**GIT**

**Table of contents**

|  |  |  |
| --- | --- | --- |
| S No | Subject | Page No |
| 1 | What is version control | 3 |
| 2 | Types of version control | 3 |
| 3 | Source Code Management | 5 |
| 4 | Git repo | 5 |
| 5 | How to initialize a repo or clone it | 5 |
| 6 | How to reach folder where you want to do git operations | 5 |
| 7 | Adding, commit and pushing files to repo | 6 |
| 8 | Upstream branch info | 6 |
| 9 | Creating, checkout and pushing branches | 6 |
| 10 | Git config | 7 |
| 11 | Bare repo | 7 |
| 12 | Git init | 8 |
| 13 | Git clone | 8 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**What is version control?**

Version control is also called source control. For developers source code is the crystal ball, it is my precious. So, to make any changes or check changes made or to experiment on code we use VCS.

Say we don’t have VCS or not using VCS then

1) You want to undo the changes made, then you need to remember it or write it down somewhere. {suppose it is a change done months ago}

2) You will also not have a history of all the changes made.

3) Say multiple developers are working on the same code then it will be troublesome.

4) Say multiple developers are working on code but one change is compatible with other change the developer making then, using VCS we can track that down.

To prevent all these problems, we are using VCS.

A few advantages are

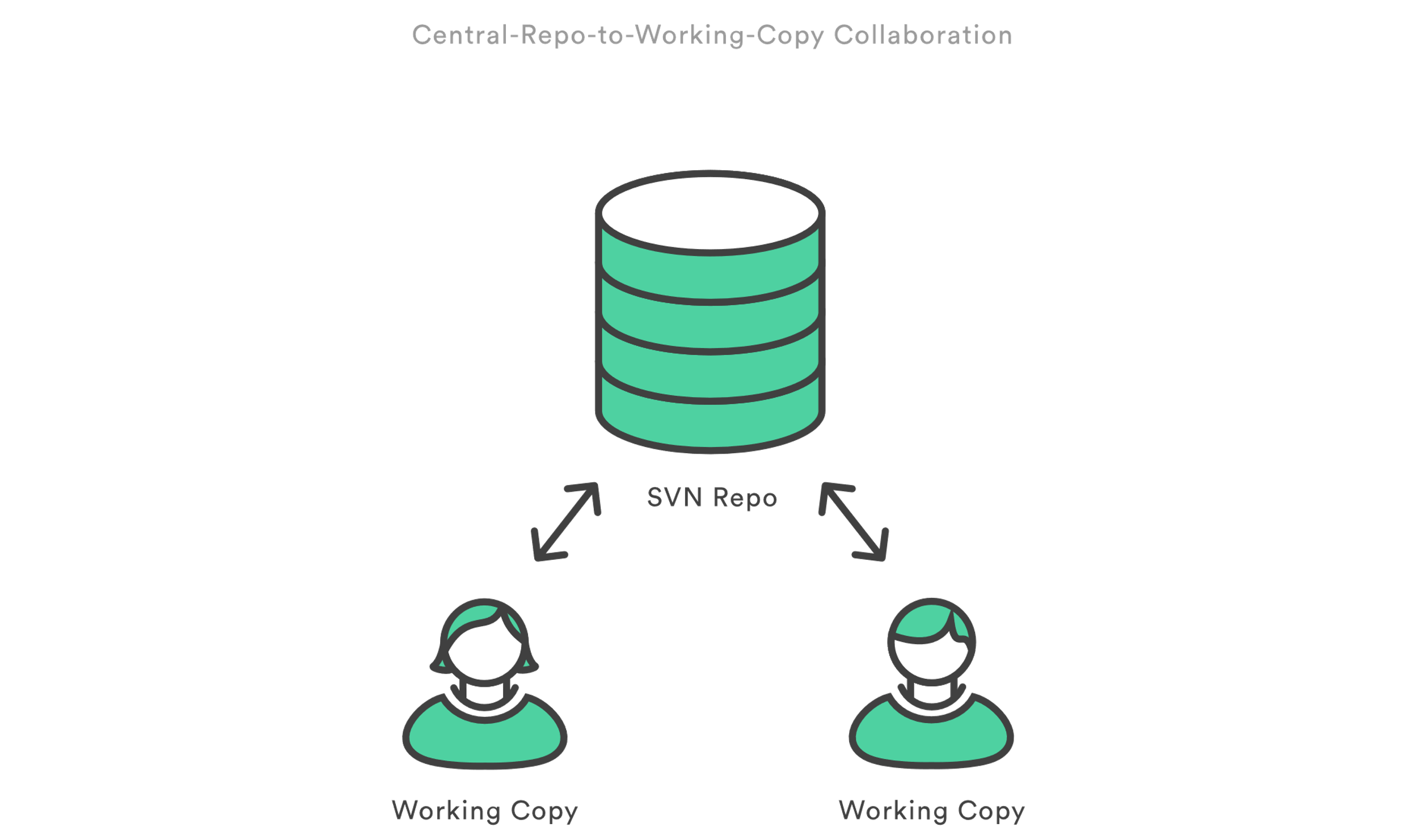
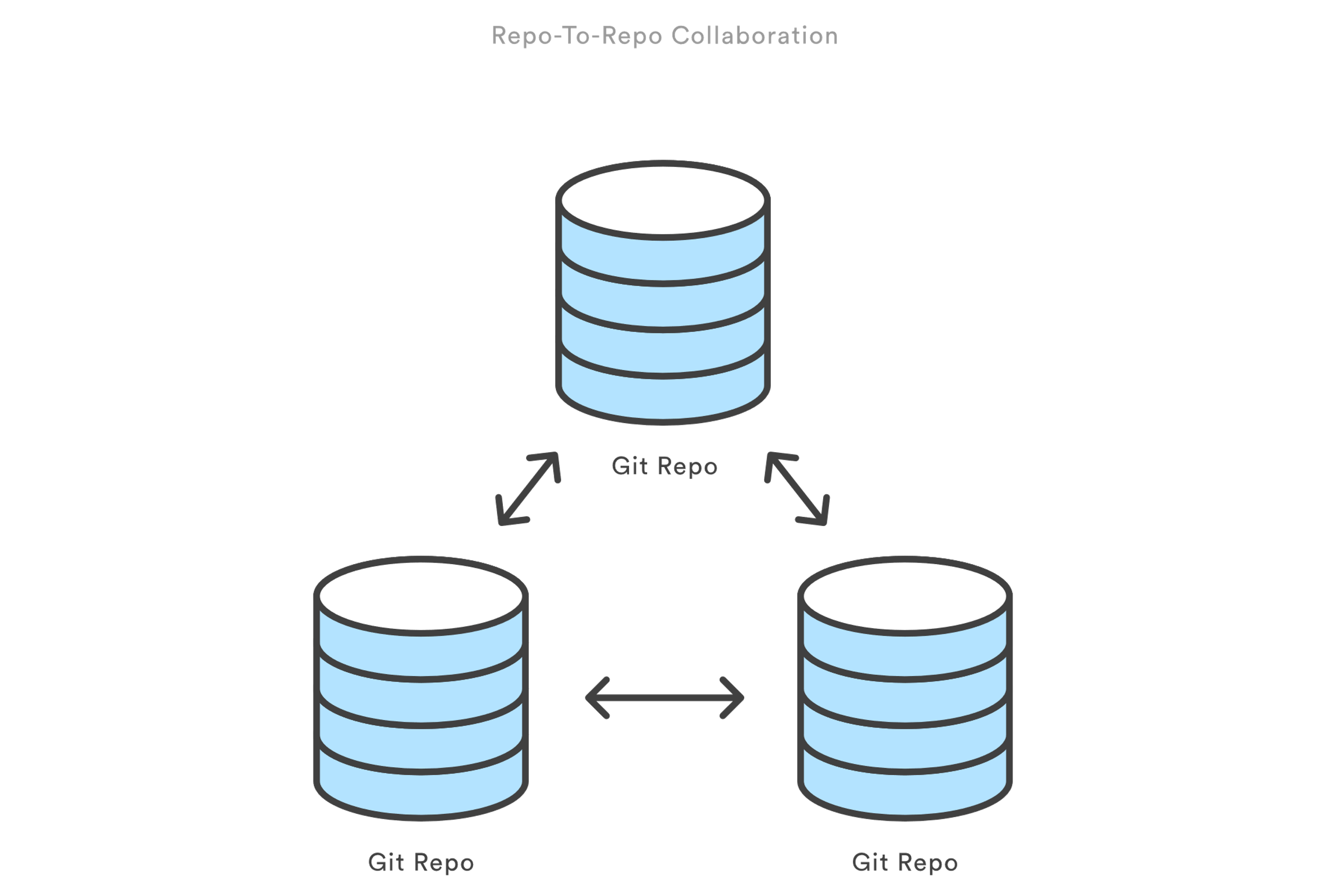
1. Branching and merging: allows you to create a test sample code, then make changes without affecting the main source code, then do testing and everything and if it works out then merge it to the main source code. This practice is adopted by all developers.
2. A lifetime history of all the changes made to code.
3. Traceability

**Types of version control**

There are mainly two types of version control, Centralized VCS (CVCS) and Distributed VCS (DVCS).

In CVCS all developers are connected to a central server, then make changes so it needs to remain connected to the server and do the thing.

In DVCS all the history and source code are copied to developer’s system, so it will be faster because load is on individual system unlike central one and can also work offline. Example is git.



**Source Code Management**

It is synchronous to VCS. It tracks all the changes made to the source code repository. Say two developers are working on same source code but on different features, and one’s code causes the crash in other guy’s code. So, to prevent this a code changes tracking system is required then. Also, SCM blocks the changes or gives warning to prevent this.

SCM brought version control safeguards to prevent loss of work due to conflict overwriting. These safeguards work by tracking changes from each individual developer and identifying areas of conflict and preventing overwrites. SCM will then communicate these points of conflict back to the developers so that they can safely review and address.

**Git Repository**

It is like storage in PC but online, companies save their source code there and make changes to it with benefits of VCS (branching, commit etc.). Also called repo sometimes.

**How to initialize a repository or clone it**

You can go to GitHub, GitLab or any VCS and create a repository there, the process is very simple. Then to bring it to your system you can clone it using command

git clone

Another way of creating a repo only in your system is

1. Select any folder in your directory
2. Git bash there
3. Type the code git init
4. This will create a repo in your local system

All repo’s, even if they are remote or local, work the same. What I mean is when you create a local one, then add file, then commit files do operations on them all are same, but not necessarily updated with each other.

Once you clone a repo then all commit history, all files all code will be there then if you work on master branch that is the main branch as long as you don’t push those changes will not be affected. It may be a bit confusing now but you will understand it as we move on main thing is all repo works same.

**How to reach the folder where you want to do git operations?**

Just right click over the directory you want go and git bash here.

**Adding, commit and pushing files to repo**

1. Put the file in the directory cloned or other way around, but this will not add it as a part of repo.
2. You need to write a code git add filename.extension in git bash.
3. Now you have added it but in git adding a file or doing modification to a file is a two-step process.
4. First is staging, which you did just now, then commit.
5. Before commit check status using code git status.
6. To commit use code git commit –m"message”.
7. In the message section provide a comment on why this commit or if you are working in company then Jira ticket number or as per company instruction, it is considered as a good practice because if you want to look back your comment will give you the reason or anything related to it.
8. After staging and committing, we push the code to the remote repo using command git push.
9. Then it will ask you for your password to git account then done. File added to repo.
10. You can check log using git log command.

**Upstream branch info**

When you create a repo in git then do the commit and push an upstream branch for your local branch is set as origin. You can also create different upstream branch using

git remote add <remote\_name> <remote\_repo\_url>

e.g git remote add master <https://github.com/austinjohny/codingQues.git>

So, when you check with code

git remote

you will get master and origin

**Creating, checkout and pushing branches**

1. To create a branch use code git branch branch\_name
2. This will create a local branch that is inside your local system only not the remote one.
3. To work on that branch, you need to checkout that branch, use code git checkout branch\_name.
4. Or you can do both the above steps in on step use code git checkout -b branch\_name. This will create a new branch and checkout it. This code will create a branch of current head.
5. Say you want to create a branch of any other branch then use code git checkout -b branch\_name existing\_branch.
6. Till this point the branch you created out of main or any other branch is present in local repo. We need to push it into remote repo that is GitHub. So, we use the code git push -u origin/upstream\_branch branch\_name. The other branch name is the upper branch. This origin or upstream\_branch is required.

**Git config**

1. Once you completed setting up your remote repo connection and upstream branch (by default it is origin but you can change it).
2. Then you add your details to git, so that when any commit is being done your name and email will be displayed there.
3. Git stores configuration options in three separate files, which lets you scope options to individual repositories (local), user (Global), or the entire system (system):

* Local: /.git/config – Repository-specific settings.
* Global: /.gitconfig – User-specific settings. This is where options set with the --global flag are stored.
* System: $(prefix)/etc/gitconfig – System-wide settings.

1. Use code git config –global user.name name and git config –global user.email email

**Bare repository**

To be filled

**Git init**

The git init command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new, empty repository. Most other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

Executing git init creates a .git subdirectory in the current working directory, which contains all of the necessary Git metadata for the new repository. This metadata includes subdirectories for objects, refs, and template files. A HEAD file is also created which points to the currently checked out commit.

Aside from the .git directory, in the root directory of the project, an existing project remains unaltered (unlike SVN, Git doesn't require a .git subdirectory in every subdirectory).

By default, git init will initialize the Git configuration to the .git subdirectory path. The subdirectory path can be modified and customized if you would like it to live elsewhere. You can set the $GIT\_DIR environment variable to a custom path and git init will initialize the Git configuration files there. Additionally, you can pass the --separate-git-dir argument for the same result. A common use case for a separate .git subdirectory is to keep your system configuration "dotfiles" (.bashrc, .vimrc, etc.) in the home directory while keeping the .git folder elsewhere.

**Git clone**

1. It is used to clone a remote repo to local machine.

2. As a convenience, cloning automatically creates a remote connection called "origin" pointing back to the original repository.

3. Shallow clone: We use code git clone -depth=1 repo\_url. It only copies the history of commits specified by the option depth=1. In this example above only the most recent commit (i.e only one commit) is included in new cloned repo. This is used when commit history is very large and we only need specific commits.

4. Clone branch: Use code git clone -branch branch\_name, then only that branch is cloned.

Git alias

1. These are shortcut to enable faster code execution.
2. For example, we can assign a different keyword for commit as co. So instead of typing git commit we can use git co.
3. Code is written like
   1. $ git config --global alias.co checkout
   2. $ git config --global alias.br branch
   3. $ git config --global alias.ci commit
   4. $ git config --global alias.st status
4. The commit and other command are still available and we can also use the alias.
5. These aliases were created with the --global flag which means they will be stored in Git's global operating system level configuration file. On Linux systems, the global config file is located in the User home directory at /.gitconfig.