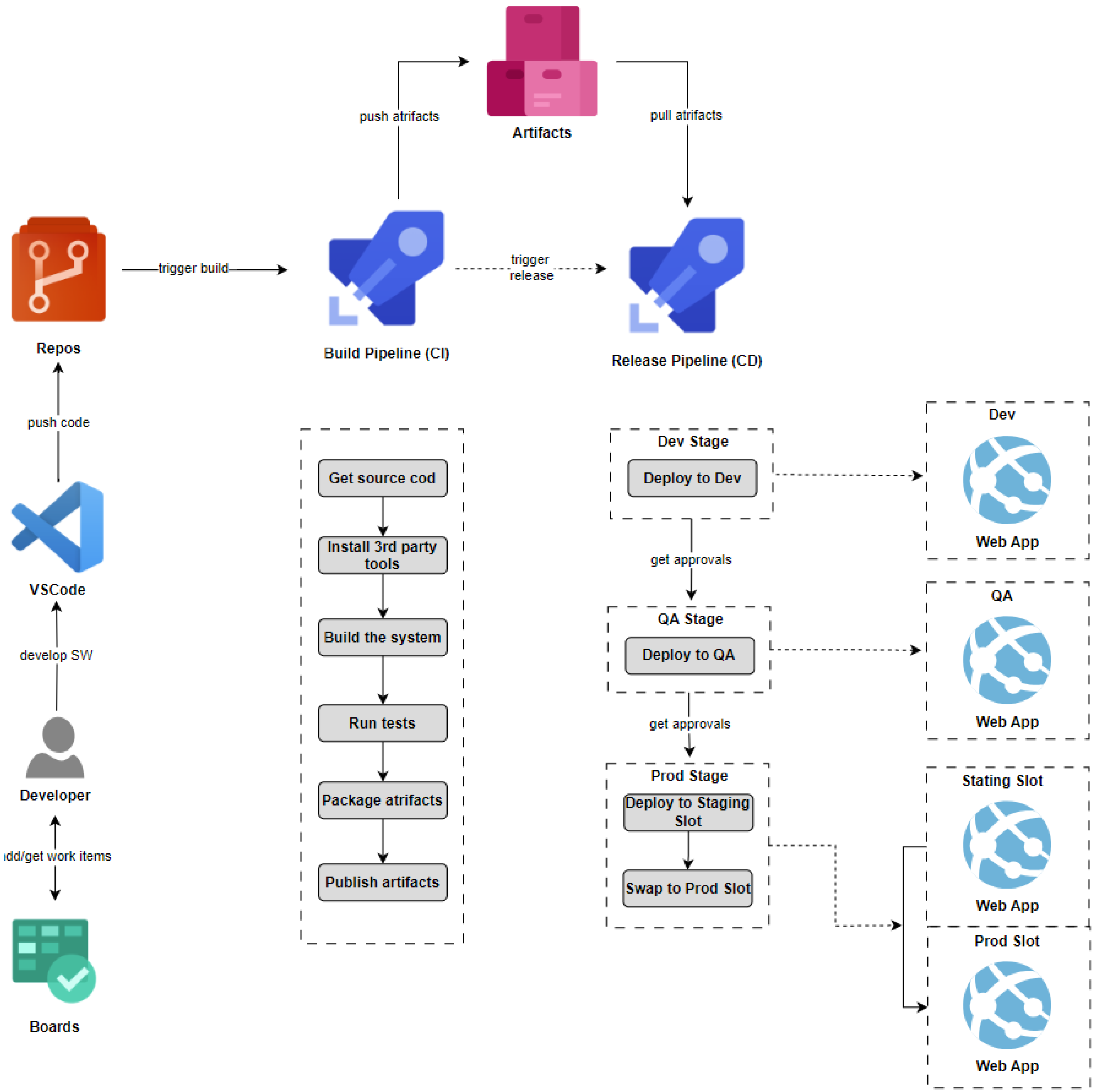
Basic git concepts #1

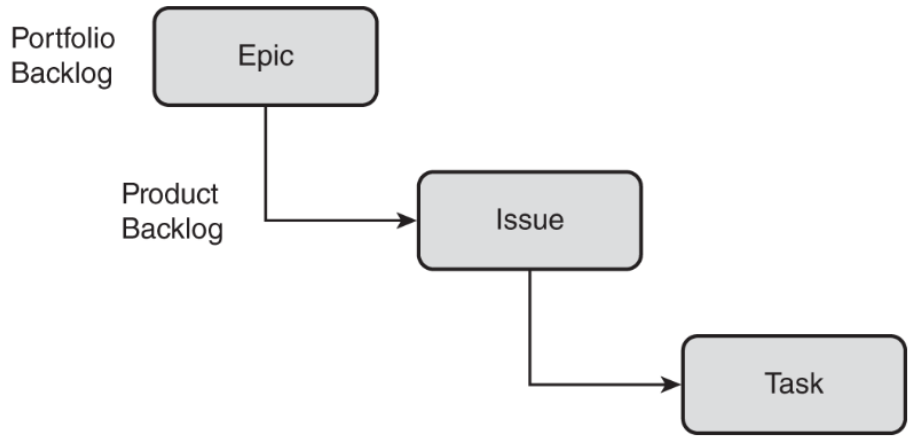
Majid Babaei


REVIEW

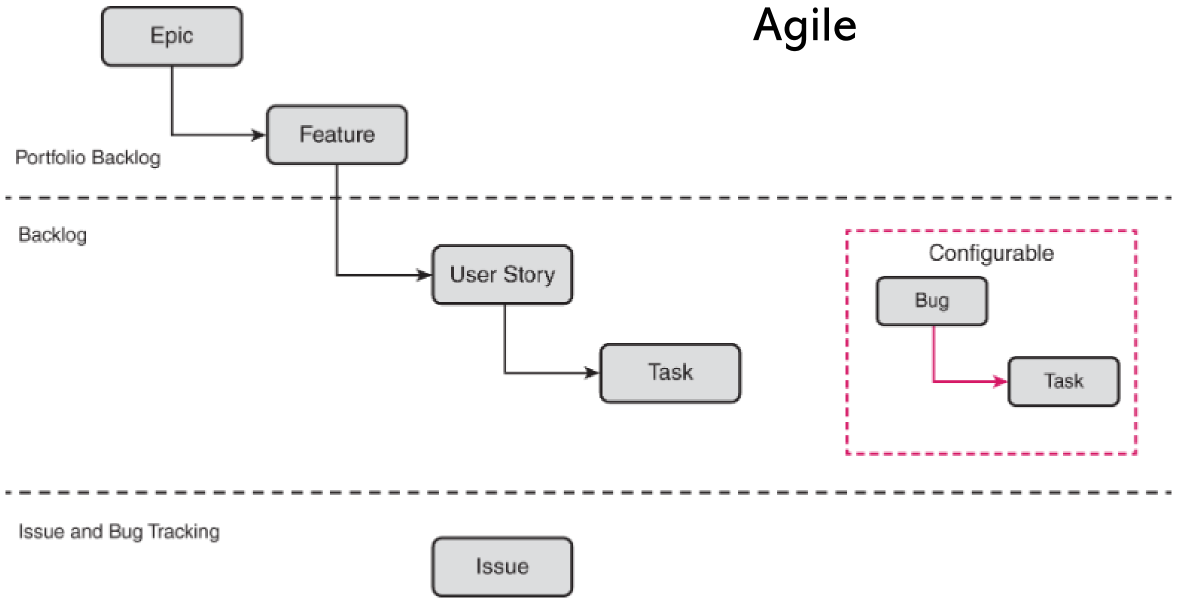


Artifacts
push atrifacts
pull atrifacts
Repos
trigger build
Build Pipeline (CI)
trigger release
Release Pipeline (CD)
push code
VSCode
develop SW
Developer
idd/get work items
Boards
Get source cod
Install 3rd party tools
Build the system
Run tests
Package atrifacts
Publish artifacts
Dev Stage
Deploy to Dev
get approvals
QA Stage
Deploy to QA
get approvals
Prod Stage
Deploy to Staging Slot
Swap to Prod Slot
Dev
Web App
QA
Web App
Stating Slot
Web App
Prod Slot
Web App

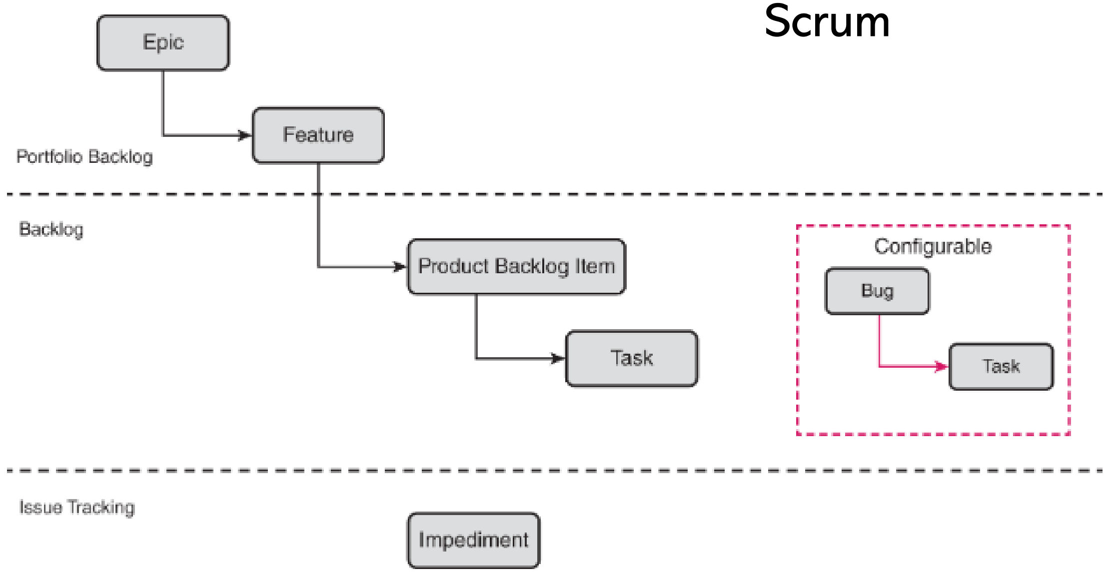
Basic

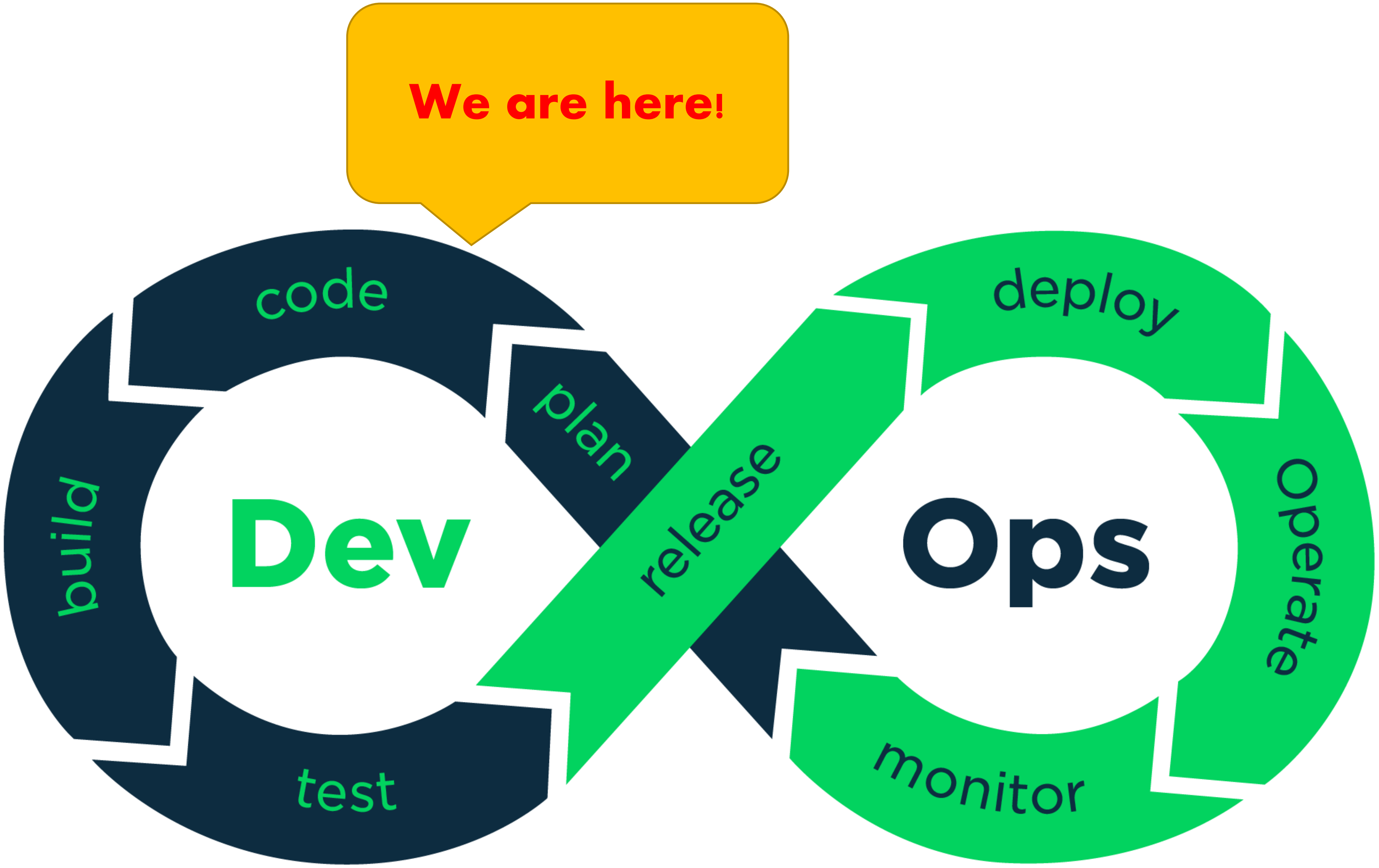


Agile



Scrum




We are here!
code
plan
build
Dev
test
release
deploy
Ops
operate
monitor

A few years ago developers encountered recurring problems as follows:

- *How to share my code with my team members*
- *How to version the update of my code*
- *How to track changes in my code*
- *How to retrieve an old state of my code or part of it*

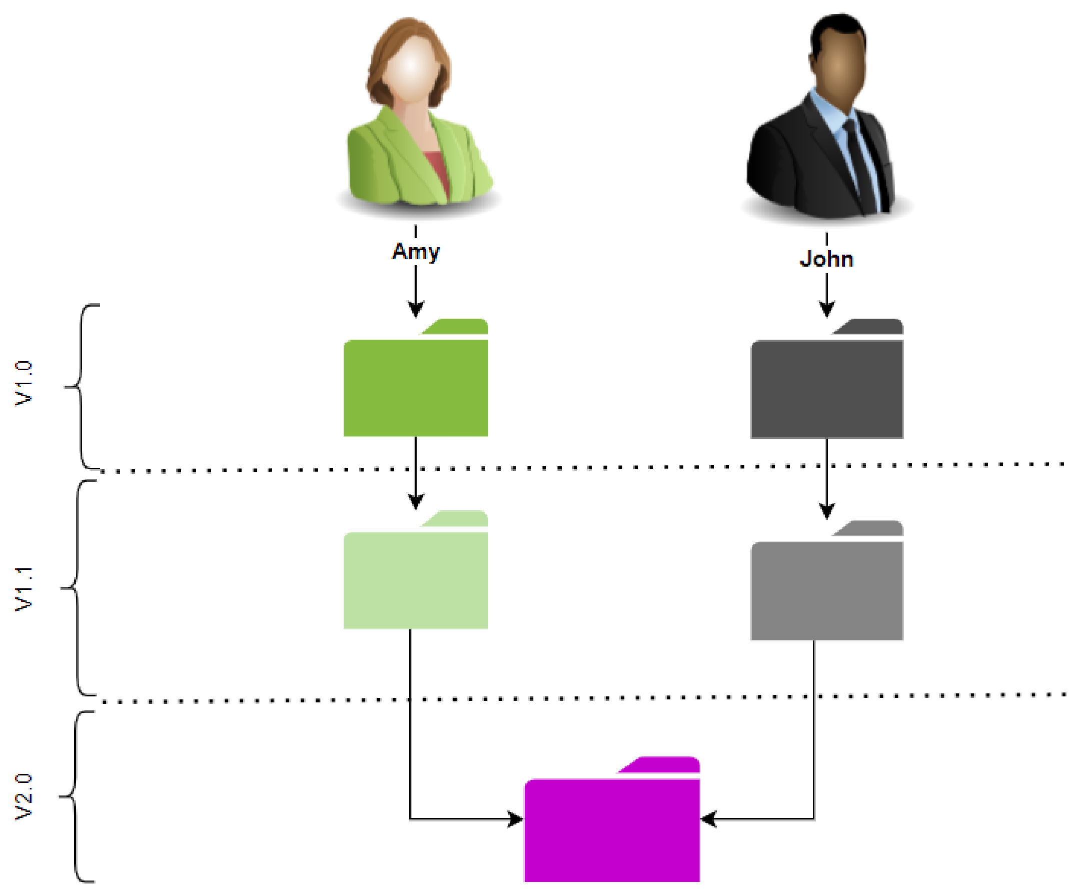
Over time, these issues have been solved with the emergence of source code managers, also called a **Version Control System** (**VCS**).

The goals of these VCSes are mainly to do the following:

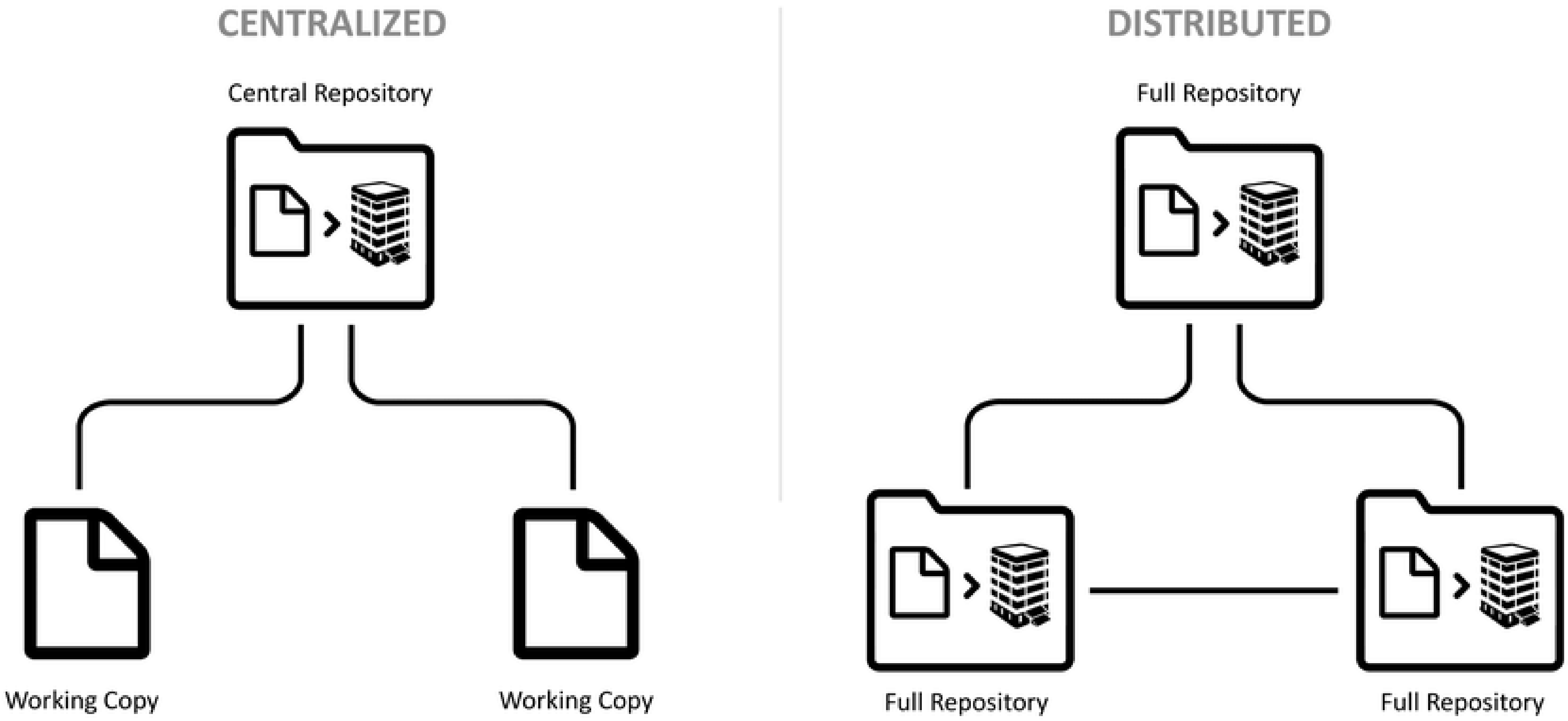
- Allow collaboration of developers, retrieve the code, and version the code.

With the advent of agile methods and DevOps culture, the use of a VCS in processes has become essential.

You would have to constantly toss around the latest code by email and then manually merge the changes!



There are two types of VCS: *centralized* and *distributed*

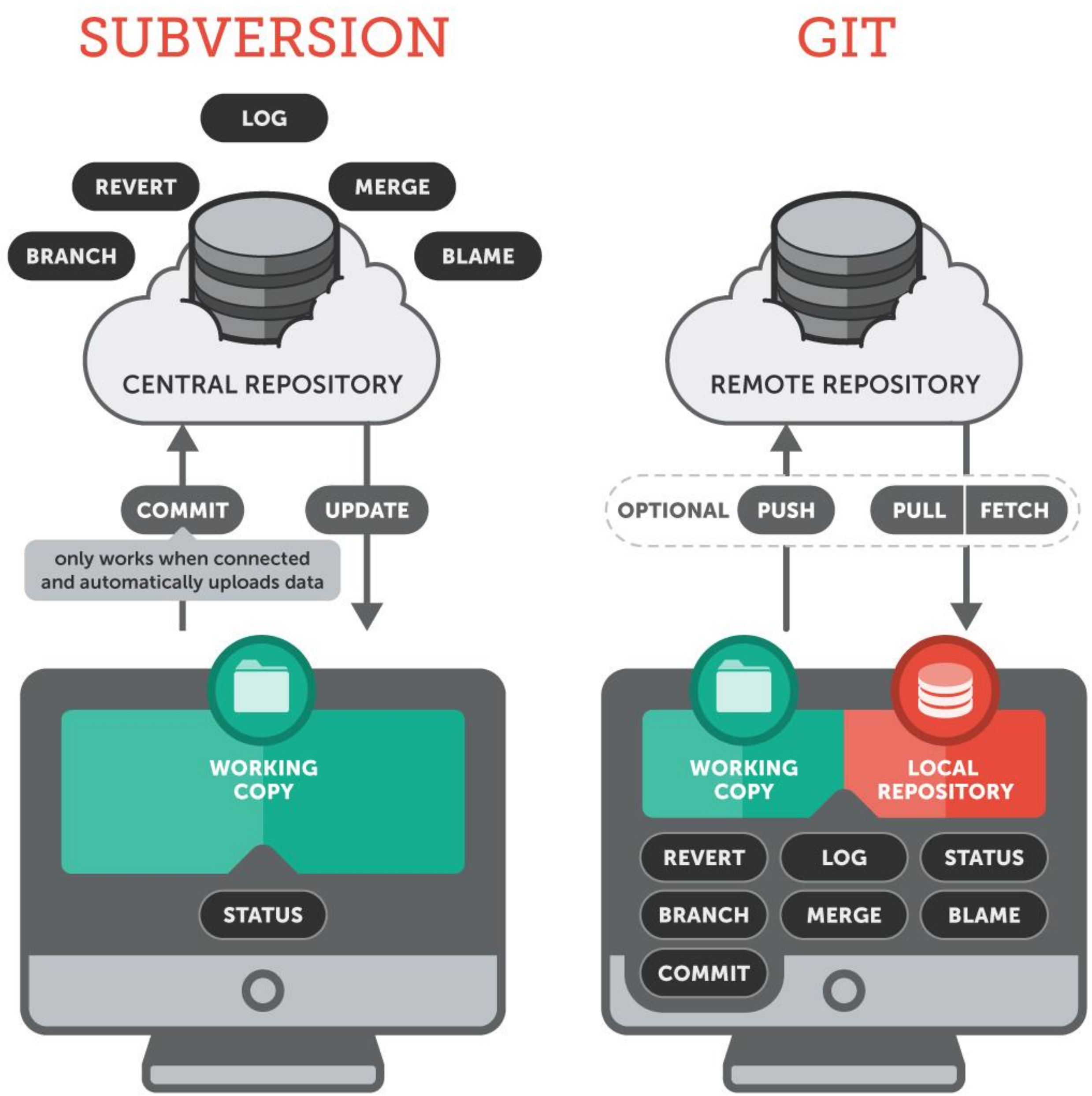


Properties of a centralized version control system

- The first type is the centralized systems, such as SVN, and Subversion.
- Single repository
- Commits requires network connection
- All history is in one place
- Easy access management
- Good option for the development of highly sensitive and critical projects

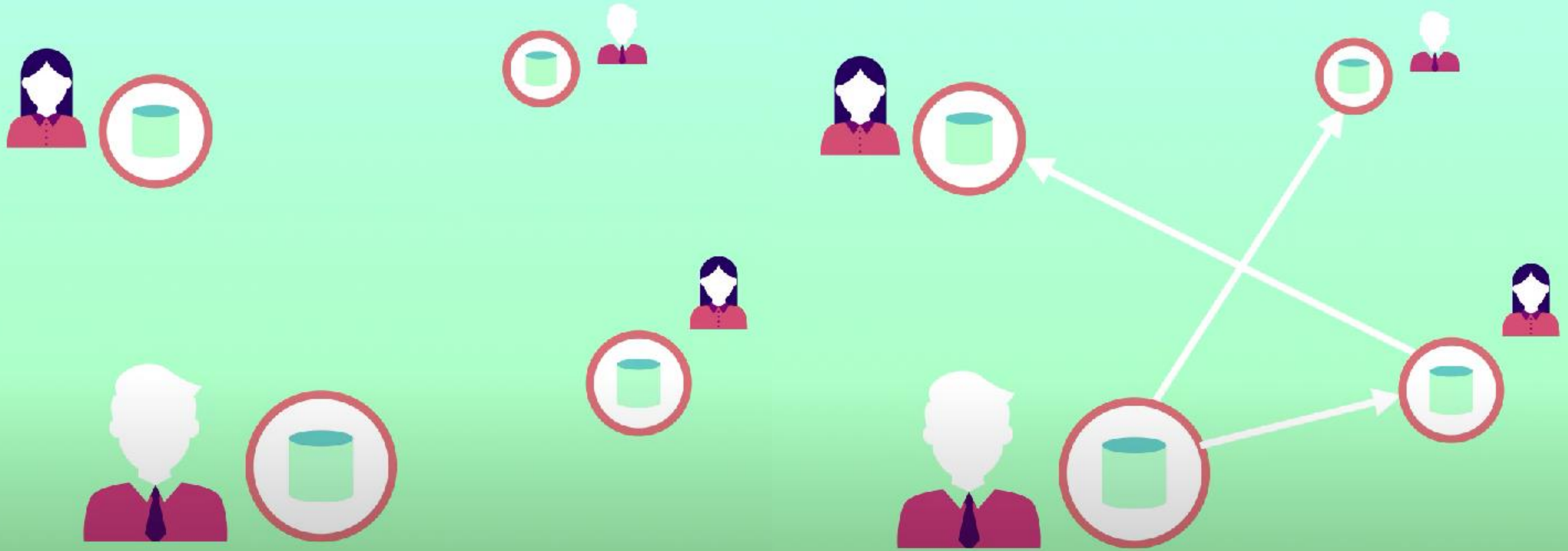
Properties of a distributed version control system

- Distributed version control systems such as *Mercurial* or *Git*.
- Multiple repositories
- Commits does not require network connection
- All history is not in one place
- Multiple branches can be developed in parallel
- Good option for the development of open-source projects


SUBVERSION
LOG
REVERT
MERGE
BRANCH
BLAME
CENTRAL REPOSITORY
COMMIT
UPDATE
only works when connected
and automatically uploads data
WORKING
COPY
STATUS
GIT
REMOTE REPOSITORY
OPTIONAL
PUSH
PULL
FETCH
WORKING
COPY
LOCAL
REPOSITORY
REVERT
LOG
STATUS
BRANCH
MERGE
BLAME
COMMIT

Distributed version control system

If the central sever goes offline developers can sync directly with each other



Distributed version control system

- With this distributed system, *even in the event of disconnection from the remote repository*, the developer can continue to work with the local repository.

- Synchronization will be done when the remote repository is accessible again. And on the other hand, a copy of the code and its history is also present in the local repository.

- *Git* is, therefore, a distributed CVS that was created in 2005 by Linus Torvalds and the Linux development community.

To learn a little more about Git's history, read this page:
https://gitscm.com/book/en/v2/Getting-Started-A-Short-History-of-Git

A bit of History of Version Control Systems!

- First generation (1972): *[source code control system (SCCS)]*
  - It was designed to track changes to individual files, and checked-out files can be edited locally by one user at a time.
  - All users connect to the same shared Unix host with their own account.

- Second generation (1982): *[Apache Subversion (SVN)]*
  - It is a client-server version control system using a concurrency model based on locking and merging.
  - It led to a centralized repository containing the main version of the project.

- Third generation (2000s): *[Git]*
  - It introduced distributed version control.
  - They are merge-based concurrency models that increase the overall history stored on each peer.

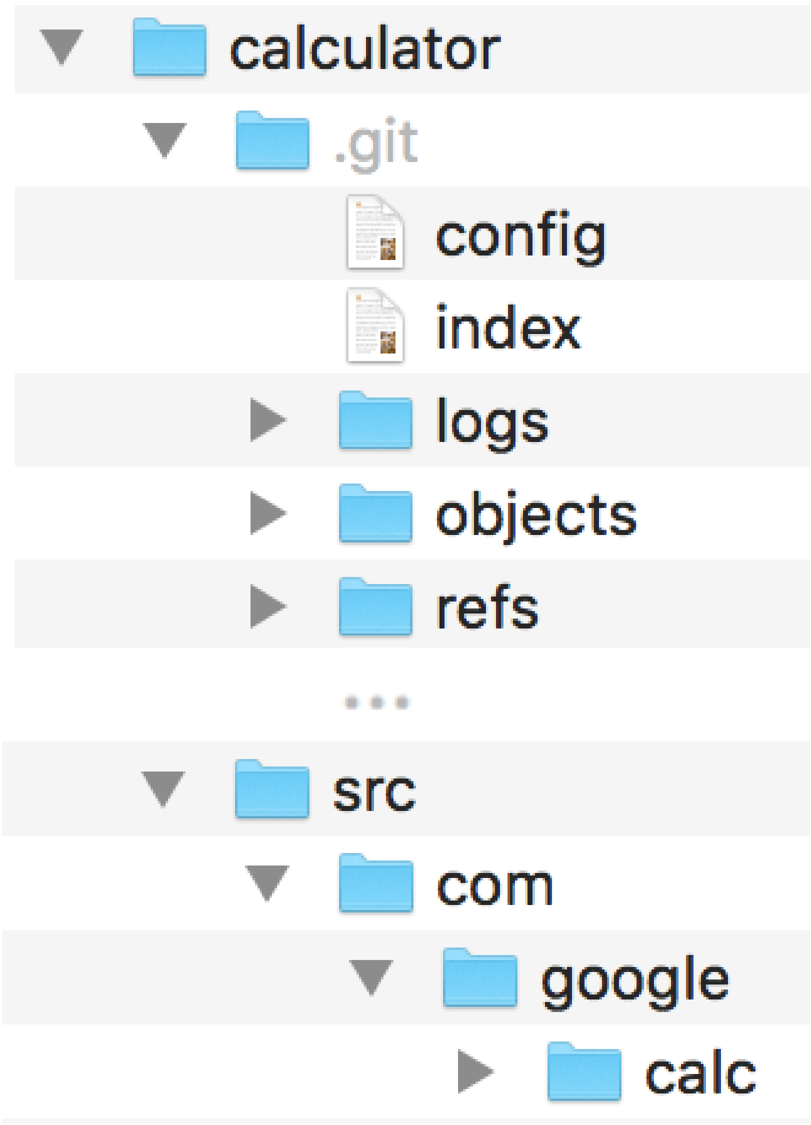
Git

- Git is a free, cross-platform tool, and it can be installed on:
  - a *local machine* for people who manipulate code. *It works for small teams*
  - but can also be installed on *servers to host and manage remote repositories*.

- Git is *a command-line tool* with a multitude of options. Nevertheless, today there are many graphical tools, such as *Git GUI*, *Git Kraken*, *GitHub desktop*, or *SourceTree*.

- Fortunately, many code editors such as Visual Studio Code, JetBrains, and SublimeText allow direct code integration with Git and remote repositories.

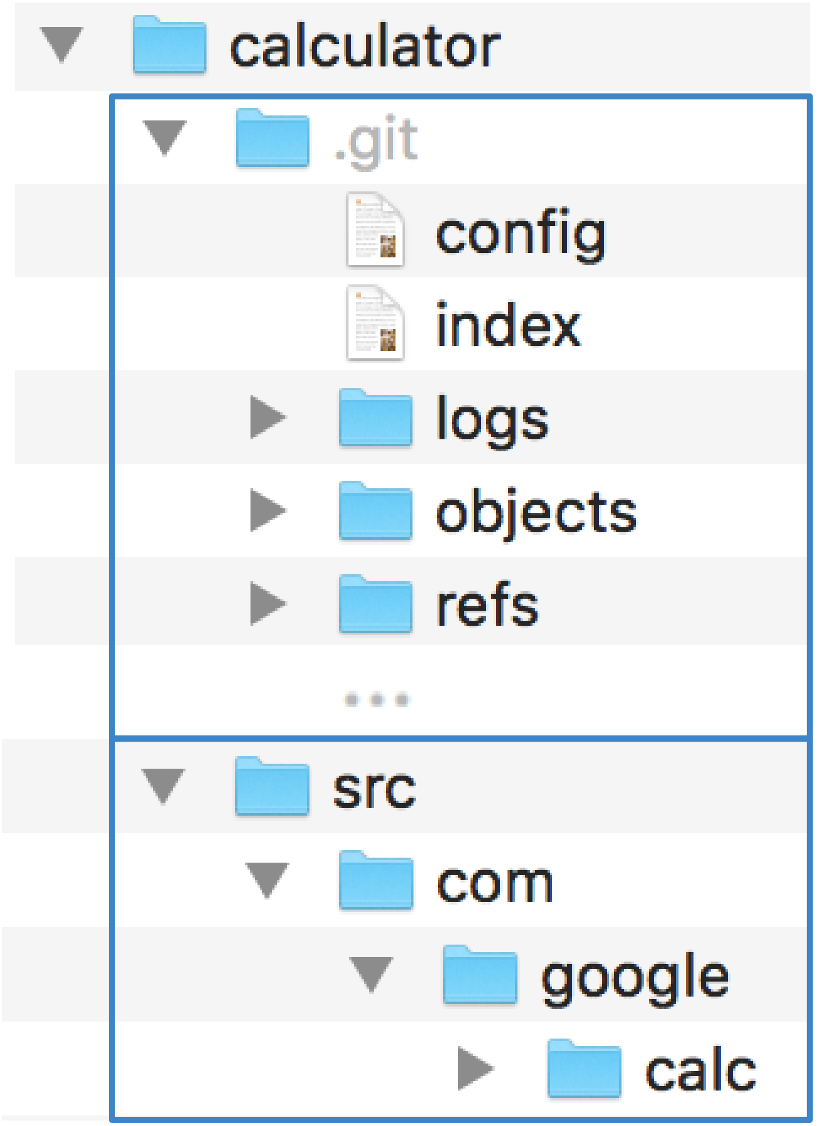
Git Repository Structure



A ***Git repository*** is created by:

- *`git init`*
- *`git clone`*

Git Repository Structure



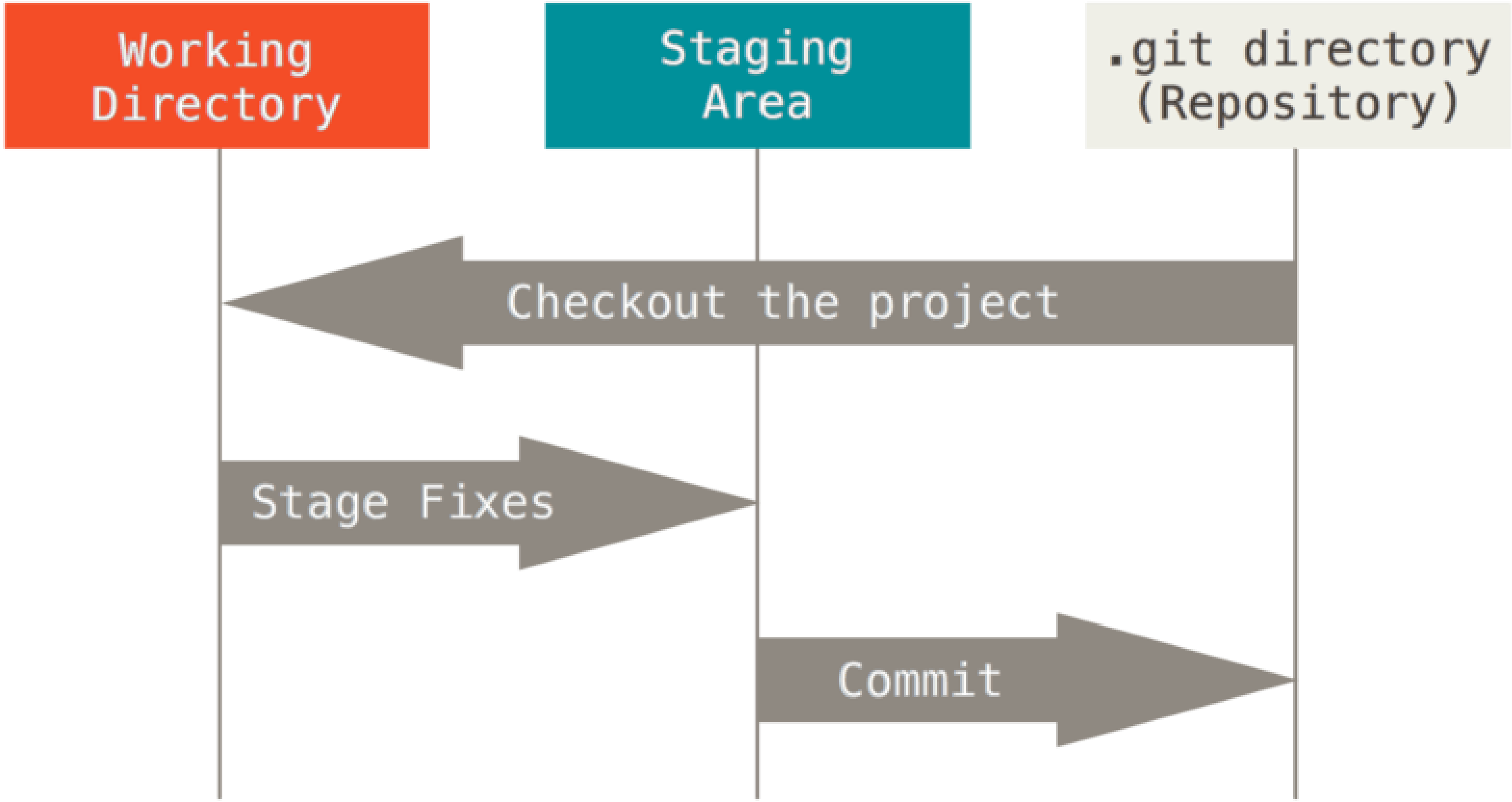
.git folder is the ***Git repository***

files/folders next to the ***.git*** folder are the ***working tree***

A ***Git repository*** has at most one ***working tree***.

Three main sections of a Git project

- Three main sections of a Git project: **the working tree**, **the staging area**, and the **Git directory**.



Three main states in Gits

- Git has three main states that your files can reside in: *modified*, *staged*, and *committed*:

- **Modified** means that you have *changed the file* but have not committed it to your *local* repository yet.

- **Staged** means that you have *marked a modified file* in its current version to go into your next commit snapshot.

- **Committed** means that the data is *safely stored* in your local repository.

Tools that allows us to use Git

- The Command Line
- Code editors & IDEs (e.g.,. VSCODE)
- Graphical user interface (e.g., Git Client, Git Kraken)

Why Using Command Line

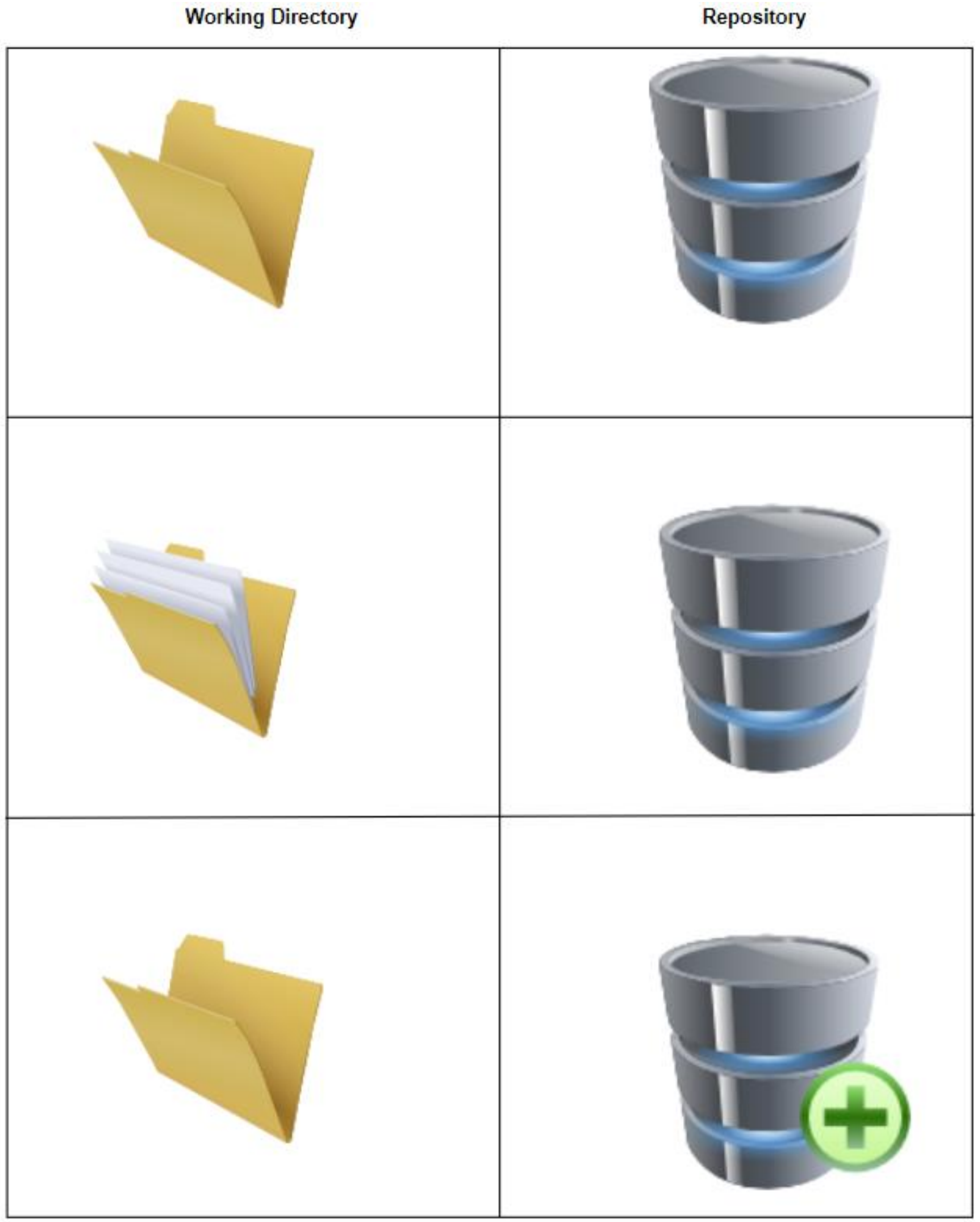
- GUI tools have limitations
- Sometimes they are more complex and confusing
- They are not always available

Initializing a git repository

```
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git init
Initialized empty Git repository in C:/Users/drbab/OneDrive/Desktop/tmp/workspace/git-practice/.git/
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> ls
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> ls -force


    Directory: C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d--h--          9/6/2023   8:39 PM                .git


PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> cd .\.git\
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice\.git> ls


    Directory: C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice\.git


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          9/6/2023   8:39 PM                hooks
d-----          9/6/2023   8:39 PM                info
d-----          9/6/2023   8:39 PM                objects
d-----          9/6/2023   8:39 PM                refs
-a----          9/6/2023   8:39 PM            130 config
-a----          9/6/2023   8:39 PM             73 description
-a----          9/6/2023   8:39 PM             23 HEAD
```

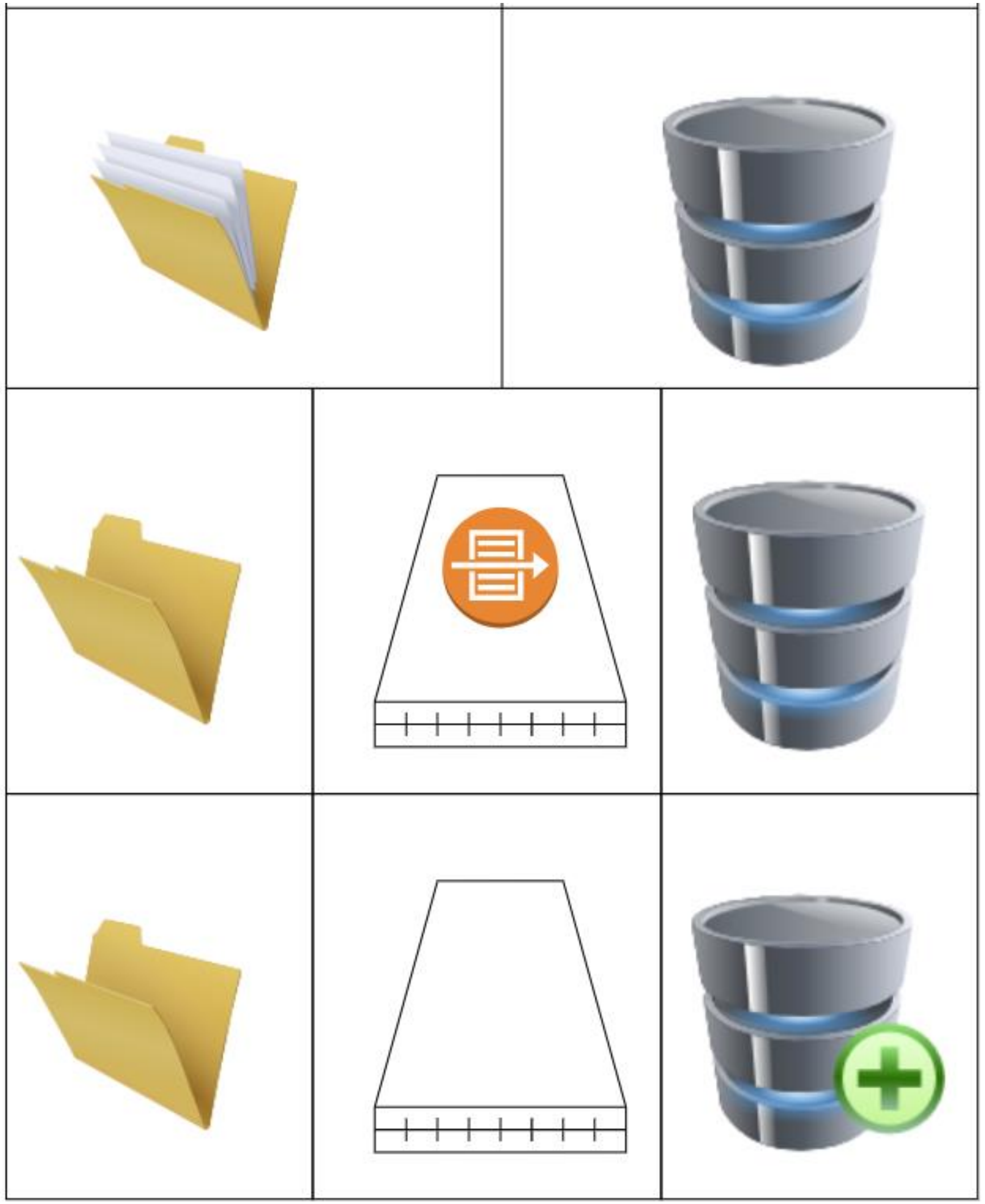
Basic Git workflow

- We have a WD and a Local Repository
- As a part of our job everyday we modify one or more files
- When it reaches the state that we want to record
- We commit those changes into the local repository
- Making a commit is like creating a snapshot of your project



Basic Git workflow

- In Git we have a special area that does not exist in most VCSs
- Staging Area / Index
- What we are proposing for the next commit
- When we are done with the changes:
  - Add to the staging area
  - Review the file
  - Commit to the repository



Let’s create two files in the working directory

```
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> echo file1 > file1.txt
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> echo file2 > file2.txt
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> ls


    Directory: C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----          9/6/2023   8:59 PM             16 file1.txt
-a----          9/6/2023   8:59 PM             16 file2.txt
```

Now add the files to the staging area

```
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git add .\file1.txt .\file2.txt
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git commit -m "new files added"
[master (root-commit) 57f2f93] new files added
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt
 create mode 100644 file2.txt
```

Then create a snapshot to the local repository.

What is inside the staging area?

A common misunderstanding is once we commit the changes the staging area becomes empty.
It is not correct!

What we currently have in Staging Area is:
the same snapshot we stored in the repository

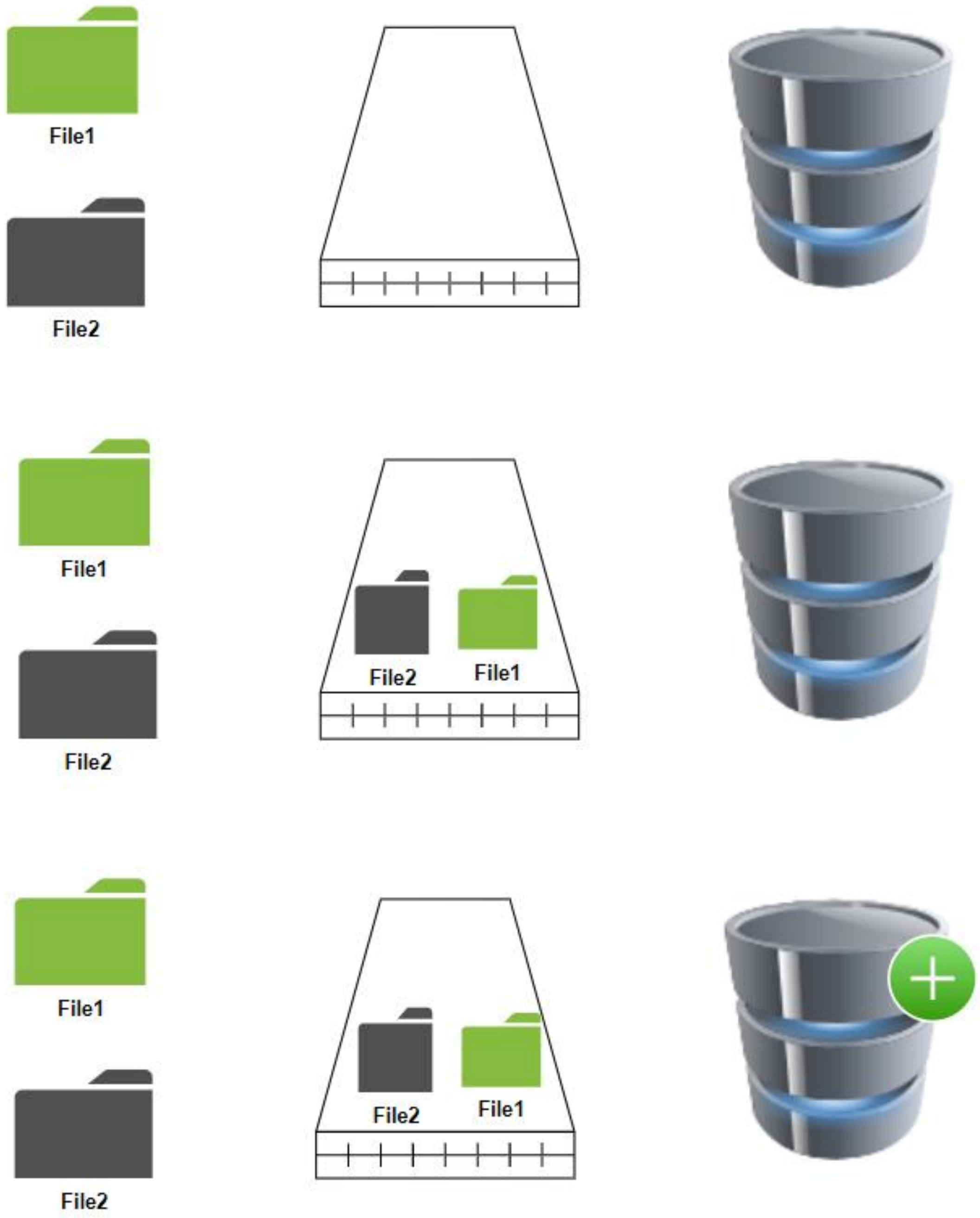
It is more like production version of the software

Show the objects in the staging area

```
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git ls-files --stage
100644 02edef5b6502e522e1b998dd55736ea4d2d039aa 0       file1.txt
100644 077d212ad1d3a20e16ce13e2236bc179c470c1ad 0       file2.txt
```

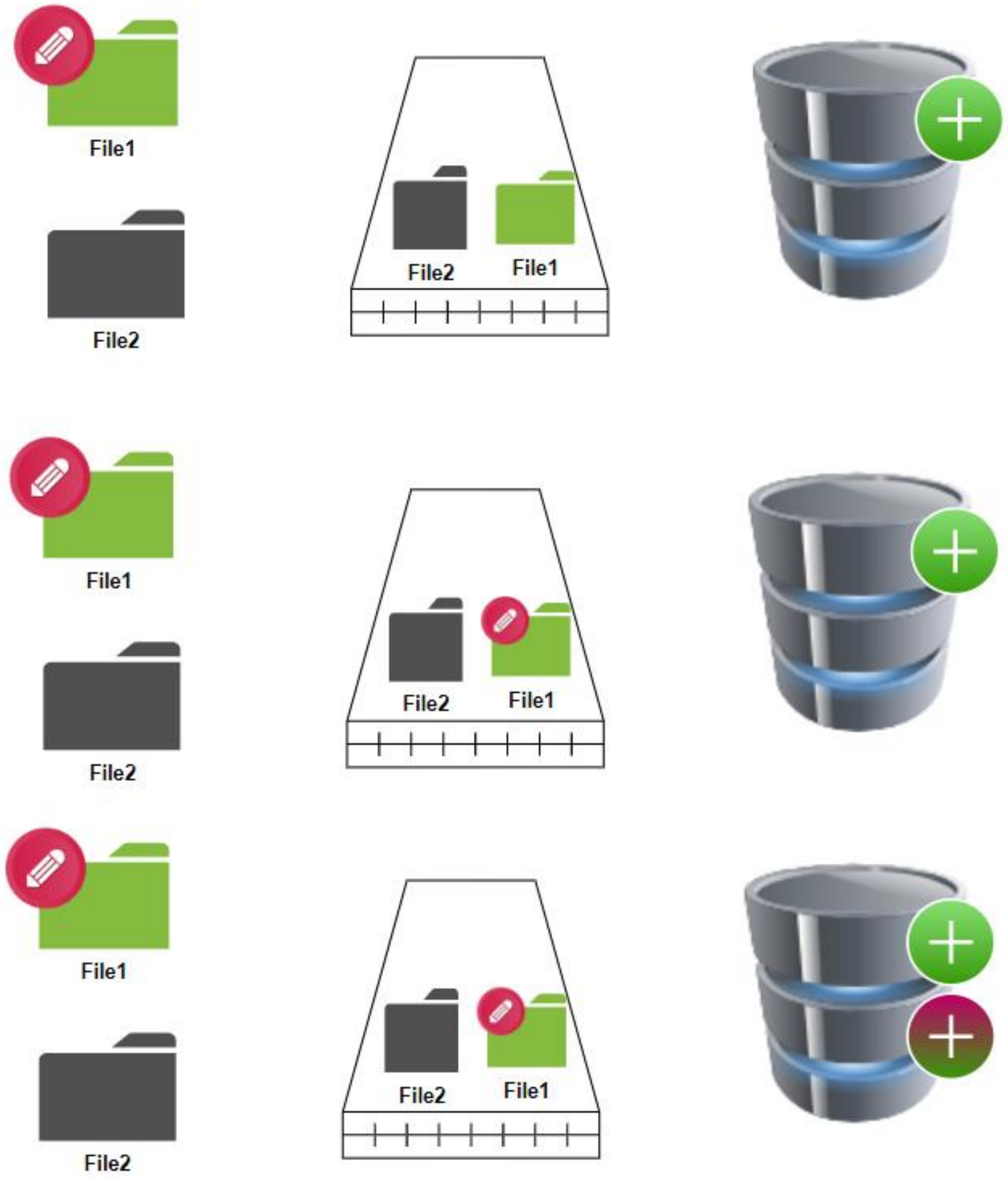
The index is a binary file (generally kept in .git/index) containing a sorted list of path names, each with permissions and the SHA1 of an object.

```
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice\.git> cat .\index
DIRC□□dù h(kdù h(k¤□□íï[e□å"á¹˜ÝUsn¤ÒÐ9ª        file1.txtdù q,ð□„dù q,ð□„¤□}!*ÑÓ¢□Î□â#kÁyÄpÁ-   file2.txtTREE□2 0
œr˜□□Ý¶M±ï˜.˜□«”Ú4b!-³<□2□Ö>Q□àPŽøßŸÇ
Ì
```


File1
File2
File1
File2
File2
File1
File1
File2
File2
File1

Let's modify the file1.txt

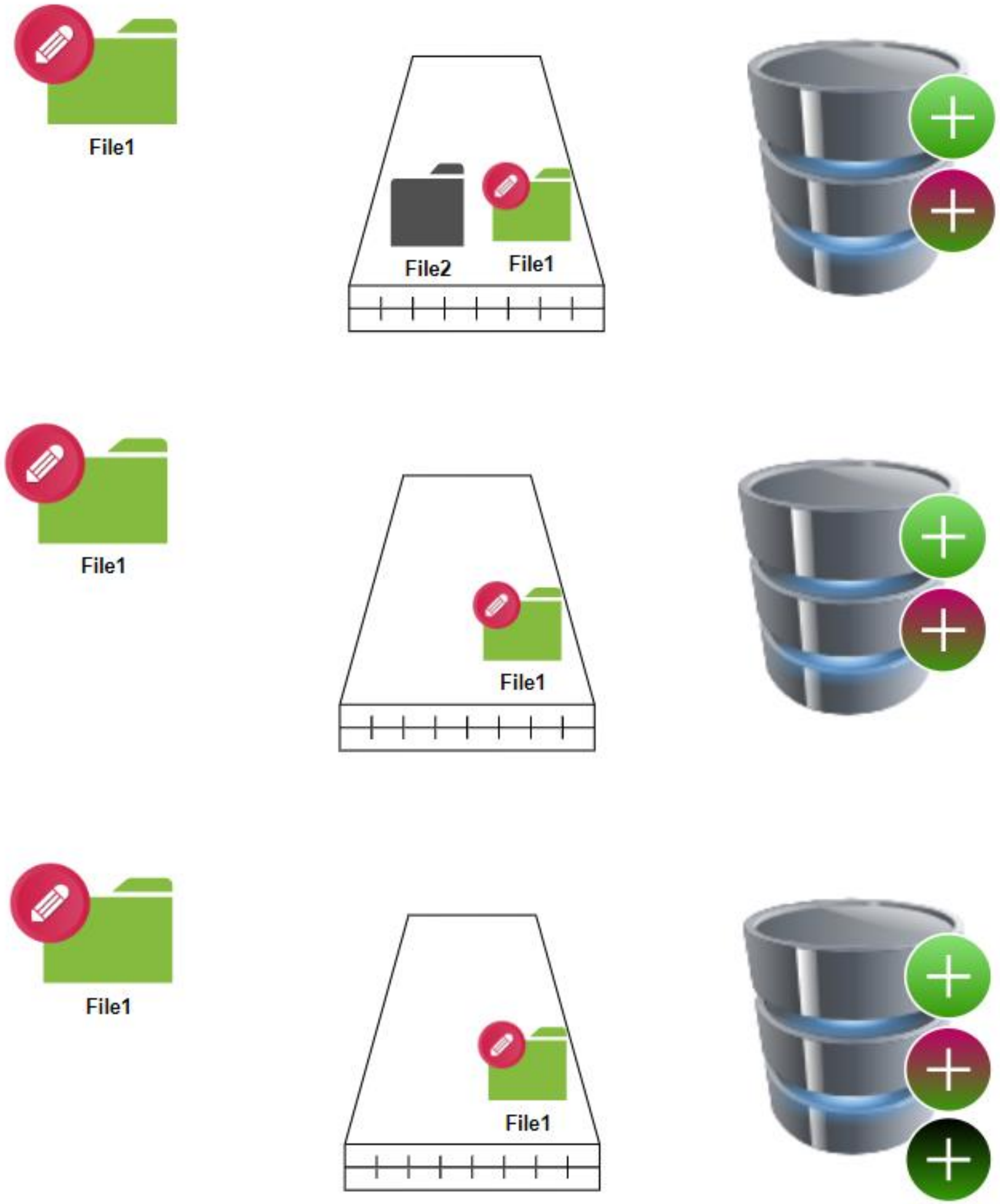
```
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git status
On branch master
nothing to commit, working tree clean
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> echo "add a new content into file1.txt" >> .\file1.txt
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> cat .\file1.txt
file1
add a new content into file1.txt
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> 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:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git add .\file1.txt
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.txt

PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git commit -m "file1.txt is updated"
[master cd4cbfd] file1.txt is updated
 1 file changed, 0 insertions(+), 0 deletions(-)
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git status
On branch master
nothing to commit, working tree clean
```



File1
File2
File2
File1
File1
File2
File2
File1
File1
File2
File2
File1

Let’s delete file2.txt

```
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> rm .\file2.txt
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    file2.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git add .\file2.txt
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    file2.txt

PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git commit -m "file2.txt is removed"
[master cb5b665] file2.txt is removed
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 file2.txt
PS C:\Users\drbab\OneDrive\Desktop\tmp\workspace\git-practice> git status
On branch master
nothing to commit, working tree clean
```


File1
File2
File1
File1
File1
File1
File1

How does git manage to perform these functionalities in a right way?

What are inside a git commit?

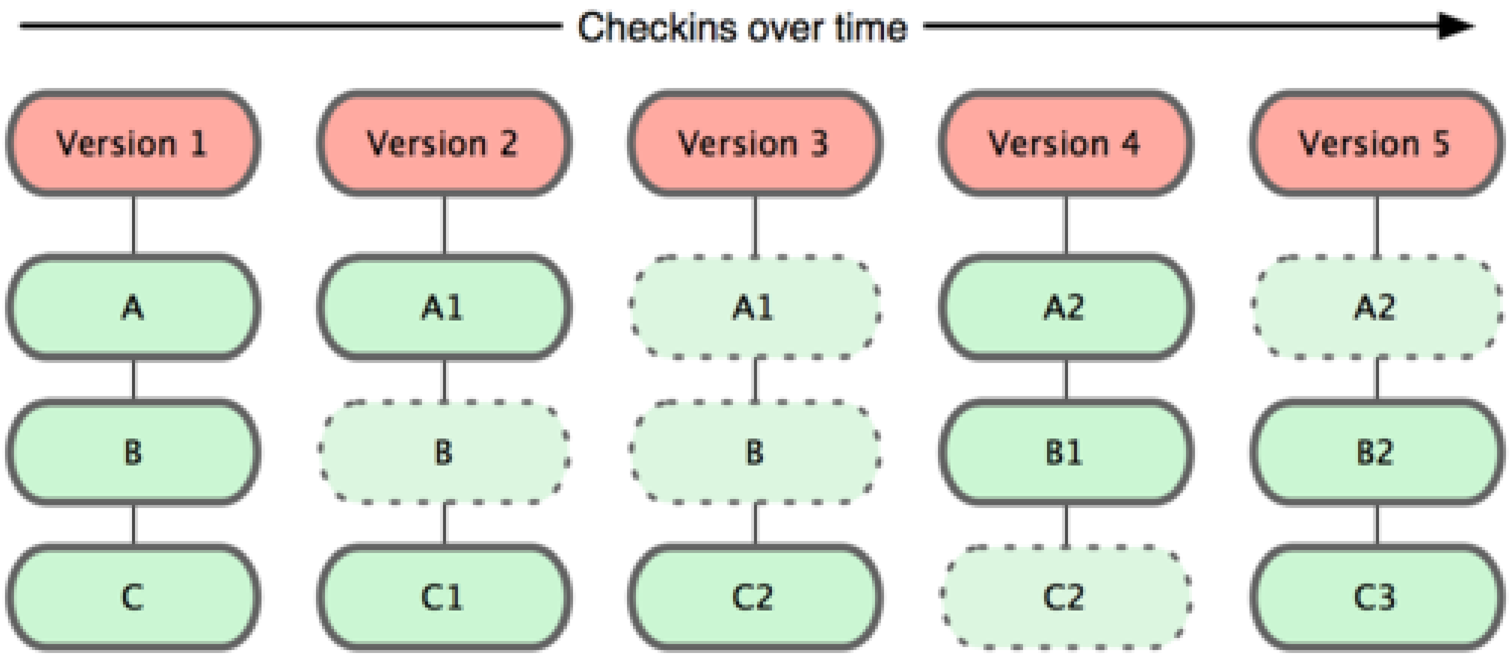
- Unlike many other VCSs, *Git doesn't store deltas* (what was changed), instead it stores the full content!
- Git can quickly restore the project to an earlier snapshot without having to compute the changes
- Git is very efficient in data storage!

Every commit contains:

- ID
- Message
- Date
- Author
- Complete snapshot

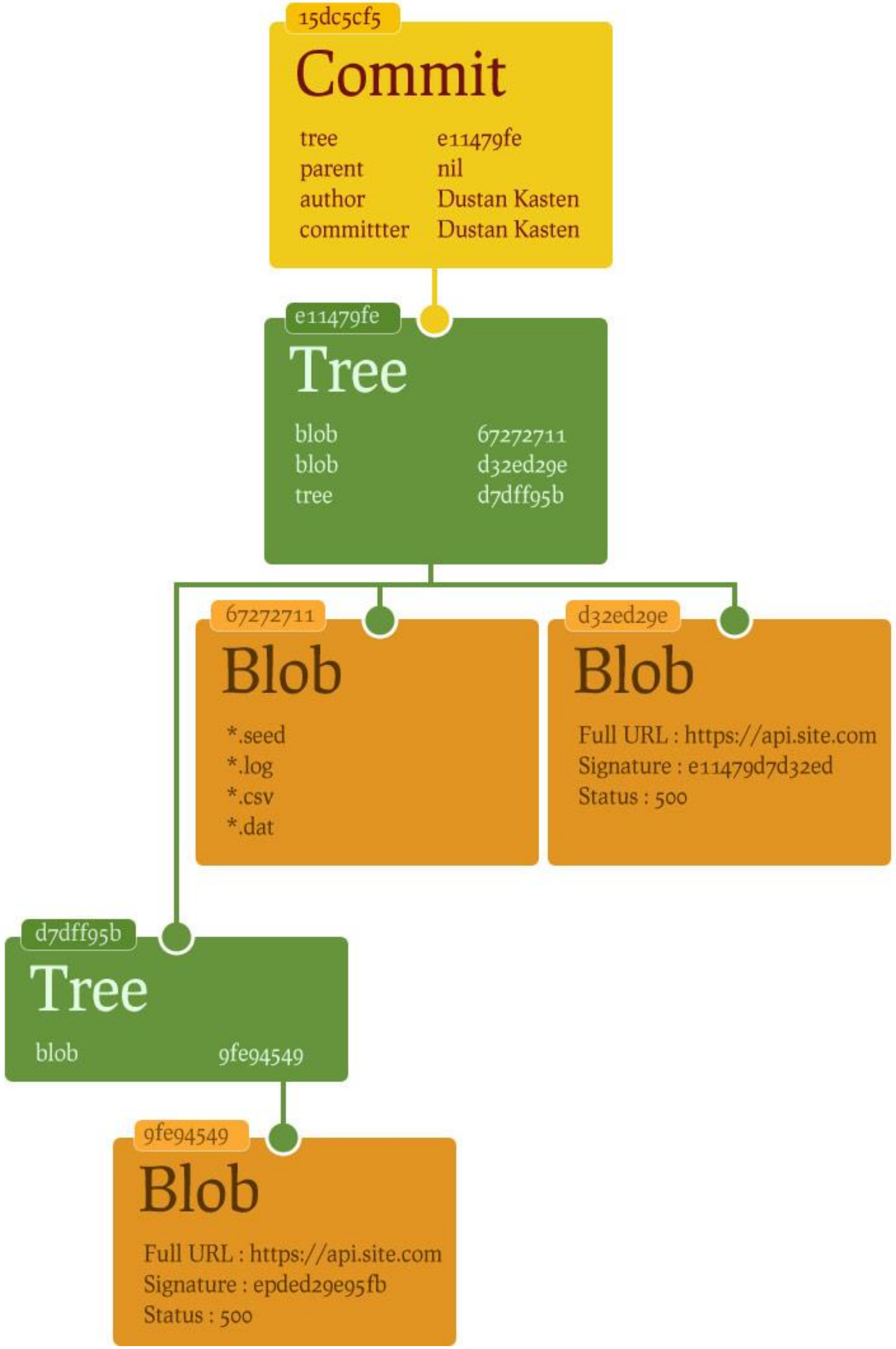
Data Storage

- Git compress data and does not store duplicate files



You don't need to know implementation detail to use Git (it may change in the future!)

Deep dive into git objects



Data Storage - *blob*

We should shift our focus only on
.git/objects content—the primary data storage in Git.

Git stores every single version of each file it tracks as a *blob*.

Git identifies blobs by the hash of their content and keeps them in .git/objects.

Any change to the file content will generate a *completely new blob object*.

The easiest way to create an object is to *add an object to the stage*.

Data Storage - *blob*

After adding our new file to the stage, inside .git/objects, we have:

We have a new folder, *34*,
and inside that folder a file
5e6aef713208c8d50cdea23b85e6ad831f0449.

```
$ ls -R .git/objects
34      info    pack

.git/objects/34:
5e6aef713208c8d50cdea23b85e6ad831f0449

.git/objects/info:

.git/objects/pack:
```

If you apply a modification and add the file again you will see a new object is created!

Data Storage - *tree*

The *tree objects* are how Git is storing folders.

Like blobs, Git identifies each tree by the *hash of its content*.

Because the tree is referencing the hash of each file it contains, any change to the content of files will cause the creation of an entirely new tree object.

One Note and One Question

- If you run *git init* in a directory with several files Git does not track these files *unless you instruct Git* to do that by adding them!

- What would be the output of the last command?
  - *git init*
  - *echo “hello” > file1.txt*
  - *echo “hello” > file2.txt*
  - *git status*

```
On branch master

No commits yet

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

        file1.txt
        file2.txt
```

Continue!

- To add the files we have multiple options:
  - *git add file1.txt; git add file2.txt;*
  - *git add file1.txt, file2.txt*
  - *git add *.txt;*
  - *git add .* (not recommended!)

They are in green because they are in the staging area

- Now if you run *git status* again

```
On branch master

No commits yet

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

        new file:   file1.txt
        new file:   file2.txt
```

Continue!

- Suppose we append something to file1.txt:
  - *echo "world" >> file1.txt*

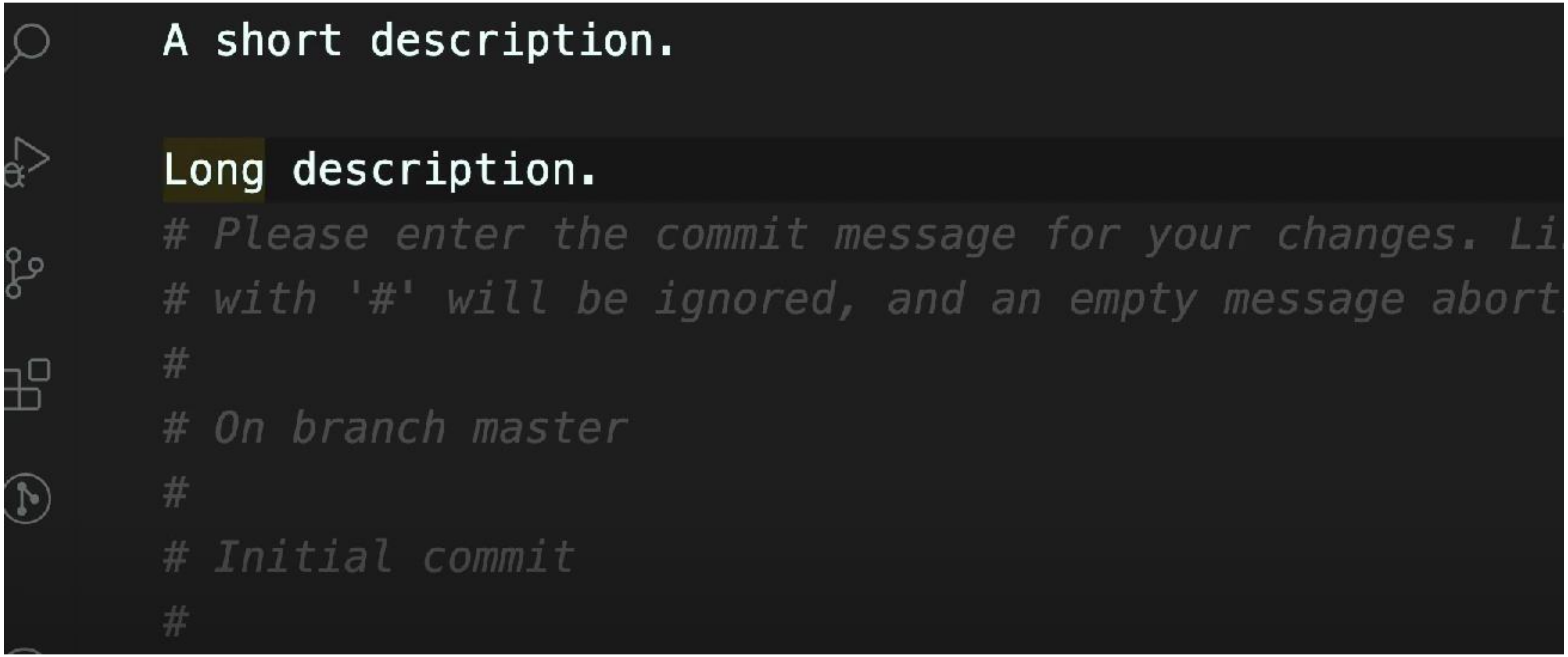
- Now if you run *git status* again

```
On branch master

No commits yet

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

        new file:   file1.txt
        new file:   file2.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
ctory)

        modified:   file1.txt
```

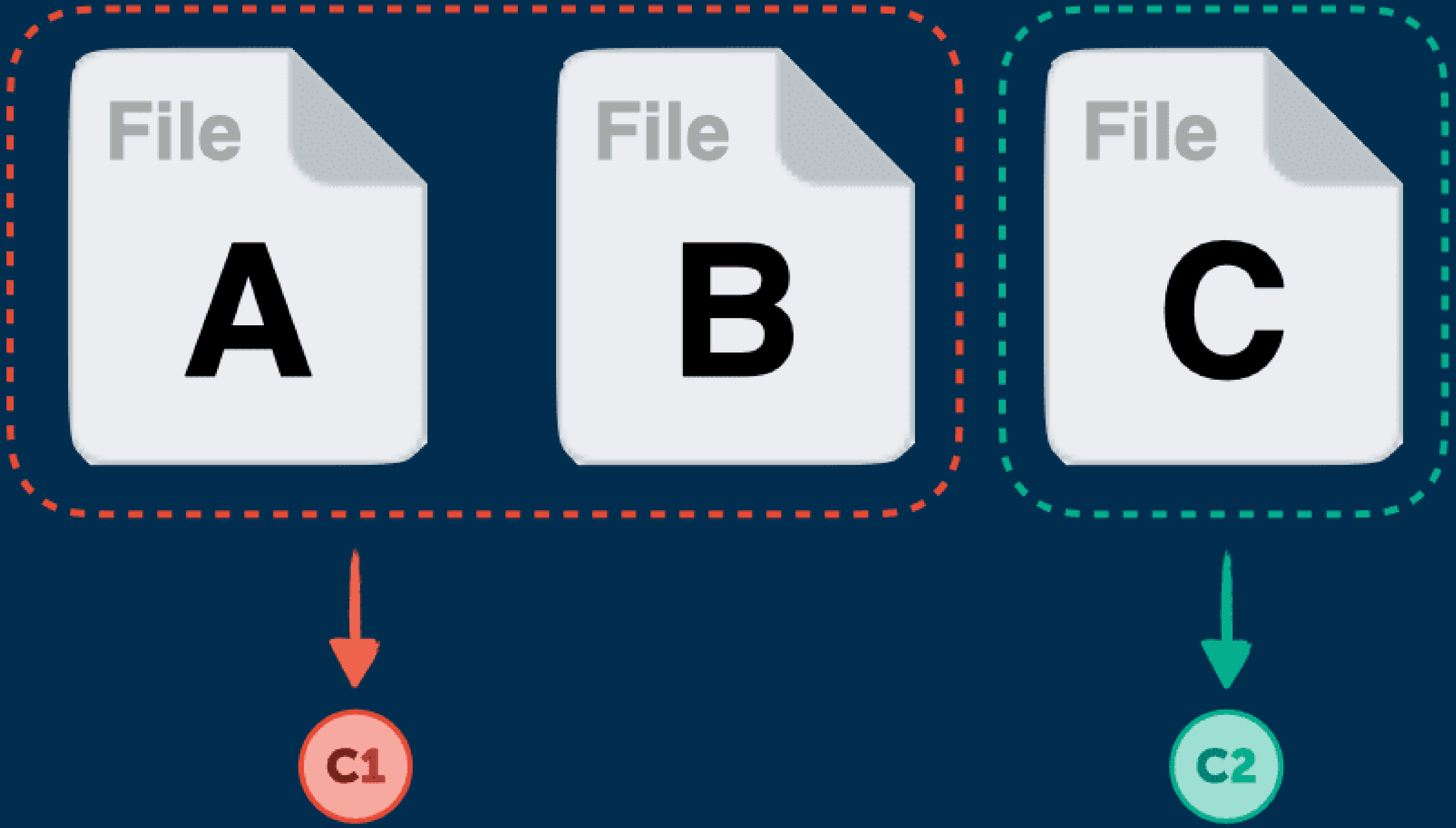
Commit with a short and long description

- You can use *git commit –m "short description"* for only giving a short description

- But if you want to say more just run *git commit*
  - It will open a text editor that allows you to add more information



The Perfect Commit



Commit size matters!

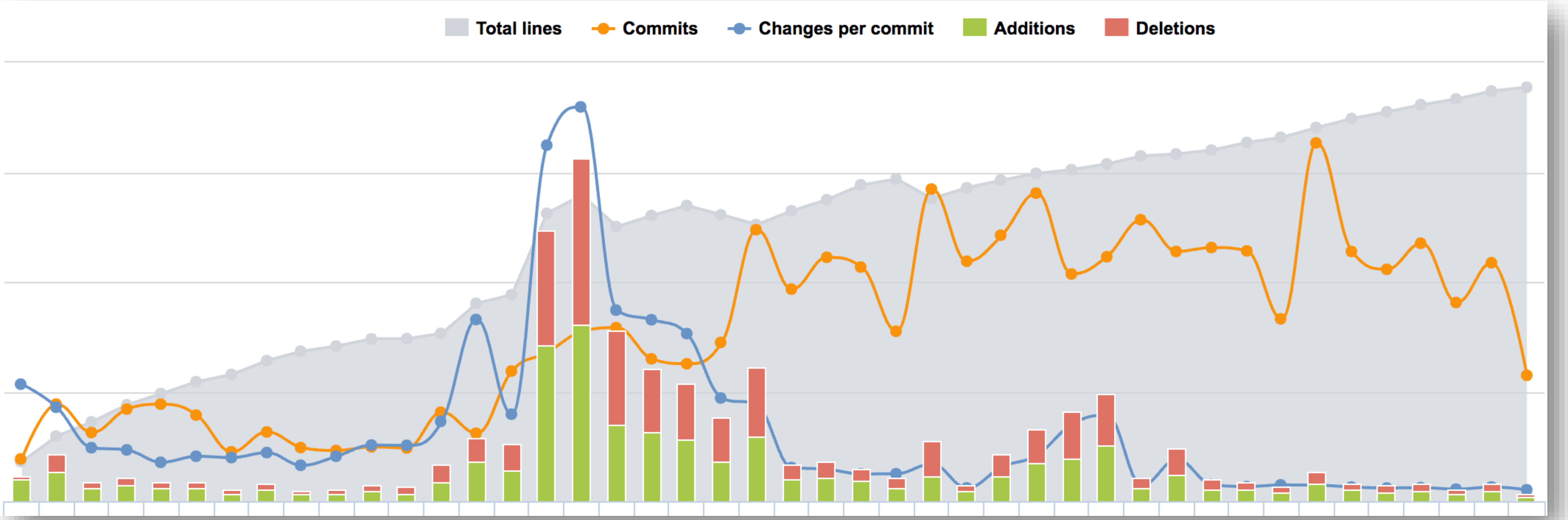
- Your commit shouldn't be *too big* or *too small*

What happens if your commits are too small?

What happens if your commits are too big?

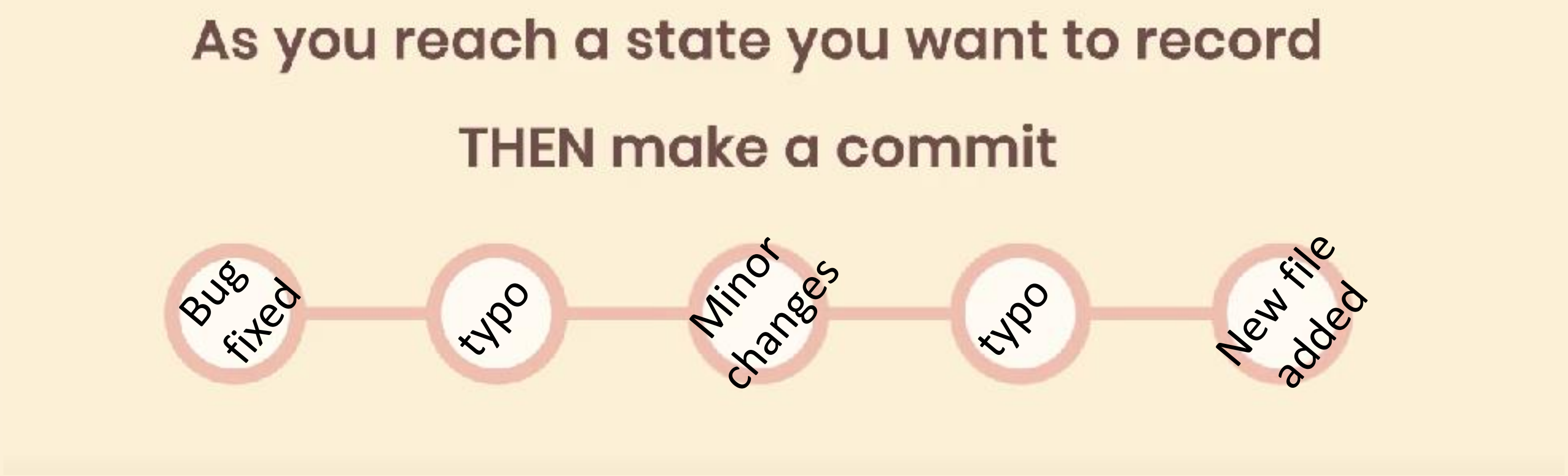
- The whole point of committing is to *record check points* as we go!
- If we screw up, we can go back and recover the code
- Commit often (5 – 10 every day depending the amount of work you do)

Git Commit Statistics



A change set

- All commits should *represent logically a change set*



If you accidentally stage both these changes you can unstage them!

Final points!

- You should give yourself the habit of creating meaningful commit messages!
- All your messages will be show up in the history
- They will be very helpful for your team members
- If you separate your commit logically it will be easy to design a good commit message for them.