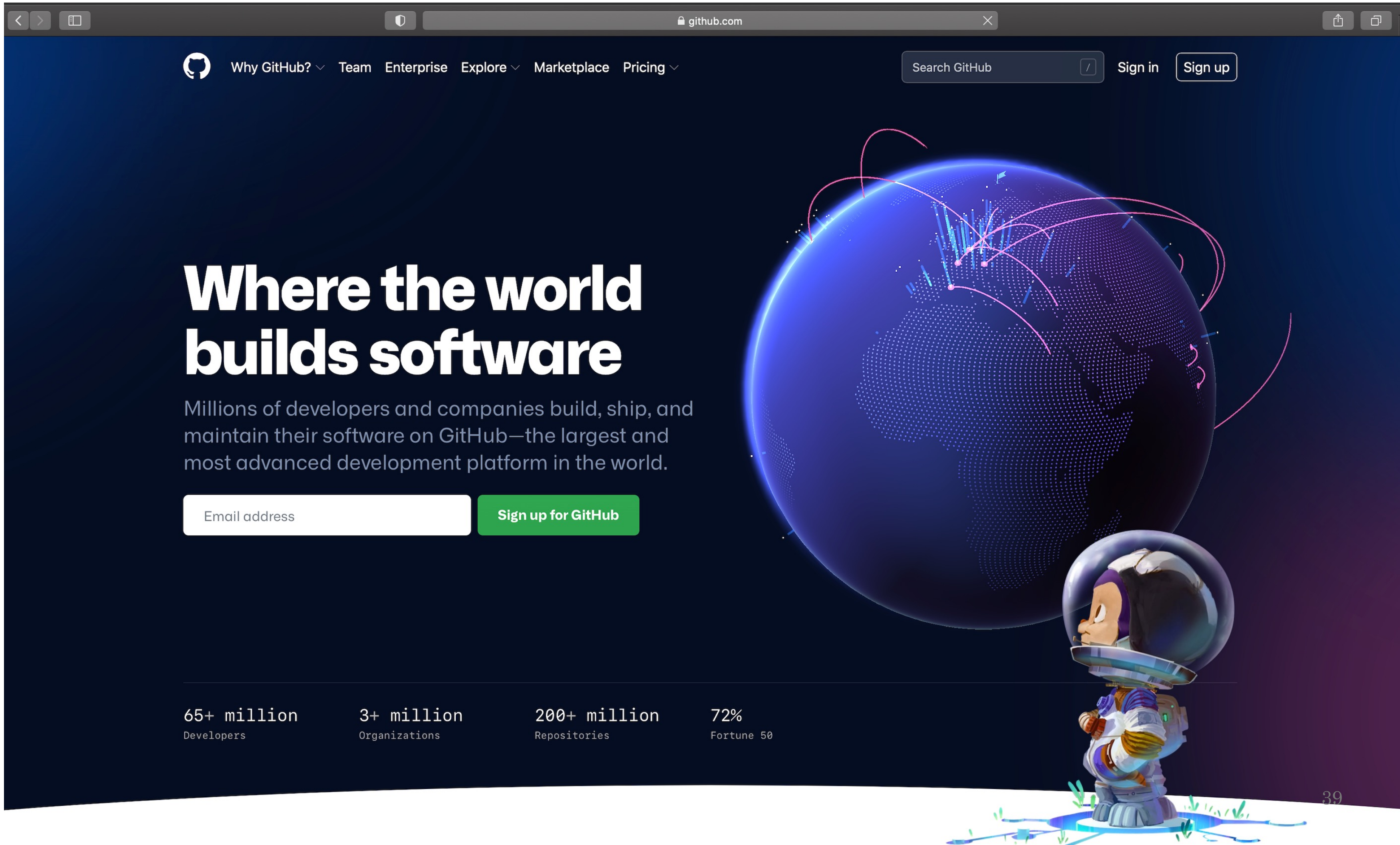
GitHub is a web-based hosting service for git repositories. It offers all of Git's DVCS SCM and has some additional features

GitHub includes collaboration functionality like project management, support ticket management, and bug tracking.

With GitHub, developers can share their repositories, access other developers' repositories, and store remote copies of repositories to serve as backups.

Creating a Personal Account on GitHub

To create your repositories on GitHub or contribute to other open source projects, you will need to create a personal account GitHub



github.com

Why GitHub? Team Enterprise Explore Marketplace Pricing Search GitHub Sign in Sign up

Where the world builds software

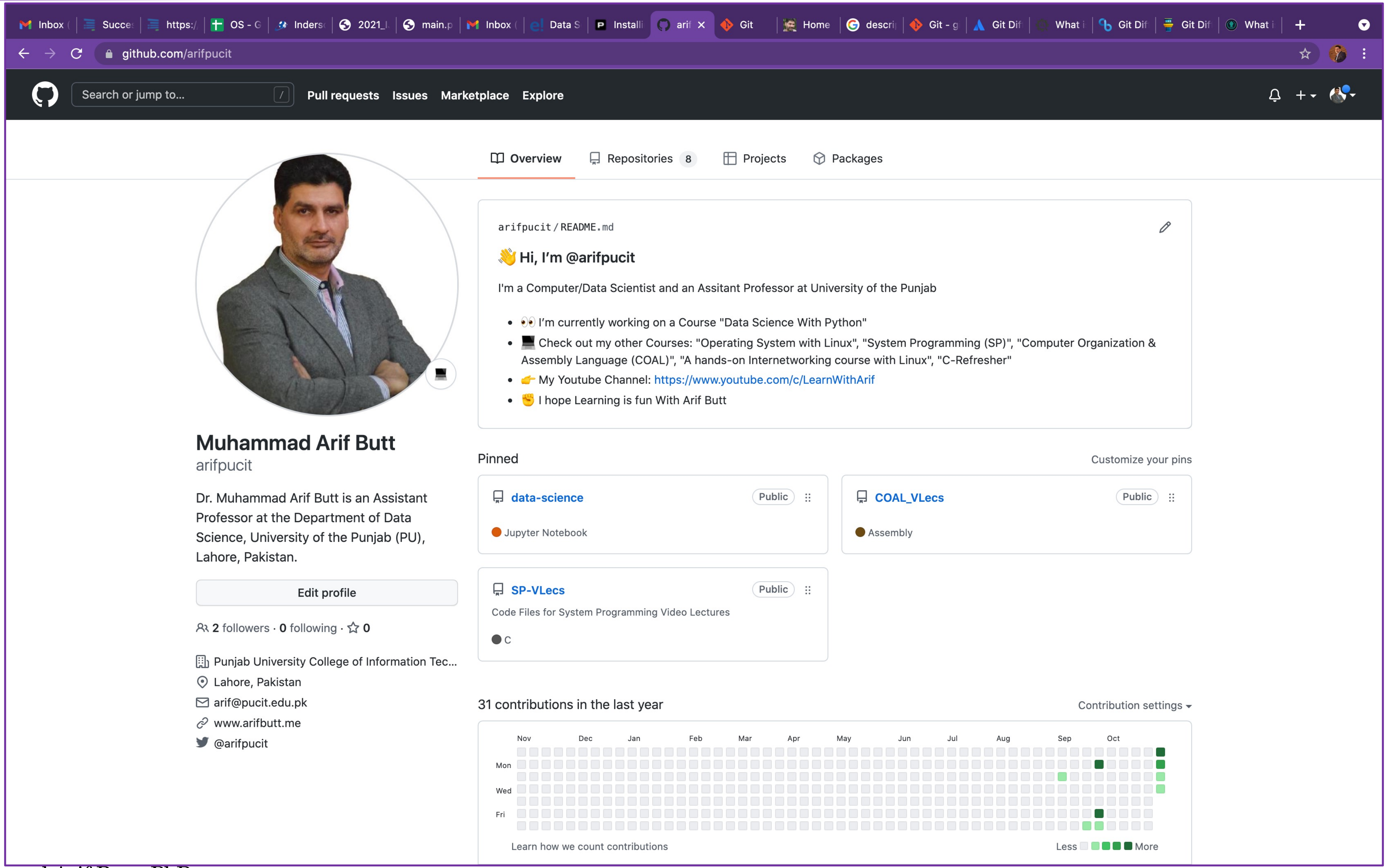
Millions of developers and companies build, ship, and maintain their software on GitHub—the largest and most advanced development platform in the world.

Email address | Sign up for GitHub

65+ million Developers
3+ million Organizations
200+ million Repositories
72% Fortune 50

Instructor: Muhammad Arif Butt, PhD

Login into your GitHub Account



github.com/arifpucit

arifpucit / README.md

👋 Hi, I'm @arifpucit

I'm a Computer/Data Scientist and an Assitant Professor at University of the Punjab

- 👀 I'm currently working on a Course "Data Science With Python"
- 💻 Check out my other Courses: "Operating System with Linux", "System Programming (SP)", "Computer Organization & Assembly Language (COAL)", "A hands-on Internetworking course with Linux", "C-Refresher"
- 👉 My Youtube Channel: https://www.youtube.com/c/LearnWithArif
- ✊ I hope Learning is fun With Arif Butt

Muhammad Arif Butt
arifpucit

Dr. Muhammad Arif Butt is an Assistant Professor at the Department of Data Science, University of the Punjab (PU), Lahore, Pakistan.

Edit profile

2 followers · 0 following · ☆ 0

Punjab University College of Information Tec...
Lahore, Pakistan
arif@pucit.edu.pk
www.arifbutt.me
@arifpucit

Pinned — Customize your pins

Repository	Visibility	Description	Language
data-science	Public		Jupyter Notebook
COAL_VLecs	Public		Assembly
SP-VLecs	Public	Code Files for System Programming Video Lectures	C

31 contributions in the last year — Contribution settings

Learn how we count contributions — Less ⬜🟩🟩🟩 More

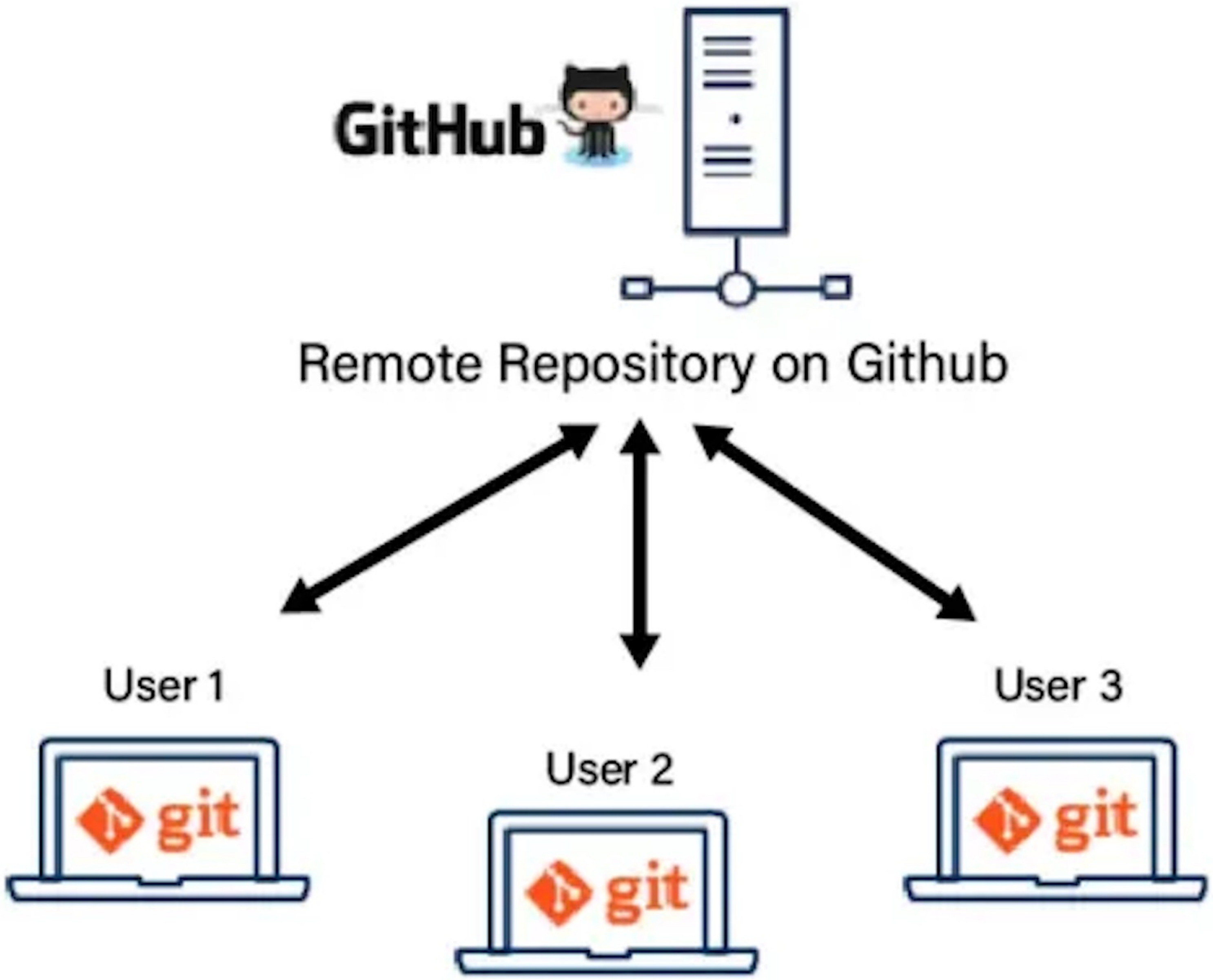
Instructor: Muhammad Arif Butt, PhD

Hosting Services for git Repositories

Since there are different users, who will be accessing the remote repo on GitHub, therefore, there has to be a way to authenticate these users. GitHub provides following ways:

- Password based authentication
- Personal Access Tokens (PATs)
- SSH Keys

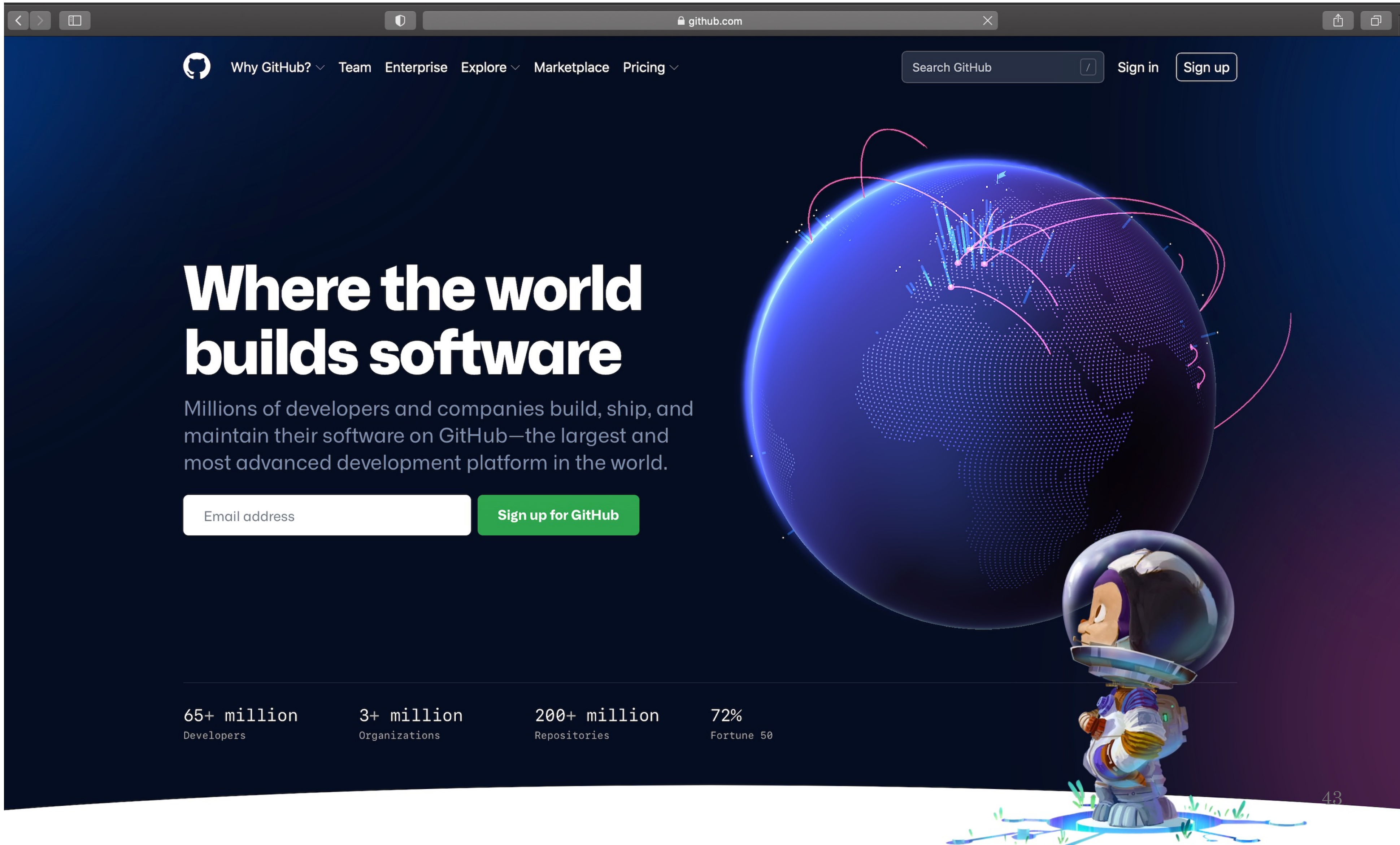
More on this in upcoming slides ☺



Creating a Remote Repo on `GitHub`

Creating a Personal Account on GitHub

To create your repositories on GitHub or contribute to other open source projects, you will need to create a personal account GitHub



Where the world builds software

Millions of developers and companies build, ship, and maintain their software on GitHub—the largest and most advanced development platform in the world.

65+ million Developers

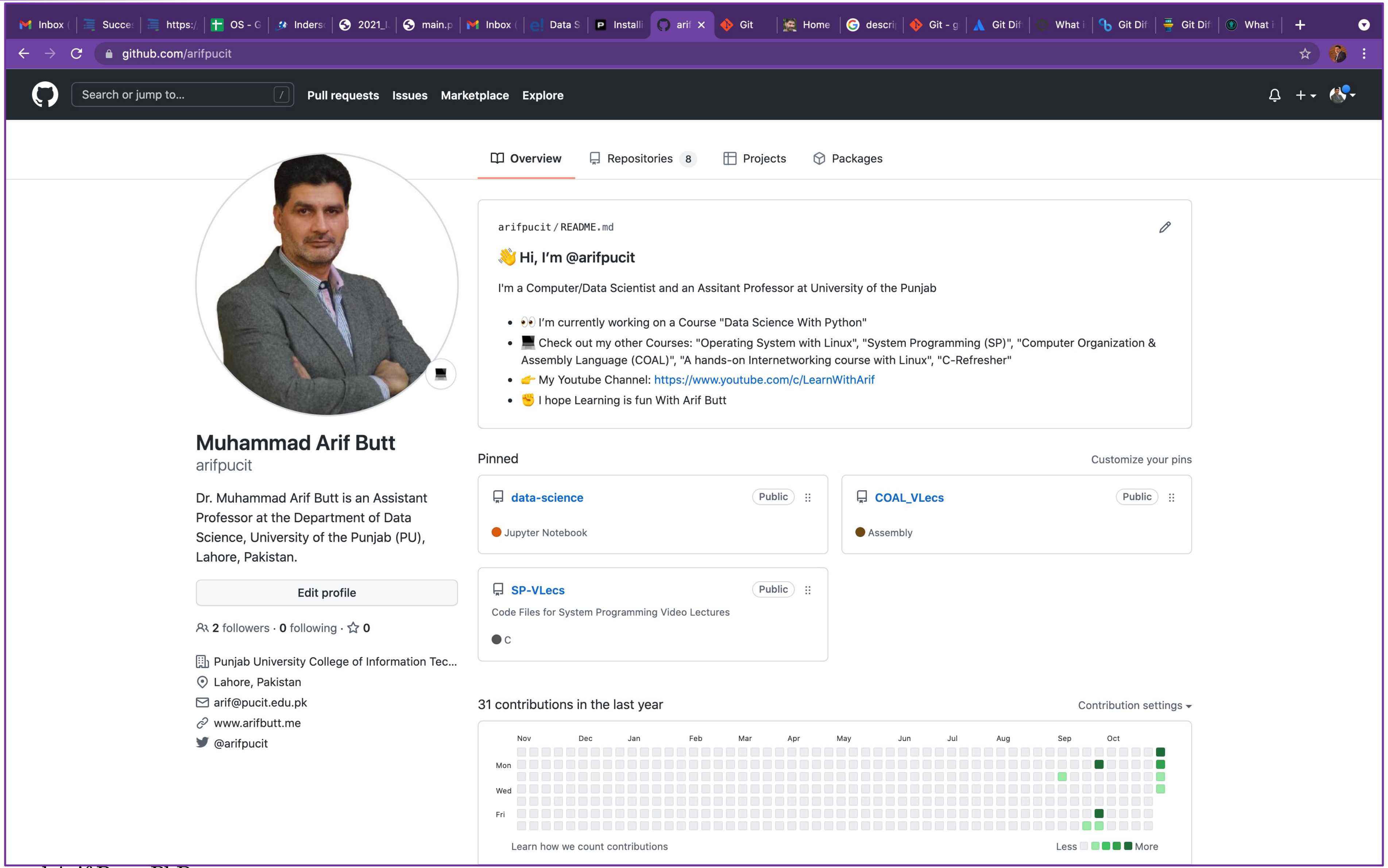
3+ million Organizations

200+ million Repositories

72% Fortune 50

Instructor: Muhammad Arif Butt, PhD

Login into your GitHub Account



github.com/arifpucit

arifpucit / README.md

👋 Hi, I'm @arifpucit

I'm a Computer/Data Scientist and an Assitant Professor at University of the Punjab

- 👀 I'm currently working on a Course "Data Science With Python"
- 💻 Check out my other Courses: "Operating System with Linux", "System Programming (SP)", "Computer Organization & Assembly Language (COAL)", "A hands-on Internetworking course with Linux", "C-Refresher"
- 👉 My Youtube Channel: https://www.youtube.com/c/LearnWithArif
- ✊ I hope Learning is fun With Arif Butt

Muhammad Arif Butt
arifpucit

Dr. Muhammad Arif Butt is an Assistant Professor at the Department of Data Science, University of the Punjab (PU), Lahore, Pakistan.

2 followers · 0 following · ☆ 0

Punjab University College of Information Tec...
Lahore, Pakistan
arif@pucit.edu.pk
www.arifbutt.me
@arifpucit

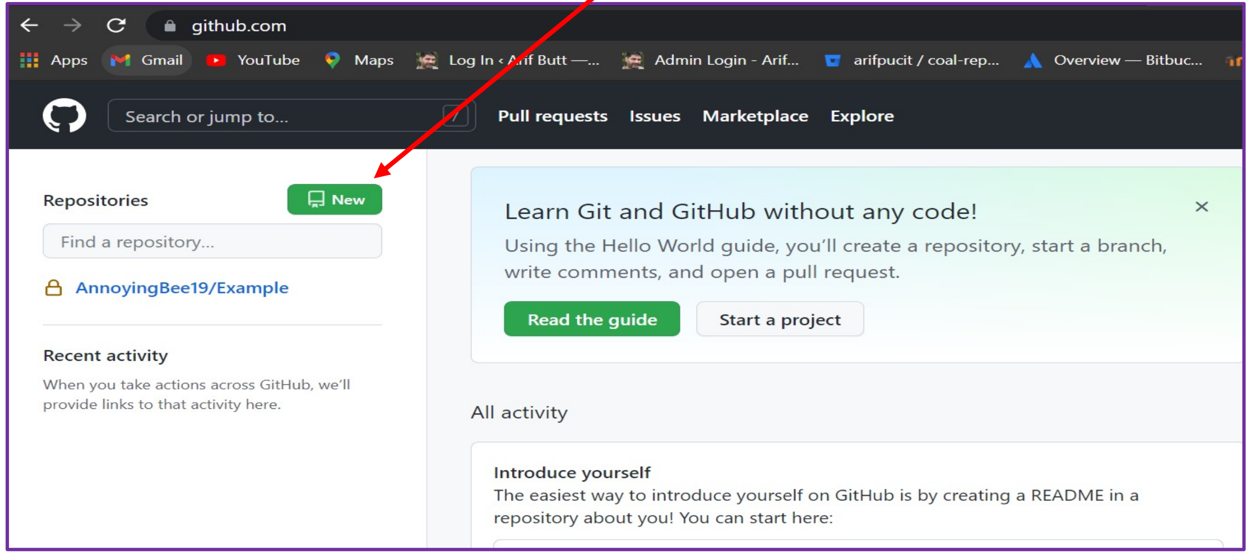
Pinned

Repository	Visibility	Description	Language
data-science	Public		Jupyter Notebook
COAL_VLecs	Public		Assembly
SP-VLecs	Public	Code Files for System Programming Video Lectures	C

31 contributions in the last year

Creating a Remote Repository on GitHub

Once you are logged in and are on the homepage, you will notice a button, that will let you to create your own Repository



Recent activity

When you take actions across GitHub, we'll provide links to that activity here.

Learn Git and GitHub without any code!

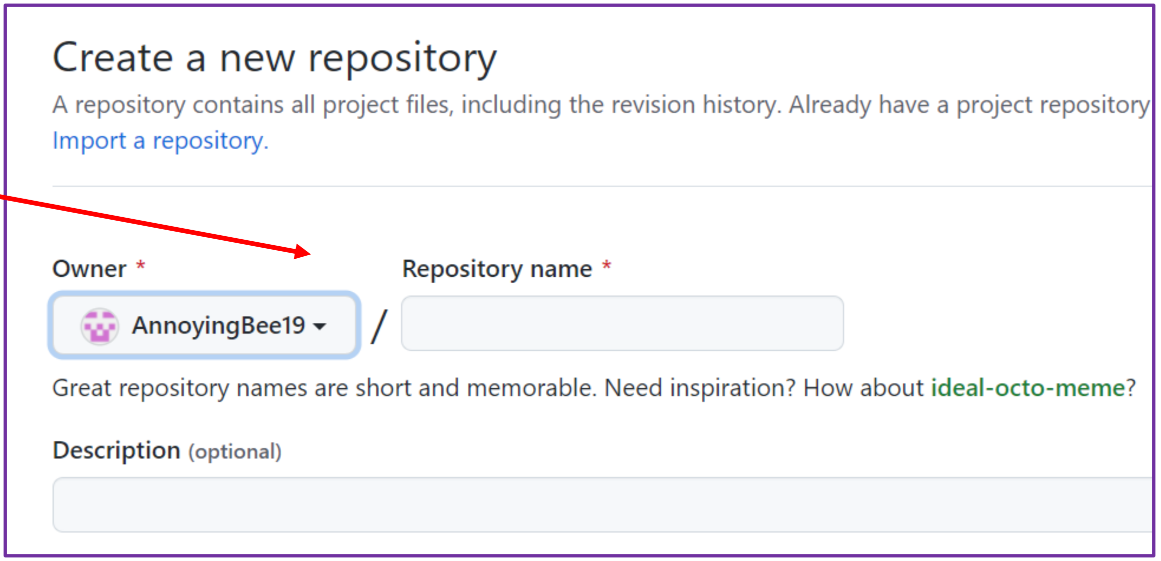
Using the Hello World guide, you'll create a repository, start a branch, write comments, and open a pull request.

All activity

Introduce yourself

The easiest way to introduce yourself on GitHub is by creating a README in a repository about you! You can start here:

Once you click on the 'New' button, GitHub will redirect you to a different page where you will have to provide a name for the repository. Additionally, you can add a description of your repository.



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository

Import a repository.

Owner * AnnoyingBee19 / Repository name *

Great repository names are short and memorable. Need inspiration? How about ideal-octo-meme?

Description (optional)

Public and Private Repositories

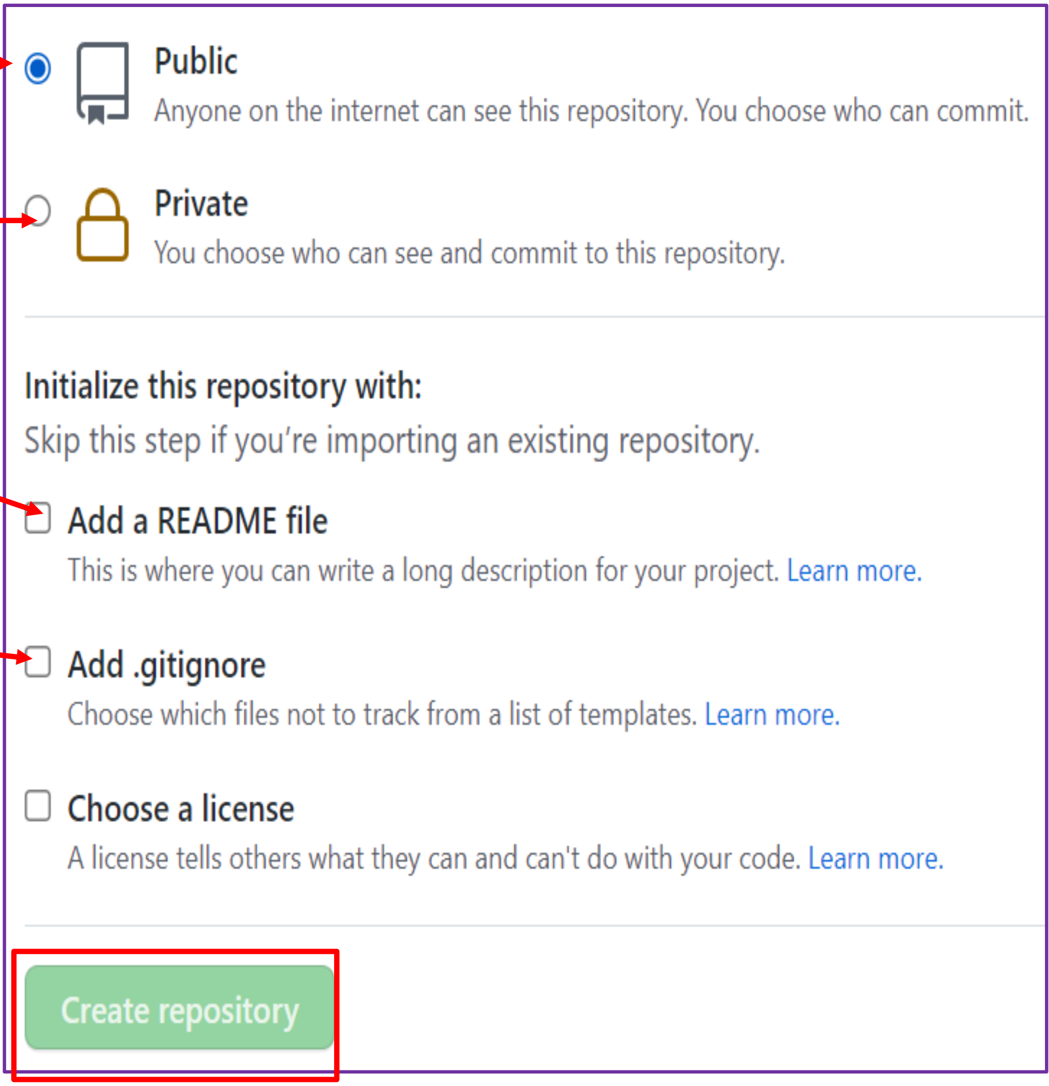
Besides providing a name/description, you need to choose whether you want your repository to be public or private.

Public repository is accessible to anyone. Anyone is able to see the codebase and clone this repository to their local machine for use.

Private repository, on the other hand, is only visible to people who you have chosen. No other person is able to view it.

Another decision you will have to make while creating a new repository is whether or not you'll create a *README* file.

Finally, you will be able to choose whether or not you want a *`.gitignore`* file. The purpose of the `.gitignore` is to filter out files and subdirectories in your repository that you do not want `git` to keep track of.



[✓ Public]
Anyone on the internet can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

Add a README file
This is where you can write a long description for your project. Learn more.

Add .gitignore
Choose which files not to track from a list of templates. Learn more.

Choose a license
A license tells others what they can and can't do with your code. Learn more.

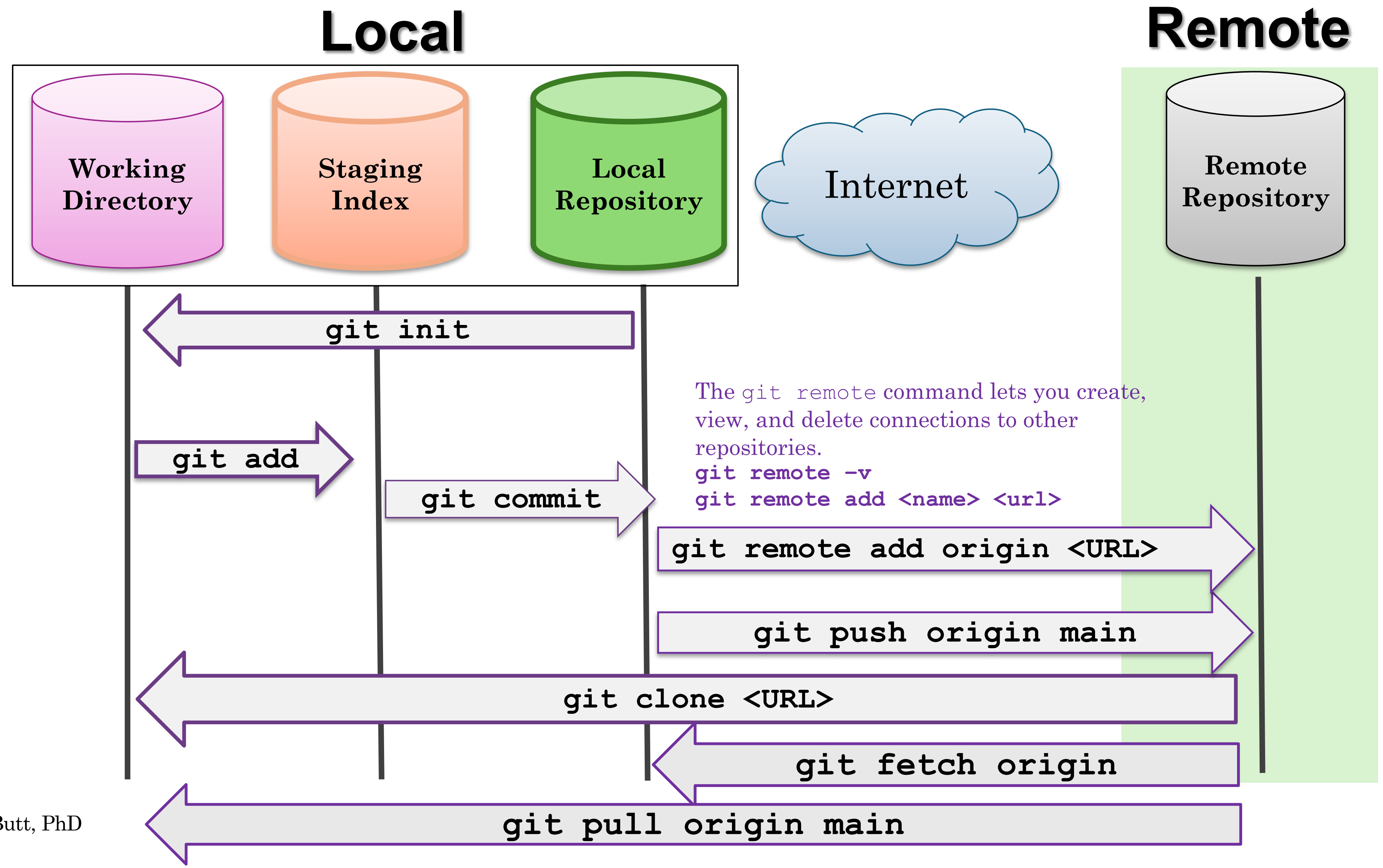
Basic Workflow of GitHub

https://www.atlassian.com/git/tutorials/syncing/git-push

https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone

https://www.atlassian.com/git/tutorials/syncing/git-pull
https://www.atlassian.com/git/tutorials/syncing/git-fetch

Workflow of Working with GitHub



The `git remote` command lets you create, view, and delete connections to other repositories.

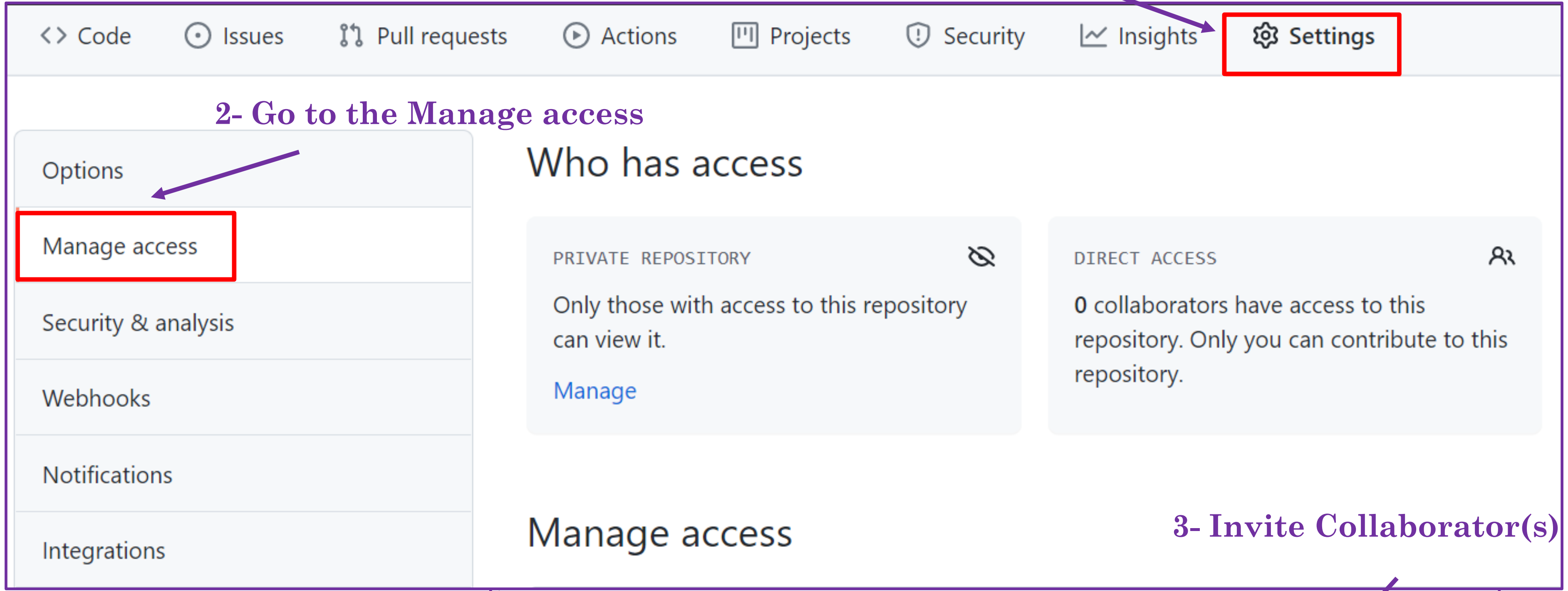
```
git remote -v
git remote add <name> <url>
```

Invite Collaborators

You can decide and manage, who can access your private repository and make collaboration.

1. After creating a private repo, click the settings tab

2- Go to the Manage access



Who has access

PRIVATE REPOSITORY

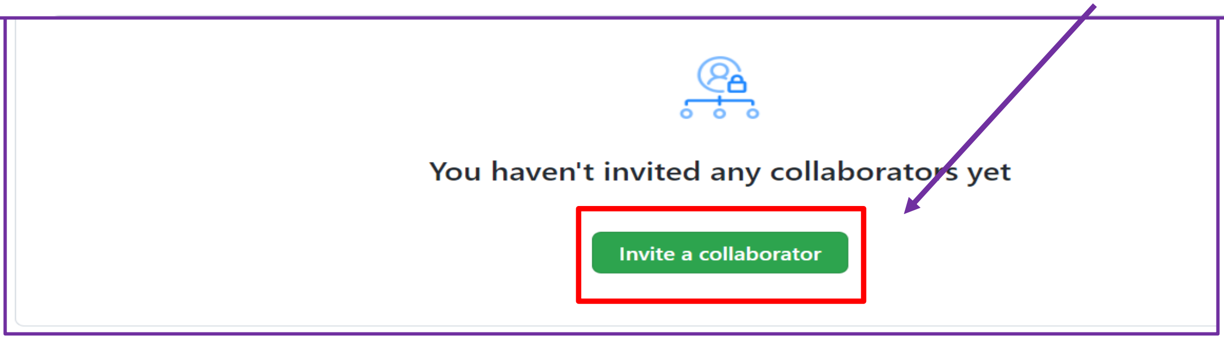
Only those with access to this repository can view it.

DIRECT ACCESS

0 collaborators have access to this repository. Only you can contribute to this repository.

Manage access

3- Invite Collaborator(s) via email or username



You haven't invited any collaborators yet

The collaborator will get an email, and he can accept it. After this the collaborator has push access to your repo.

What a Collaborator can do?

Action	Public Repo (Non-collaborator)	Public Repo (Collaborator)	Private Repo (Collaborator)	
View repository	Yes	Yes	Yes	Browse repo contents
Clone repository	Yes	Yes	Yes	Download local copy
Fork repository	Yes	Yes	Yes (if owner permits)	Make personal copy
Create issues	Yes	Yes	Yes	Report or suggest changes
Submit pull requests	Yes (via fork)	Yes	Yes	Propose code updates
Push directly	❌ No	Yes (Write+)	Yes (Write access only)	Upload commits directly
Merge PRs	❌ No	Yes (Write+)	Yes (Write+)	Integrate pull requests
Manage settings	❌ No	Yes (Maintain+)	Yes (Maintain+)	Change repo settings

Note: A non-collaborator cannot do anything on a private repo unless invited.

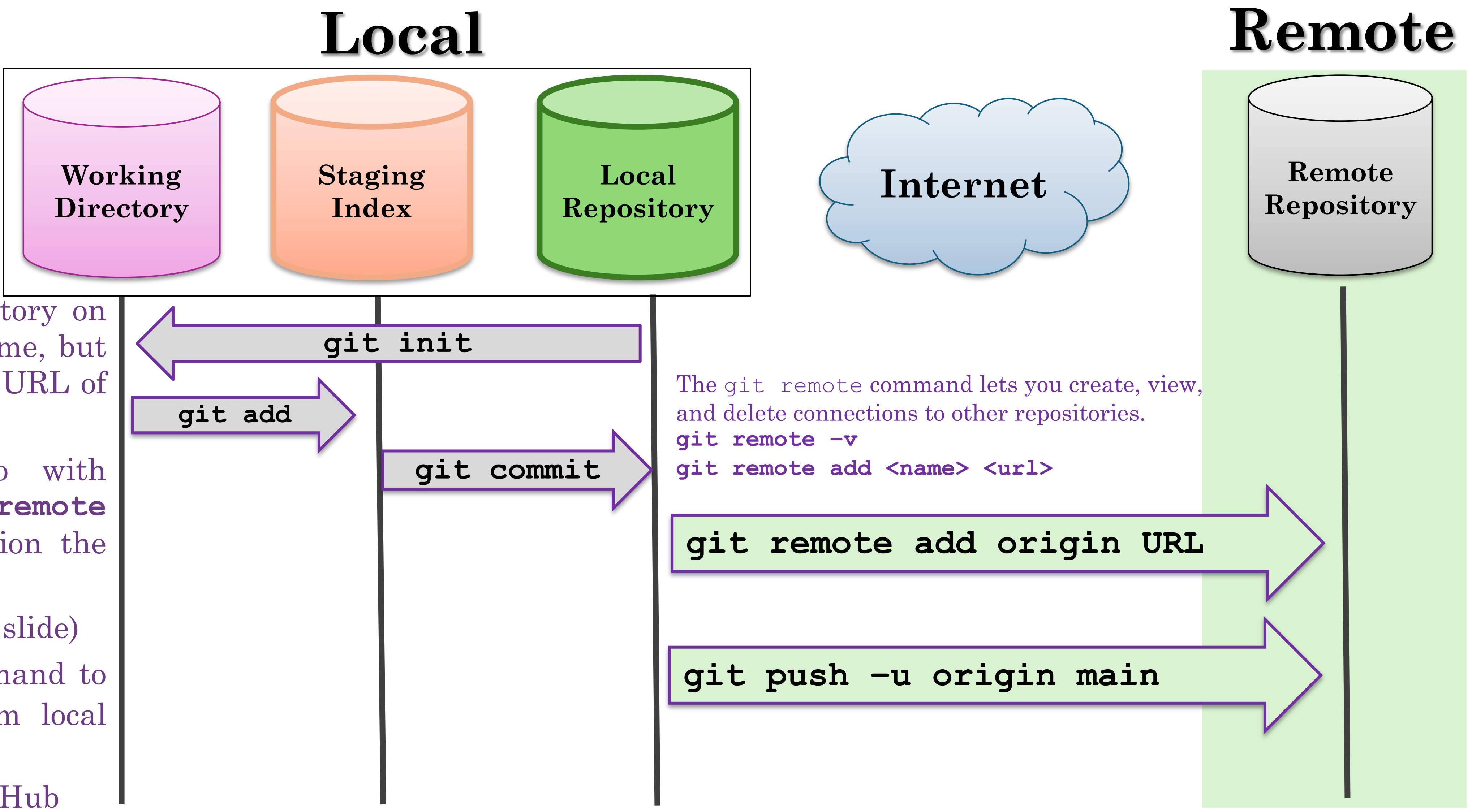
More on this in upcoming slides ☺

Push your Local Repo on GitHub

Pushing a Local Repo to GitHub

Suppose you have created a project using `git` and all the files are there on your local machine. In order to share your project repo with your colleagues you need to follow following steps:

1. Create a new repository on GitHub, give it a name, but keep it empty. Copy URL of remote repo
2. Connect local repo with GitHub using **`git remote`** command and mention the GitHub username
3. Authentication (next slide)
4. Use `git push` command to push your code from local repo to remote repo
5. Verify by visiting GitHub



The `git remote` command lets you create, view, and delete connections to other repositories.

```
git remote -v
git remote add <name> <url>
```

GitHub Authentication and Integrity Methods

- For logging into your GitHub account using a browser, you just need to use your GitHub Password with may be 2FA or Passkeys.
- If you want to push/clone repos via HTTPS, you need to generate Personal Access Tokens (PATs) from the Developer Settings of your profile page of GitHub.
- I personally, prefer using SSH keys, that provide a secure, password-less authentication method to avoid re-entrying the credentials with every push/pull:
  - ➢ You need to generate a private-public key pair locally and save them inside ~/.ssh/ directory. Copy the public key and paste it inside the SSH & GPG keys section in your profile settings.

```
$ ssh-keygen -t ed25519 -C “arifpucit@gmail.com”
```

Generate ~/.ssh/id_ed25519 and ~/.ssh/id_ed25519.pub

```
$ eval “$(ssh-agent -s)”
$ ssh-add ~/.ssh/id_ed25519
```

The ssh agent will keep the private key loaded, so you don’t have to type passphrase every time

```
$ cat ~/.ssh/id_ed25519.pub
```

Copy the public key and paste it to your GitHub account Settings → SSH and GPG Keys → New SSH key

```
$ ssh -T git@github.com
```

This will verify that the above process is a success

```
$git remote add origin git@github.com:arifpucit/<repo.git>
$ git push -u origin main
```

Adding a Feature in your Local Repo and push it to GitHub

Step 1: Adding a new Feature

Scenario: Suppose I have a local repo, which has been pushed on my own GitHub repo and I want to

- Add a new feature.
- Tag a version.
- Push the tag to GitHub.
- Publish a release on GitHub.

Adding a new feature in your local repo:

- Create a new feature branch.
- Make your changes and commit.
- Merge feature branch into main branch
- Push changes to your own repo at GitHub.

Step 2: Create a Tag and push it to GitHub

What are Tags in Git? A tag is like a bookmark in Git history. It points to a specific commit. Commonly used to mark versions (e.g., v1.0, v2.0). There are two types:

- **Lightweight Tag** → just a pointer to a commit.
- **Annotated Tag** → includes metadata (author, date, message).

- Create a lightweight Tag or annotated tag (as per your need):

```
$ git tag v1.0.0
$ git tag -a v1.0.0 -m "Release version 1.0.0"
```

- Push tag to GitHub:

```
$ git push origin v1.0.0
```

- To delete a tag:

```
$ git delete -d <tag-name>
```

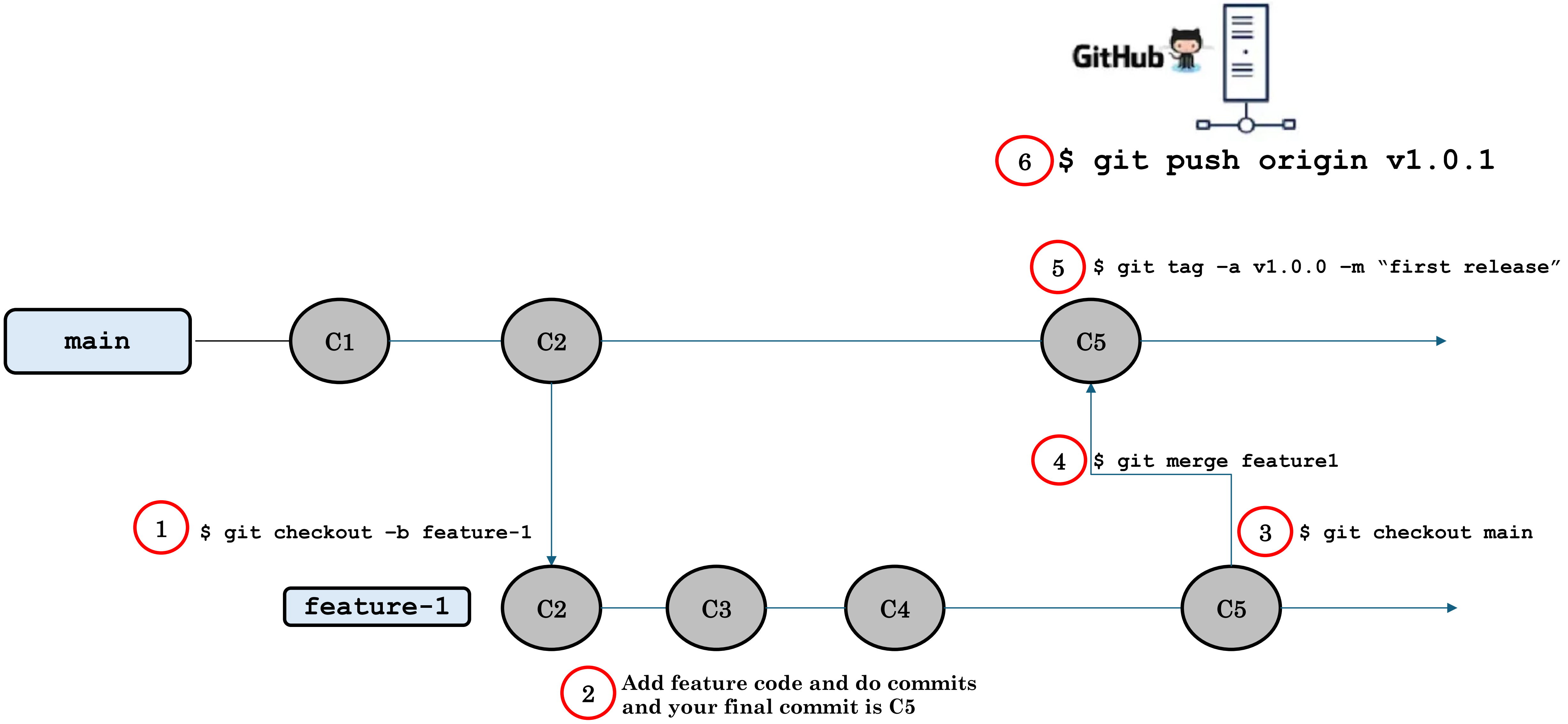
- To see all tags:

```
$ git tag
```

- To view details about a specific tag:

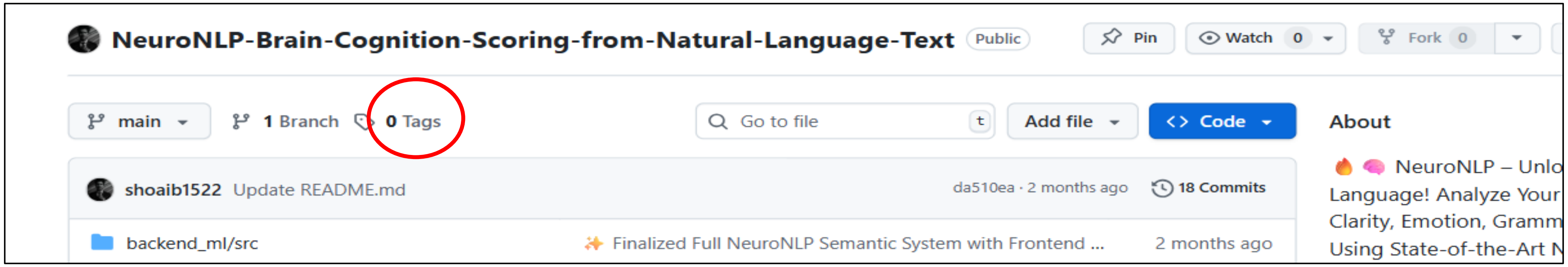
```
$ git show <tag-name>
```

Step 2: Create a Tag and push it to GitHub (...)


GitHub
6
$ git push origin v1.0.1
5
$ git tag -a v1.0.0 -m “first release”
main
C1
C2
C5
4
$ git merge feature1
1
$ git checkout -b feature-1
3
$ git checkout main
feature-1
C2
C3
C4
C5
2
Add feature code and do commits and your final commit is C5

Step 3: Create a new Release on GitHub

- Now inside a browser, go to your GitHub repo, where you will see the tags as shown in the following screenshot.
- Click the Tags tab, which will display the tags available for this repository.



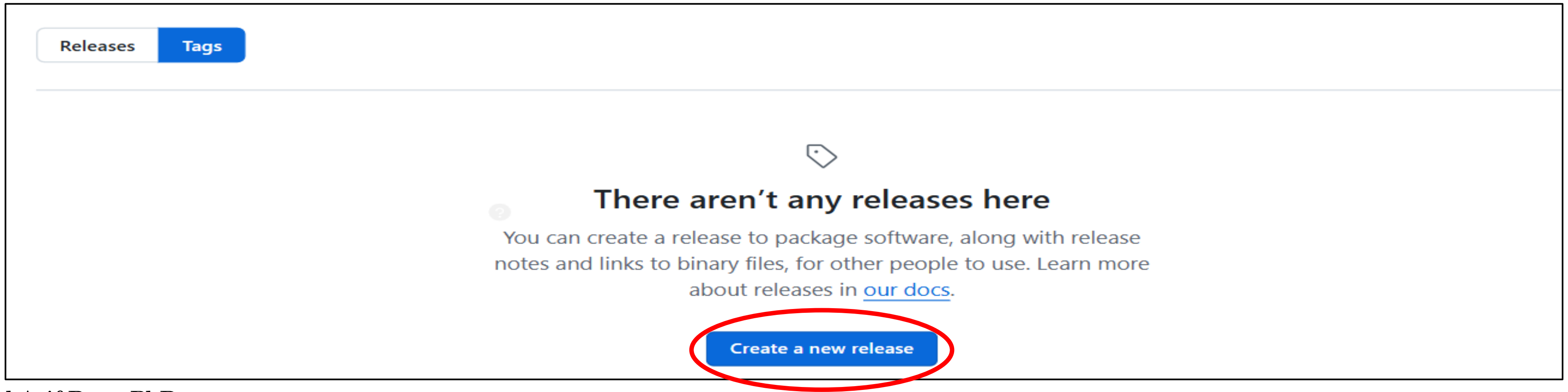
NeuroNLP-Brain-Cognition-Scoring-from-Natural-Language-Text Public

shoaib1522 Update README.md da510ea · 2 months ago 18 Commits

backend_ml/src ✨ Finalized Full NeuroNLP Semantic System with Frontend ... 2 months ago

🔥 🧠 NeuroNLP – Unlo
Language! Analyze Your
Clarity, Emotion, Gramm
Using State-of-the-Art N

- On the Tags page, click Create a new release.
- This option appears if no release exists yet, as shown in the screenshot below.

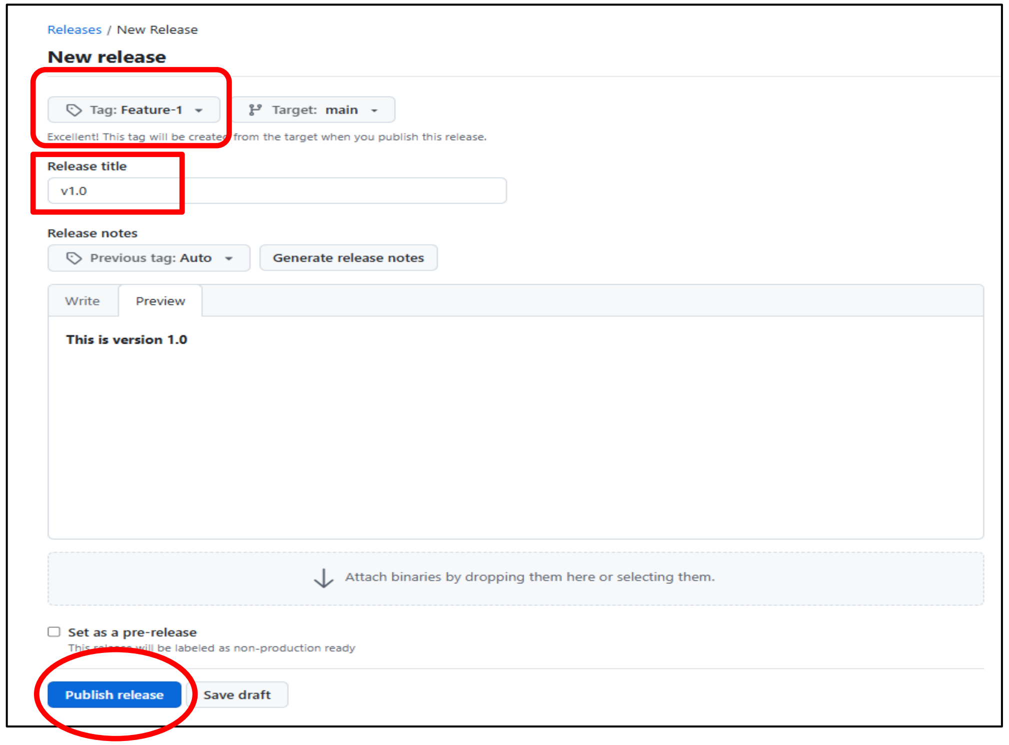


There aren't any releases here

You can create a release to package software, along with release notes and links to binary files, for other people to use. Learn more about releases in our docs.

Step 4: Publish the Release on GitHub

- The opposite screenshot displays the “Create a new release” form.
- Select the tag, you just pushed (`v1.0.0`).
- Add release notes/description.
- Click Publish release.
- You will now see your release in the Releases section, which is linked to the tag you created.



Releases / New Release

New release

Tag: Feature-1 · Target: main

Excellent! This tag will be created from the target when you publish this release.

Release title: v1.0

Release notes: Previous tag: Auto · Generate release notes

Write · Preview

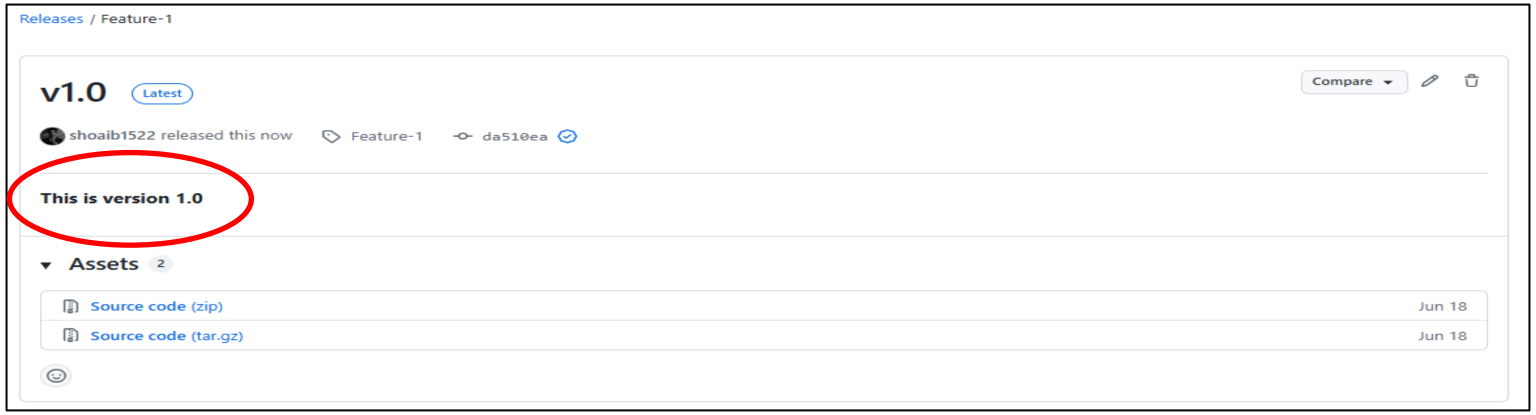
This is version 1.0

Attach binaries by dropping them here or selecting them.

☐ Set as a pre-release
This release will be labeled as non-production ready

Publish release · Save draft

- The opposite screenshot displays your release in the Releases section, which is linked to the tag you created.



Releases / Feature-1

v1.0 Latest · Compare

shoaib1522 released this now · Feature-1 · da510ea

This is version 1.0

Assets 2

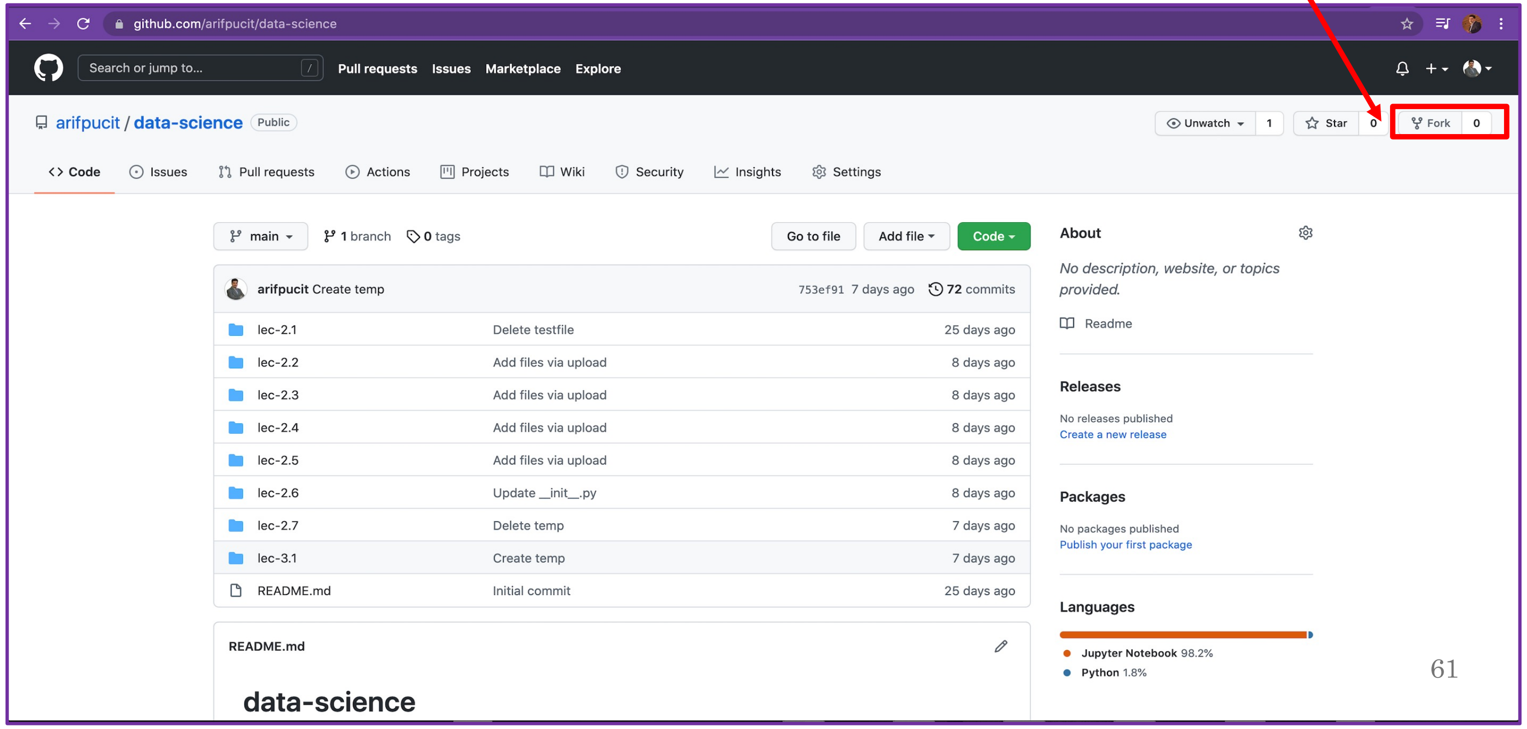
Source code (zip)	Jun 18
Source code (tar.gz)	Jun 18

Fork, Clone and Contribute to a Friends Repository using Pull Request

Fork a Repository from GitHub

- Forking means creating a copy of complete repo from some one else's GitHub account on your GitHub account. You can do this to collaborate on a open source project, or use the existing state of the project as a starting point for your own project
  - ✓ On GitHub navigate to someone's repository that you want to fork, and click the Fork button, then check the repository availability on your GitHub account.
  - ✓ Clone this repo on your local machine, make a new branch, fix a bug, add/enhance a functionality, and then push it back to your own remote repo
  - ✓ Finally click pull request to open a new pull request to the actual project owner

Click the fork button



github.com/arifpucit/data-science

arifpucit / data-science Public

arifpucit Create temp 753ef91 7 days ago 72 commits

lec-2.1	Delete testfile	25 days ago
lec-2.2	Add files via upload	8 days ago
lec-2.3	Add files via upload	8 days ago
lec-2.4	Add files via upload	8 days ago
lec-2.5	Add files via upload	8 days ago
lec-2.6	Update __init__.py	8 days ago
lec-2.7	Delete temp	7 days ago
lec-3.1	Create temp	7 days ago
README.md	Initial commit	25 days ago

README.md

data-science

About

No description, website, or topics provided.

Readme

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

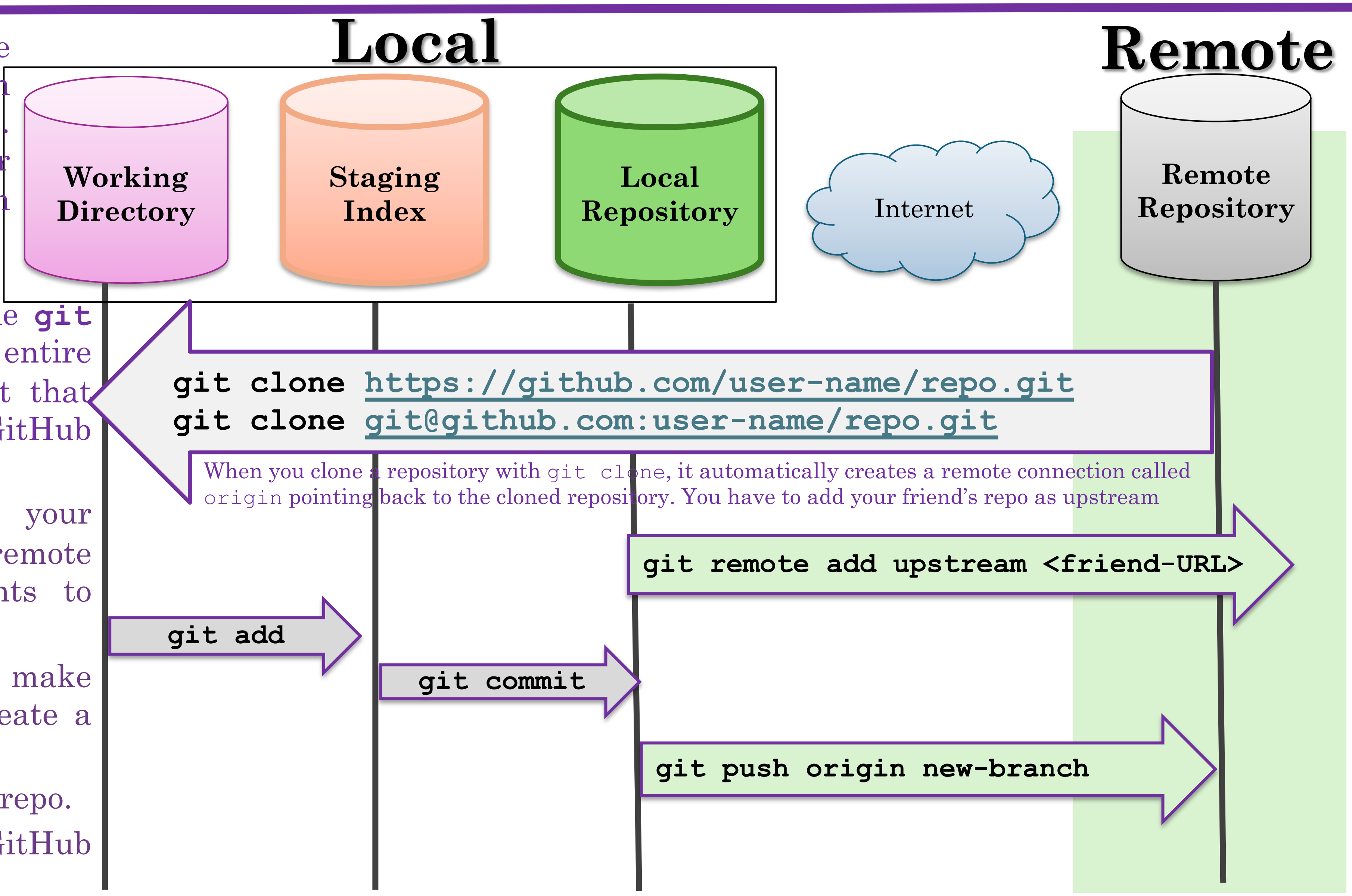
Languages

- Jupyter Notebook 98.2%
- Python 1.8%

Clone Your Remote Repo to Local Repo

From your GitHub web interface **fork** your friend's public repo on which you want to collaborate. Copy the URL (https or ssh) of your friend's repo as well as your own forked repo (both on GitHub).

1. From your local machine use the **git clone** command to copy the entire codebase of your friend's project that you have forked on your own GitHub account.
2. Use **git remote** mentioning your friends-repo-url to add a remote reference "upstream" that points to friends original repo.
3. Create new feature branch, make changes, commits and finally create a tag on your local repo.
4. Push changes to your own forked repo.
5. Create a **pull request** via your GitHub web interface (next slide)



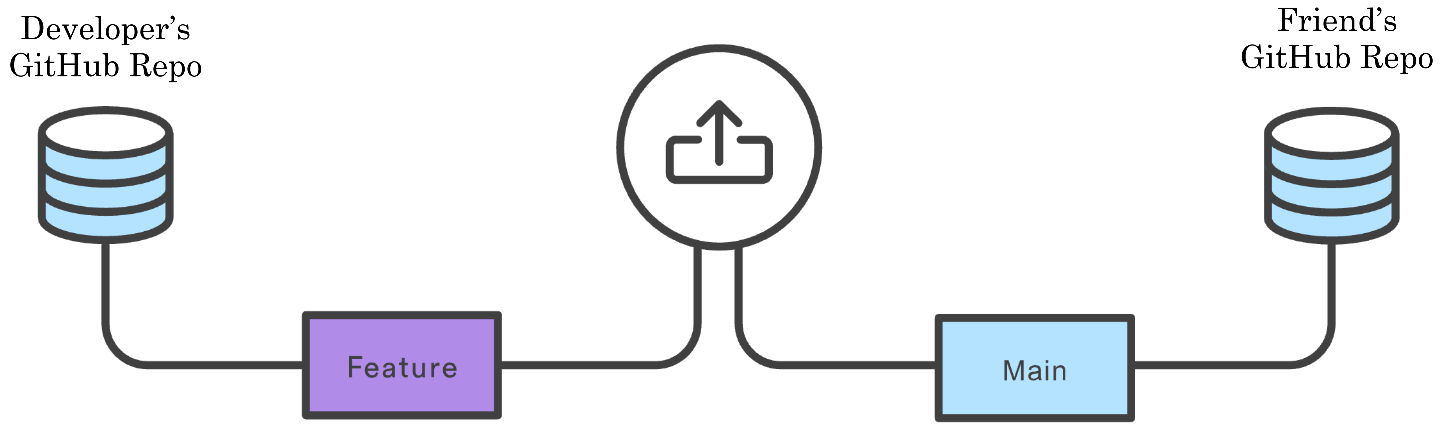
```
git clone https://github.com/user-name/repo.git
git clone git@github.com:user-name/repo.git
```

When you clone a repository with `git clone`, it automatically creates a remote connection called `origin` pointing back to the cloned repository. You have to add your friend's repo as upstream

```
git remote add upstream <friend-URL>
git push origin new-branch
```

Forking Workflow with a Pull Request?

- **Pull Request (PR)** is a `GitHub/GitLab/Bitbucket` feature that is a request to merge code from one branch into another. It's not a `git` command but rather a collaboration tool provided by Git hosting platforms. A pull request allows you to:
    - Propose changes from your feature branch to be merged into another branch
    - Enable code review and discussion before merging
    - Run automated tests and checks
    - Document what changes you're making and why
    - Get approval from team members before the merge happens



What happens after a Pull Request?

- After the collaborator has pushed the changes to his own forked repo, he needs to get to his GitHub, where he will find two branches. He will then go to "Pull requests" tab and click "New pull request". Fill out PR details by writing a clear, descriptive title and description. Add labels, assignees and reviewers if needed.
- Wait for a response from your friend, (the maintainer of the original repo), who will receive an email to review the pull request. He will go to his GitHub and review the changes you have made. If he is satisfied with this feature he will approve, otherwise he will send his feedback for improvement.
- Once approved, the PR can be merged by you or by your friend (maintainer).
- The maintainer or you yourself, will got to the GitHub original repo, and click the "Merge Pull Request" button.
- Finally do the Clean up: Deleting the feature branch after merging (on GitHub), switch back to main branch locally, pull the updated main, and delete your local feature branch.

https://www.atlassian.com/git/tutorials/making-a-pull-request

CI/CD Pipeline

https://www.spec-india.com/blog/ci-cd-pipeline

Overview of CI / CD pipeline?

CI/CD stands for **Continuous Integration** and **Continuous Delivery/Deployment**. It is a modern software development practice that automates the process of:

- Integrating code changes.
- Testing them for quality.
- Delivering or deploying them to users.

The **aim** is to release software faster, more reliably, with fewer errors, and with seamless team collaboration.

- **Continuous Integration (CI):** Automatically builds, tests, and validates code changes as soon as they are committed to the repository to catch integration issues early.
- **Continuous Delivery (CD):** Extends CI by automating the delivery process up to a staging or pre-production environment, where code is ready for production deployment but requires manual approval.
- **Continuous Deployment:** Goes a step further than continuous delivery by automatically deploying code changes directly to production without manual intervention after passing all automated tests.

What is CI / CD pipeline?

CI/CD works through an automated pipeline a series of steps code passes through from development to production.

Continuous Integration (CI)

A software development practice where developers merge code changes into a shared repository.

Process:

- Developer commits code.
- The system automatically **builds** the application.
- Automated **tests** run to verify functionality.
- Process ends here with **feedback** (Pass/Fail).

Purpose: Detect integration issues early, improve code quality, & enable faster development cycles.

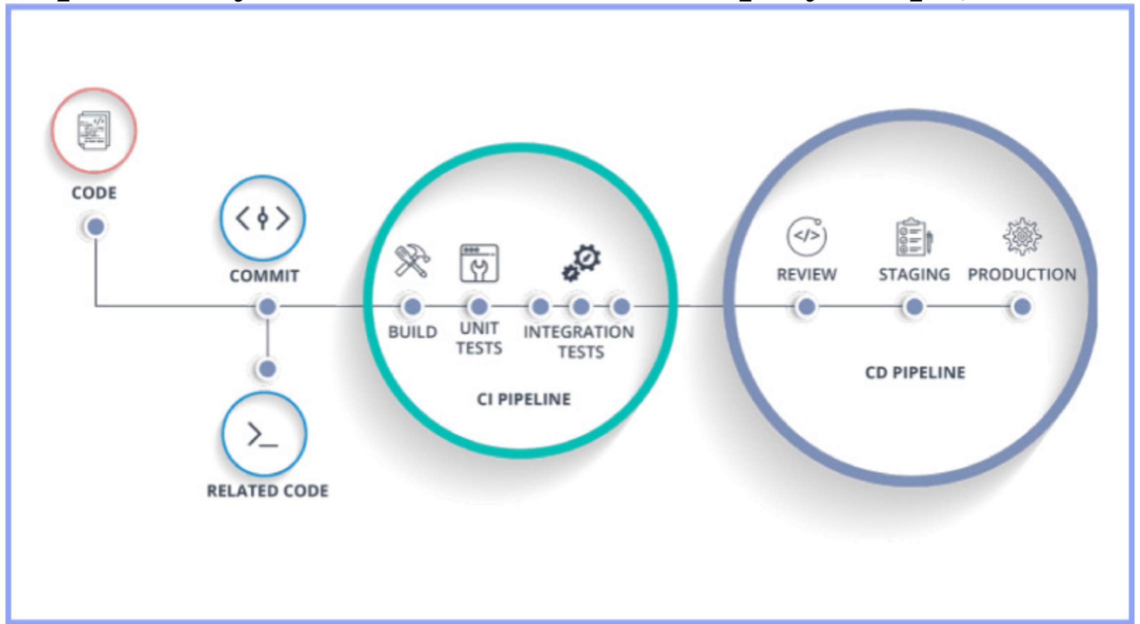
Continuous Deployment (CD)

Extends continuous delivery by **automatically** deploying every change that passes tests directly to production.

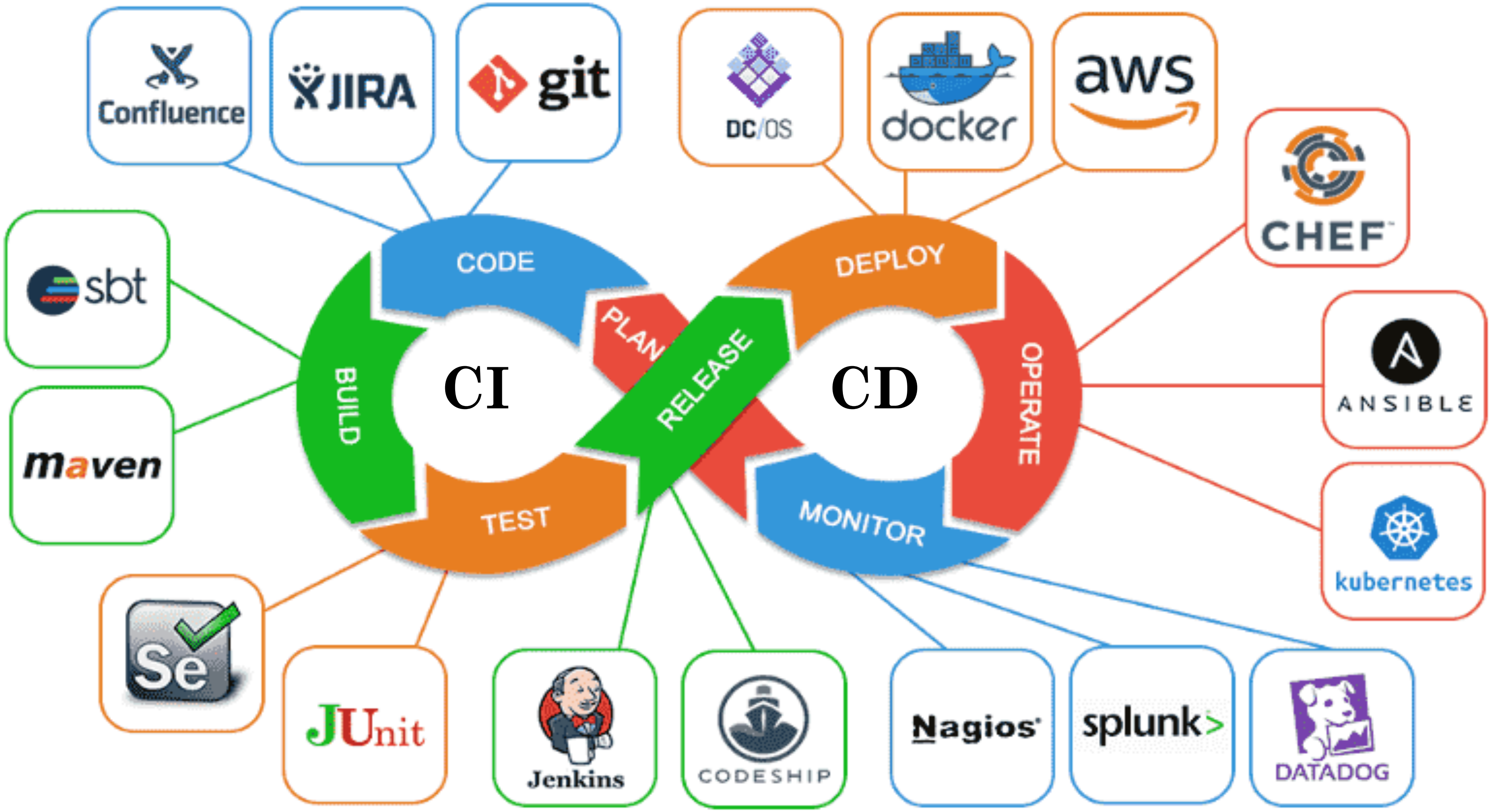
Process:

- Commit code → Build → Automated checks.
- **No human approval needed**.
- Code is deployed straight to production.

Purpose: Speed up releases, eliminate manual deploy steps, & ensure customers get instant updates

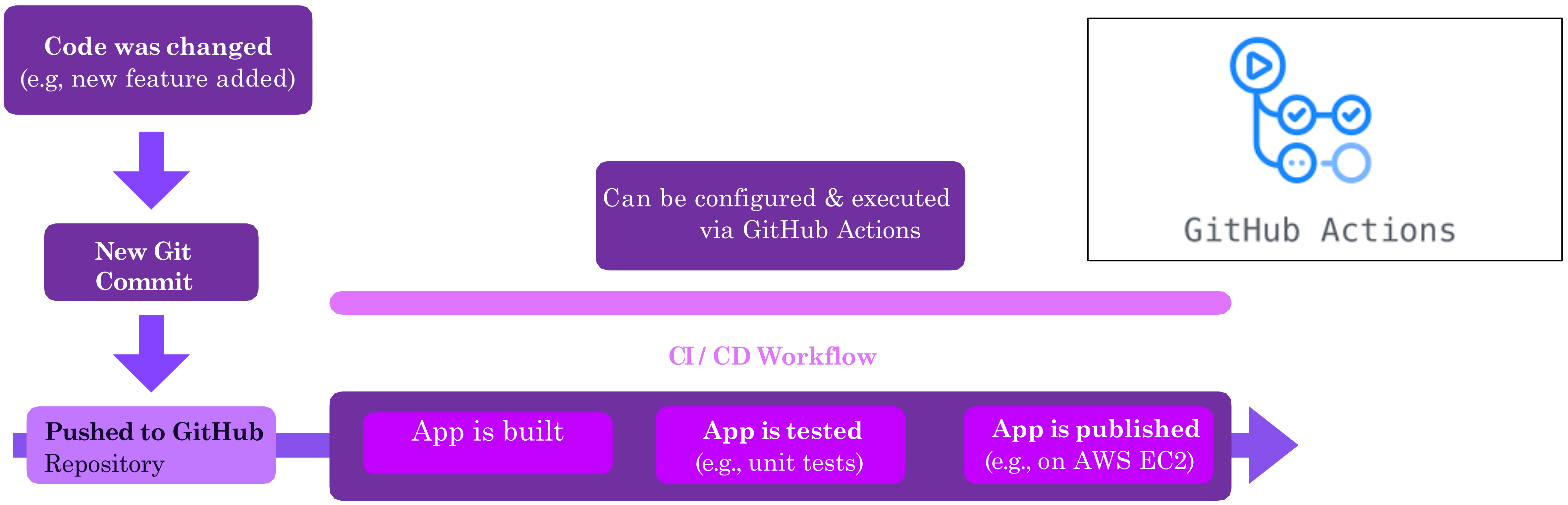


Lifecycle and Tools for CI / CD pipeline



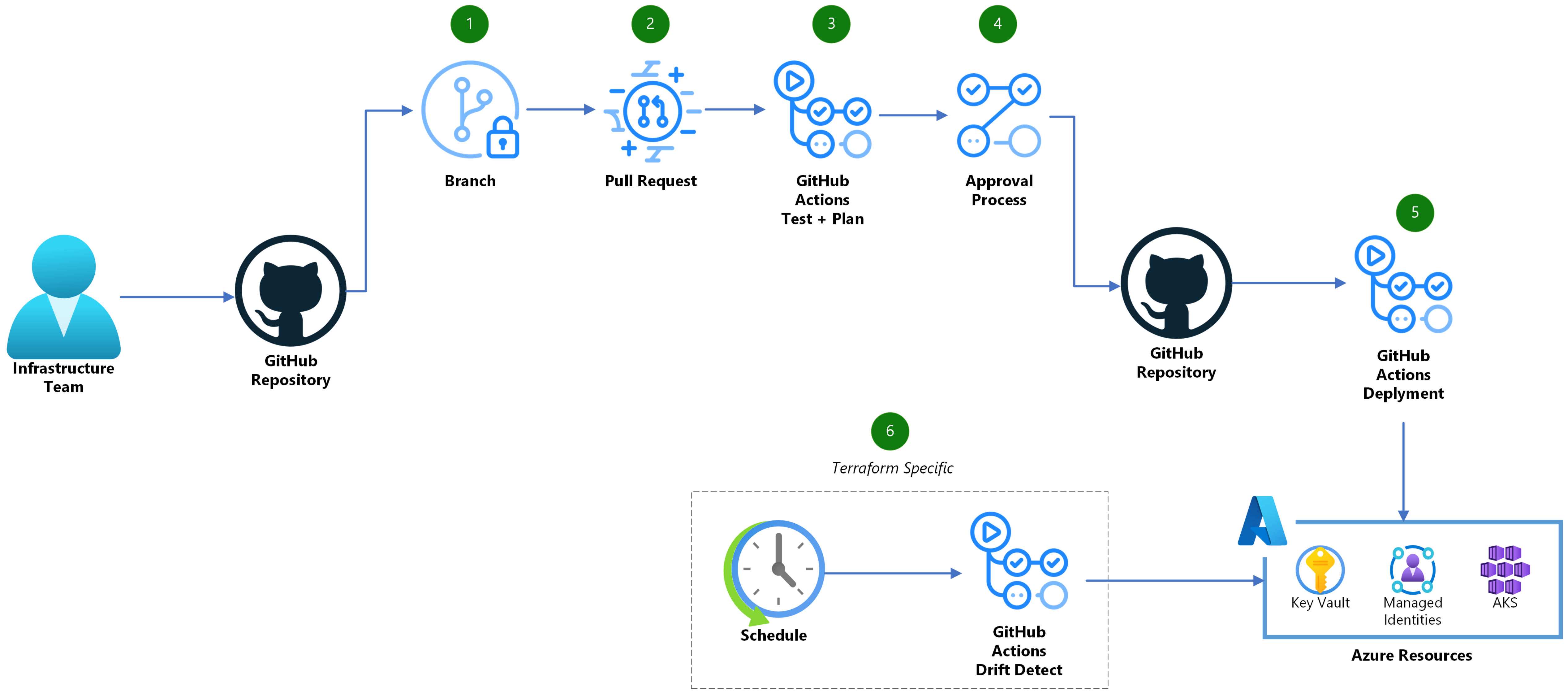
CI / CD Workflow with GitHub Actions

GitHub Actions is a GitHub's built-in CI/CD platform integrated directly within GitHub repositories, enabling the automation of software development workflows. It allows users to define and execute automated processes, called workflows, in response to various events within their repositories like commits, pull requests or releases.



Code was changed
(e.g, new feature added)
New Git Commit
Pushed to GitHub Repository
Can be configured & executed via GitHub Actions
GitHub Actions
CI / CD Workflow
App is built
App is tested (e.g., unit tests)
App is published (e.g., on AWS EC2)

Workflow of GitHub Actions



Instructor: Muhammad Arif Butt, PhD

How GitHub Actions Work?

Following four steps describes at an abstract level as to how GitHub Actions work:

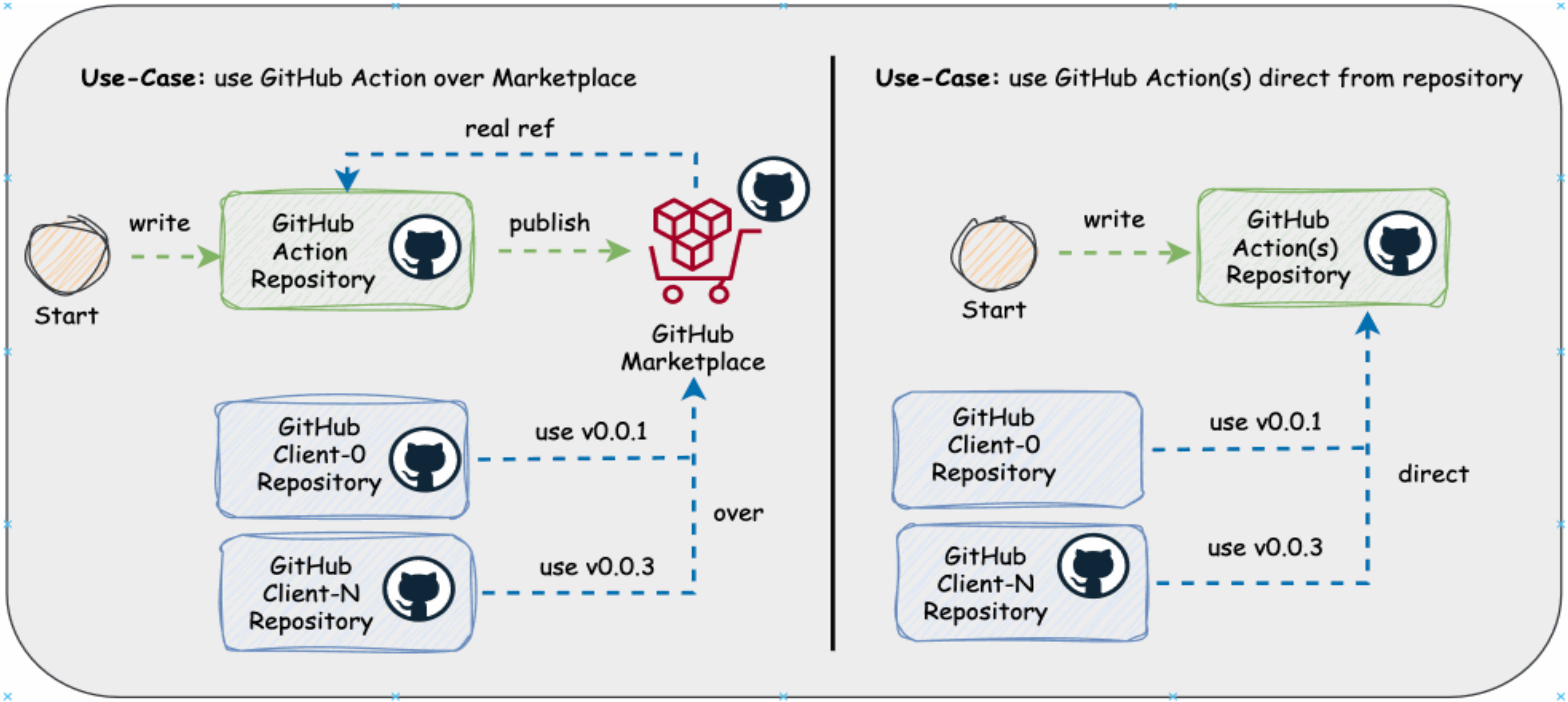
- **Create a YAML workflow file** (`.yml` or `.yaml`) in the `.github/workflows/` directory of your repository.
- **Define trigger events** that initiate GitHub events (such as `on: [push]`, `on: [pull_request]`, `on: [schedule]`)
- **Specify jobs** with steps containing either shell commands (`run:`) or reusable, pre-built automation components from the GitHub Actions marketplace (`uses:`)
- Monitor GitHub's automatic execution of the workflow with real-time logs and status reporting in the Actions tab

```yaml
# .github/workflows/simple-ci.yml
name: Simple CI Pipeline

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: npm install
```

This minimal example of a yml file shows a simple a GitHub Actions workflow having:

- **Two triggers**: `push` and `pull_request` events on the main branch
- **One pre-built action**: `actions/checkout@v4` (most commonly used action for getting repository code)
- **One shell command**: `npm install` for installing dependencies

GitHub Actions in Action



Use-Case: use GitHub Action over Marketplace
real ref
write
GitHub Action Repository
publish
Start
GitHub Marketplace
GitHub Client-0 Repository
use v0.0.1
over
GitHub Client-N Repository
use v0.0.3
Use-Case: use GitHub Action(s) direct from repository
write
GitHub Action(s) Repository
Start
GitHub Client-0 Repository
use v0.0.1
direct
GitHub Client-N Repository
use v0.0.3

To Do

- Install **`git`** on your machine and practice working on a local repository by performing lots of commits, create branches and merge them.
- Create your GitHub account using your RollNo and official email ID
- Create a private and a public repository and share it with your friends and TAs.
- Try to **`fork`** your friend's repo, add a feature or fix a bug or just improve documentation and submit a **`pull request`** to the repository owner. This will equip to do the lab tasks as well as the upcoming Programming assignments.

- Watch DS video on git: https://www.youtube.com/watch?v=WTjxs3em8SM&list=PL7B2bn3G_wfAs3C49i12i_rblzvuU1dFN&index=3&t=3368s

Coming to office hours does NOT mean that you are academically weak!