innovate achieve lead

BITS Pilani

Pilani Campus

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

By Prof A R Rahman

CSIS Group WILP

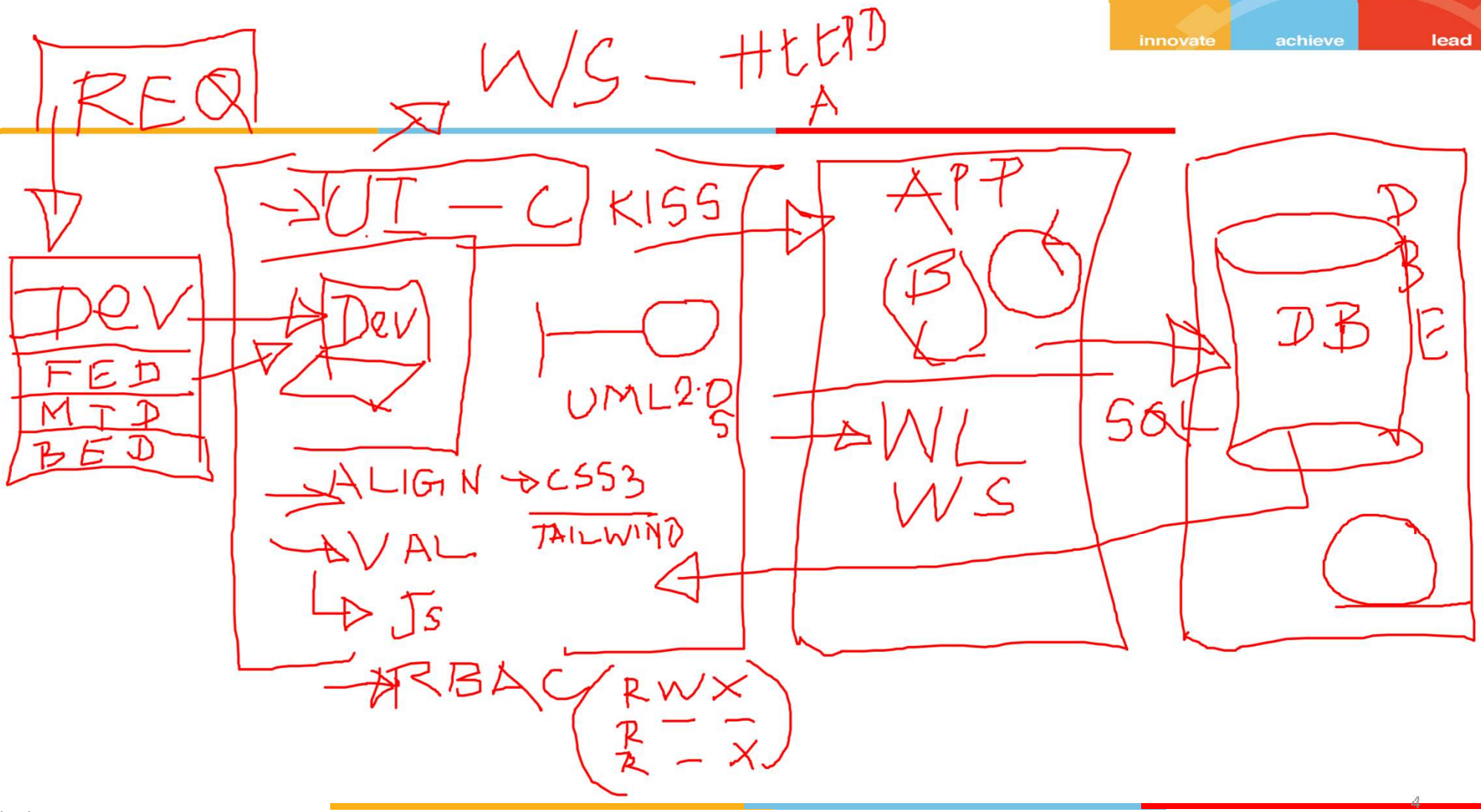
innovate achieve lead

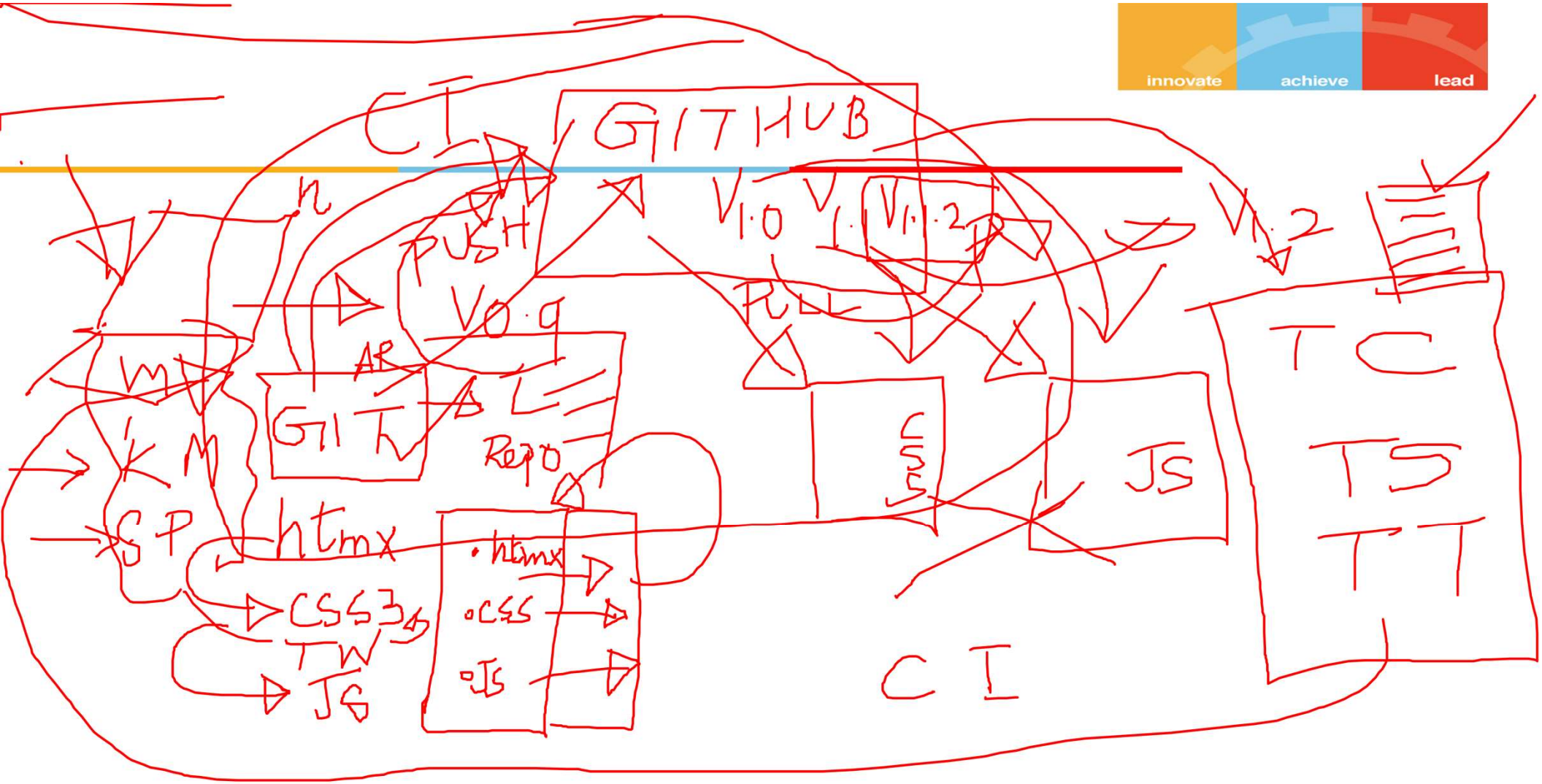
BITS Pilani
Pilani Campus

Course Name: Introduction to DevOps
Course Code : CSI ZG514/SE ZG514

By Prof A R Rahman
CSIS Group WILP

CS - 5 Source Code Management

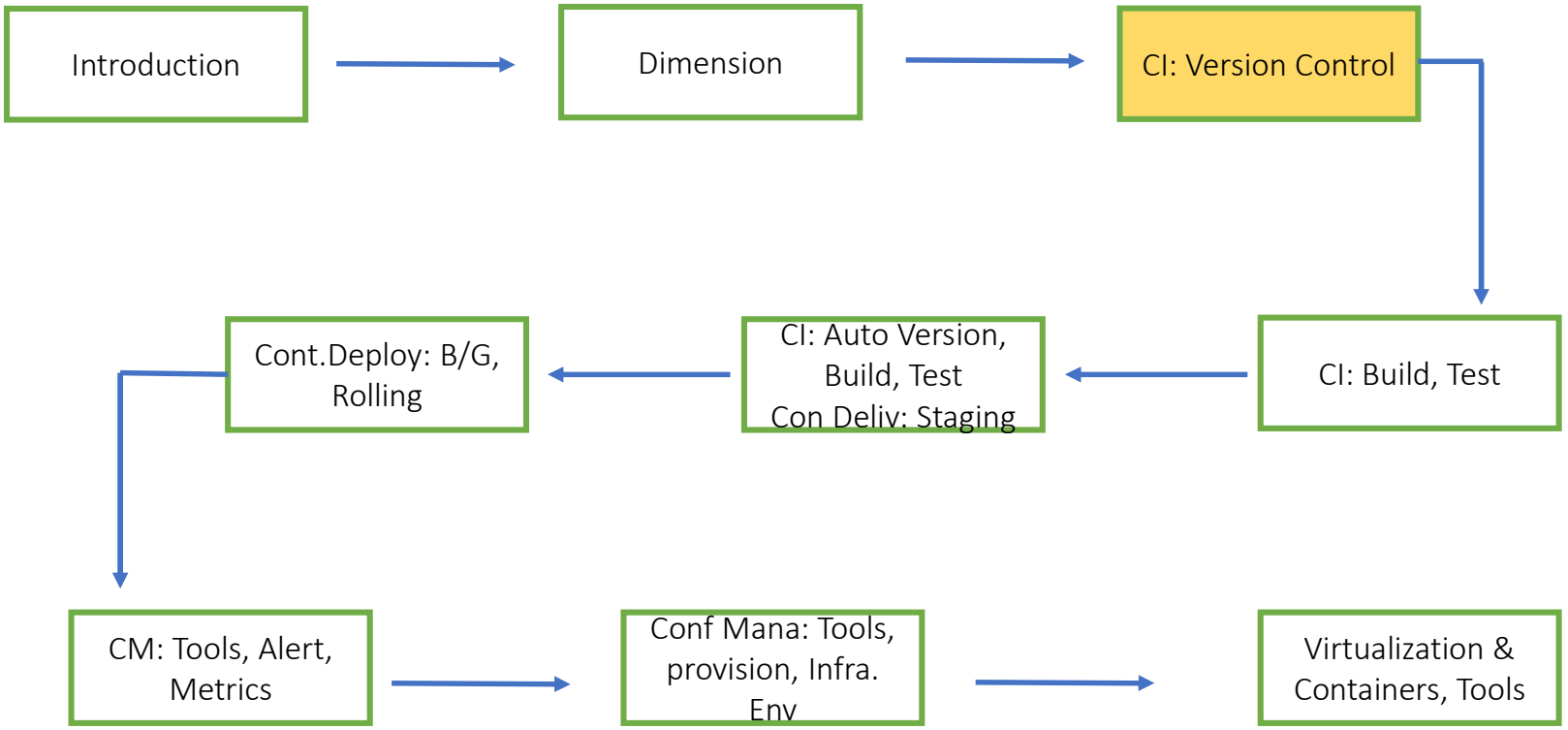

REQ
WS – HTTP
UI – C
KISS
APP
DEV
FED
MID
BED
Dev
UML2.0
DB
ALIGN → CSS3
TAILWIND
WL
WS
SQL
VAL
JS
RBAC
RWX
R – –
R – X


CI
GITHUB
PUSH
V1.0
V1.1
V1.1.2
V0.9
PULL
V1.2
GIT
Repo
JS
TC
TS
TT
SP
htmx
CSS3
TW
JS
CI

Coverage

- Version control system and its types
- Introduction to GIT
- GIT Basics commands (Creating Repositories, clone, push, commit, review)
- Git workflows- Feature workflow, Master workflow, Centralized workflow
- Feature branching
- Managing Conflicts
- Tagging and Merging
- Best Practices- clean code

DevOps


Introduction
Dimension
CI: Version Control
CI: Build, Test
CI: Auto Version, Build, Test
Con Deliv: Staging
Cont.Deploy: B/G, Rolling
CM: Tools, Alert, Metrics
Conf Mana: Tools, provision, Infra. Env
Virtualization & Containers, Tools

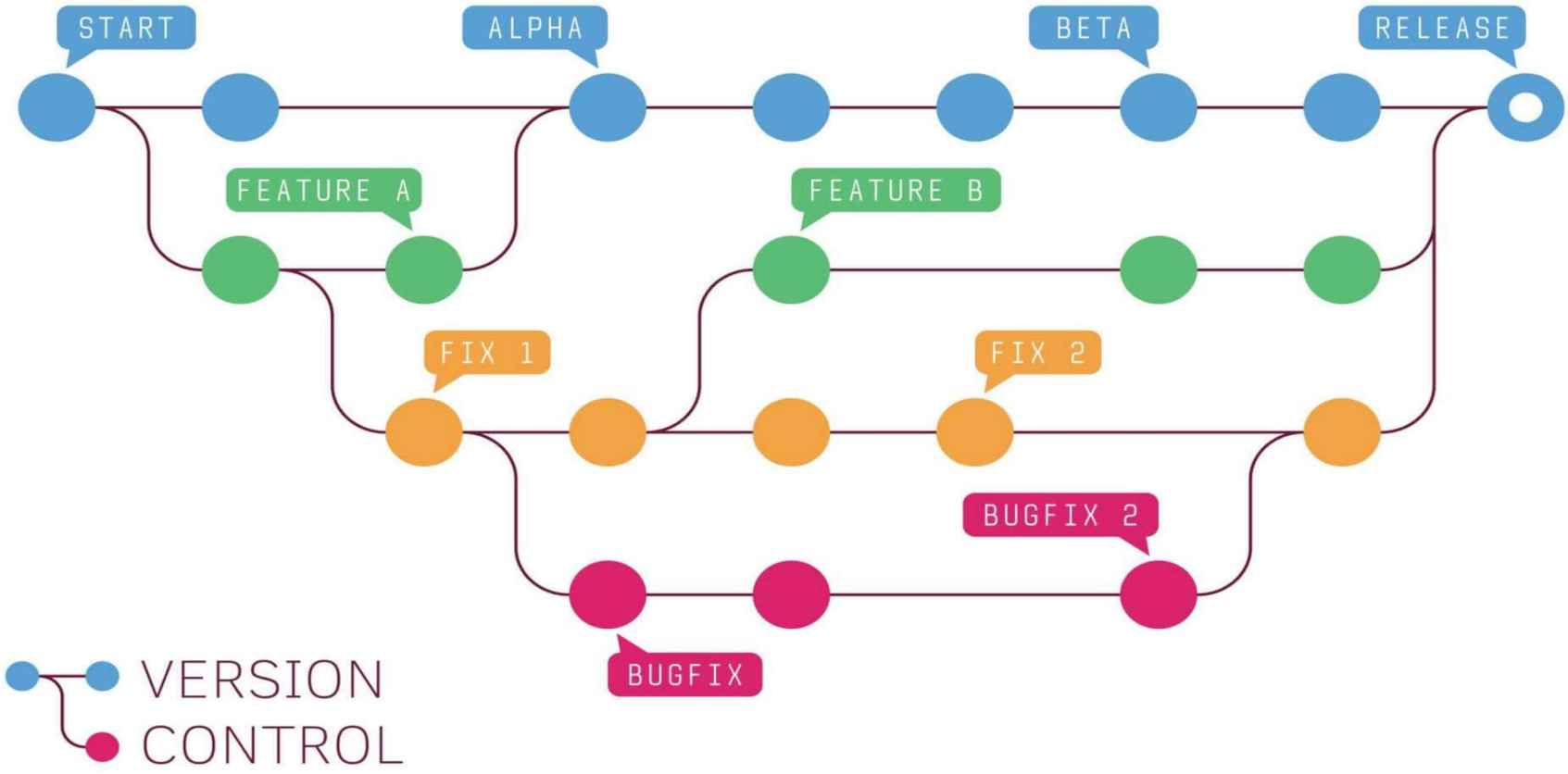
Version Control

- Source code is precious asset whose value must be protected
- Multiple people to simultaneously work on a single project.
- Each person edits his or her own copy of the files
- How to share the changes with the rest of the team?.

Version Control

- Make sure temporary or partial edits by one person do not interfere with another person's work.
- Need software tools that help a team manage changes to source code over time.
- It should keep track of every modification to the code in a special kind of database.
- If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Version Control



Version Control System

Version control is the management of changes to documents, computer programs, large websites, and other collections of information.

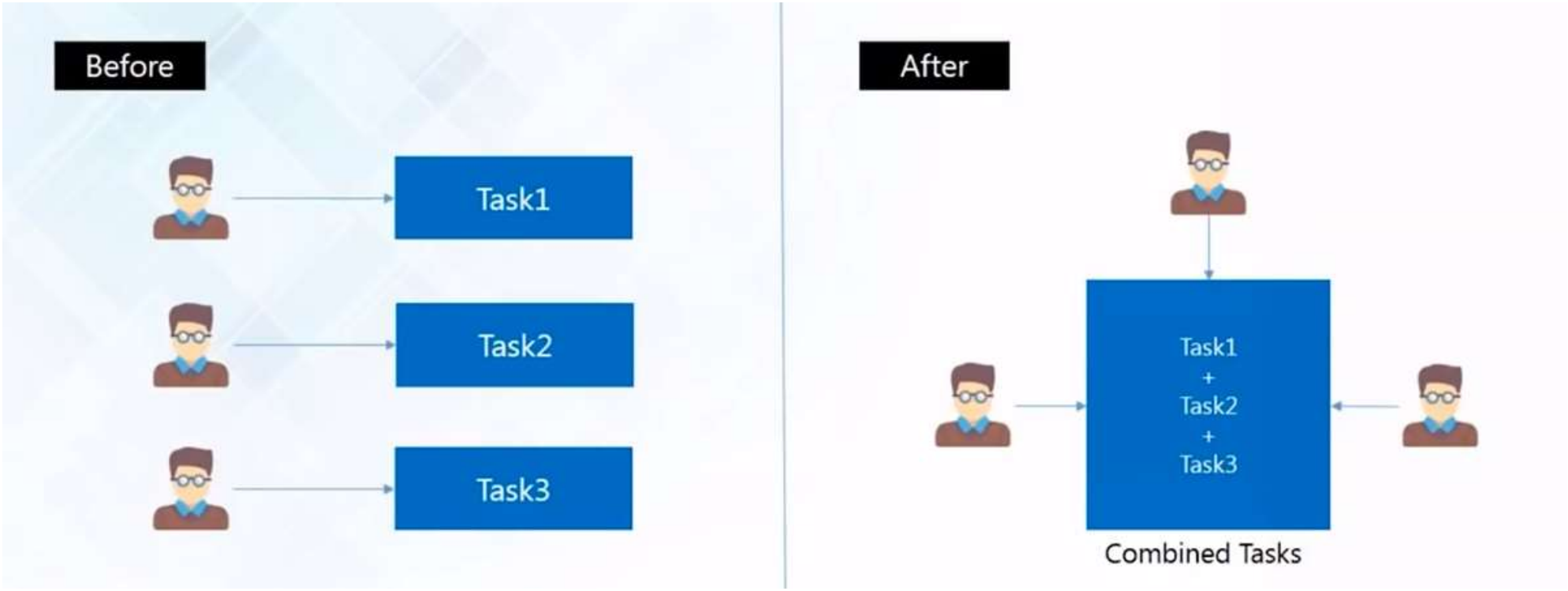
These changes are usually named “versions”

Version Control System

- Version control, also known as source control, is the practice of tracking and managing changes to software code.
  - Tools that help software teams manage changes to source code over time.
  - Accelerate development environments by helping software teams work faster and smarter.
  - Help teams to reduce development time and increase successful deployments.

a.k.a. Source Code Management (SCM) and Revision Control System (RCM)

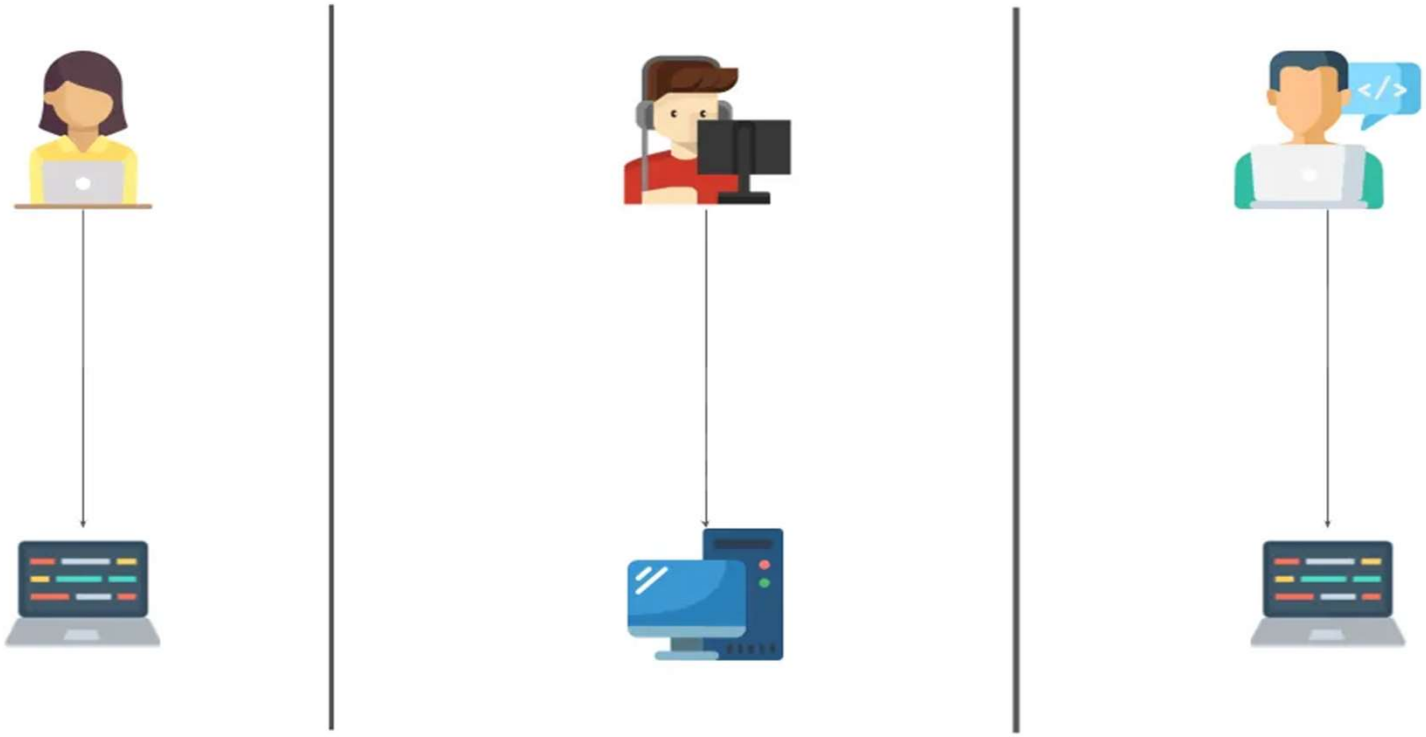
Collaboration



Types of Version Control System

- 1.Local Version Control Systems:
- Think of this as a VCS but without a remote repo.
- You manage and version all the files only within your local system.
- There is no remote server in this scenario.
- All the changes are recorded in a local database.
- As we can see in the image below, every developer has their own computers and are not sharing anything.

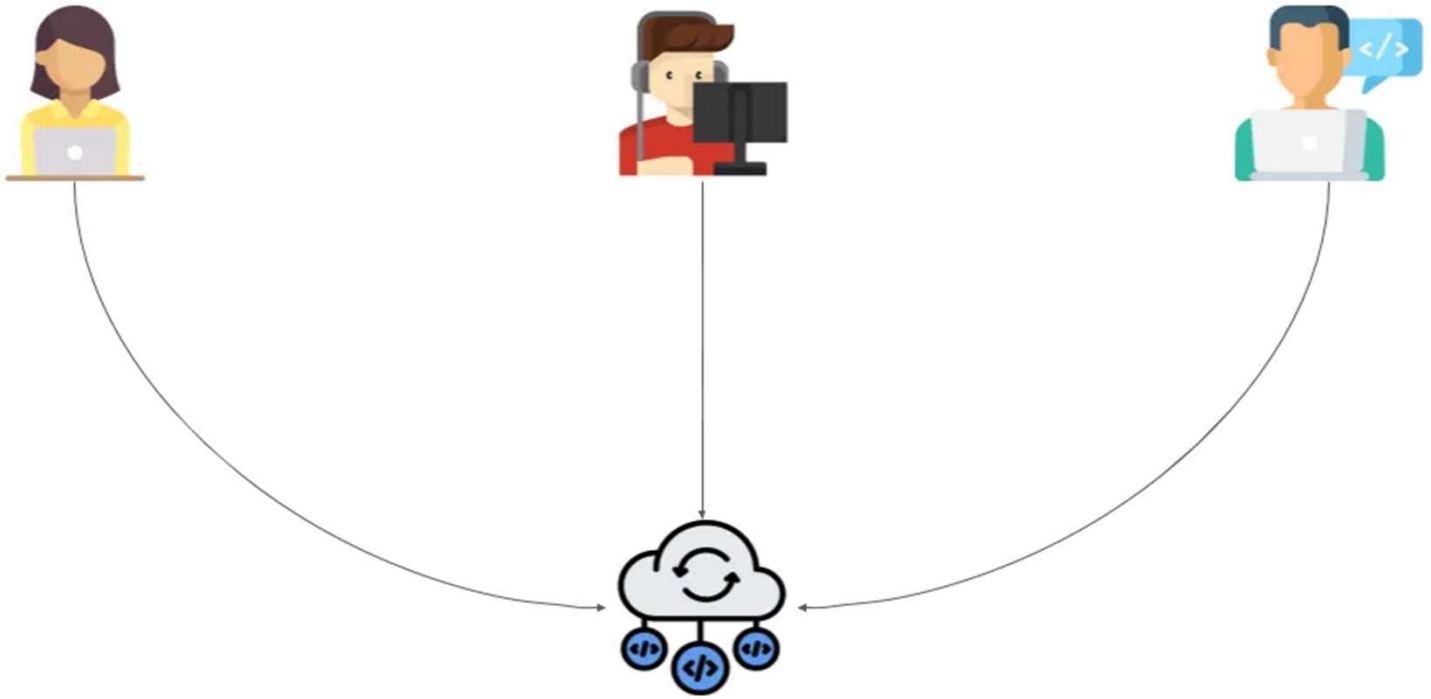
Types of Version Control System



Types of Version Control System

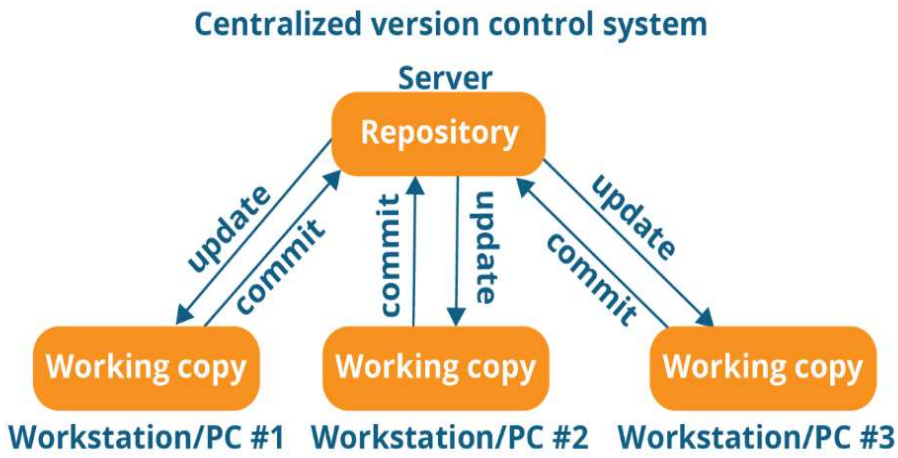
- 2. Centralized Version Control Systems:
- Here, there's a central repo shared with all the developers, and everyone gets their own working copy.
- Whenever you commit, the changes get reflected directly in the repo.
- Unlike distributed systems, developers directly commit to the remote.
- Which means they may affect files knowingly or unknowingly.

Types of Version Control System



Version Control - Centralized

- single “central” copy of project on a server, and programmers will “commit” their changes to this central copy
- “Committing” a change simply means recording the change in the central system.
- Other programmers can then see this change. They can also pull down the change, and the version control tool will automatically update the contents of any files that were changed.



Benefits:

- Centralized systems are typically easier to understand and use
- Can grant access level control on directory
- performs better with binary files

Version Control - Centralized

- CVS (Concurrent Versions System)
  - For Many years, CVS was the best known and most popular version control system in the world, mainly because it was the only free VCS
  - CVS brought a number of innovations both to versioning and to the software development process.

  - Despite its innovations, CVS has many problems
    - Branching in CVS involves copying every file into a new copy of the repository.
    - This can take a long time and use a lot of disk space if you have a big repository
    - Merging from one branch into another can give lots of phantom conflicts and does not automatically merge newly added files from one branch into another.
    - Tagging in CVS involves touching every file in the repository
    - Check-ins to CVS are not atomic

Version Control - Centralized

- SVN(Subversion)
  - Fixes many of CVS problems, and in general can be used as a superior replacement to CVS in any situation.
  - Retains essentially the same command structure as CVS.
  - allows to mount a remote repository to a specified directory in your repository
  - Revision numbers apply globally to the repository rather than to individual files.
  - Branching and tagging in Subversion are also much improved. Instead of updating each individual file, Subversion leverages the speed and simplicity of its copy-on-write repository

Version Control - Centralized

- Allows to keep a local copy of the version of every file as it existed when last checked it out from the central repository.

- Many operations can be performed locally, making them much faster than in CVS.

- Limitations
- You can only commit changes while online
- While server operations are atomic, client-side operations are not.
- If a client-side update is interrupted, the working copy can end up in an inconsistent state.
- Revision numbers are unique in a given repository, but not globally unique across different repositories.
- Commercial Centralized Version control systems: Perforce, AccuRev, Microsoft's Team Foundation Server (TFS)

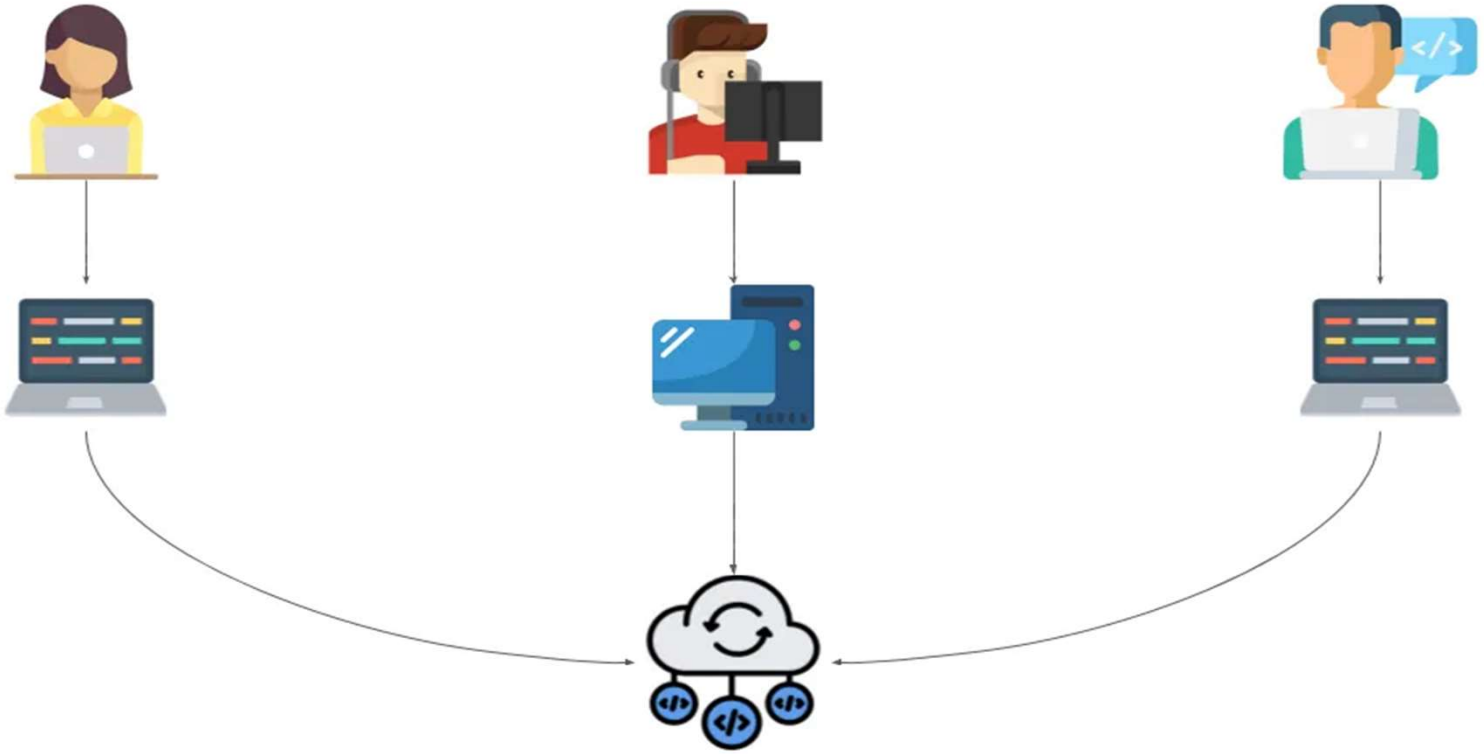
SVN typical workflow

1. svn up—Get the most recent revision.
2. Write some code.
3. svn up—Merge my changes with any new updates to the central repository and fix any conflicts.
4. Run the commit build locally.
5. svn ci—Check my changes, including my merge, into version control.

Types of Version Control System

- 3. Distributed Version Control Systems:
- In distributed systems, there is a local copy of the repo for every developer on their computers.
- They can make whatever changes they want and commit without affecting the remote repo.
- They first commit in their local repo and then push the changes to the remote repo.
- This is the type used majorly today.

Types of Version Control System



Centralized Version Control Systems

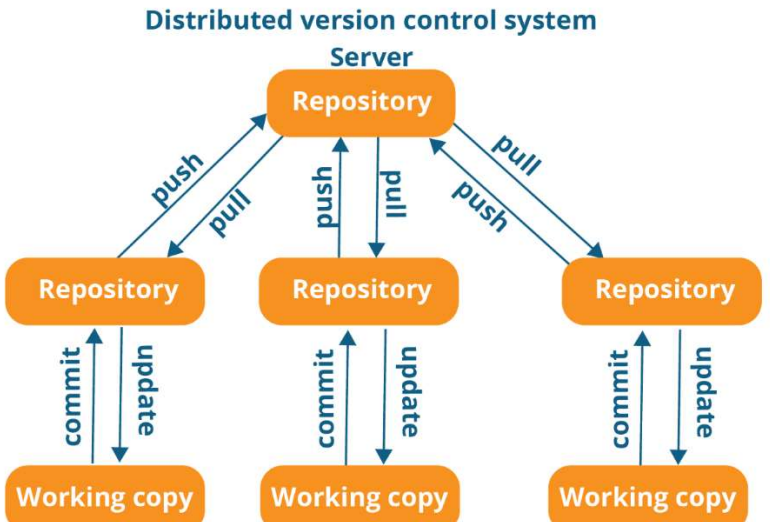
- In the Centralized VCS, every commit was directly happening to the, say, remote repo.
- But in distributed VCS, it first happens in the developer's local repo, and then they send those changes to the remote.
- *Remote repo is a git repo on a server hosted by any such providers like GitHub, Bitbucket, etc.*
- When you commit changes in a centralized system, the changes are directly visible to others as you commit directly in the remote.

Distributed Version Control Systems

- But in distributed, you first commit locally, and it will not allow others to see until you *push* it to the remote.
- Likewise, when others commit a change in a centralized system, you can directly see them, but you need to *pull* changes in the local repo in distributed systems after they push it.
- So, distributed systems can also be beneficial in having only necessary/updated/completed code with everyone rather than having even experimental code like in Centralized systems.

Distributed Version Control Systems

- Every developer “clones” a copy of a repository and has the full history of the project on their own hard drive. This copy (or “clone”) has all of the metadata of the original.
- Modern systems also compress the files to use even less space
- The act of getting new changes from a repository is usually called “pulling,” and the act of moving your own changes to a repository is called “pushing”



Benefits:

- Performance of distributed systems is better
- Branching and merging is much easier
- With a distributed system, you don’t need to be connected to the network all the time (complete code repository is stored locally on PC)

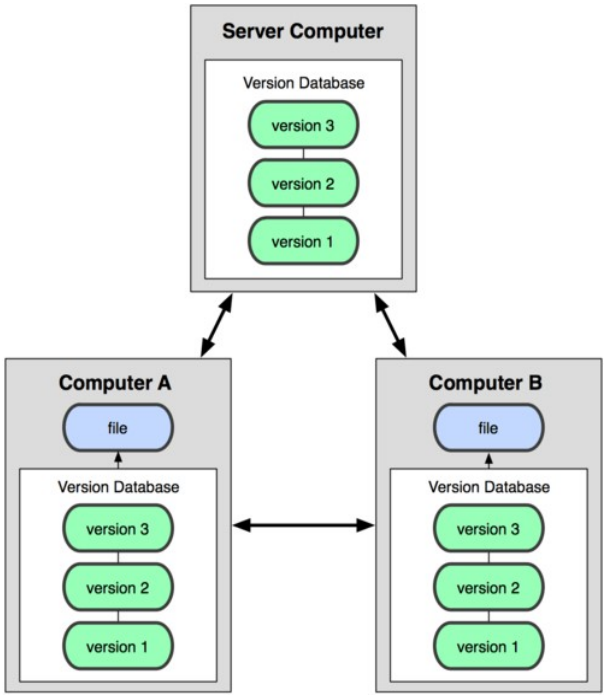
Distributed Version Control Systems

A “remote master” repository is typically hosted on a server.

Clones of the remote repository are maintained on each user’s computer

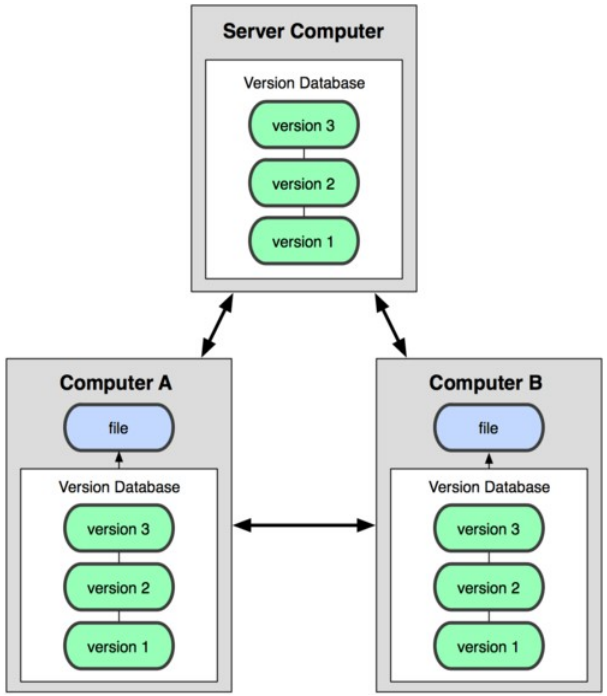
If the server goes down, users can continue sharing code (provided they can connect over a network) by synch’ing to each others’ repositories

If the network is unavailable, users can continue reading & writing to their own local repository – synching with others later

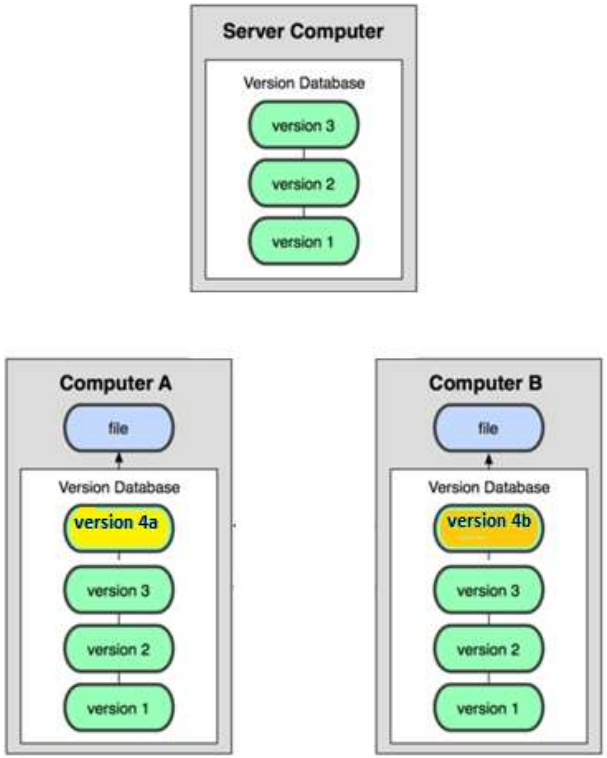


Git and Mercurial use this model

Merging in a DVCS, part 1

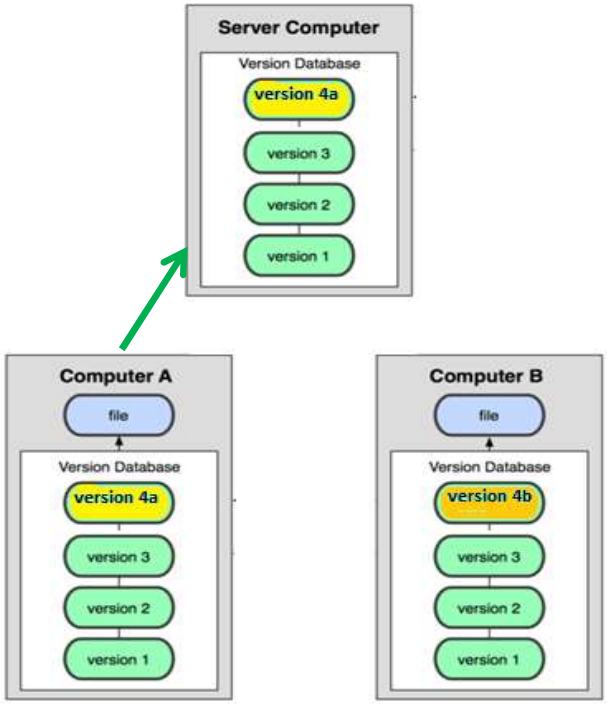


1. Initially, both users' repositories are synched to the remote master; All files are identical.

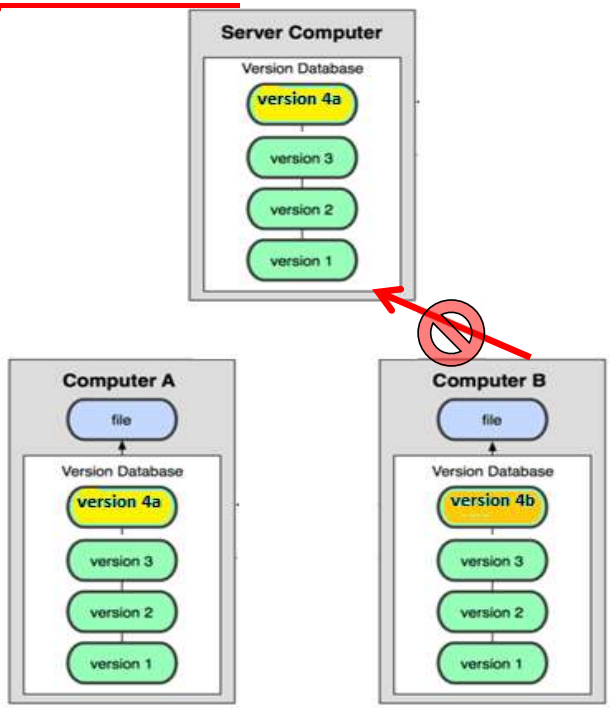


2. Both users edit the same file, and their local repositories reflect their changes

Merging in a DVCS, part 2



3. User A "pushes" her repository to the remote master to synch; this succeeds.

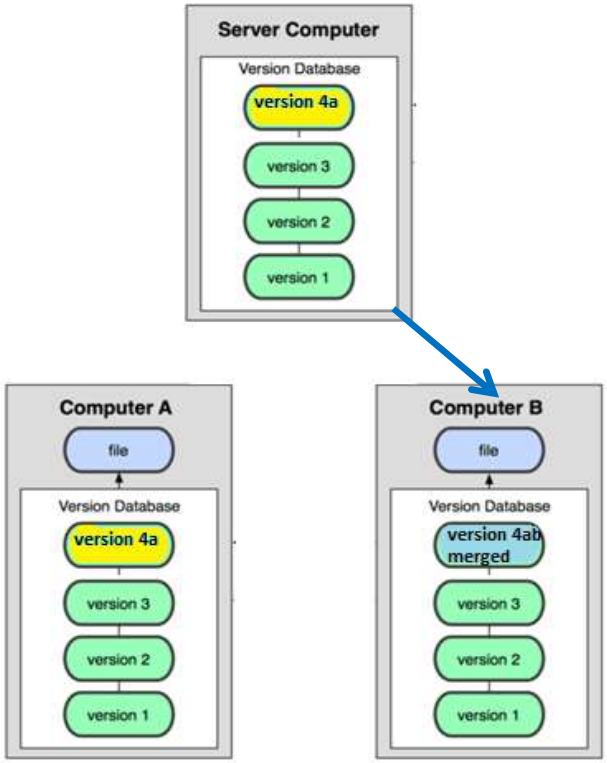


4. User B attempts to "push" his repository to the remote to synch; this fails due to User A's earlier push. Remote does not change.

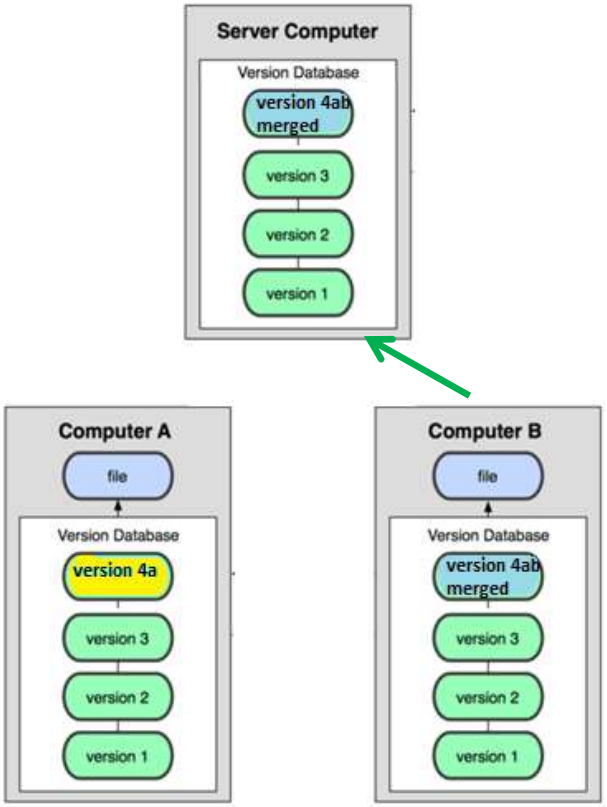
Git and Mercurial use this model

Merging in a DVCS, part 3

Git and Mercurial use this model

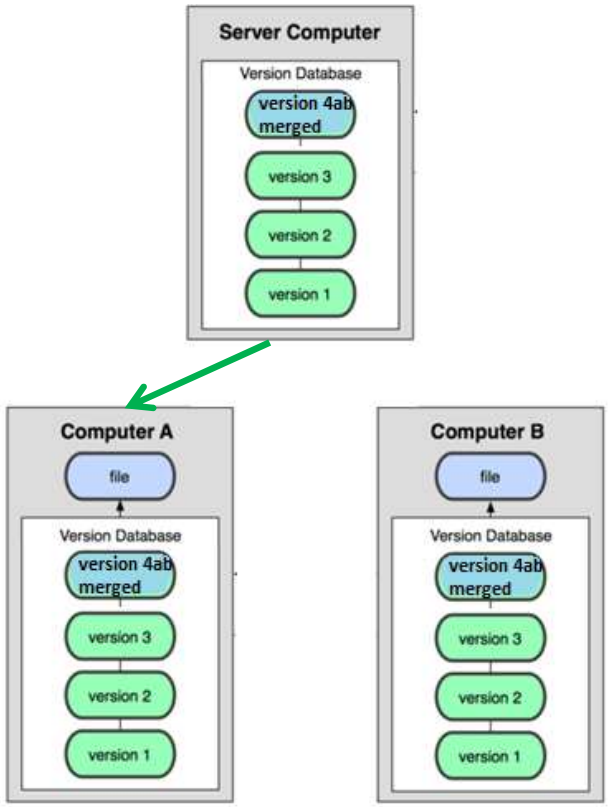


5. User B "pulls" User A's updates from the remote master, merging A and B together. **Merging is automatic with some exceptions.**



6. User B "pushes" his repository to the remote to synch; this now succeeds

Merging in a DVCS, part 4



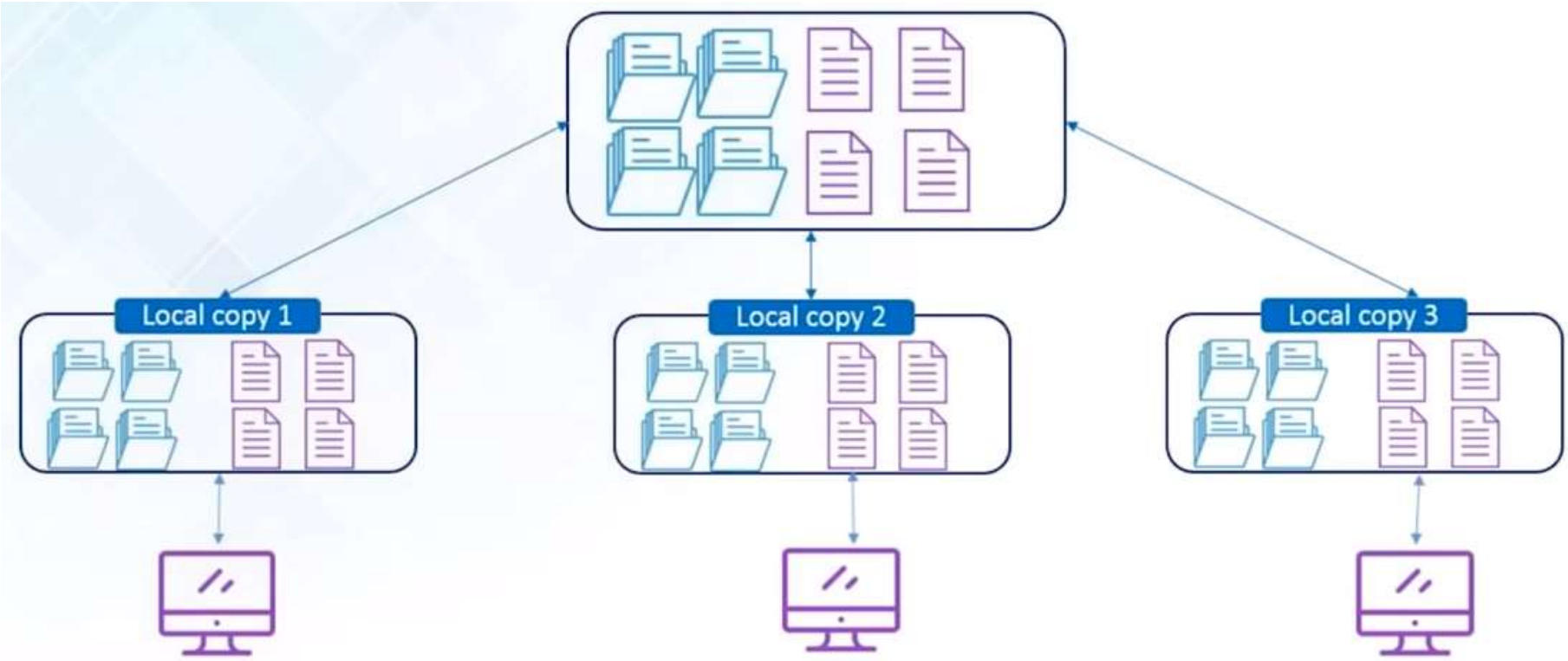
7. User B "pulls" from the remote, and everyone is in synch again.

System Failures

- In a Local VCS, say, if your hard disk gets corrupted or you lose the file, you lose everything related to it, including all the changes.
- Similarly, in the Centralized VCS, if you lose the repo, everything is just gone.
- But in a Distributed VCS, even if someone's local repo is corrupted or the remote doesn't work or is deleted, or anything happens, there is a copy available with other developers.
- So, you can take it from them, push to remote, and everything is back on track, just like before.
- There is no problem of single-point failure in Distributed VCS.
- You always have your code safe with fellow developers.

Backup

In any case if your central server crashes, a backup is always available in your local servers



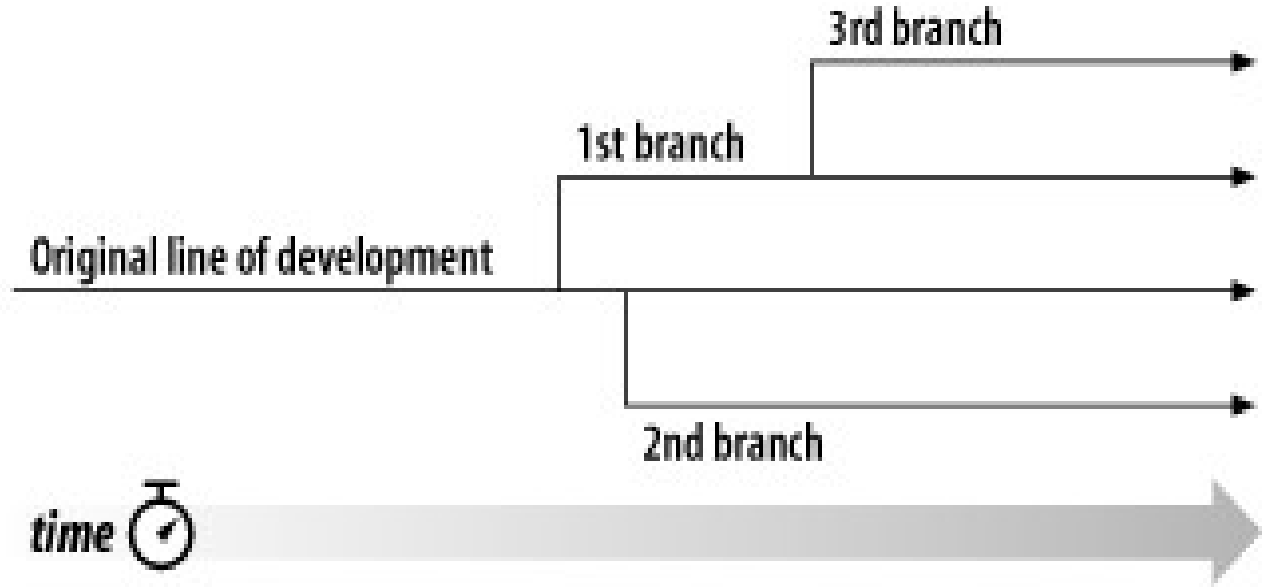
Analyze

When you change version –

- VCS provides with proper description
- What exactly was changed
- When it was changed and by whom

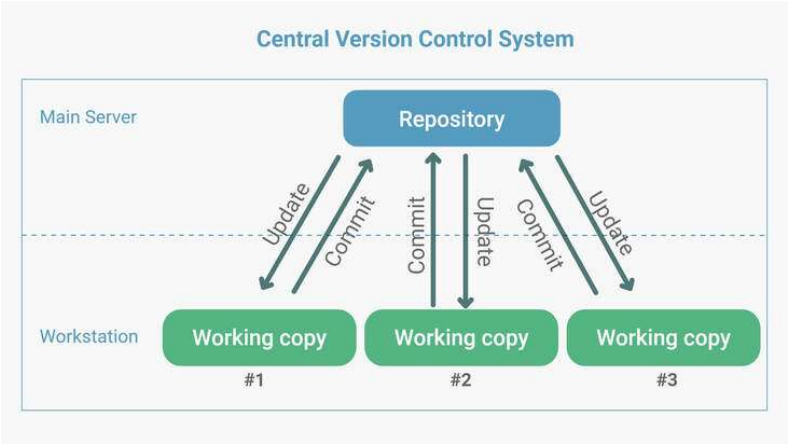
You can analyze how your project evolved between versions

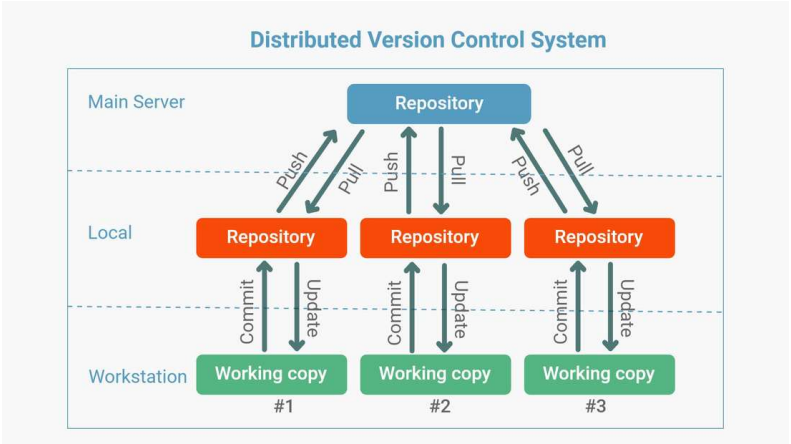
Repositories can also manage and track differences between parallel revisions of a document



Typically, a software product will undergo parallel development on different **branches** (e.g. for new/future features) while primary development takes place on some main branch

Version Control Systems Types





- Location: central server
- Need to be online, not locally available
- Need to connect to the central server for every command
- Single point of failure
- Rely on the backups

- Location: Distributed in nature with local copy of entire history
- not necessary to be online
- Only required during pushing or pulling the code
- Better performance
- Local backup, if remote crashes
- Remote backup, if local crashes

Version Control – Why should use?

- Any project that has more than one developer maintaining source code files should absolutely use a Version Control
- Collaboration
  - team is able to work absolutely freely — on any file at any time.
  - later allow to merge all the changes into a common version
  - Always latest version of a file or the whole project is in a common, central place
- Rollback and undo changes to source code
  - keeps a history of changes and the state of the source code throughout a project's history
  - possibility to "undo" or rollback a source code project to a last well-known state
  - If any bug, the code can be quickly reverted to a known stable version
- Understanding What Happened
  - Short description/Tagging on new version helps to understand what was changed
  - can see what exactly was changed in the file's content
  - helps to understand how your project evolved between versions

But why to manage changes ?

Answering Who, Why, What, When, for Whom ….

Some benefits of such VCS

- Concurrency (e.g. Google Doc vs working on same docx file through email)
- Preserving collaborative history
- Rollback and undo changes
- Offsite source code backup
- Branching and merging
- Traceability
- Conflict resolution

Version Control Systems Goals

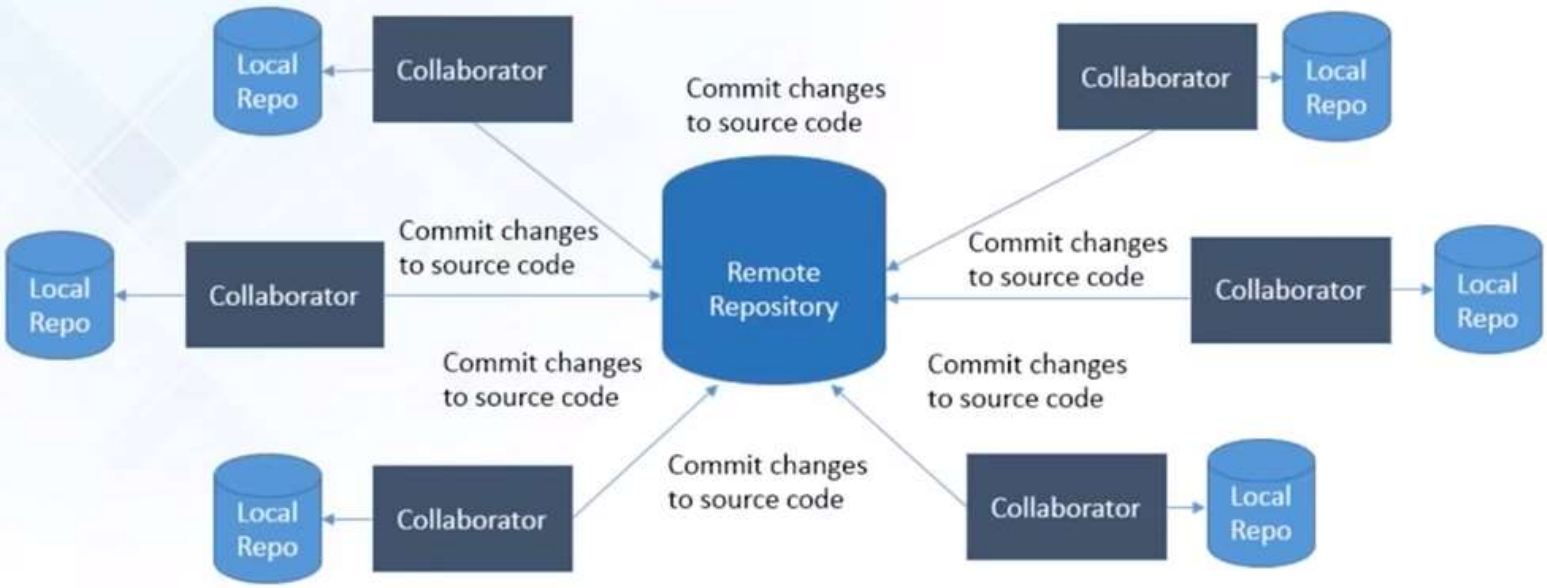
Some goals of version control system:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)
- Data integrity

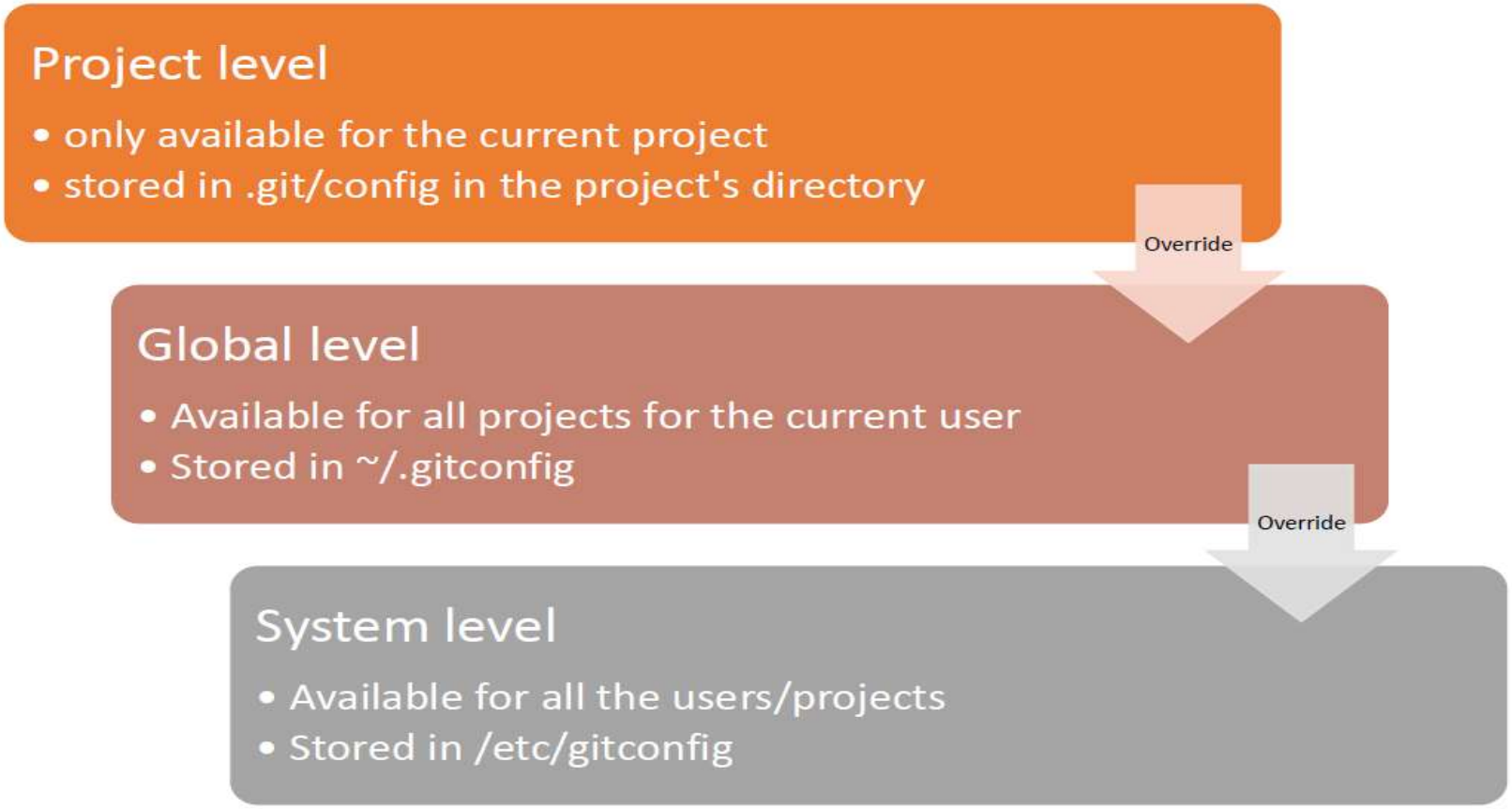
Version Control System Tools

GIT

Git is a Distributed Version Control tool that supports distributed non-linear workflows by providing data assurance for developing quality software.

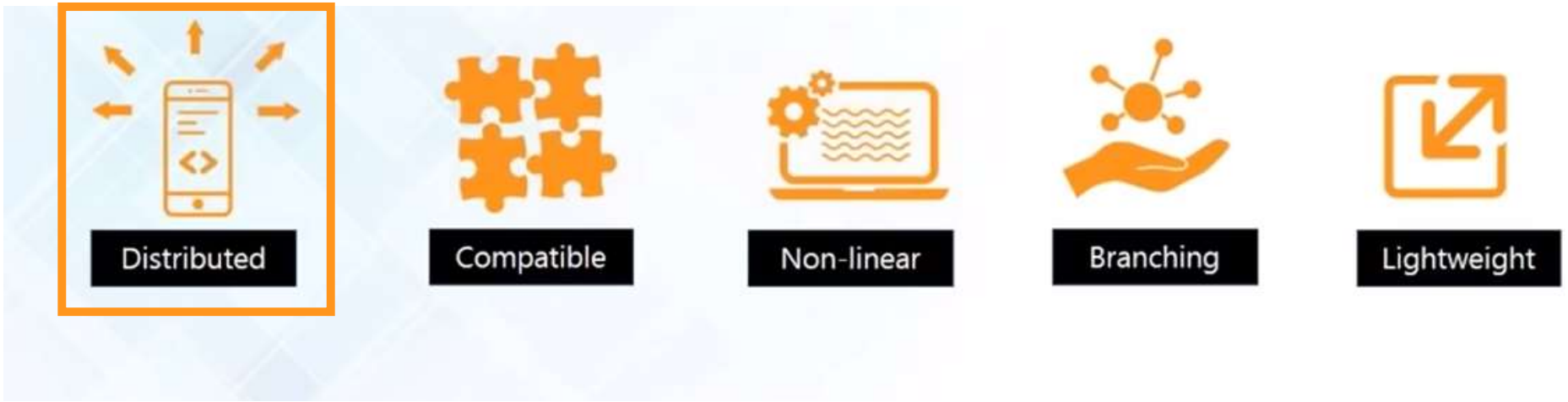


Git -- Configuration



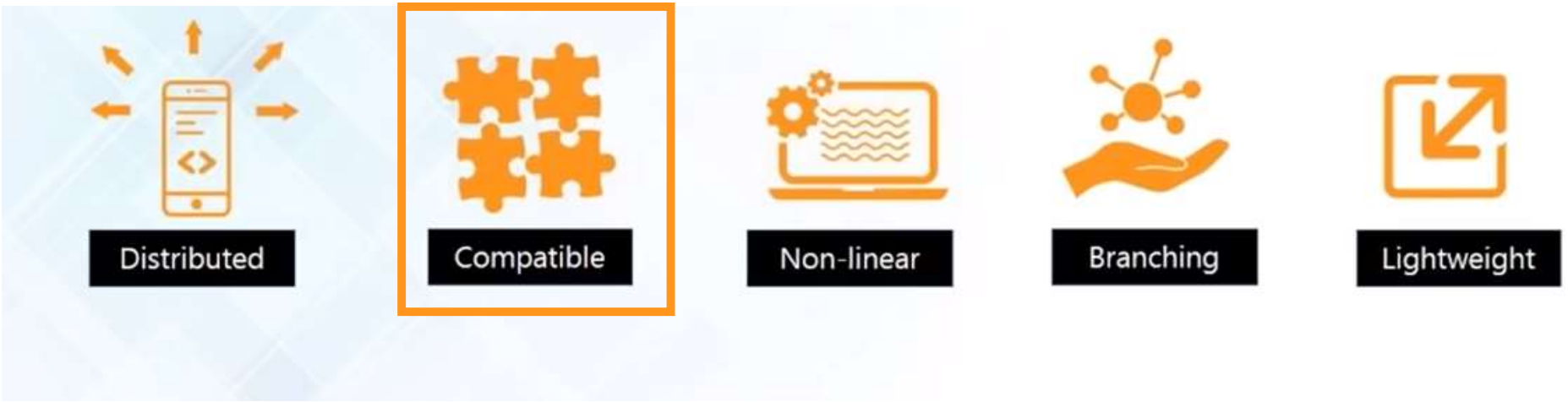
Git Features

Git Features



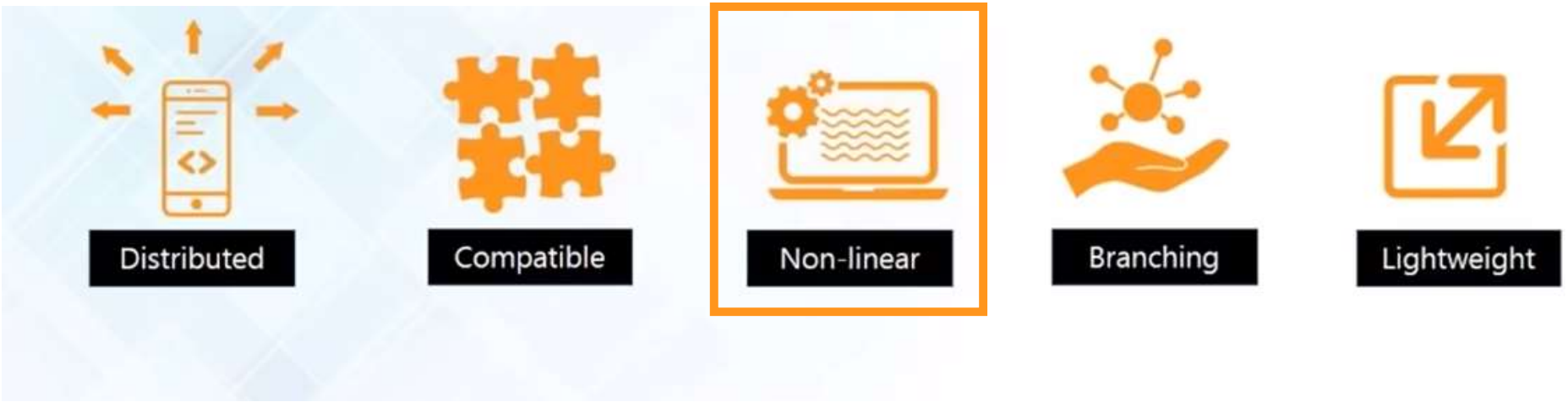
- Allows distributed development of code
- Every developer has a local copy of the entire development history and changes are copied from one repository to another

Git Features



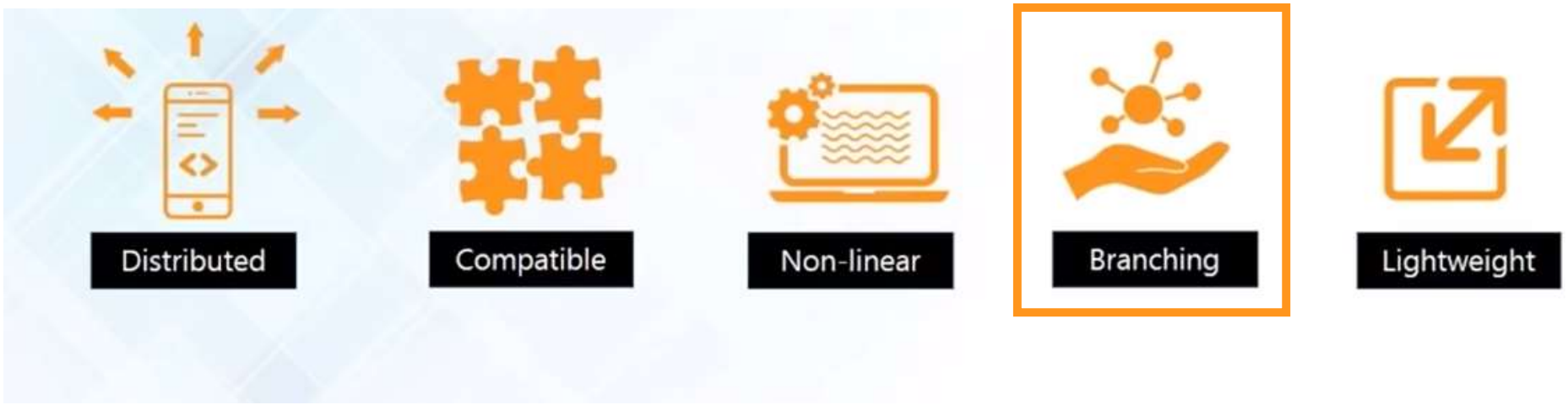
- Compatible with current systems and protocols
- SVN and SVK repositories can be directly accessed using Git-SVN

Git Features



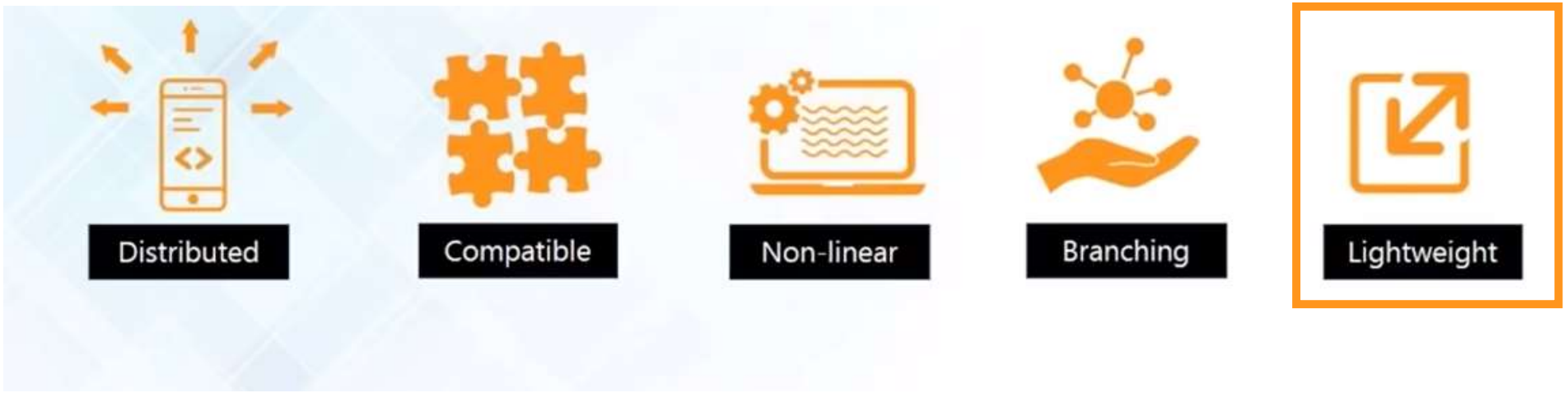
- Supports non-linear development of software
- Includes various techniques to navigate and visualize non-linear development history

Git Features



- It takes only few seconds to create and merge branches
- Master branch always contains production quality codec

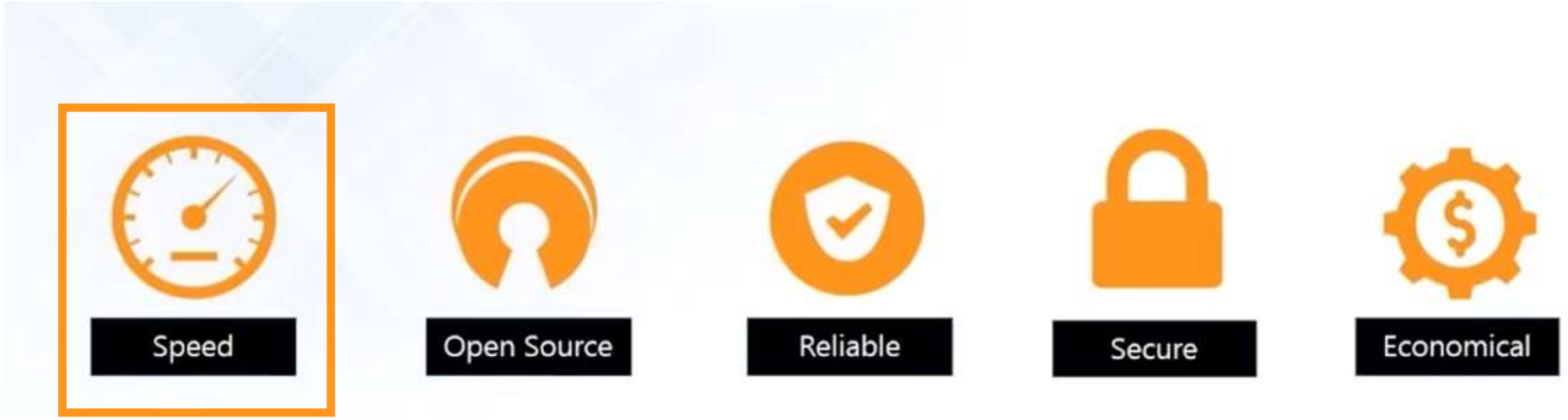
Git Features



- Uses lossless compression technique to compress data on the client's side

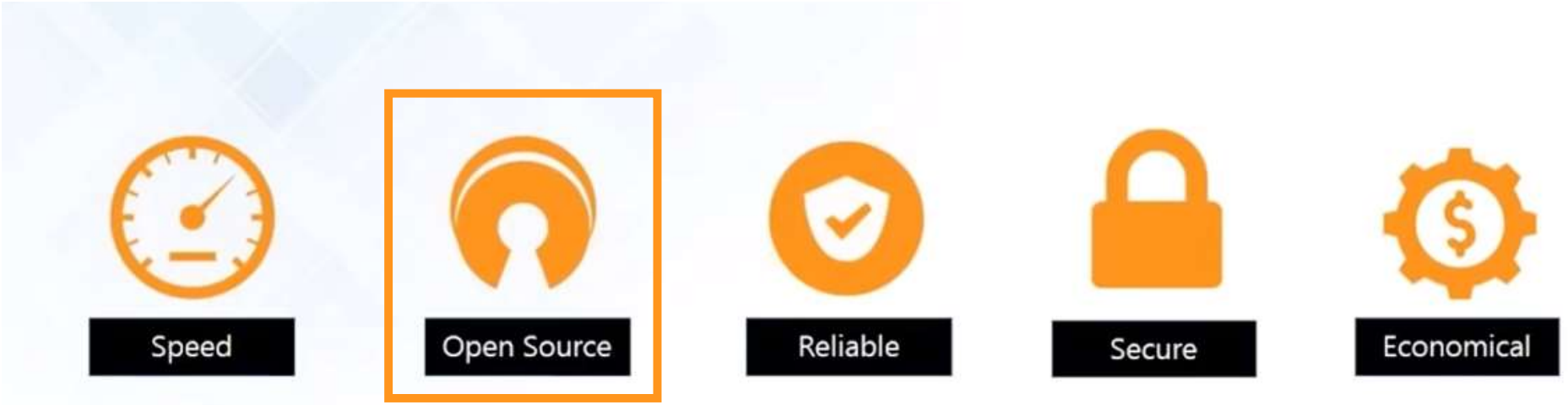
Git Features

- Fetching data from local repository is 100 times faster than remote repository
- Git is one order of magnitude faster than other VCS tools



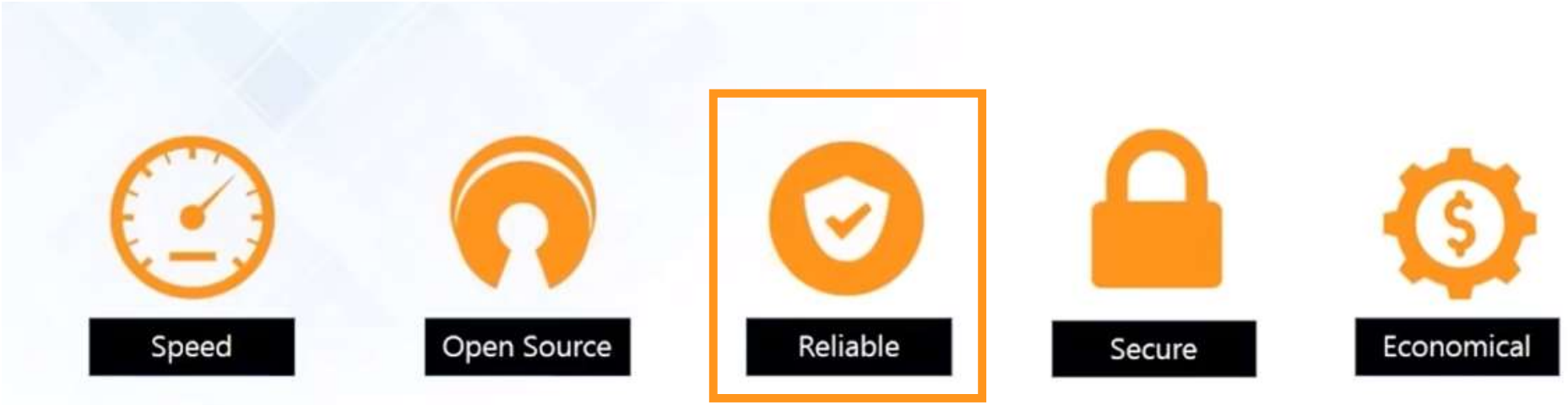
Git Features

- You can modify its source code according to your needs



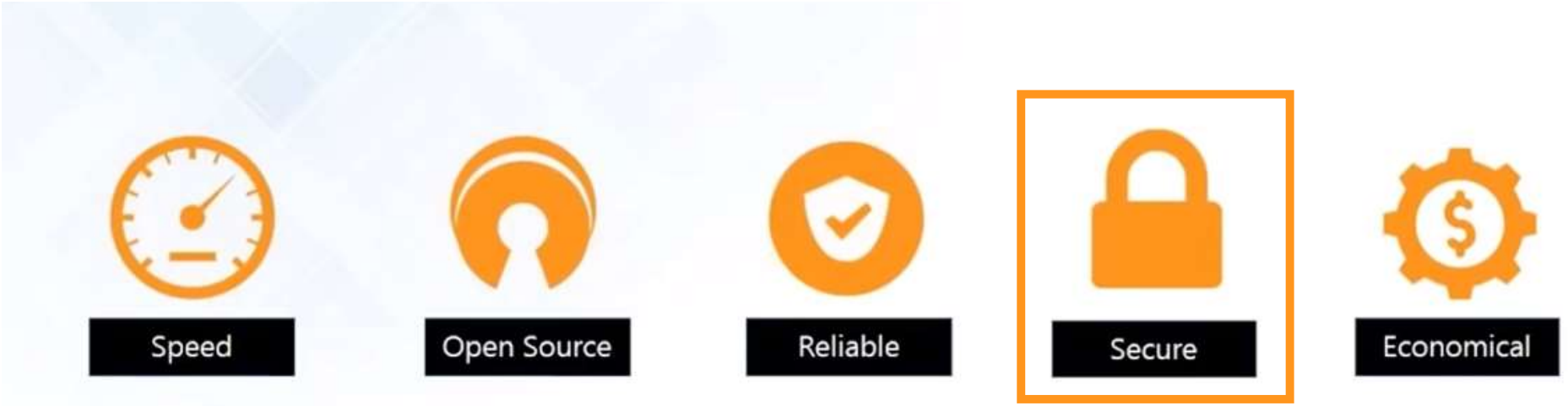
Git Features

- On events of system crash, the lost data can be easily recovered from any local repositories of the collaborators



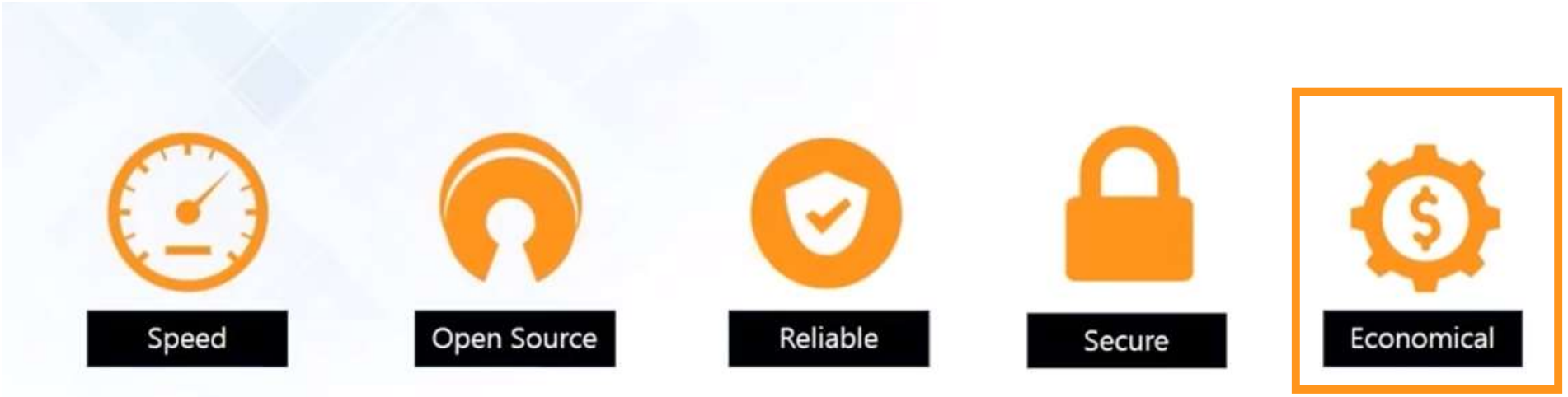
Git Features

- Uses SHA1 to name and identify objects
- Every file and commit is checksummed and is retrieved by its checksum at time of checkout



Git Features

- Released under GPL's license (It is for free)
- All heavy lifting is done on client-side, hence a lot of money can be saved on costly servers



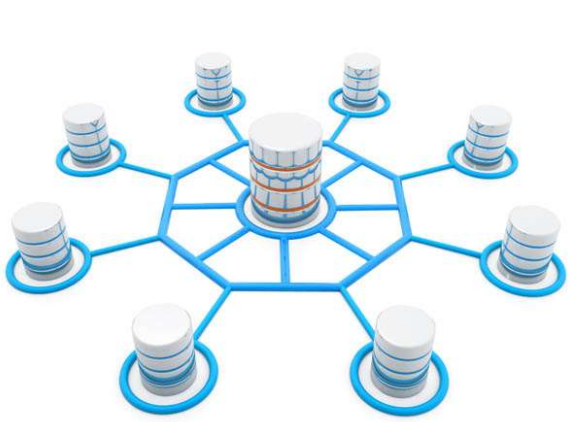
What is a Repository?

A directory of storage space where your projects can live.

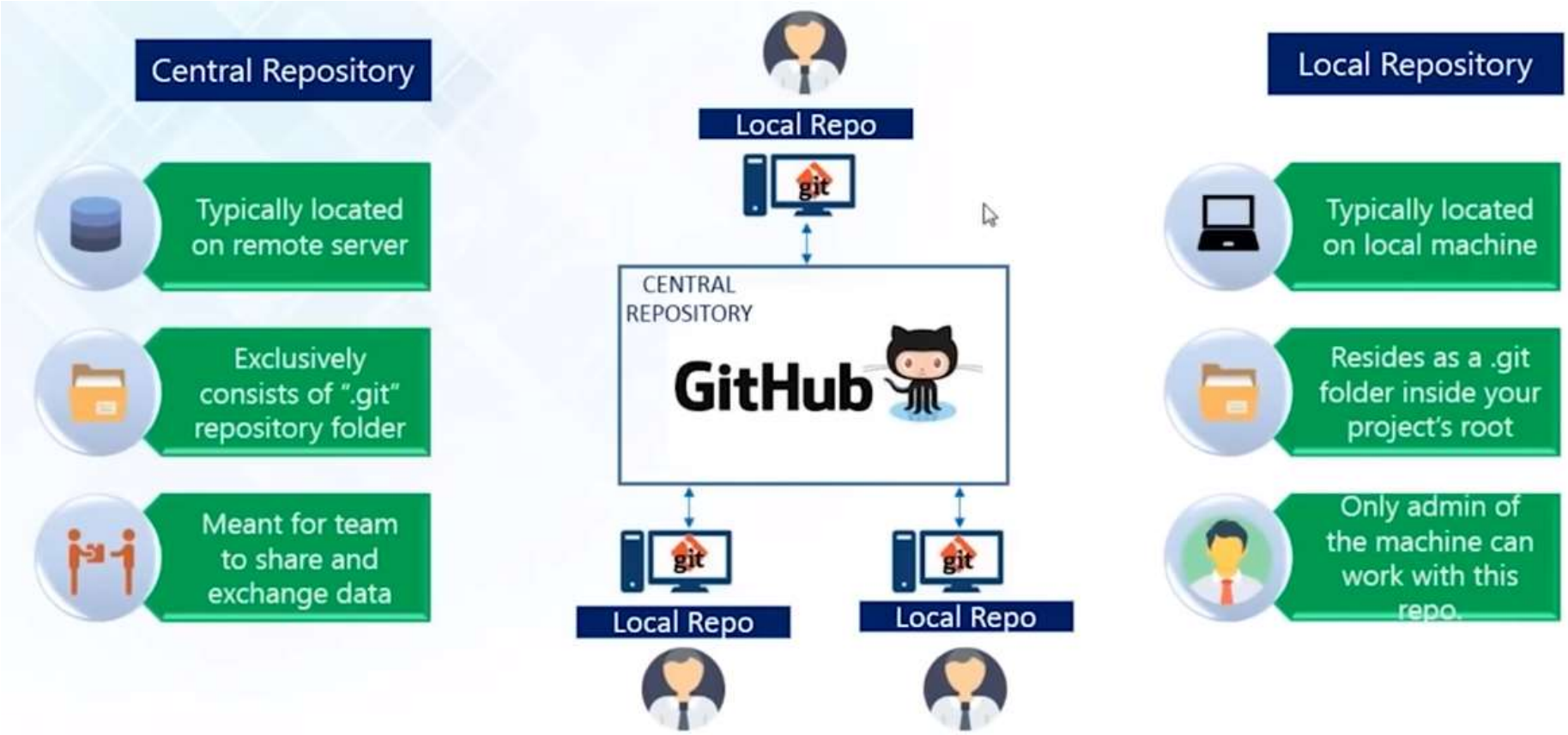
It can be local to a folder on your computer, or an online storage space

There are two main types of repositories:

- Central Repository
- Local Repository



Central and Local Repository

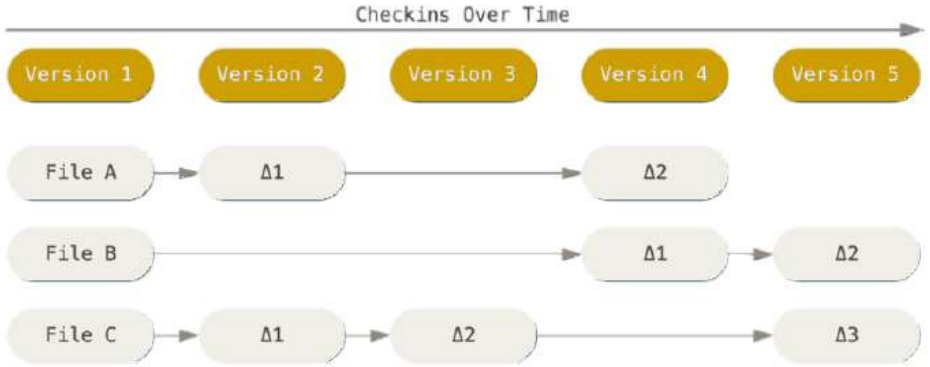


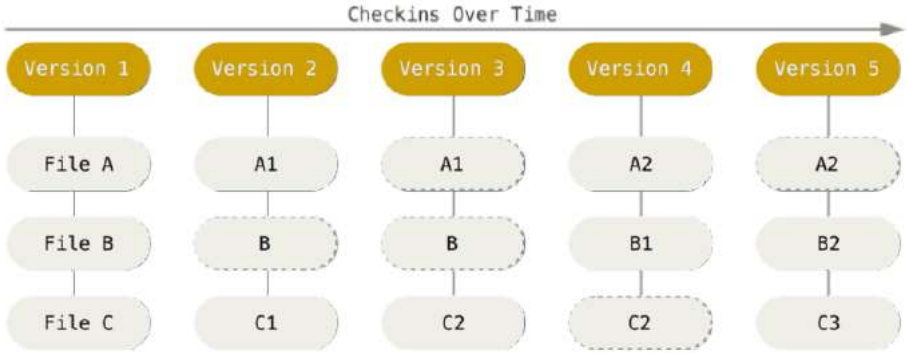
Storing Versions

- Snapshots of all versions are properly documented and stored
- Versions are also named accurately

GIT

- The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data.
- Conceptually, most other systems store information as a list of file-based changes



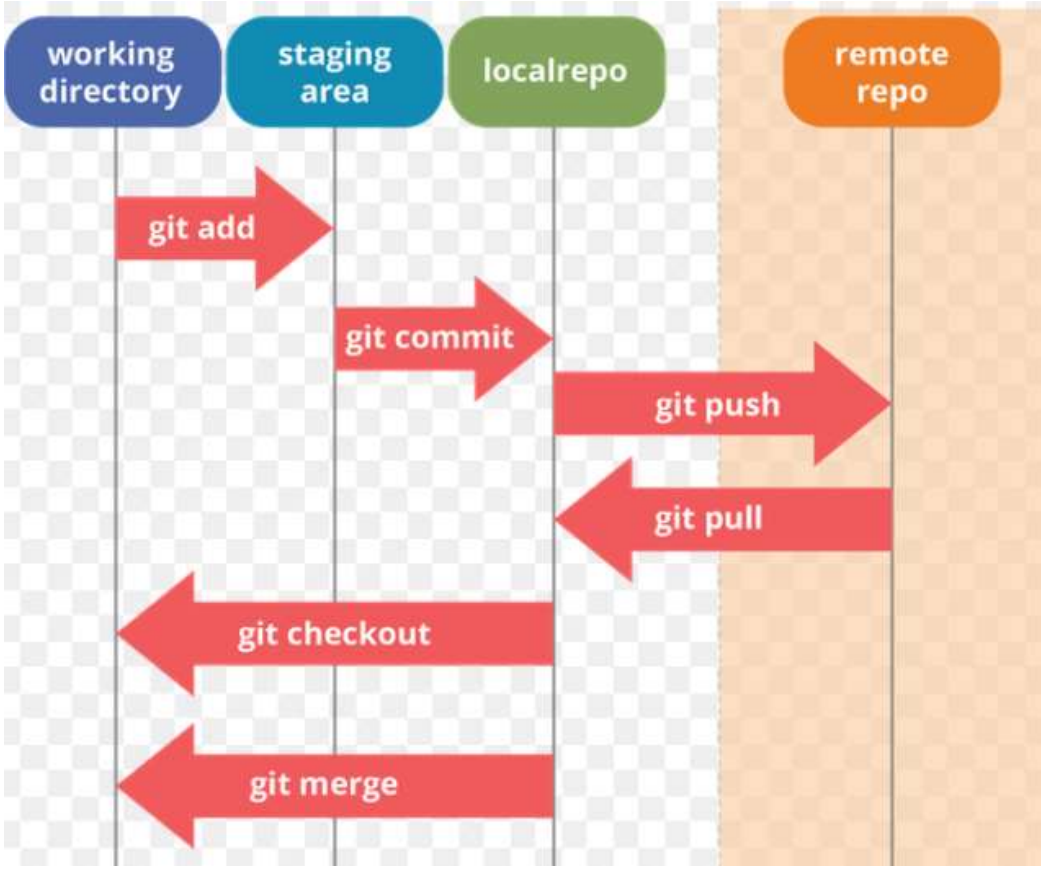


GIT

- Linux kernel developers used BitKeeper for their development.
- The Decision to use it was wildly regarded as a bad move and made many people unhappy.
- Their fears were confirmed in 2005 when BitKeeper SCM stopped being free.
- Since it was closed source, the developers lost their favorite Version Control System.
- The community (led by Linus Torvalds) had to find another VCS, and since an alternative was not available, they decided to create their own.
- Thus, Git was born Git community is very active, and there are many contributors involved in its development https://git-scm.com/.
  - Go back and forth between versions
  - Review the differences between those versions
  - Check the change history of a file
  - Tag a specific version for quick referencing
  - Exchange “changesets” between repositories
  - Review the changes made by others

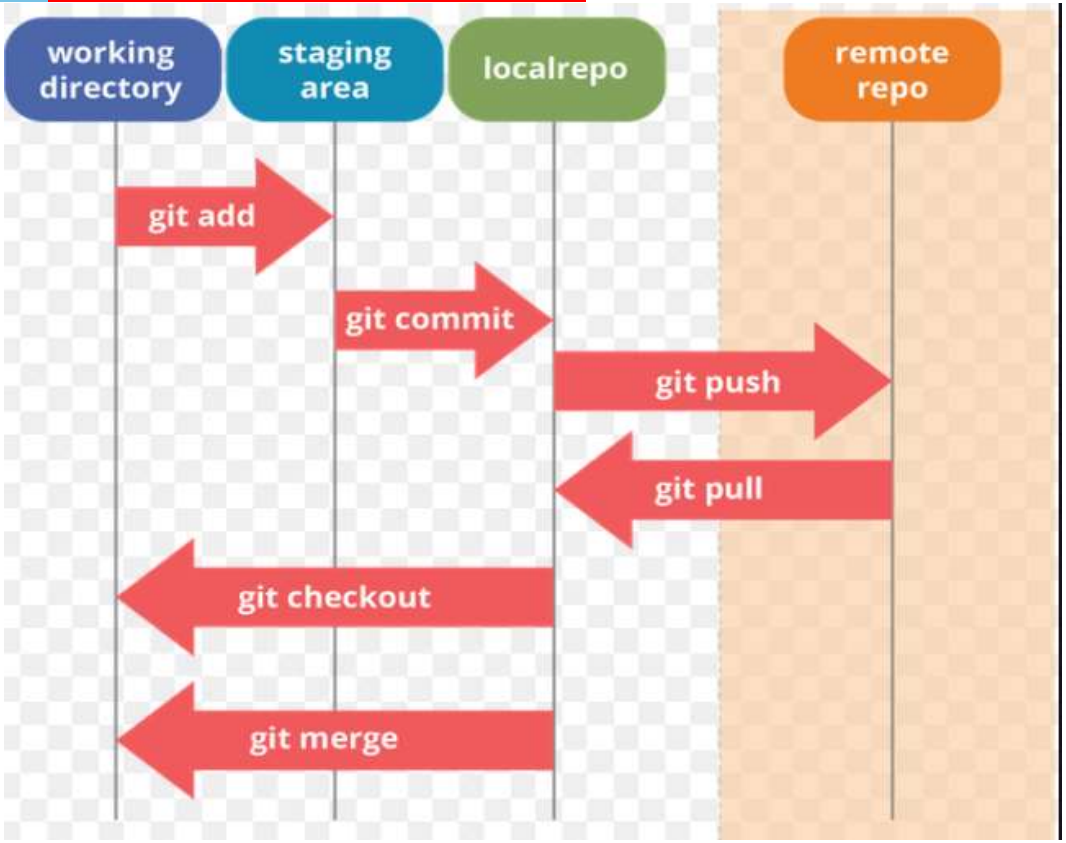
GIT

- Git has three main states that files can reside in: modified, staged and committed.
- Modified means that changed the file but have not committed it to local repo yet.
- Staged means that marked a modified file in its current version to go into next commit snapshot.
- Committed means that the data is safely stored in local repo.



GIT

- Local Repo directory is where Git stores the metadata and object database for project. This is the most important part of Git, and it is what is copied when clone a repository from another server.
- Staging area is a file, generally contained in local repo that stores information about what will go into next commit. It's sometimes referred to as the "index", but it's also common to refer to it as the staging area.
- Working directory is a single checkout of one version of the project. These files are pulled out of the compressed database in the local repo and placed on disk for you to use or modify.

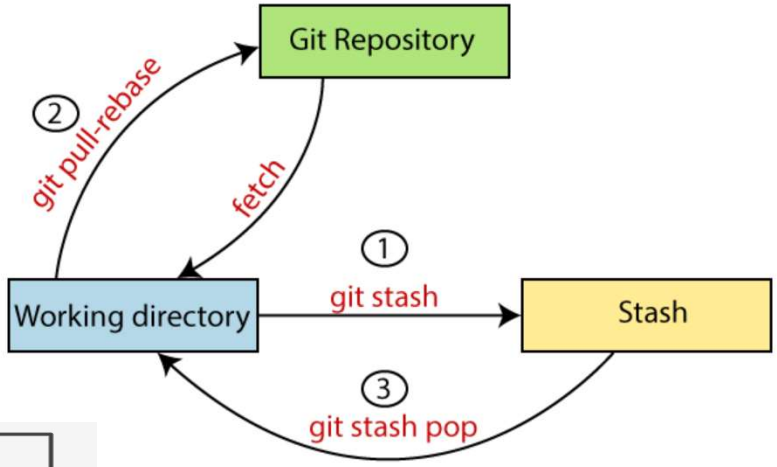


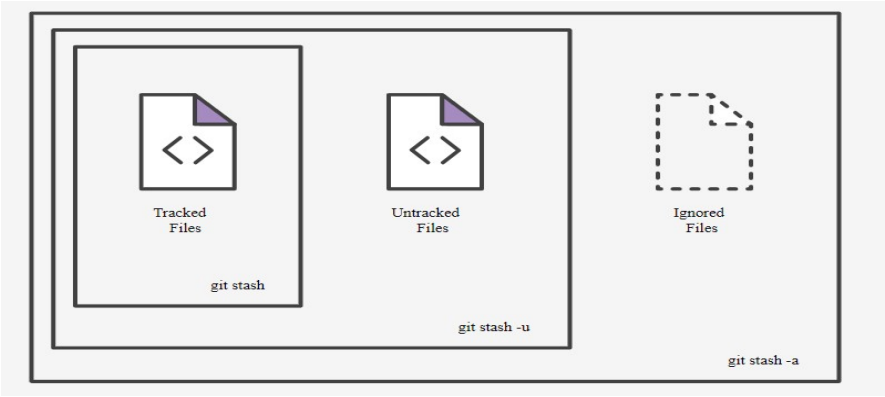
GIT – Installation and configuration

- When you install Git is to set your user name and email address.
- This is important because every Git commit uses this information
- $ git config --global user.name “arahman"
- $ git config --global user.email arahman@wilp.bits-Pilani.ac.in
- $ git config –list
- $ git init ->To create a new blank repository and also used to make an existing project as a Git project
- $ git add test.cc -> Add file contents to the Index (Staging Area)
- $ git commit -m 'initial project version’
- $ git clone https://github.com/arahman/Git-Example.git -> copy of the remote repository
- $ git clone https://github.com/libgit2/libgit2
- $ git clone -b <Branch name><Repository URL>
- $ git clone -b master https://github.com/arahman/Git-Example.git "new folder(2)"
- $ git diff
- $ git rm
- $ git log -> Commit history

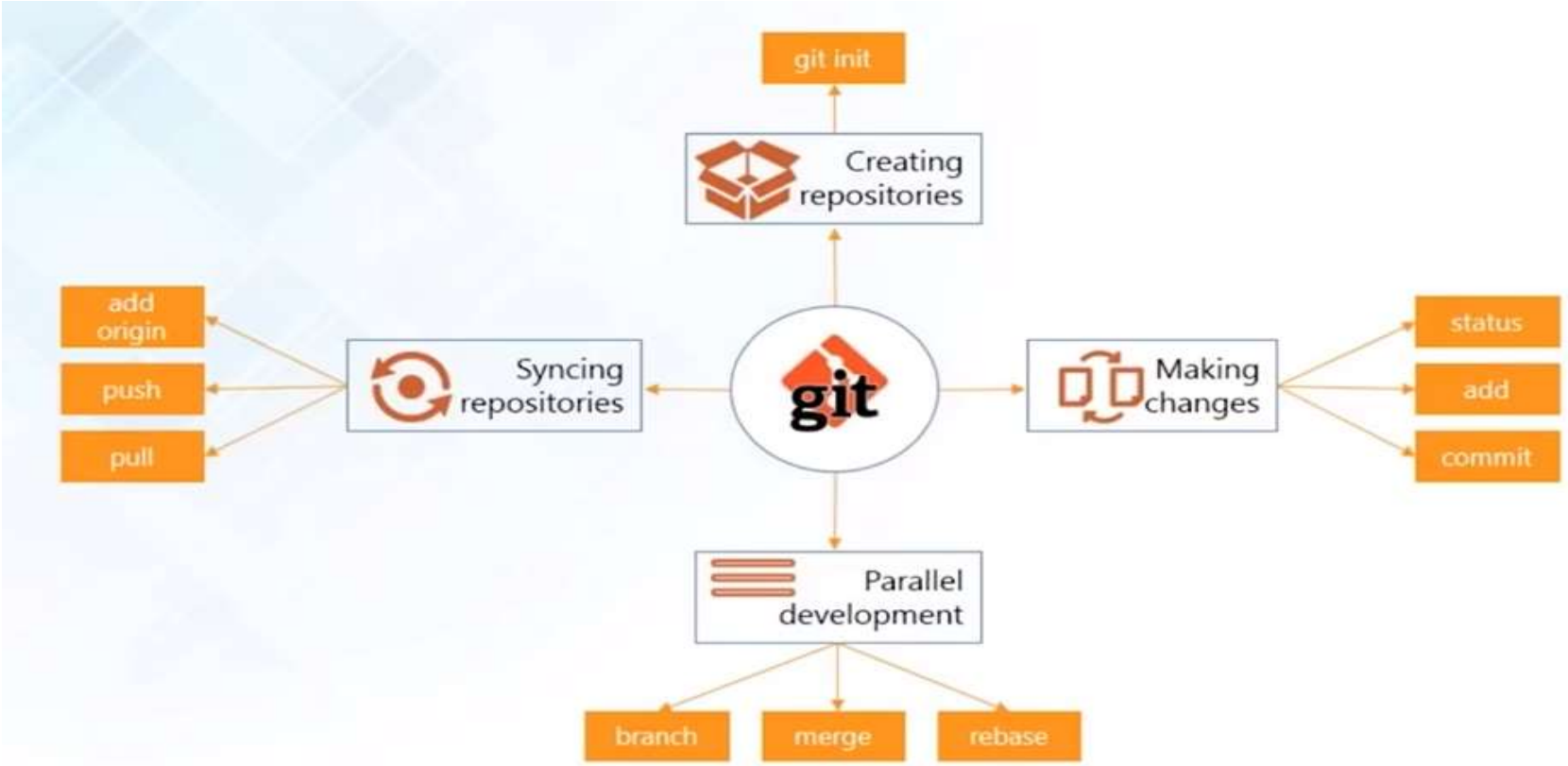
GIT – Installation and configuration

$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit –amend
$ git stash and $ git stash pop
$ git reset --hard
.gitignore -> Lets Git know that it should ignore certain files and not track them





Git Operations and Commands



GIT – Installation and configuration

- One of the main features of Git is its Branching system
- Stashing is the act of safely putting away your current edits so that you have clean environment to work on something completely different.
  - $ git init # Initialize a new git database
  - $ git clone # Copy an existing database
  - $ git status # Check the status of the local project
  - $ git diff # Review the changes done to the project
  - $ git add # Tell Git to track a changed file
  - $ git commit # Save the current state of the project to database
  - $ git push # Copy the local database to a remote server
  - $ git pull # Copy a remote database to a local machine
  - $ git log # Check the history of the project
  - $ git branch # List, create or delete branches
  - $ git merge # Merge the history of two branches together
  - $ git stash # Keep the current changes stashed away to be used later

GIT – Installation and configuration

- $ git remote
- $ git remote –v

Add remote repository git remote add <shortname> <url>

- $ git remote add <name> https://github.com/arrahman/bitsian
- $ git fetch <name> -> to retrieve the latest meta-data info from the original
- git fetch command only downloads the data to your local repository – it doesn't automatically merge it with any of your work or modify what you're currently working on.
- use the git pull command to automatically fetch and then merge that remote branch into your current branch.
- $git pull = $git fetch origin HEAD + $ git merge HEAD (HEAD ref pointing to current branch)
- $ git push [remote-name] [branch-name].
- $ git tag

GIT – Installation and configuration

1.**Verify the installation** by running:

git –version

Step 2: Configure Git

Before using Git, it's essential to configure your identity.

This information will be included in your commits.

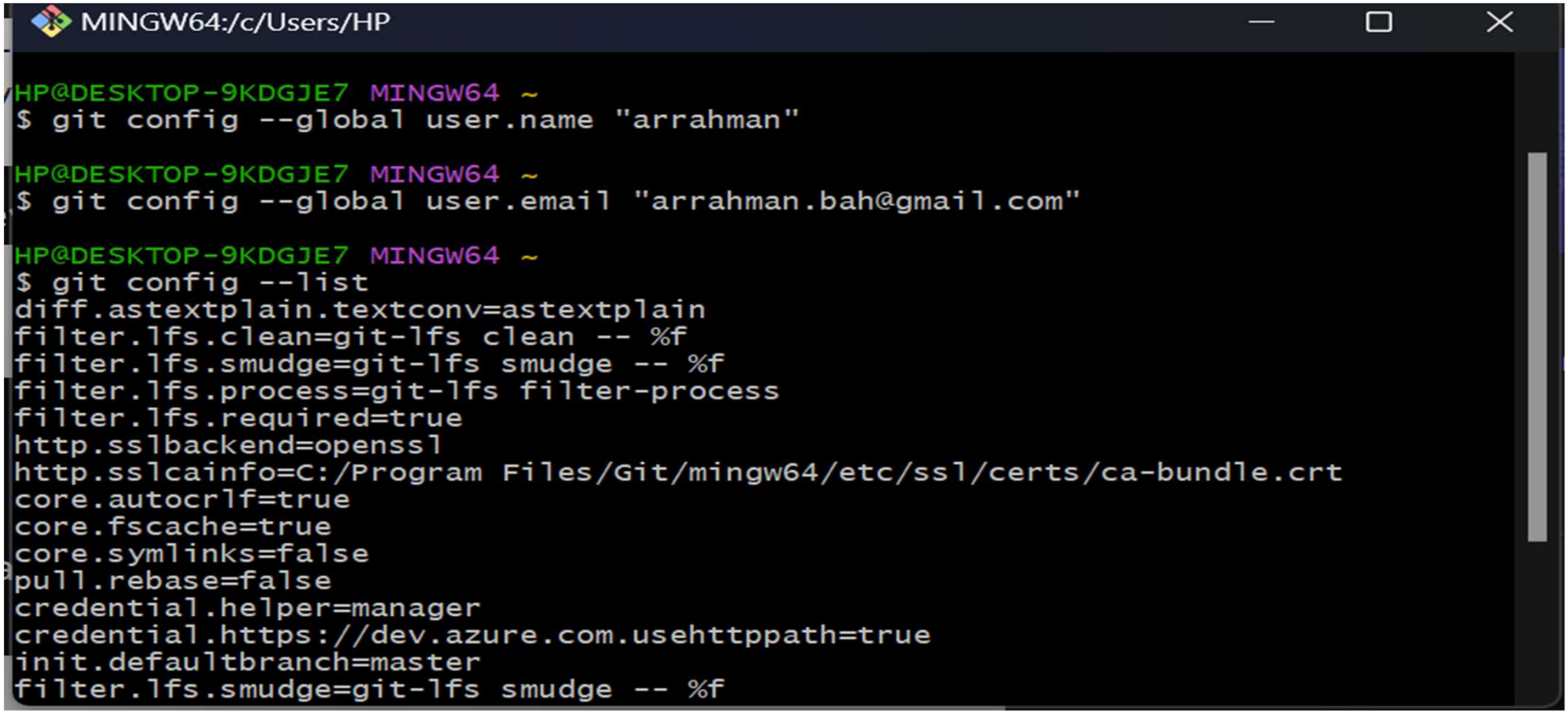
1. **Set your username**:git config --global user.name "Your Name"

2. **Set your email**:git config --global user.email your.email@example.com

You can check your configuration settings at any time using:

git config --list

GIT – Installation and configuration



GIT – Installation and configuration

GitHub

Create your central repository on GitHub

git init

git

Install git on your local machine and use “git init” to create local repository

git clone

Download or clone your repository from GitHub using “git clone”

GIT – Installation and configuration

Step 3: Initialize a New Repository

Creating a New Repository

1.Navigate to your project directory: cd /path/to/your/project

2.Initialize the repository: git init

This command creates a new .git directory in your project folder, marking it as a Git repository.

Adding Files to the Repository

1.Add files to the staging area: git add .

This command stages all the files in your project directory for the initial commit.

You can also add specific files by listing them individually.

2.Commit the files: git commit -m "Initial commit“

This command commits the staged files to the repository with a message describing the commit.

GIT – Installation and configuration

git status

Tells you which files are added to the index

git add

Let's you add files to your index

git commit

Let's you add files to your index

GIT – Installation and configuration

Step 4: Working with a Remote Repository

To collaborate with others or keep a backup of your repository, you can use remote repositories hosted on platforms like GitHub, GitLab, or Bitbucket.

Creating a Repository on GitHub

1. **Sign in to GitHub** and navigate to GitHub.
2. **Click on the "+" icon** in the top right corner and select "New repository".
3. **Fill in the repository details** (name, description, etc.) and click "Create repository".

Syncing Repositories

- Use “git add origin <link>” to add remote repo
- Pull files with “git pull”
- Push your changes into central repo with “git push”

GIT – Installation and configuration

Connecting Your Local Repository to the Remote Repository

1.Add the remote repository URL: git remote add origin https://github.com/yourusername/your-repository

2.Push your local commits to the remote repository: git push -u origin main

Step 5: Cloning an Existing Repository

If you want to start working on an existing project, you can clone a remote repository to your local machine.

1.Navigate to the desired directory where you want to clone the repository:cd /path/to/directory

2.Clone the repository:git clone https://github.com/username/repository

3.This command creates a copy of the remote repository on your local machine.

GIT Basics commands(Creating Repositories, clone, push, commit, review)

- Commit
  - The “commit” command is used to save your changes to the repository
  - Every set of changes implicitly creates a new, different version of your project.
  - Every commit also marks a specific version.
  - It’s a snapshot of your complete project at that certain point in time.
  - The commit knows exactly how all of your files and directories looked and can be used to restore the project to that certain state.
  - Every commit item consists (amongst other things) following metadata
    - unique id/hash — every commit has a unique identifier.
    - date — information when commit happened. It helps later on to lists the commits in chronological order
    - author — information who performed changes
    - message — the author of a commit has to comment what he did in a short “commit message”

GIT Basics commands(Creating Repositories, clone, push, commit, review)

- unique id/hash
  - In centralized version control systems(CVS,SVN), an ascending revision number is used
  - In distributed version control (Git) multiple people can work in parallel, committing their work offline. So, Git is using a 40-character checksum called the “commit hash”
- Commit only related changes
  - When crafting a commit, it’s very important to only include changes that belong together.
  - Should never mix up changes from multiple, different topics in a single commit.
  - Commit should only wrap related changes: fixing two different bugs should produce (at the very least) two separate commits
- Write Good Commit Messages
  - Time spent on crafting a good commit message is time spent well:
  - It will make it easier to understand what was the intention when making these changes (and after some time also for yourself)

GIT Basics commands(Creating Repositories, clone, push, commit, review)

- Branch
  - In every project, there are always multiple different contexts where work happens.
  - Each feature, bugfix, experiment of product can be seen as its own “topic”, clearly separated from other topics.
  - In real-world projects, work always happens in multiple of these contexts in parallel.
  - Need branching to solve issues
    - Handle multiple topics in a single context would lead into a chaos (copying your complete project folder is not a solution)
    - Another team member can block you if they break something up
    - Providing a bug fix for customer without bringing new feature implementation which already started might be a challenging or not possible at all

There are two ways to create another branch

1. For example, create a local **dev** branch and then push that branch to the remote

```
$ git branch dev
$ << Note:the command executes silently>>
```

2. Create a remote **dev** branch (using the Bitbucket web interface), and then fetch that branch locally
   - <<Demo of remote branch creation>>

```
$ git fetch && git checkout dev
  From bitbucket.org:se2800/MyRepository
 * [new branch]      dev        -> origin/dev
Branch dev set up to track remote branch dev from origin.
Switched to a new branch dev'
```

GIT Basics commands(Creating Repositories, clone, push, commit, review)

Git commands for branch status

Use git branch to see what local branch you are currently working on:

```
$ git branch
  * dev
    master
```

- Use git fetch && git branch -r to list all remote branches:

```
$ git fetch
$ git branch -r
  origin/master
  origin/dev
```

Git commands for switching branches

Use git checkout master to switch your working directory to the **master** branch:

```
$ git checkout master
    dev
  * master
```

- Use git checkout dev to switch your working directory to the **dev** branch:

```
$ git checkout dev
  * dev
    master
```

NOTE: *If you currently have nothing to commit*, and you switch branches, the code in your working directory will be replaced with the code that is in the branch you're switching to.

GIT Basics commands(Creating Repositories, clone, push, commit, review)

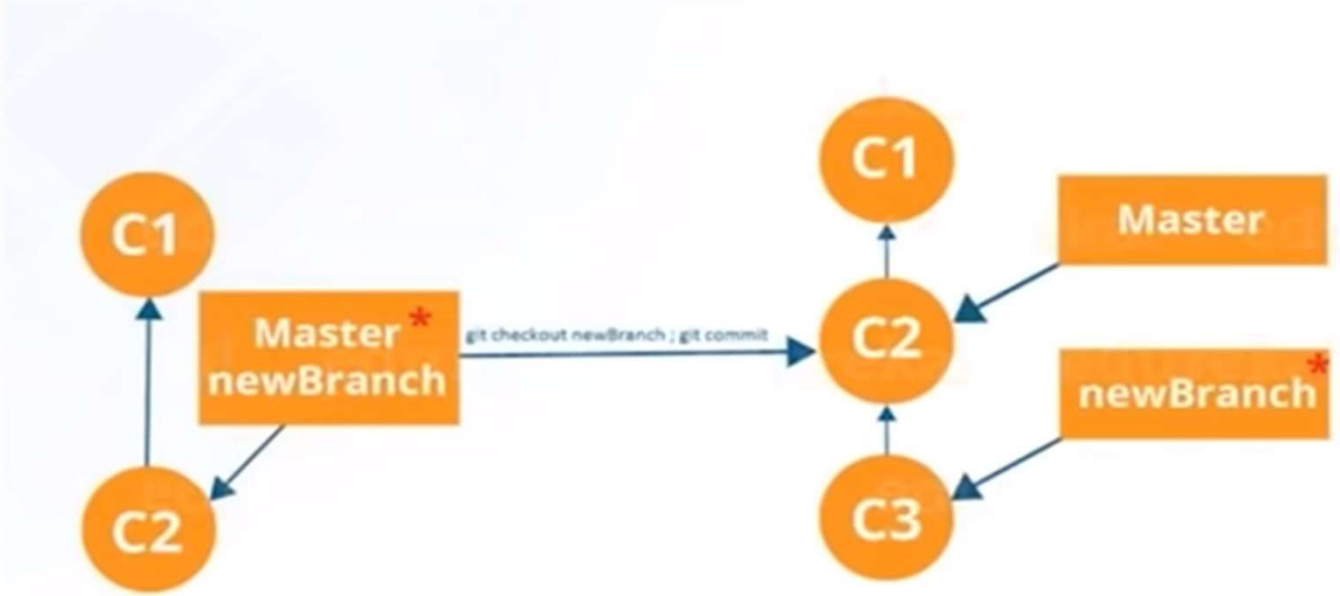
- Branch
- Branch represents exactly such a context in a project and helps you keep it separate from all other contexts
- All the changes you make at any time will only apply to the currently active branch and all other branches untouched
- Freedom to both work on different things in parallel and, above all, to experiment
- Use Branches Extensively
  - Branching is one of most powerful features.
  - Branches are the perfect tool to help you avoid mixing up different lines of development.
  - Should use branches extensively in your development workflows: for new features, bug fixes, experiments, ideas…

GIT Basics commands(Creating Repositories, clone, push, commit, review)

Branches are pointers to a specific commit

Branches are of two types:

- Local branches
- Remote-tracking branches



GIT Basics commands(Creating Repositories, clone, push, commit, review)

- Tag

- A tag represents a version of a particular branch at a moment in time (tag mark a specific commit in your repository history).

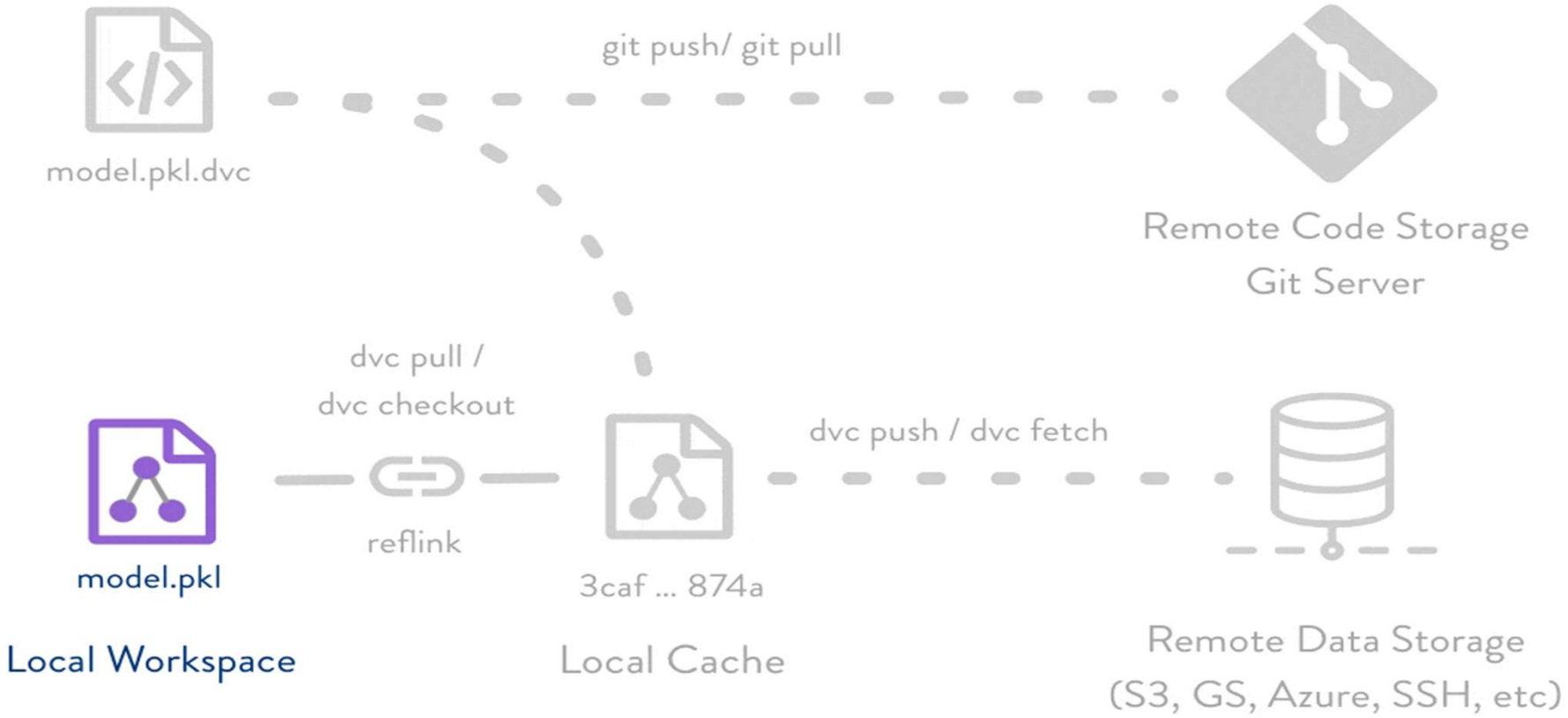
- Tags are commonly used to mark release versions, with the release name as the tag name (i.e. v1.0.1).

- A tag is like a branch that doesn't change. Unlike branches, tags, after being created, have no further history of commits.

- When tag a commit, all the changes are included before it.

- Helps later compare tags to see the difference between two points in history.

Data Versioning



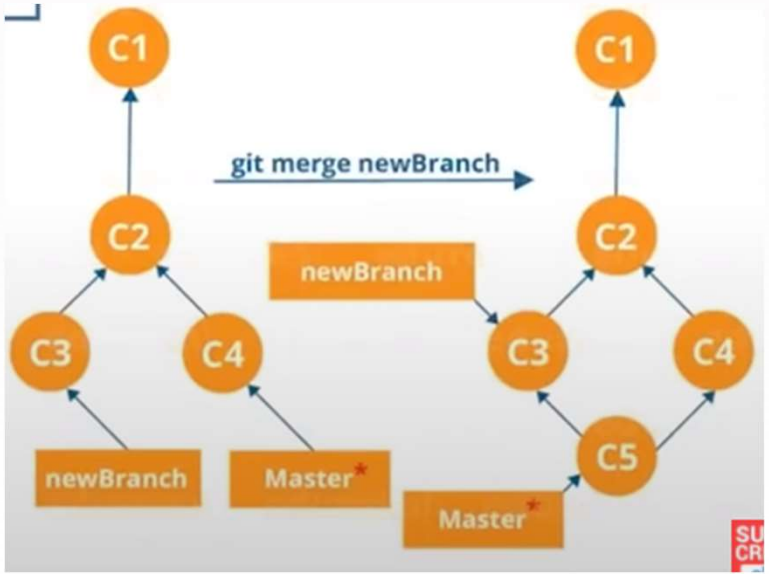
GIT Basics commands(Creating Repositories, clone, push, commit, review)

- Merge
- Integrating Branches — Not Keeping commits in the separate context of a branch is a huge help.
- But there will come a time when you want to integrate changes from one branch into another.
- For example, when you finished developing a feature and want to integrate it into your "production" branch.
- you're not yet finished working on your feature, but so many things have happened in the rest of the project in the meantime that you want to integrate these back into your feature branch. Such an integration is called "merging"
- Individual Commits
  - When starting a merge, no need have to (and cannot) pick individual commits that shall be integrated.
  - Instead, tell Git which branch you want to integrate and Git will figure out which commits you don't have in your current working branch. Only these commits will then be integrated as a result.

Parallel Development - Merging

It is a way to combine the work of different branches together

Allows to branch off, develop a new feature and combine it back

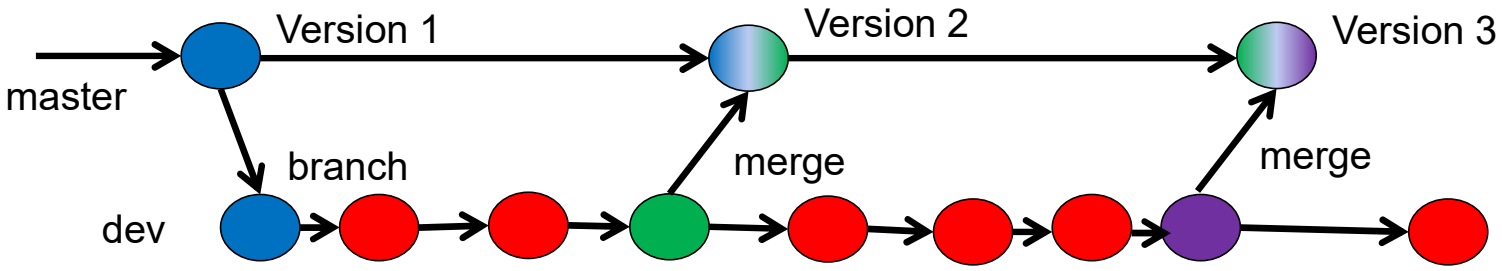


The master branch should only be used for maintaining production-ready code (and related files)

Code modified or created during a development cycle should be maintained on separate branches

A second standard branch – often named "**dev**" is typically used to maintain code that is being actively worked on

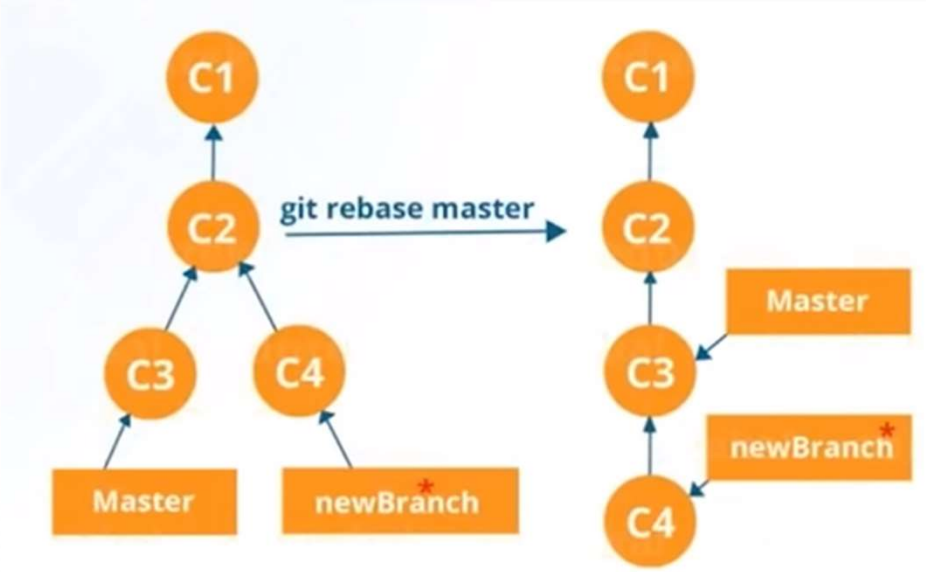
- At the beginning of a cycle, the **dev** branch is identical to master
- During development, the **dev** branch maintains the evolving code
- At the end of a cycle, the **dev** branch contains the completed code.
  - The **dev** branch is then merged into the **master** branch, and the process repeats with the next cycle.



Parallel Development - Rebasing

This is also a way of combining the work between different branches

It can be used to make a linear sequence of commits



GIT Basics commands(Creating Repositories, clone, push, commit, review)

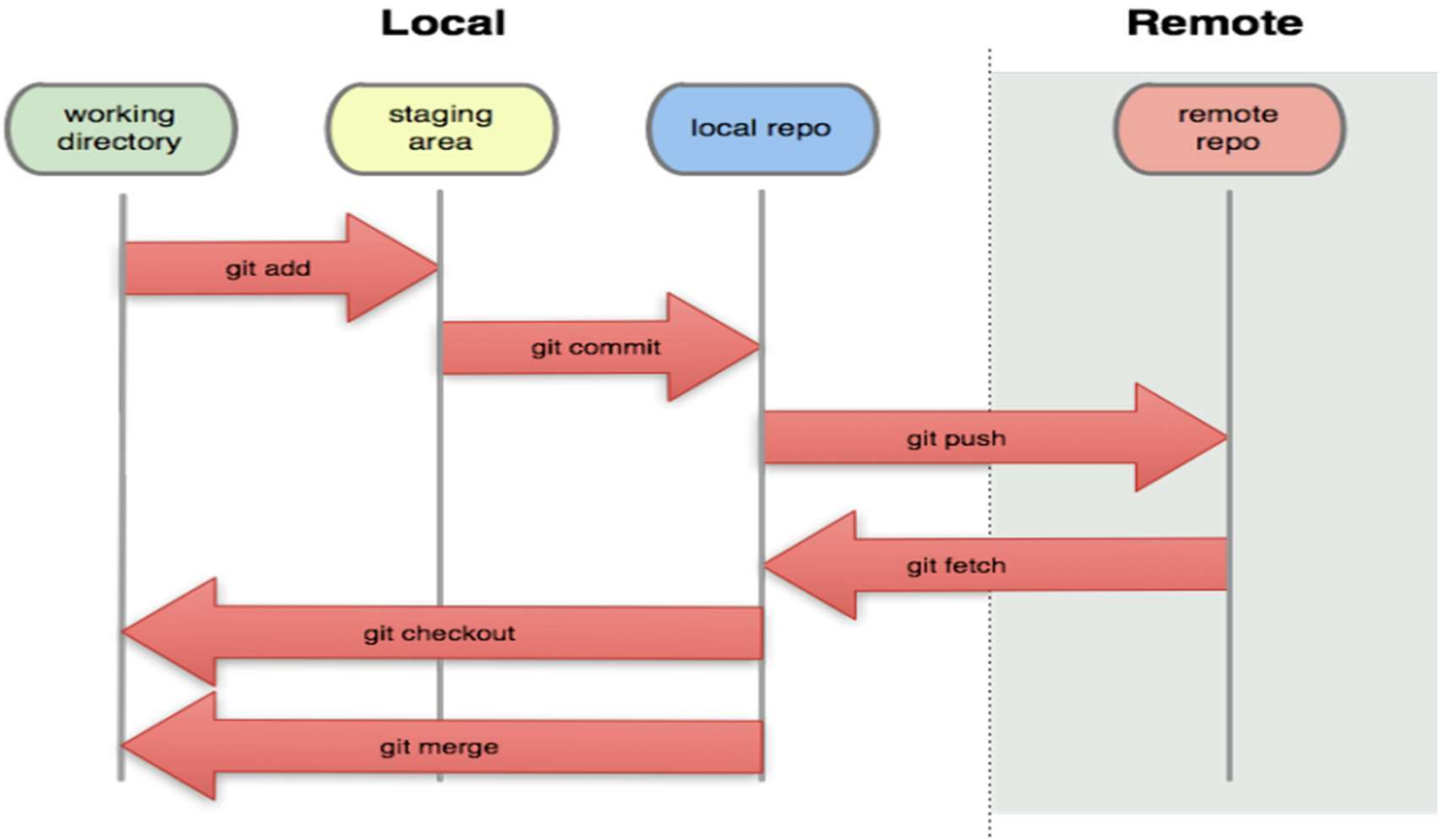
When you submit a change, git review does the following things:

- it looks up which branch to push to (production or whatever) in the .gitreview file.
- If it can't find this information, it pushes to master
- it figures out what "topic" to put on the revision (you can set the topic manually with -t)
    - if you're resubmitting a downloaded change, it will reuse the tag of the original change
    - if your commit summary contains a bug number like bug 12345, the tag will be bug/12345 otherwise, the tag will be the name of your local branch
- it rebases your change against the HEAD of the branch you're pushing to (use -R to skip this)
- if you are submitting more than one change at once, or submitting a change that's based on another unmerged change, it will ask you whether you really meant to do that (use -y to skip this)
- it pushes the change for review

GIT Basics commands(Creating Repositories, clone, push, commit, review)

- Any project that has more than one developer maintaining source code files should absolutely use a Version Control
- Collaboration
  - team is able to work absolutely freely — on any file at any time.
  - later allow to merge all the changes into a common version
  - Always latest version of a file or the whole project is in a common, central place
- Rollback and undo changes to source code
  - keeps a history of changes and the state of the source code throughout a project's history
  - possibility to "undo" or rollback a source code project to a last well-known state
  - If any bug, the code can be quickly reverted to a known stable version
- Understanding What Happened
  - Short description/Tagging on new version helps to understand what was changed
  - can see what exactly was changed in the file's content
  - helps to understand how your project evolved between versions

Git workflows- Feature workflow, Master workflow, Centralized workflow



Git workflows- Feature workflow, Master workflow, Centralized workflow

- Featured Branch Workflow

- The Feature Branch Workflow still uses a central repository, and master still represents the official project history.
- But, instead of committing directly on their local master branch, developers create a new branch every time they start work on a new feature.
- Feature branches should have descriptive names as *login-template-header, login-http-resource, refactoring-login-service*, etc.
- Now your team is working alongside of the remote master branch, everyone is working on a feature branch.
- When each feature is done and are up to put on development the branch should be merged with remote master by a Pull Request.

Git workflows- Feature workflow, Master workflow, Centralized workflow

- Pull Request.
- Pull requests are a feature that makes it easier for developers to collaborate using a git client (GitHub, Bitbucket, Stash, etc.).

- They provide a user-friendly web interface for discussing proposed changes before integrating them into the official project.

- Pull request is a dedicated forum for discussing the proposed feature.

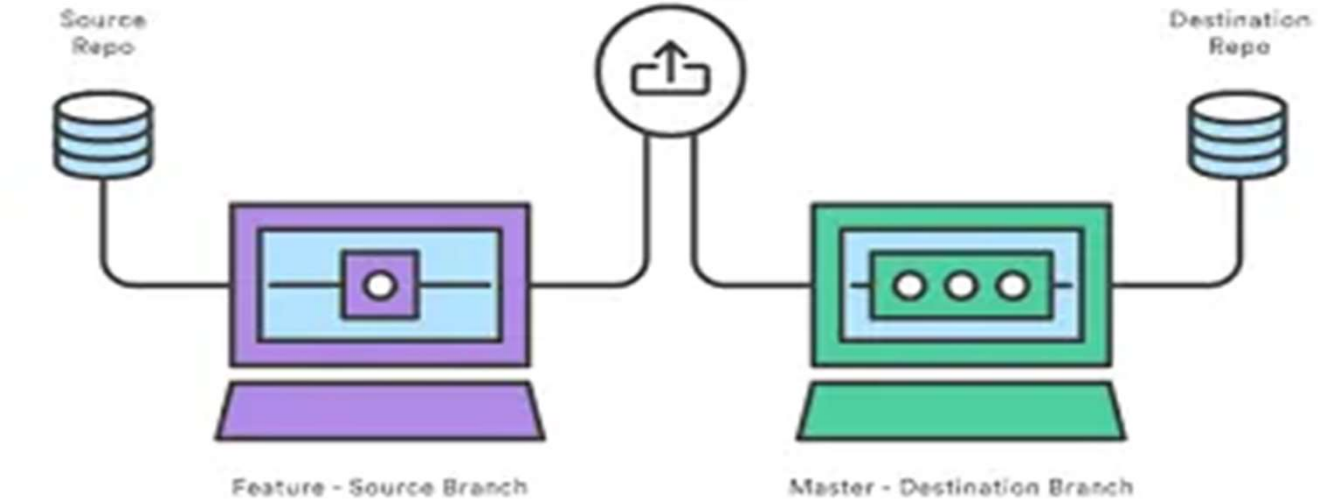
- If there are any problems with the changes, teammates can post feedback in the pull request and even tweak the feature by pushing follow-up commits.

- All of this activity is tracked directly inside of the pull request.

Git workflows- Feature workflow, Master workflow, Centralized workflow

- Essentially a PR should be closed after a Code Review with others developers.
- Code review is a major benefit of pull requests, but they're actually designed to be a generic way to talk about code.
- You can think of pull requests as a discussion dedicated to a particular branch.
- This means that they can also be used much earlier in the development process.
- For example, if a developer needs help with a particular feature, all they have to do is file a pull request. Interested parties will be notified automatically, and they'll be able to see the question right next to the relevant commits. After the Code Review is finished your code will be able to become part of the remote master code.

Git workflows- Feature workflow, Master workflow, Centralized workflow



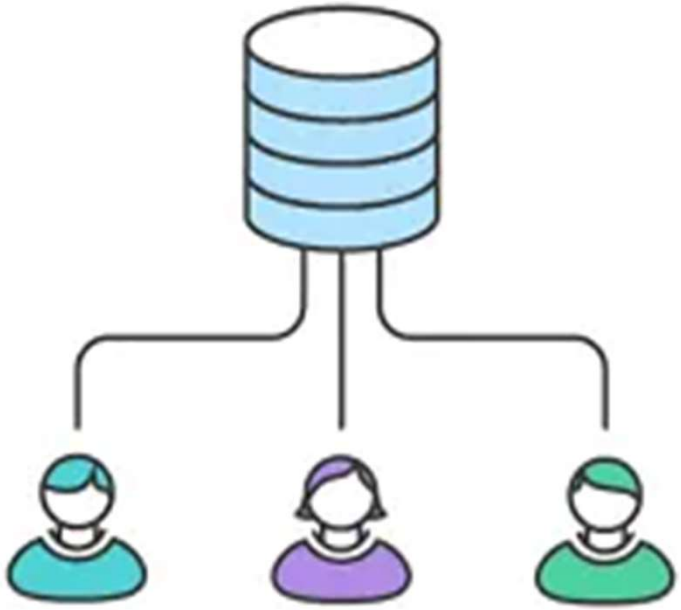
Git workflows- Feature workflow, Master workflow, Centralized workflow

- **Centralized Workflow -** This is the "**troublemaker**", the cause of discords.
- The Centralized Workflow uses a central repository to serve as the entry for all changes to the project.

- The default development branch is called master and all changes are committed into this branch.
- This workflow doesn't require any other branches besides master.

- The client starts by cloning the central repository and in their own local copies of the project, they edit files and commit changes as they would with SVN; however, these new commits are stored locally completely isolated from the central repository.
- This lets developers defer synchronizing upstream until they're at a convenient break point.
- It is the simplest way to work with Git, you work directly on the master locally and when you finish doing changes you commit to the central repository, simple right?

Git workflows- Feature workflow, Master workflow, Centralized workflow



Git workflows- Feature workflow, Master workflow, Centralized workflow

- Imagine that you have a team and everyone is working on your local copy of **master** and one of your teammates, **Dev - A,** push all his changes about the feature B to the central repositor but you are working on feature A locally yet.

- Friday's you finish your feature and your leader tells you to deploy on production this feature as soon as possible, BUT the Dev - A 's feature cannot be deployed together. See the big picture?

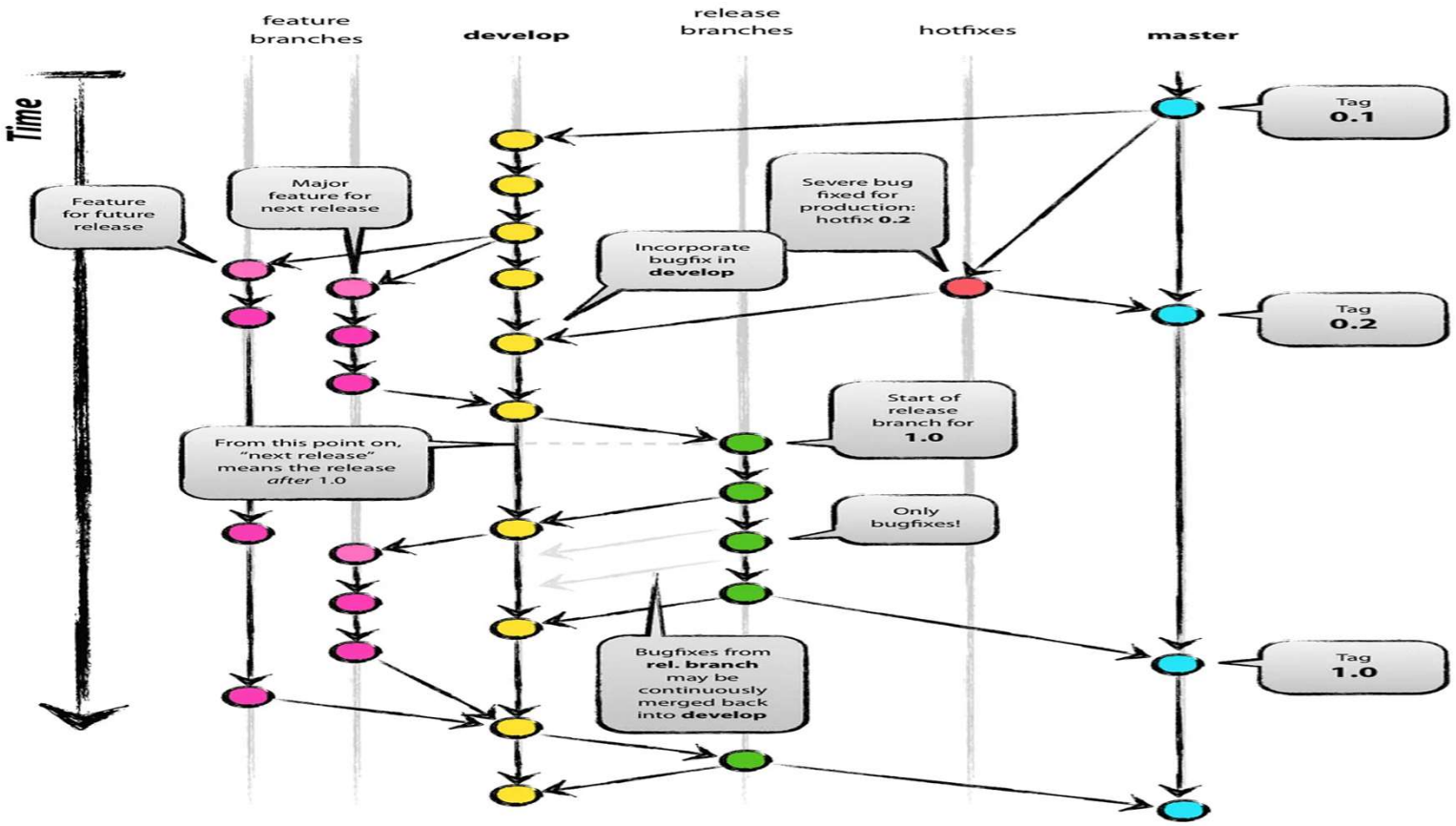
- You have to synchronize your local repository with master to deploy something but in **master**

- you have code who is not applicable to be deployed.
- It will be cool if you can work on "another copy of central repository", right?

Git workflows- Feature workflow, Master workflow, Centralized workflow

- Git Workflow.
- This is the most complete workflow on git, you have many possibilities.
- We always thought this is the exactly workflow which big teams work on the same repository would adopt and the small teams should avoid.

Git workflows- Feature workflow, Master workflow, Centralized workflow



Git workflows- Feature workflow, Master workflow, Centralized workflow

- Looking the big picture, you can see the possibilities you can do with this flow.
- You have hotfixes branch that should be solved as soon as possible on the minor “distance” of master, to simplify and do the change as faster as can, you have release branches which are the preparation to the next release on production, develop branch and as you can note by the name is on the code merged from release branch are, nothing directly on master anymore, which now is only tags.
- This is the most “beautiful” workflow on git, you have a big consistency working with a big team but is an overengineering when you are working in a small or medium (maybe) team, because you have more control to manage the changes.

Git workflows- Feature workflow, Master workflow, Centralized workflow

- Git Flow is an evolution of Featured Branch Workflow, so it reuse all principles of each pattern and include more as develop, hotfixes and release branches.
- These patterns are very important and I guess for each team you will use one of them.
- One isn’t better them another in every case, forget it!
- You will not find a solution for all your problems with only one of them instead you will have more options to work with small, medium and big teams and see how any of these patterns works better.

Git Feature Branching

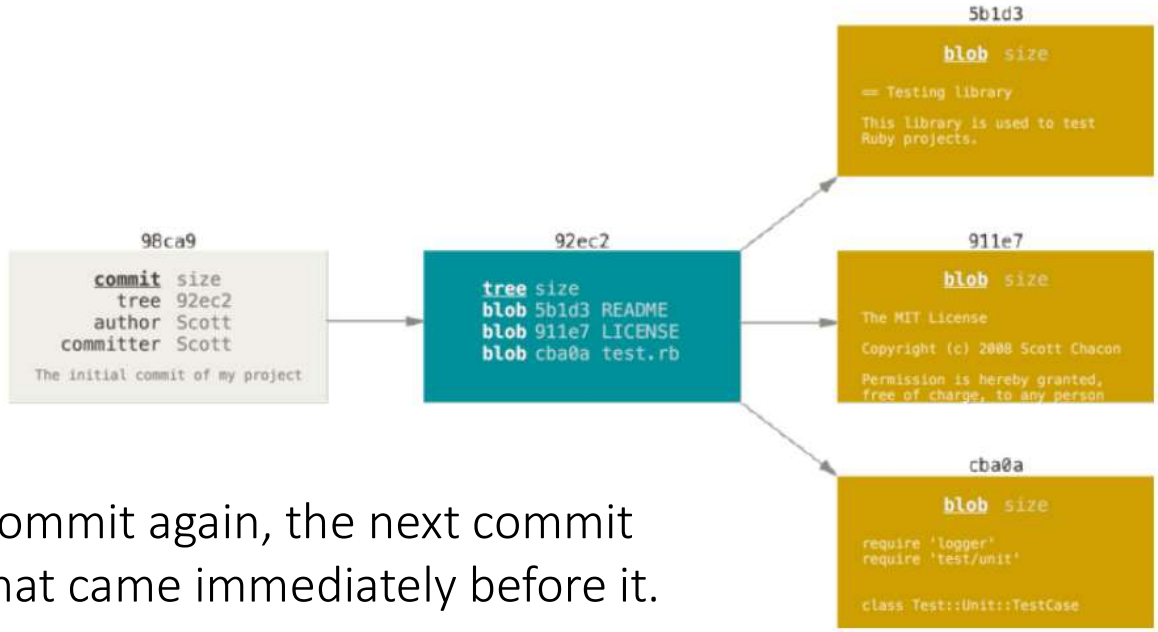
- When make a commit, Git stores a commit object that contains a pointer to the snapshot of the content staged.
- This object also contains the author's name and email, the message that typed, and pointers to the commit or commits that directly came before this commit (its parent or parents):
- Zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.
- Create the commit by running git commit, Git checksums each subdirectory (in this case, just the root project directory) and stores those tree objects in the Git repository.
- Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

Git Feature Branching

- When make a commit, Git stores a commit object that contains a pointer to the snapshot of the content staged.
- This object also contains the author's name and email, the message that typed, and pointers to the commit or commits that directly came before this commit (its parent or parents):
- Zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.
- Create the commit by running git commit, Git checksums each subdirectory (in this case, just the root project directory) and stores those tree objects in the Git repository.
- Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

Git Feature Branching

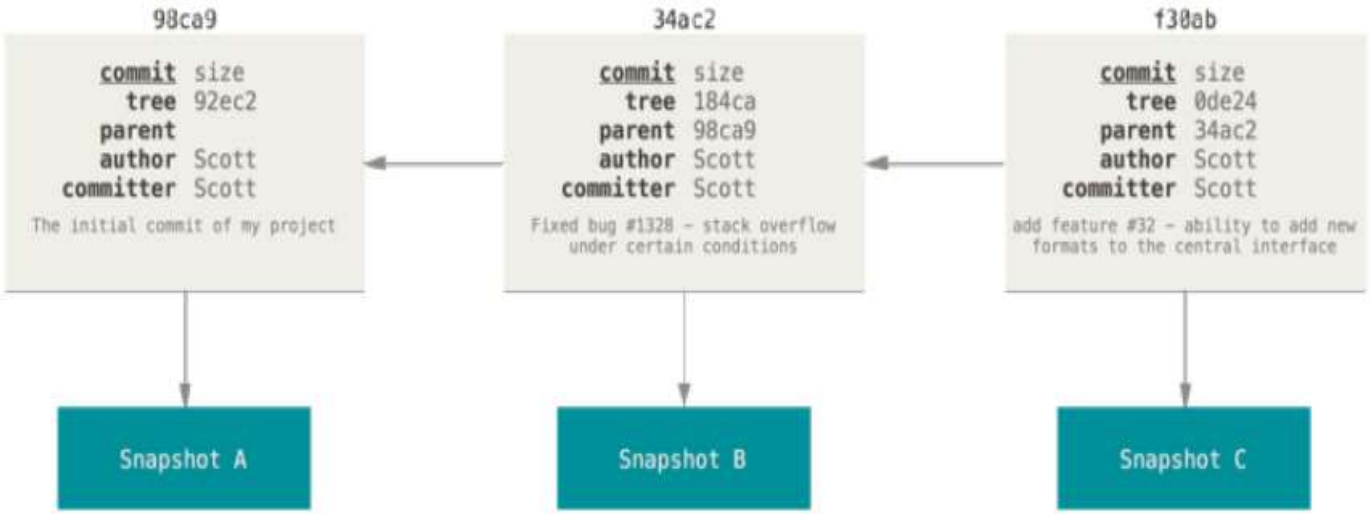
- Git repository now contains five objects: one blob for the contents of each of files, one tree that lists the contents of the directory and specifies which file names are stored as which blobs, and one commit with the pointer to that root tree and all the commit metadata.



If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.

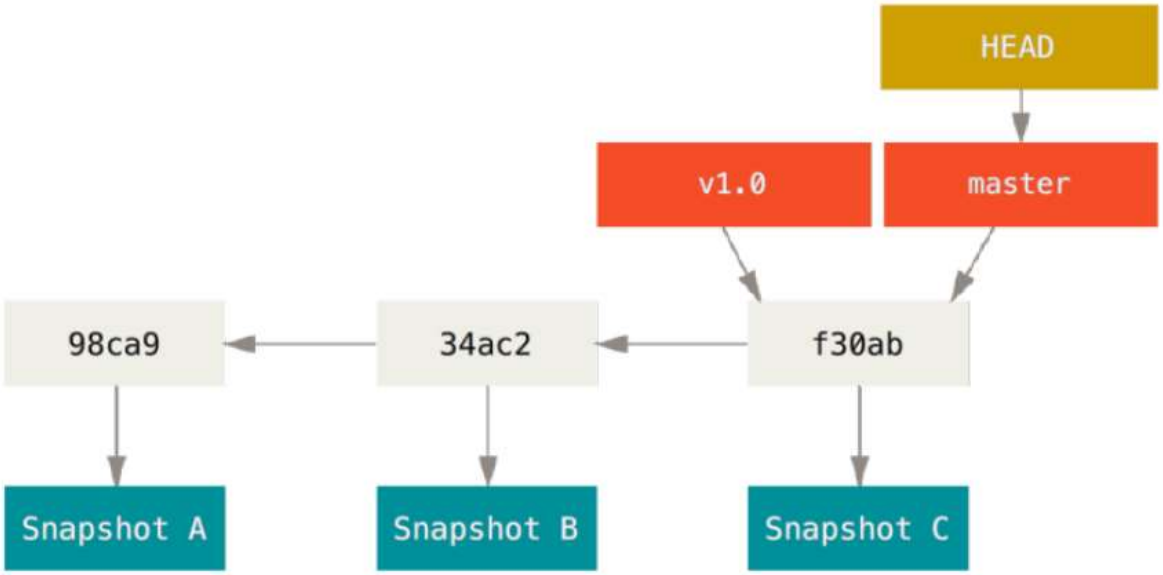
Git Feature Branching

- A branch in Git is simply a lightweight movable pointer to one of these commits.
- The default branch name in Git is master.
- As you start making commits, you're given a master branch that points to the last commit you made.
- Every time you commit, it moves forward automatically.



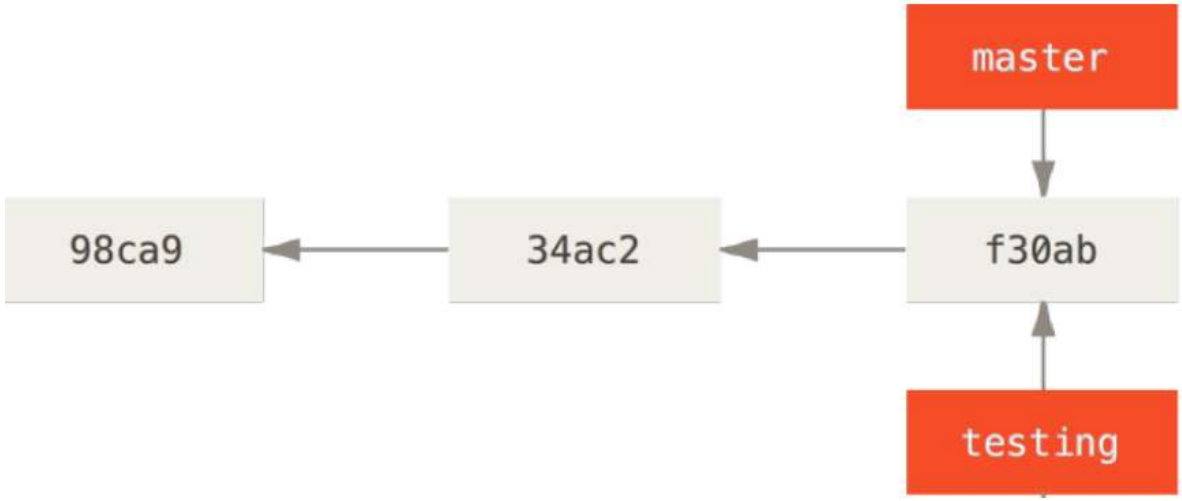
Git Feature Branching

- Master Branch



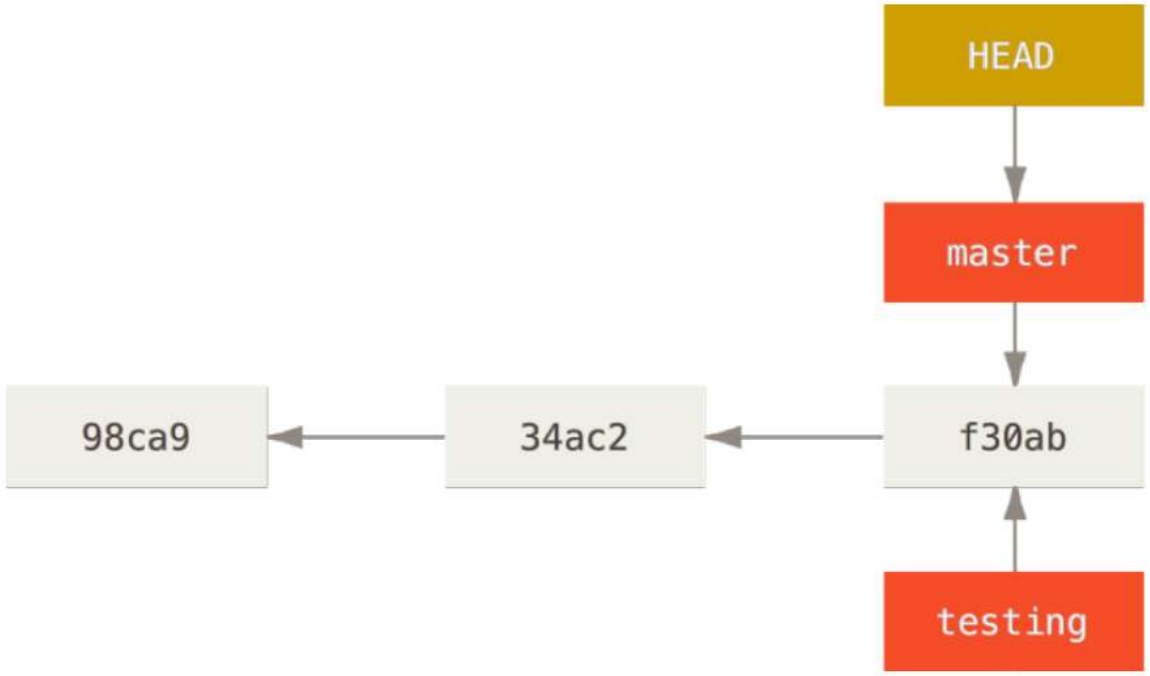
Git Feature Branching

- What happens if you create a new branch?
- Doing so creates a new pointer for you to move around.
- Let's say you create a new branch called testing.
- $ git branch testing



Git Feature Branching

- How does Git know what branch you're currently on? It keeps a special pointer called HEAD.
- In Git, this is a pointer to the local branch you're currently on. In this case, you're still on master.
- The git branch command only created a new branch – it didn't switch to that branch.



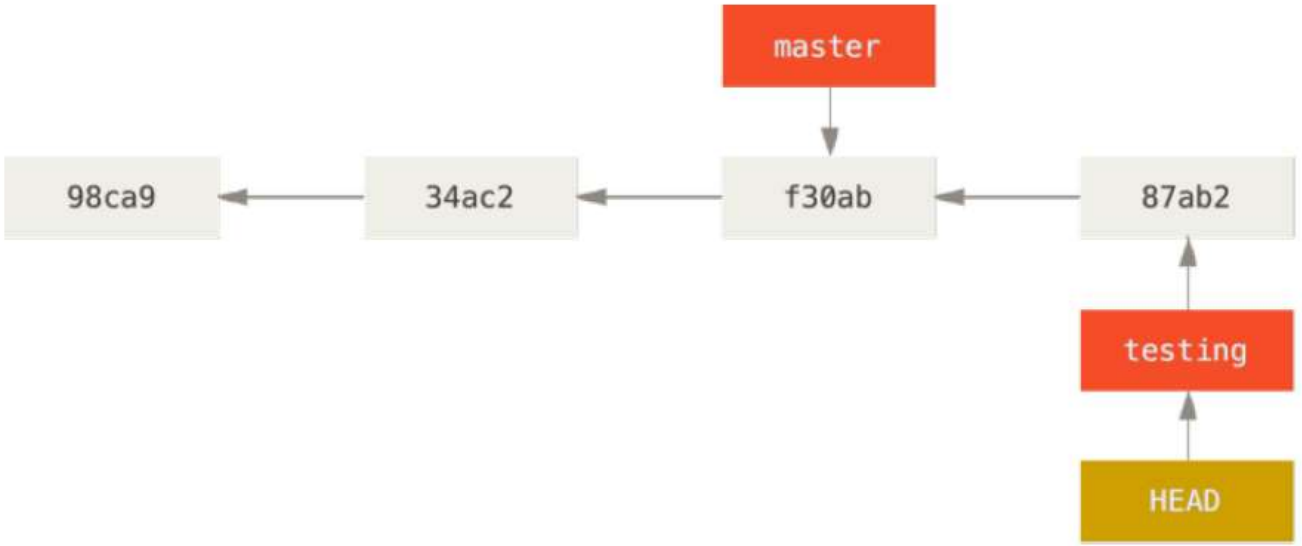
Git Feature Branching

Switch Branch:

$ git checkout testing -> moves HEAD to point to the testing branch

Git Feature Branching

- $ vim test.rb
- $ git commit -a -m 'made a change'



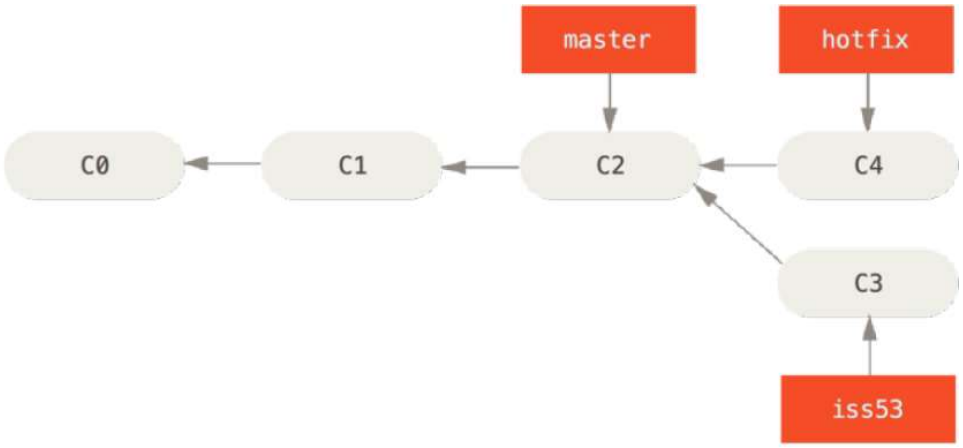
- Testing branch has moved forward, but master branch still points to the commit were on when you ran git checkout to switch branches

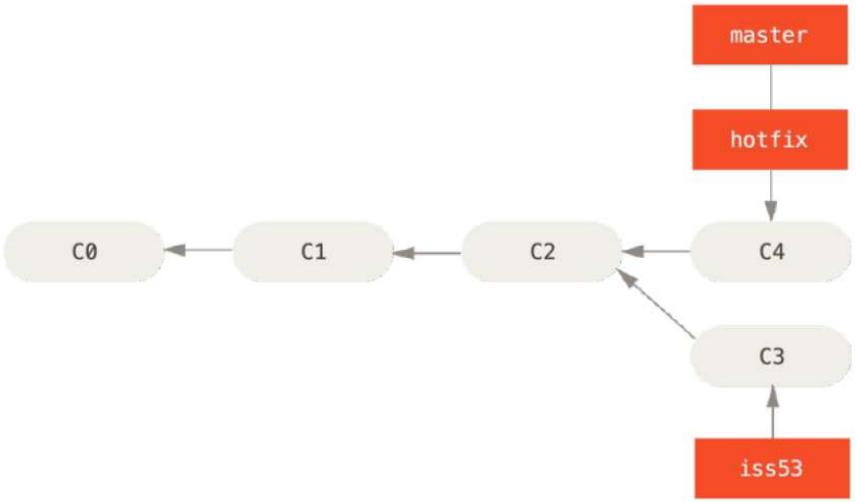
Git Feature Branching

- To work on new feature, Bug create new branch and checkout
- $ git checkout -b iss53
- You can run your tests, make sure the hotfix is what you want, and merge it back into your master branch to deploy to production

  $ git checkout master

  $ git merge hotfix

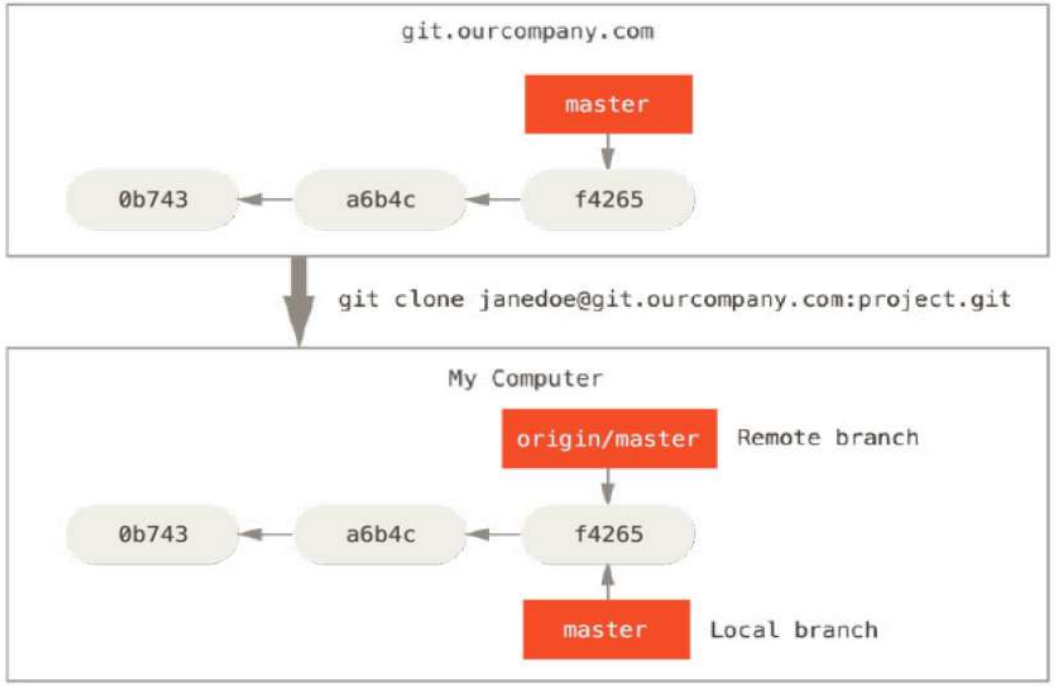




$git branch -d hotfix

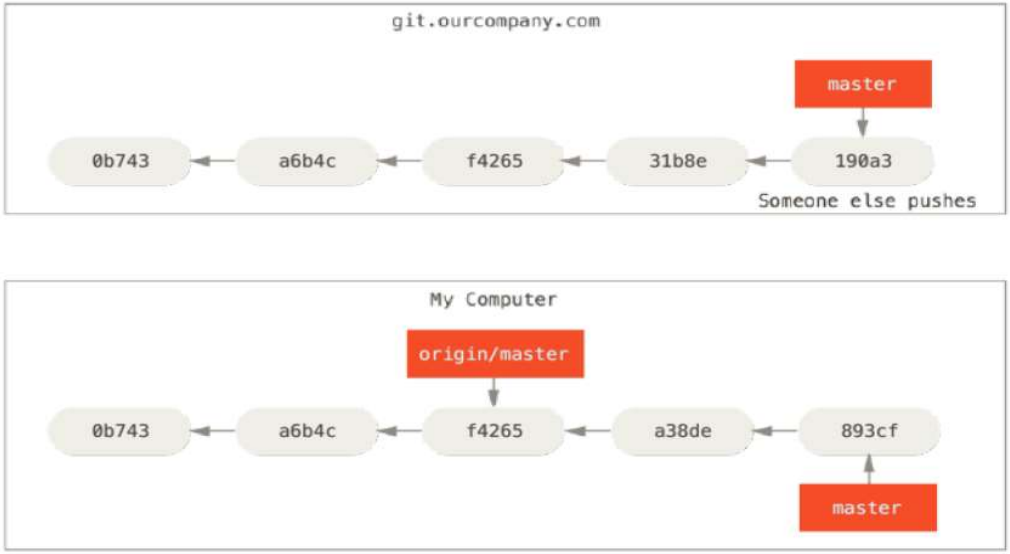
Remote Branches

- Remote references are references (pointers) in your remote repositories, including branches, tags, and so on. You can get a full list of remote references explicitly with git ls-remote



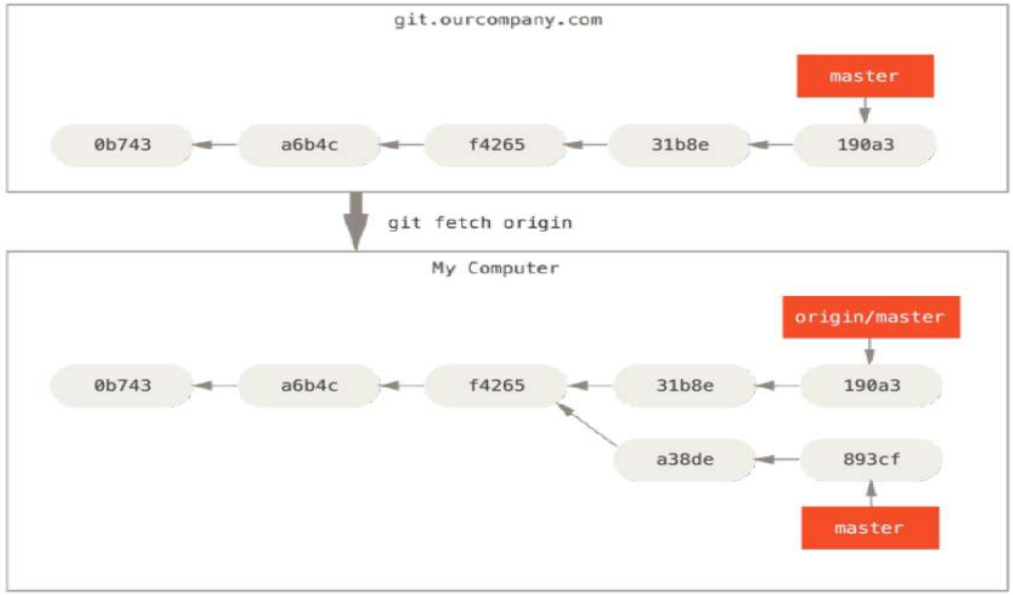
Remote Branches

- If you do some work on your local master branch, and, in the meantime, someone else pushes to git.ourcompany.com and updates its master branch, then your histories move forward differently.
- Also, as long as you stay out of contact with your origin server, your origin/master pointer doesn't move.



Remote Branches

- To synchronize your work, you run a git fetch origin command.
- This command looks up which server “origin” is (in this case, it’s git.ourcompany.com), fetches any data from it that you don’t yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position.



Remote Branches - Pushing

- When you want to share a branch with the world, you need to push it up to a remote that you have write access to.
- Your local branches aren't automatically synchronized to the remotes you write to – you have to explicitly push the branches you want to share.
- you can use private branches for work you don't want to share, and push up only the topic branches you want to collaborate on.

- If you have a branch named serverfix that you want to work on with others, you can push it up the same way you pushed your first branch.

- Run git push <remote> <branch>
- $git push origin serverfix
- git push origin --delete serverfix -> Delete remote branch serverfix

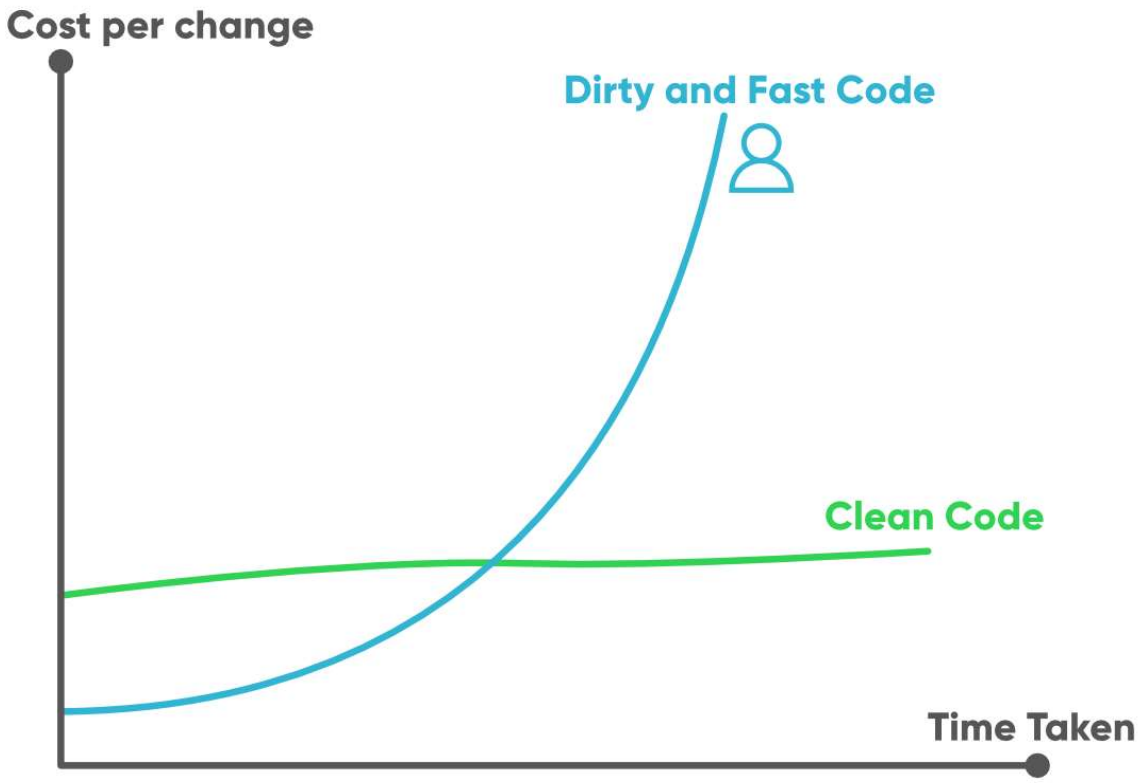
Tips To Write Clean And Better Code

- As a software engineer or **software developer**, you are expected to write *good software*.
- So, the question is what makes good software?
- Good software can be judged by reading some pieces of code written in the project.
- If the code is ***easy to understand*** and ***easy to change*** then definitely it's a good software and developers love to work on that.

Tips To Write Clean And Better Code

- It's a common thing in development that nobody wants to continue a project with horrible or messy code (It becomes a nightmare sometimes...).
- Sometimes developers avoid writing clean code due to deadline pressure.
- They rush to go faster but what happens actually is they end up going slower.
- It creates more bugs which they need to fix later going back on the same piece of code.
- This process takes much more time than the amount of time spent on writing the code.
- A study has revealed that *the ratio of time spent reading code versus writing is well over 10 to 1.*

Tips To Write Clean And Better Code



Tips To Write Clean And Better Code

- It doesn't matter if you are a beginner or an experienced programmer, you should always try to become a *good programmer* (not just a programmer...).
- Remember that *you are responsible for the quality of your code* so make your program good enough so that other developers can understand and they don't mock you every time to understand the messy code you wrote in your project.
- Before we discuss the art of writing clean and better code let's see some characteristics of it...

Tips To Write Clean And Better Code

- What Makes a Code "a Clean Code"?
- Clean code should be ***readable***. If someone is reading your code, they should have a feeling of reading poetry or prose.
- Clean code should be ***elegant***. It should be pleasing to read and it should make you smile.
- Clean code should be ***simple*** and easy to understand. It should follow the ***single responsibility principle (SRP)***.
- Clean code should be ***easy to understand, easy to change,*** and ***easy to take care of***.
- Clean code should ***run all the tests***.

Tips To Write Clean And Better Code

- Suggestions for Writing Cleaner and More Effective Code
- Writing Clean Code is important for better collaboration among developers.
- You can easily debug the issues in clean and readable code.
- Also, a clean and readable code reduce the maintenance cost drastically and enhance the overall quality of software.
- So, below are some of the best practices and tips that software developers follow to write clean and better code.

"Clean code is simple and direct. Clean code reads like a well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control." **-Grady Booch** (*Author of Object-Oriented Analysis and Design with Applications*)

Suggestions for Writing Cleaner and More Effective Code

- Writing Clean Code is important for better collaboration among developers.
- You can easily debug the issues in clean and readable code.
- Also, a clean and readable code reduce the maintenance cost drastically and enhance the overall quality of software.
- So, below are some of the **best practices and tips that software developers follow to write clean and better code.**

Suggestions for Writing Cleaner and More Effective Code

1. Use Meaningful Names

- You will be writing a lot of names for **variables**, **functions**, **classes**, **arguments**, **modules**, **packages**, **directories**, and things like that.

- Make a habit to use meaningful names in your code.

- Whatever names you mention in your code it should fulfill three purposes...

- *what it does*, *why it exists,* and *how it is used*.

- For Example:

- int user_no; // number of users.

Suggestions for Writing Cleaner and More Effective Code

- In the previous example, you need to mention a comment along with the name declaration of a variable which is not a characteristic of a good code.
- The name that you specify in your code should reveal its intent.
- It should specify the purpose of a variable, function, or method.
- So, for the above example, a better variable name would be:- *int* user_no.
- It may take some time to choose meaningful names but it makes your code much cleaner and easy to read for other developers as well as for yourself.
- Also, try to limit the names to three or four words.

Suggestions for Writing Cleaner and More Effective Code

- 2. Single Responsibility Principle (SRP)
- Classes, Functions, or methods are a good way to organize the code in any programming language so when you are writing the code you really need to take care that how to write a function that communicates its intent.

- Most beginners do this mistake they write a function that can handle and do almost everything (perform multiple tasks).

- It makes your code more confusing for developers and creates problems when they need to fix some bugs or find some piece of code.
- So, when you are writing a function you should remember two things to make your function clean and easy to understand...
- They should be small.
- They should do only one thing and they should do it well.

Suggestions for Writing Cleaner and More Effective Code

- The above two points clearly mention that your function should follow the **single responsibility principle**.
- This means it shouldn't have a nested structure or it should not have more than two indent levels.
- Following this technique make your code much more readable and other developers can easily understand or implement another feature if your function fulfills a specific task.
- Also, make sure that your function should not have more than three arguments.

Suggestions for Writing Cleaner and More Effective Code

- More arguments perform more tasks so try to keep the arguments as less as possible.
- Passing more than three arguments makes your code confusing, quite large, and hard to debug if any problem would be there.
- If your function has a try/catch/finally statement then make a separate function containing just the try-catch-finally statements.
- Take care of your function name as well.
- Use a descriptive name for your function which should clearly specify what it does.

Suggestions for Writing Cleaner and More Effective Code

Example:

```
function subtract(x, y) {
return x - y;
}
function add(x, y) {
return x + y;
}
```

In the above example, the function name clearly shows that its purpose is to perform subtraction for two numbers, also it has only two arguments.

Suggestions for Writing Cleaner and More Effective Code

- 3. Avoid Writing Unnecessary Comments

- It's a common thing that developers to use comments to specify the purpose of a line in their code.

- It's true that comments are really helpful in explaining the code and what it does but it also requires more maintenance of your code.

- In development, code move here and there but if the comment remains in the same place, then it can create a big problem.

Suggestions for Writing Cleaner and More Effective Code

- It can create confusion among developers and they get distracted as well due to useless comments.

- It's not like you shouldn't use comments at all, sometimes it is important, for example...if you are dealing with third-party API where you need to explain some behavior there you can use comments to explain your code but don't write comments where it's not necessary.

- Today modern programming languages syntax are English-like through and that's good enough to explain the purpose of a line in your code.

- To avoid comments in your code, use meaningful names for variables, functions, or files.

Good code is its own best documentation. As you're about to add a comment, ask yourself, "How can I improve the code so that this comment isn't needed?" Improve the code and then document it to make it even clearer. -Steve McConnell

Suggestions for Writing Cleaner and More Effective Code

- 4. Write Readable Code For People
- A lot of people especially beginners make mistakes while writing code they write everything in a single line and don't give proper whitespace, indentation, or line breaks in their code.
- It makes their code messy and difficult to maintain.
- There's always a chance that another human will get to your code and they will have to work with it.
- It wastes other developers' time when they try to read and understand the messy code.

Suggestions for Writing Cleaner and More Effective Code

- So always pay attention to the formatting of your code.
- You will also save your time and energy when you will get back to your own code after a couple of days to make some changes.
- So, make sure that your code should have proper indentation, space, and line breaks to make it readable to other people.
- The coding style and formatting affect the maintainability of your code.
- A **software developer** is always remembered for the coding style he/she follows in his/her code.

Code formatting is about communication, and communication is the professional developer's first order of business. -Robert C. Martin

Suggestions for Writing Cleaner and More Effective Code

```
// Bad Codeclass CarouselRightArrow extends Component{render(){return (<a href=""
className="carousel__arrow carousel__arrow--left" onClick={this.props.onClick}> <span
className="fa fa-2x fa-angle-left"/> </a> );}};
// Good Codeclass CarouselRightArrow extends Component {
        render() {
            return (
                <a
                href="#"
                className="carousel__arrow carousel__arrow--left"
        onClick={this.props.onClick}>
                 <span className="fa fa-2x fa-angle-left" />
                </a>
                );
            }
};
```

Suggestions for Writing Cleaner and More Effective Code

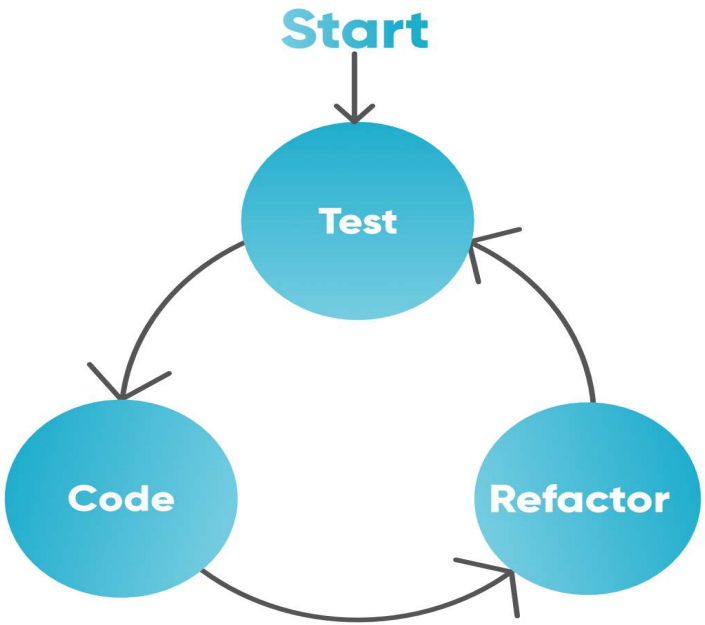
- 5. Write Unit Tests
- Writing Unit tests is very important in development.
- It makes your code **clean**, **flexible**, and **maintainable**.
- Making changes in code and reducing bugs becomes easier.
- There is a process in software development which is called **Test Driven Development** (**TDD**) in which requirements are turned into some specific test cases then the software is improved to pass new tests.
- According to Robert C. Martin, the three laws of TDD demonstrates that...

Suggestions for Writing Cleaner and More Effective Code

1.You are not allowed to write any production code unless it is to make a failing unit test pass.

2.You are not allowed to write any more of a unit test than is sufficient to fail, and compilation failures are failures.

3.You are not allowed to write any more production code than is sufficient to pass the one failing unit test.



After I jumped on board the unit testing bandwagon, the quality of what I deliver has increased to the point that the amount of respect I get from my colleagues makes me feel awkward. They think I'm blessed or something. I'm not gifted or blessed or even that talented, I just test my code. – Justin Yancey, Senior Systems Development Engineer, Amazon

Suggestions for Writing Cleaner and More Effective Code

- 6. Be Careful With Dependencies
- In software development, you really need to be careful about your dependencies.
- If possible, your dependencies should always be a singular direction.
- It means....let's say we have a kitchen class that is dependent on a dishwasher class.
- As long as the dishwasher doesn't also dependent on the kitchen class this is a one-directional dependency.
- The kitchen class is just using the dishwasher but the dishwasher doesn't really care and anyone can use it.

Suggestions for Writing Cleaner and More Effective Code

- It doesn't have to be a kitchen.
- This example of one-directional dependency is easier to manage however it's impossible to always have one-directional dependency but we should try to have as many as possible.
- When dependency goes in multiple directions, things get much more complicated.
- In bidirectional dependency both entities depend on each other, so they have to exist together even though they are separate.
- It becomes hard to update some systems when their dependencies don't form a singular direction. So always be careful about managing your dependencies.

Suggestions for Writing Cleaner and More Effective Code

- 7. Make Your Project Well Organized
- This is a very common problem in software development that we add and delete so many files or directories in our project and sometimes it becomes complicated and annoying for other developers to understand the project and work on that.

- We agree that you can not design a perfect organization of folders or files on day one but later on, when your project becomes larger you really need to be careful about the organization of your folder, files, and directories.

- A well-structured folder and file make everything clear and it becomes easier to understand a complete project, search some specific folder and make changes in it.

- So, make sure that your directory or folder structure should be in an organized manner (The same applies to your code as well).

Benefits of Clean Code

- Better Use of Your Time
- Easier Onboarding for New Team Members
- Easier Debugging
- More Efficient Maintenance
- Simplicity

Suggestions for Writing Cleaner and More Effective Code

- Conclusion

- For the purpose of creating high-quality software that is ultimately **simple to maintain**, **test**, and **update**, cleaner and better code must be written.

- In a Project different developers have to add multiple features on a shared project, so it become important for a developer to write , more clean and readable code that can be easily understandable by its naming convention itself.

- Developers can produce software that is reliable, scalable, and simple to use by adhering to the 7 recommended practices listed above and writing the code in a **clear**, **organized**, **legible**, and **efficient manner**.

QUESTIONS AND DISCUSSION
THANK YOU