GIT

Git is a free, open source distributed version control system tool designed to handle everything from small to very large projects with speed and efficiency. It was created by Linus Torvalds in 2005 to develop Linux Kernel. Git has the functionality, performance, security and flexibility that most teams and individual developers need.

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

What is Git – Why Git Came Into Existence?

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

Version control is a system that records changes to a file or set of files over time so that you can recall specific **versions** later

There are two types of VCS:

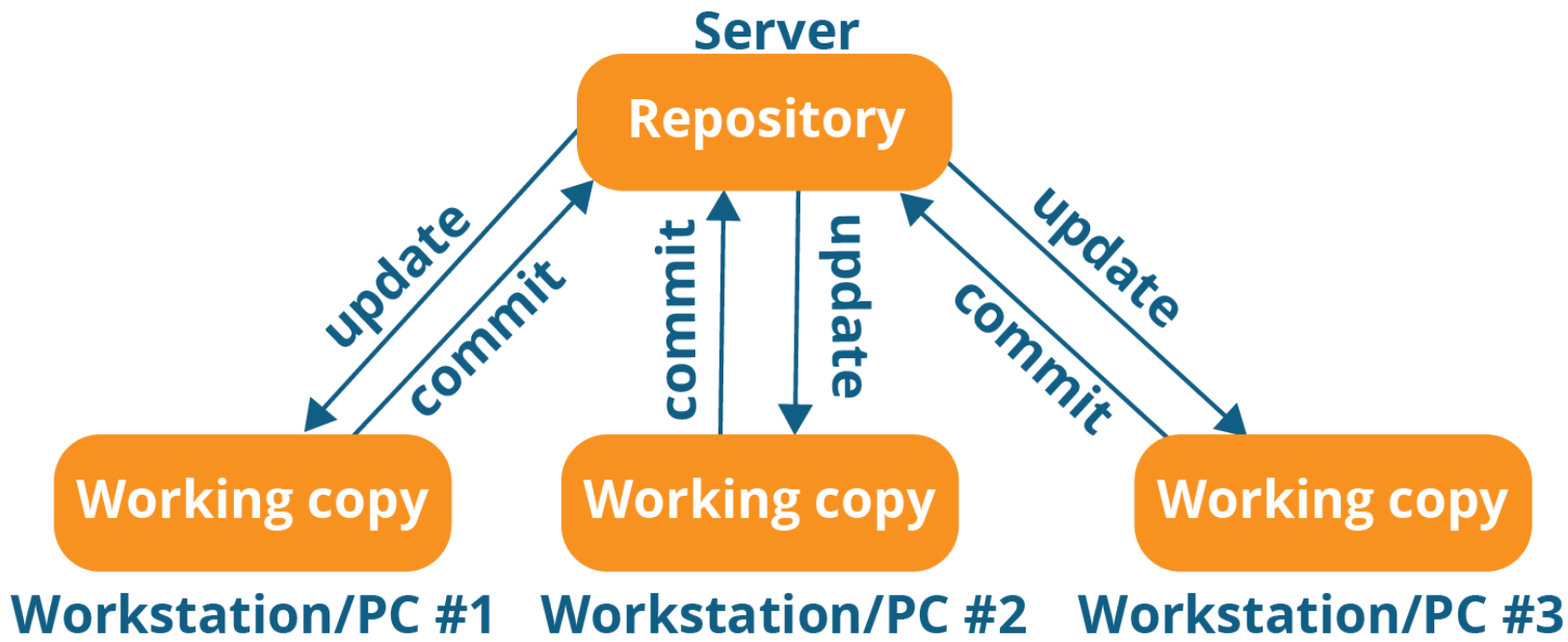
- Centralized Version Control System (CVCS)
- Distributed Version Control System (DVCS)

Centralized VCS

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. It works on a single repository to which users can directly access a central server.

Please refer to the diagram below to get a better idea of CVCS:

Centralized version control system



The repository in the above diagram indicates a central server that could be local or remote which is directly connected to each of the programmer's workstation.

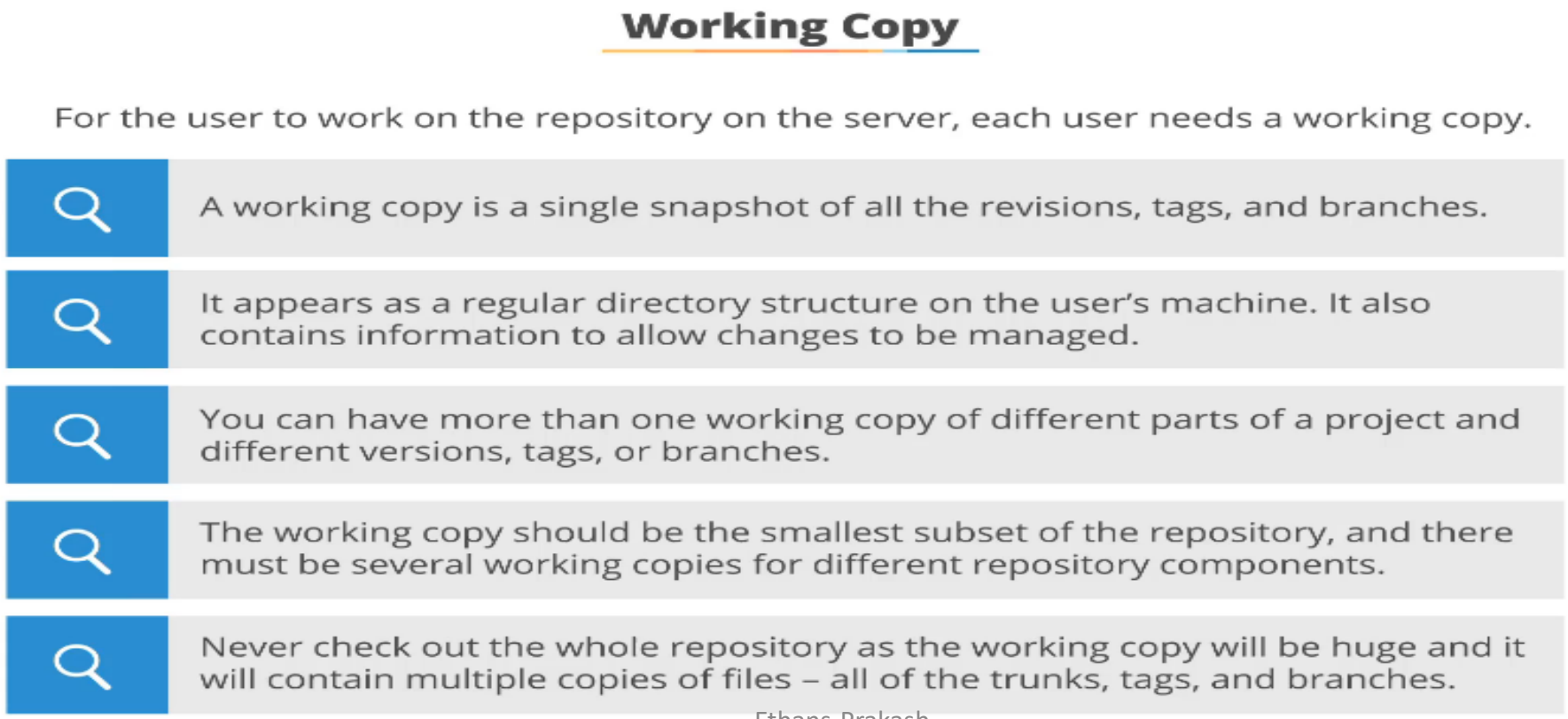
Every programmer can extract or **update** their workstations with the data present in the repository or can make changes to the data or **commit** in the repository. Every operation is performed directly on the repository.

Even though it seems pretty convenient to maintain a single repository, it has some major drawbacks. Some of them are:

- It is not locally available; meaning you always need to be connected to a network to perform any action.
- Since everything is centralized, in any case of the central server getting crashed or corrupted will result in losing the entire data of the project.

EXP: SVN tool is Centralize version control management tool.

This is when Distributed VCS comes to the rescue.

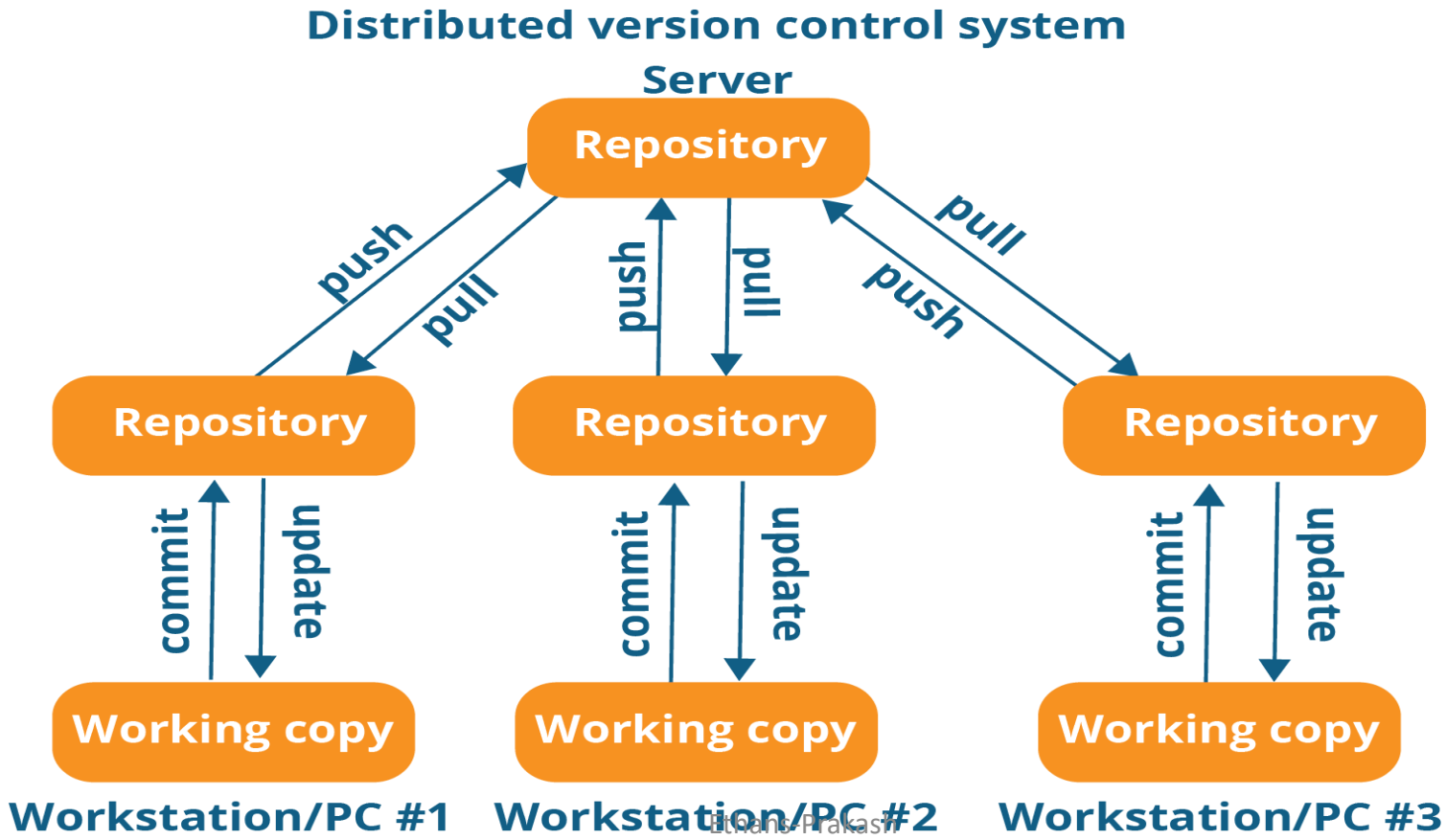


Distributed VCS

These systems do not necessarily rely on a central server to store all the versions of a project file.

In Distributed VCS, every contributor has a local copy or “clone” of the main repository i.e. everyone maintains a local repository of their own which contains all the files and metadata present in the main repository.

You will understand it better by referring to the diagram below:



As you can see in the above diagram, every programmer maintains a local repository on its own, which is actually the copy or clone of the central repository on their hard drive. They can commit and update their local repository without any interference.

They can update their local repositories with new data from the central server by an operation called "**pull**" and affect changes to the main repository by an operation called "**push**" from their local repository.

The act of cloning an entire repository into your workstation to get a local repository gives you the following advantages:

- All operations (except push & pull) are very fast because the tool only needs to access the hard drive, not a remote server. Hence, you do not always need an internet connection.
- Committing new change-sets can be done locally without manipulating the data on the main repository. Once you have a group of change-sets ready, you can push them all at once.
- Since every contributor has a full copy of the project repository, they can share changes with one another if they want to get some feedback before affecting changes in the main repository.
- If the central server gets crashed at any point of time, the lost data can be easily recovered from any one of the contributor's local repositories.

Features Of Git:

Free-and-open-source:
Git is released under GPL's (General Public License) open source license. You don't need to purchase Git. It is absolutely free. And since it is open source, you can modify the source code as per your requirement.

Speed:
Since you do not have to connect to any network for performing all operations, it completes all the tasks really fast. Performance tests done by Mozilla showed it was an order of magnitude faster than other version control systems. Fetching version history from a locally stored repository can be one hundred times faster than fetching it from the remote server. The core part of Git is written in C, which avoids runtime overheads associated with other high level languages.

Scalable:

Git is very scalable. So, if in future , the number of collaborators increase Git can easily handle this change. Though Git represents an entire repository, the data stored on the client's side is very small as Git compresses all the huge data through a lossless compression technique.

Reliable :

Since every contributor has its own local repository, on the events of a system crash, the lost data can be recovered from any of the local repositories. You will always have a backup of all your files.

Secure:

Git uses the ***SHA1*** (Secure Hash Function) to name and identify objects within its repository. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. The Git history is stored in such a way that the ID of a particular version (a *commit* in Git terms) depends upon the complete development history leading up to that commit. Once it is published, it is not possible to change the old versions without it being noticed.

Economical :

In case of CVCS, the central server needs to be powerful enough to serve requests of the entire team. For smaller teams, it is not an issue, but as the team size grows, the hardware limitations of the server can be a performance bottleneck. In case of DVCS, developers don't interact with the server unless they need to push or pull changes. All the heavy lifting happens on the client side, so the server hardware can be very simple indeed.

Supports non-linear development:

Git supports rapid branching and merging, and includes specific tools for visualizing and navigating a non-linear development history. A core assumption in Git is that a change will be merged more often than it is written, as it is passed around various reviewers. Branches in Git are very lightweight. A branch in Git is only a reference to a single commit. With its parental commits, the full branch structure can be constructed.

Easy Branching:

Branch management with Git is very simple. It takes only few seconds to create, delete, and merge branches. Feature branches provide an isolated environment for every change to your code.

When a developer wants to start working on something, no matter how big or small, they create a new branch. This ensures that the master branch always contains production-quality code.

Distributed development:

Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another. These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch.

Compatibility with existing systems or protocol:

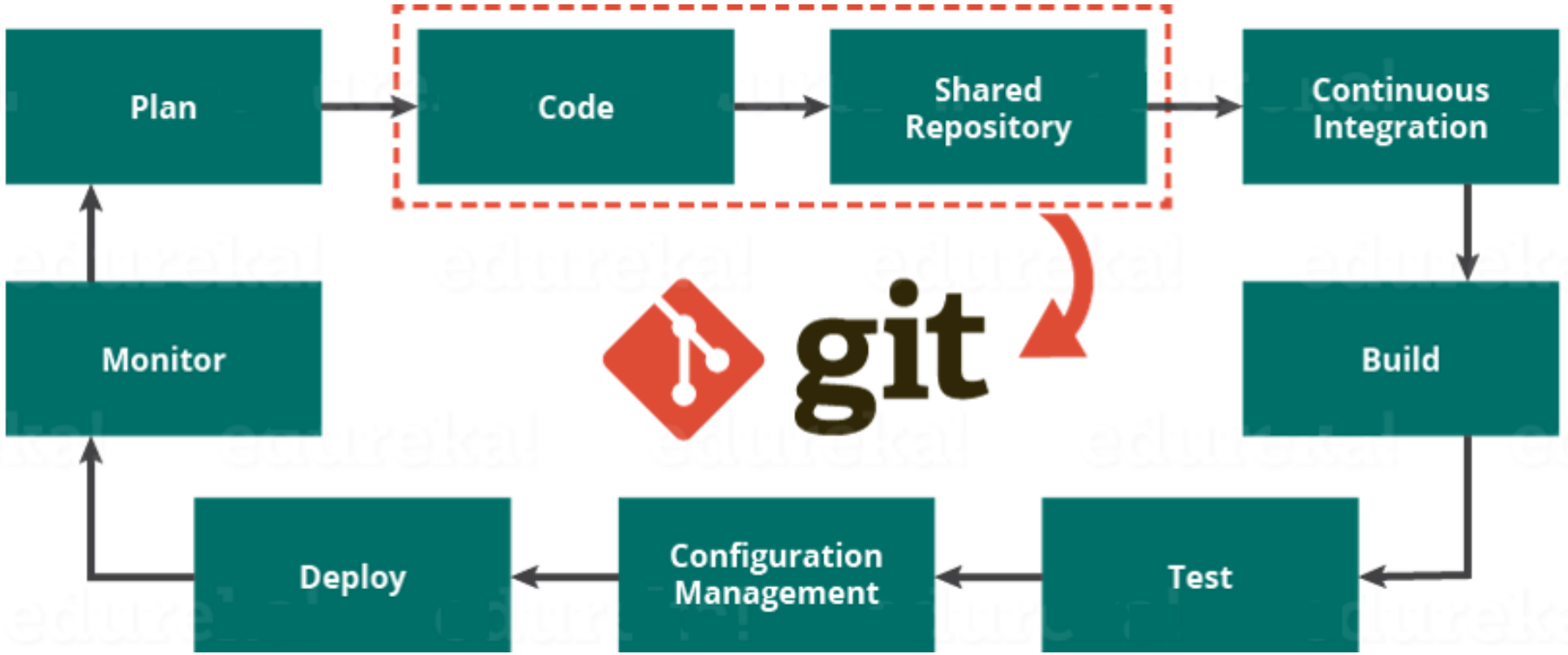
Repositories can be published via http, ftp or a Git protocol over either a plain socket, or ssh. Git also has a Concurrent Version Systems (CVS) server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories. Apache SubVersion (SVN) and SVK repositories can be used directly with Git-SVN.

Role Of Git In DevOps?

Now that you know what is Git, you should know Git is an integral part of DevOps.

DevOps is the practice of bringing agility to the process of development and operations. It’s an entirely new ideology that has swept IT organizations worldwide, boosting project life-cycles and in turn increasing profits. DevOps promotes communication between development engineers and operations, participating together in the entire service life-cycle, from design through the development process to production support.

The diagram below depicts the Devops life cycle and displays how Git fits in Devops.

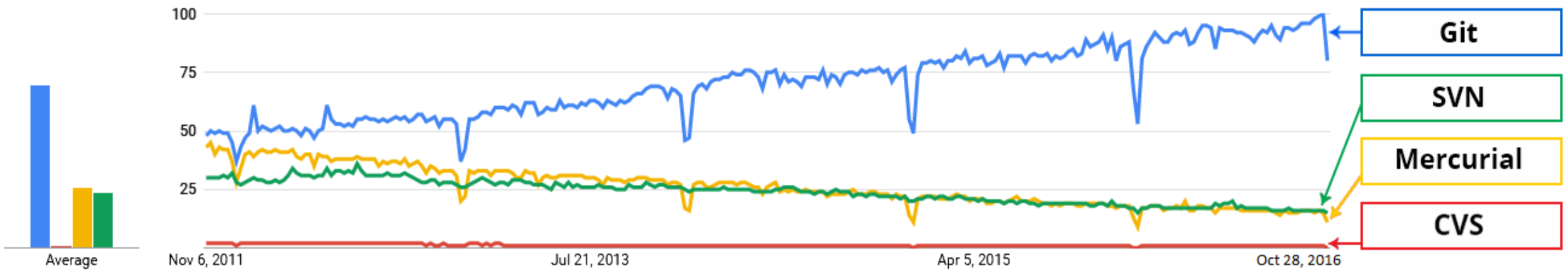


The diagram above shows the entire life cycle of Devops starting from planning the project to its deployment and monitoring. Git plays a vital role when it comes to managing the code that the collaborators contribute to the shared repository. This code is then extracted for performing continuous integration to create a build and test it on the test server and eventually deploy it on the production

Tools like Git enable communication between the development and the operations team. When you are developing a large project with a huge number of collaborators, it is very important to have communication between the collaborators while making changes in the project. Commit messages in Git play a very important role in communicating among the team. The bits and pieces that we all deploy lies in the Version Control system like Git. To succeed in DevOps, you need to have all of the communication in Version Control. Hence, Git plays a vital role in succeeding at DevOps.

Companies Using Git:

Git has earned way more popularity compared to other version control tools available in the market like Apache Subversion(SVN), Concurrent Version Systems(CVS), Mercurial etc. You can compare the interest of Git by time with other version control tools with the graph collected from *Google Trends* below:



In large companies, products are generally developed by developers located all around the world. To enable communication among them, Git is the solution.

Some companies that use Git for version control are: Facebook, Yahoo, Zynga, Quora, Twitter, eBay, Salesforce, Microsoft and many more.

Lately, all of Microsoft's new development work has been in Git features. Microsoft is migrating .NET and many of its open source projects on GitHub which are managed by Git.

One of such projects is the LightGBM. It is a fast, distributed, high performance gradient boosting framework based on decision tree algorithms which is used for ranking, classification and many other machine learning tasks.

Here, Git plays an important role in managing this distributed version of LightGBM by providing speed and accuracy.

git init

This command turns a directory into an empty Git repository. This is the first step in creating a repository. After running git init, adding and committing files/directories is possible.

Usage:

change directory to codebase

$ cd /file/path/to/code

make directory a git repository

$ git init

In Practice:

change directory to codebase

$ cd /Users/prakash-pc/Documents/website

make directory a git repository

$ git init

Initialized empty Git repository in /Users/prakash-pc /Documents/website/.git/

git config --global user.email "prakash@gmail.com"

git add

Adds files in the to the staging area for Git. Before a file is available to commit to a repository, the file needs to be added to the Git index (staging area). There are a few different ways to use git add, by adding entire directories, specific files, or all unstaged files.

Usage:

$ git add <file or directory name>

In Practice:

To add all files not staged:

$ git add

To stage a specific file:

$ git add index.html

To stage an entire directory:

$ git add css

git commit

Record the changes made to the files to a local repository. For easy reference, each commit has a unique ID.

It’s best practice to include a message with each commit explaining the changes made in a commit. Adding a commit message helps to find a particular change or understanding the changes.

Usage:

Adding a commit with message

$ git commit -m "Commit message in quotes"

In Practice:

$ git commit -m "My first commit message"

[SecretTesting 0254c3d] My first commit message

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 homepage/index.html

git status

This command returns the current state of the repository. git status will return the current working branch. If a file is in the staging area, but not committed, it shows with git status. Or, if there are no changes it'll return nothing to commit, working directory clean.

In Practice:

Message when files have not been staged (git add)

$ git status

On branch SecretTesting

Untracked files:

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

homepage/index.html

Message when files have been not been committed (git commit)

$ git status

On branch SecretTesting

Your branch is up-to-date with 'origin/SecretTesting'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: homepage/index.html

Message when all files have been staged and committed

$ git status

On branch SecretTesting

nothing to commit, working directory clean

git config

With Git, there are many configurations and settings possible. *git config* is how to assign these settings. Two important settings are user user.name and user.email. These values set what email address and name commits will be from on a local computer. With *git config*, a *--global* flag is used to write the settings to all repositories on a computer. Without a *--global* flag settings will only apply to the current repository that you are currently in.

There are many other variables available to edit in *git config*. From editing color outputs to changing the behavior of *git status*

Usage:

$ git config <setting> <command>

In Practice:

Running git config globally

$ git config --global user.email "my@emailaddress.com"

$ git config --global user.name “prakash"

Running git config on the current repository settings

$ git config user.email "my@emailaddress.com"

$ git config user.name “prakash"

git branch

To determine what branch the local repository is on, add a new branch, or delete a branch.

Usage:

Create a new branch

$ git branch <branch_name>

List all remote or local branches

$ git branch -a

Delete a branch

$ git branch -d <branch_name>

In Practice:

Create a new branch

$ git branch new_feature

List branches

$ git branch -a

* SecretTesting

new_feature

remotes/origin/stable

remotes/origin/staging

remotes/origin/master -> origin/SecretTesting

Delete a branch

$ git branch -d new_feature

Deleted branch new_feature (was 0254c3d).

git checkout

To start working in a different branch, use git checkout to switch branches.

Usage:

```
# Checkout an existing branch
$ git checkout <branch_name>

# Checkout and create a new branch with that name
$ git checkout -b <new_branch>
```

In Practice:

```
# Switching to branch 'new_feature'
$ git checkout new_feature
Switched to branch 'new_feature'

# Creating and switching to branch 'staging'
$ git checkout -b staging
Switched to a new branch 'staging'
```

git merge

Integrate branches together. git merge combines the changes from one branch to another branch. For example, merge the changes made in a staging branch into the stable branch.

Usage:

Merge changes into current branch

$ git merge <branch_name>

In Practice:

Merge changes into current branch

$ git merge new_feature

Updating 0254c3d..4c0f37c

Fast-forward

homepage/index.html | 297 ++

1 file changed, 297 insertions(+)

create mode 100644 homepage/index.html

git remote

To connect a local repository with a remote repository. A remote repository can have a name set to avoid having to remember the URL of the repository.

Usage:

Add remote repository

$ git remote <command> <remote_name> <remote_URL>

List named remote repositories

$ git remote -v

In Practice:

Adding a remote repository with the name of beanstalk

$ git remote add origin git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git

List named remote repositories

$ git remote -v

origin git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git (fetch)

origin git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git (push)

git clone

To create a local working copy of an existing remote repository, use git clone to copy and download the repository to a computer. Cloning is the equivalent of git init when working with a remote repository. Git will create a directory locally with all files and repository history.

Usage:

$ git clone <remote_URL>

In Practice:

$ git clone git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git

Cloning into 'repository_name'...

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 5 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (5/5), 3.08 KiB | 0 bytes/s, done.

Checking connectivity... done.

git pull

To get the latest version of a repository run git pull. This pulls the changes from the remote repository to the local computer.

Usage:

$ git pull <branch_name> <remote_URL/remote_name>

In Practice:

```
# Pull from named remote
$ git pull origin staging
From account_name.git.beanstalkapp.com:/account_name/repository_name
 * branch            staging    -> FETCH_HEAD
 * [new branch]      staging    -> origin/staging
Already up-to-date.

# Pull from URL (not frequently used)
$ git pull git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git staging
From account_name.git.beanstalkapp.com:/account_name/repository_name
 * branch            staging    -> FETCH_HEAD
 * [new branch]      staging    -> origin/staging
Already up-to-date.
```

git push

Sends local commits to the remote repository. git push requires two parameters: the remote repository and the branch that the push is for.

Usage:

$ git push <remote_URL/remote_name> <branch>

Push all local branches to remote repository

$ git push —all

In Practice:

Push a specific branch to a remote with named remote

$ git push origin staging

Counting objects: 5, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (5/5), 734 bytes | 0 bytes/s, done.

Total 5 (delta 2), reused 0 (delta 0)

To git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git

  ad189cb..0254c3d SecretTesting -> SecretTesting

Push all local branches to remote repository

$ git push --all

Counting objects: 4, done.

git stash

To save changes made when they're not in a state to commit them to a repository. This will store the work and give a clean working directory.
For instance, when working on a new feature that's not complete, but an urgent bug needs attention.

Usage:

Store current work with untracked files

$ git stash -u

Bring stashed work back to the working directory

$ git stash pop

In Practice:

Store current work

$ git stash -u

Saved working directory and index state WIP on SecretTesting: 4c0f37c Adding new file to branch

HEAD is now at 4c0f37c Adding new file to branch

Bring stashed work back to the working directory

$ git stash pop

git log

To show the chronological commit history for a repository. This helps give context and history for a repository. git log is available immediately on a recently cloned repository to see history.

Usage:

Show entire git log

$ git log

Show git log with date pameters

$ git log --<after/before/since/until>=<date>

Show git log based on commit author

$ git log --<author>="Author Name"

In Practice:

Show entire git log

$ git log

Ignoring Files and Folders using GitIgnore:

GitIgnore file is a file that specifying the files or folders that we want to ignore.

- The first one is specifying by the specific filename. Here is an example, let's say we want to ignore a file called **readme.txt**, then we just need to write **readme.txt** in the **.gitignore file**
- The second one we can also write the name of the extension. For example, we are going to ignore **all .txt files**, then write ***.txt**
- There is also a method to ignore a whole folder. Let's say we want to ignore folder named **test**. Then we can just write **test/** in the file.

In Practice:

vi index1.html

git status (you can see file in working area)

now if you want to ignore it then

vi .gitignore

then add index1.html (refer below screen shot then save it)

git status

```
# Ignore file name called index1.html
index1.html

# Ignore folder name devops-demo
devops-demo/

# Ignore all .sh file
*.sh
```

How to move files from staging to working area:

In Practice:

vi index1.html

git add index1.html

git status

git reset HEAD <file name>

When you mistyped the last commit's message or forgot to add a change

- git commit --amend -m "New commit message"

How to commit files without adding to staging area (file should be committed once

- Note: not for new files, it will not staged file new files)

- git commit -a -m “your message”

How to merge specific a commit in git:

- You want to merge a particular commit from the current branch to the other branch of your requirement. Merging a specific commit from one branch to another can be achieved by **cherry-pick** command**.**

In Practice: Create a file (index1.html) on master branch and commit it. Now create a new branch (b1) from master, then create second file (index2.html) on b1 and commit it, create one more file (index3) on b1 and commit it. Now if you want only index3 to get merge with master then **switch to master** and use git cherry-pick command.

- **Syntax: git cherry-pick <commit ID>**

git rm

Remove files or directories from the working index (staging area). With git rm, there are two options to keep in mind: force and cached. Running the command with force deletes the file. The cached command removes the file from the working index. When removing an entire directory, a recursive command is necessary.

Usage:

To remove a file from the working index (cached):

$ git rm --cached <file name>

To delete a file (force):

$ git rm -f <file name>

To remove an entire directory from the working index (cached):

$ git rm -r --cached <directory name>

To delete an entire directory (force):

$ git rm -r -f <file name>

- **git revert** is used to **undo a previous commit**. In git, you can't alter or erase an earlier commit. (Actually, you can, but it can cause problems.) So instead of editing the earlier commit, revert introduces a new commit that reverses an earlier one.
- EX: create a new directory, create a new file and **commit** it. Now if you don't want those changes then use revert command

- **Syntax: git revert <commit ID>**

In Practice:

- *git init*
- *vi index1.html*
- *git add index1.html*
- *git commit -m "created index1.html"*
- *vi index2.html*
- *git add index2.html*
- *git commit -m "created index2.html"*
- *git status*
- *git log --oneline*
- *git revert <first commit ID>*
- *ls* *----->we can see only index2.html. Git has undo chnages of first commit ID*
- *vi index3.html*
- *git add index3.html*
- *git commit -m "created index3.html"*
- *git log --oneline*
- *ls*
- *git revert <reverted commit ID> ---------> Git has undo changes of first revert commit.*
- *ls* *----------> Now we can index1.html*

Summary

These are common Git commands used in various situations:

clone : Clone a repository into a new directory

init : Create an empty Git repository or reinitialize an existing one

work on the current change

add :Add file contents to the index

mv :Move or rename a file, a directory, or a symlink

reset :Reset current HEAD to the specified state

rm :Remove files from the working tree and from the index

examine the history and state

bisect :Use binary search to find the commit that introduced a bug

grep :Print lines matching a pattern

log :Show commit logs

show :Show various types of objects

status :Show the working tree status

branch :List, create, or delete branches
checkout :Switch branches or restore working tree files
commit :Record changes to the repository
diff :Show changes between commits, commit and working tree, etc
merge :Join two or more development histories together
rebase :Reapply commits on top of another base tip
tag :Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
fetch :Download objects and refs from another repository
pull :Fetch from and integrate with another repository or a local branch
push :Update remote refs along with associated objects