Common Terminologies

* **Repository** A unit of storage and change tracking that represents a directory whose contents are tracked by Git
* **Branch** A version of repository that represents the current state of the set of files that constitute the changes
* **Master** The default or main branch, it is a version of the repository that is considered the single source of truth
* **Index** This is an area where Git holds files that have been changed, added, or removed in readiness for a commit
* **Commit** This is an entry into Git's history that represents a change made to a set of files at a given point in time
* **Merge** A merge is the process of incorporating changes from one branch to another branch
* **Workflows** It refer to the approach a team takes to introduce changes to a code base. It characterized by a distinct approach in the usage of branches to introduce changes into a repository
  + **Gitflow Workflow**
    - This uses two branches: master and develop
    - The master branch is used to track release history, while the develop branch is used to track feature integration into the product.
  + **Centrailzed Workflow**
    - This approach uses the master branch as the default development branch
    - The changes are committed to the master branch
    - It’s a suitable workflow for small size teams and teams transitioning from Apache Subversion
    - In Apache Subversion, the trunk is the equivalent of the master branch
  + **Feature Branch Workflow**
    - In this workflow, feature development is carried out in a dedicated branch
    - The branch is then merged to the master once the intended changes are approved

GitHub

* Version control with Git takes on a distributed nature
* The code resides on each local computer where the code base is being worked on, as well as on a central remote point where every individual who wishes to work on the code base can obtain it.
* GitHub is one such central remote point
* GitHub hosts repositories and enables users to obtain, alter, and integrate changes to a code base through Git: local repository to Git remote repository

 Organizations:

* Organizations spur seamless coordination of work through the following features availed by GitHub:
  + Role-based membership
    - Owner, Billing manager, member
      * Each personal account that is added to an organization can belong to one of the above-mentioned roles
      * The owner role is the most superior and is used to conduct administrative procedures
  + Repository Level permissions
    - Teams or their respective members can be assigned read, write, or admin-level permissions to a repository
    - Each level dictates activities that the assigned members undertake, with a varying degree of limitations
  + Teams
    - These are members of an organization that can be grouped into teams, with the option of nesting the teams to match an organization’s structure.
  + Multi-factor authentication
    - Organization support the enforcement of two-factor authentication as well as business-specific single sign-on approaches such as **Security Assertions Markup Language (SAML)** and **System for Cross-domain Identity Management (SCIM)**

**Runtime Config**

* Runtime configurations are set using git config command
* Git configurations are set in three levels:
  + System-wide configuration
    - These options are set in the /etc/gitconfig file
    - To access these settings you use git config --system
  + User specific configuration
    - These options are set in ~/.gitconfig
    - The user specific settings are accessed via git config --global
  + Repository-specific configuration
    - Repository specific settings are set in the path\_to\_repository/.git/config
    - An example of configuration here is the GitHub URL of a repository, which set at this level
    - These settings are accessed via git config --local
* To set name and email user level:
  + Git config --global user.name BhuppalKumar
  + Git config --global user.email bhuppal@hotmail.com
  + Git config --global --list
* To remove :
  + Git config --global --unset [section.name].[section\_variable]

**SSH Configuration**

* Git supports it by using a combination of a username and password, or using an SSH key to authenticate a connection or requests made to GitHub from your local environment
* The use of SSH keys ensures enhanced security and averts the need to provide a username and password for each request

Command: ssh-keygen -t rsa -b 4096 -C bhuppal@hotmail.com

Note: all the prompts are optionals

bhupp@8huppaILaptop MING.•.'64 / 
S ssh-keygen -t rsa -b 4096 -C bhuppal@hotmail.com 
Generating public/private rsa key pair. 
Enter file in which to save the key (/c/users/bhupp/.ssh/id_rsa): 
Created directory '/c/users/bhupp/. ssh' 
Enter passphrase (empty for no passphrase): 
Enter same passphrase again: 
Your identification has been saved in / c/ Users/bhupp/. ssh/id_rsa 
Your public key has been saved in / c/ Users/bhupp/.ssh/id_rsa. pub 
The key fingerprint is: 
SHA256:/+KwXJOC16v7uEOQHVJu/PILVD6+S4geapf/uT5VASg bhuppal@hotmail.com 
The key's randomart image is: 
-CRSA 4096] 
.00* .008 
- CSHA256] 

After adding into the Github.com, under security and SSH key session, locally issue the following command:

SSH -T git@github.com

Jhupp@8huppaILaptop MING'.€64 /usr 
S SSH -T git@github.com 
The authenticity of host 'github.com (140. 82.112.4)' can't be established. 
RSA key fingerprint is SHA256:nThbg6kXUpJWG17E11GocspRomTxdCARLviKw6E5SY8. 
Are you sure you want to continue connecting (yes/no/ Cfingerprint])? yes 
Warning: Permanently added 'github. com,140. 82.112.4' (RSA) to the list of known hosts. 
Hi bhuppal! You 've successfully authenticated, but GitHub does not provide shell access. 
Jhupp@8huppaILaptop MING'.€64 /usr 
S ssh -T git@github.com 
Warning: Permanently added the RSA host key for IP address '140. 82.114.4' to the list of known hosts. 
Hi bhuppal! You 've successfully authenticated, but GitHub does not provide shell access. 

**Create Repository**

Version control requires that the files and associated changes that need to be tracked are organized in a repository which is the unit that Git identifies as the candidate for source control. To commence a piece of work, we need to create a repository. In this section, we shall explore two approaches that you may use to initialize a repository.

Git Commands:

Git init

Git remote add origin <url> (ssh or http)

.gitignore file – To add file paths and files details to ignore for git process

Git add <filename>

Git commit -m “Message”

Git pull origin master

Git push origin master

Git clone <url> (ssh or http) – To clone remote repository to locally

Tags: used for the purpose of identifying specific significant points on a repository’s history.

* Lightweight tags – it act as pointers to a specific commit. It only stores the references to the commit. Git tag v2.5
* Annotated tags – acts as pointers to a specific commit and additionally store information about the creator of the tag, email, and the date of creation. Git tag -a v2.6 -m “Support sdk version 3”

Git Description: under Edit button.

README.md – More introduction explains

Git add <filename> && git commit -m “test” “&&” -> you can enter command sequentially.

In Git, files can have the following statues.

Untracked : A file that exits in the working tree whose changes are not being monitored by Git and aren’t listed in the gitignore file.

Unstaged : A file whose changes are being tracked by Git, the file has been changed since the last commit and has been moved to the index.

Staged : A file whose changes are being tracked by Git; the file has been changed since the last comit and has been moved to the index.

Git status:

* It’s a utility that is used by Git.
* It’s used to retrieve the details of the files that are untracked, unstaged, or stated.
* Git status lists files in order of their statues
* The git status output is lengthy in nature
* Using the –s or –short option with the git status to view a brief list and status

**Branch:**

Git branch <branchname>

Git checkout <branchname>

Git checkout -b <branchname> (if doesn’t exits, it creates one)

**To see the differences diff :**

git diff

git diff <filename>

git diff HEAD (compare with last the commit)

git diff HEAD c366f89 (compare with commit)

git diff master (comparing working branch with the master)

git diff development master

git diff --cached (the changes)

git diff ---cached c366f89

**To reset the changes**

Git reset –hard

**Git Add**

* The “git add” command is used to add files to the index from the working tree.
* The “git add” command uses the following syntax: git add [options] [path\_to\_files]

The options used with git add include -n and –dry-run. This options simulates the behavior of git add for the specified file.

-f or –force

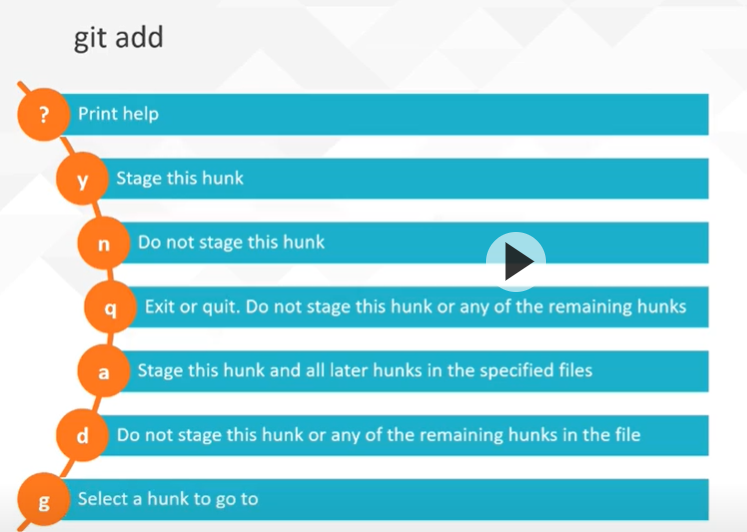
This option adds ignored files to the index

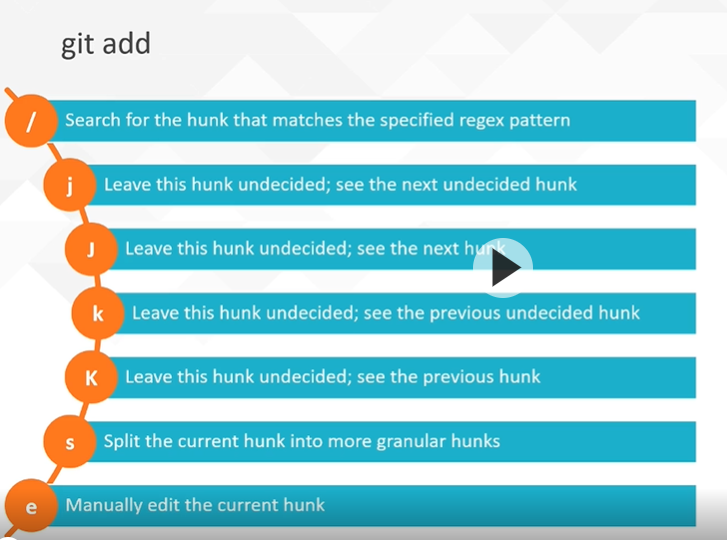
-I or –interactive

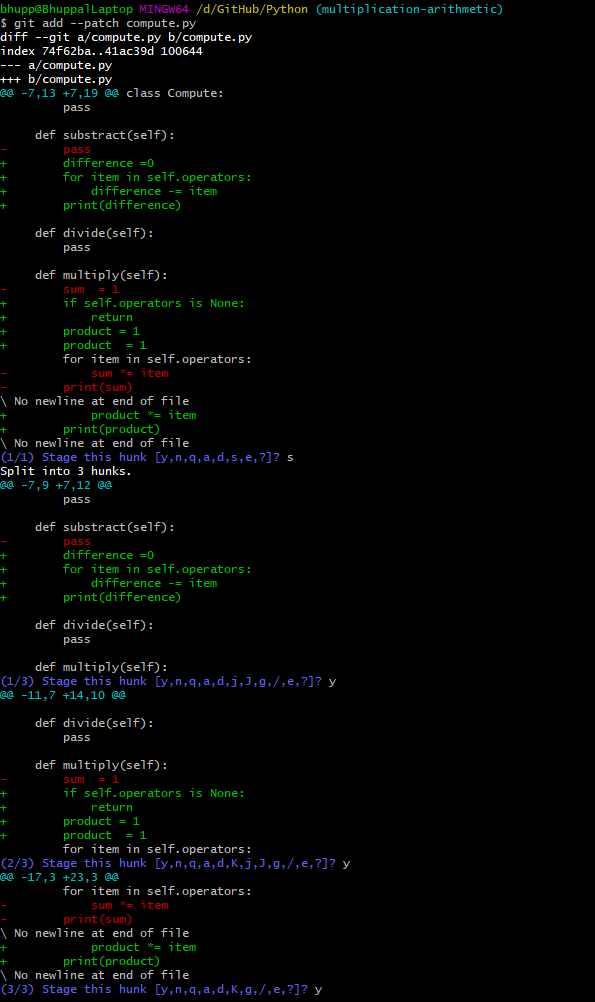
This option creates an interactive prompt that can be used for adding files from the working tree to the index.

-por or –patch

The –path option caters for adding portions of a file to the index







**Git commit**

* Git commit command saves the files in the index
* The commit operations stores a message along with the commit
* This message describes the additions or alternations associated with the created snapshot.

Git -commit m or – message associate the index file with the commit action

Git commit -a or -all stage tracked files that are unstaged

Git commit -p or –patch – interactive patch tool

Commit -c commit hash or –reuse-message=[commit hash] reuse a commit message and the author information of the specified commit hash

Git commit -F [file] or –file[file] specifies a file from which a commit message should be obtained

Git commit – t [file] or --[template] file specifies the commit message template file

Git commit -e or –edit edits the provided commit message

Git commit –no-edit uses the specified message as is

Gti commit –author overrides the details of a commit author

Git commit –date overrides the date details used in commit

Git commit -q or –quite suppresses the summary message that’s returned after running the git commit command

**Git rm**

The git rm command performs two roles:

* Remove files from the working directory and the index
* Remove files from the index

**Git mv**

* The git mv command is used to rename or move a file or directory
* This command has two forms of implementation:
  + Git mv [options][source][destination] – used to rename a file
  + Git mv [options][source]…[destination] – used to move a file

**Git log**

Git log

Git log hash#1 hash#2

Git log –follow <filename>

Git log –decorate=full or short or no

Git log -l 5:20 <filename> display changes to specific set of lines of a file

Git log -n 3 or Git log 3 or git log –skip=4

Git log –since =dd/mm/yyyy

Git log –after=dd/mm/yyyy

Git log –pretty=oneline

Git log –pretty=short

Git log –pretty=medium

Git log –pretty=format:”%h %an”

**Amending commits**

The most recent commit can be edited using the –amend option of the git commit command

Git commit –amend

Git rebase -i HEAD~4

To insert or update some code in the middle of the commit:

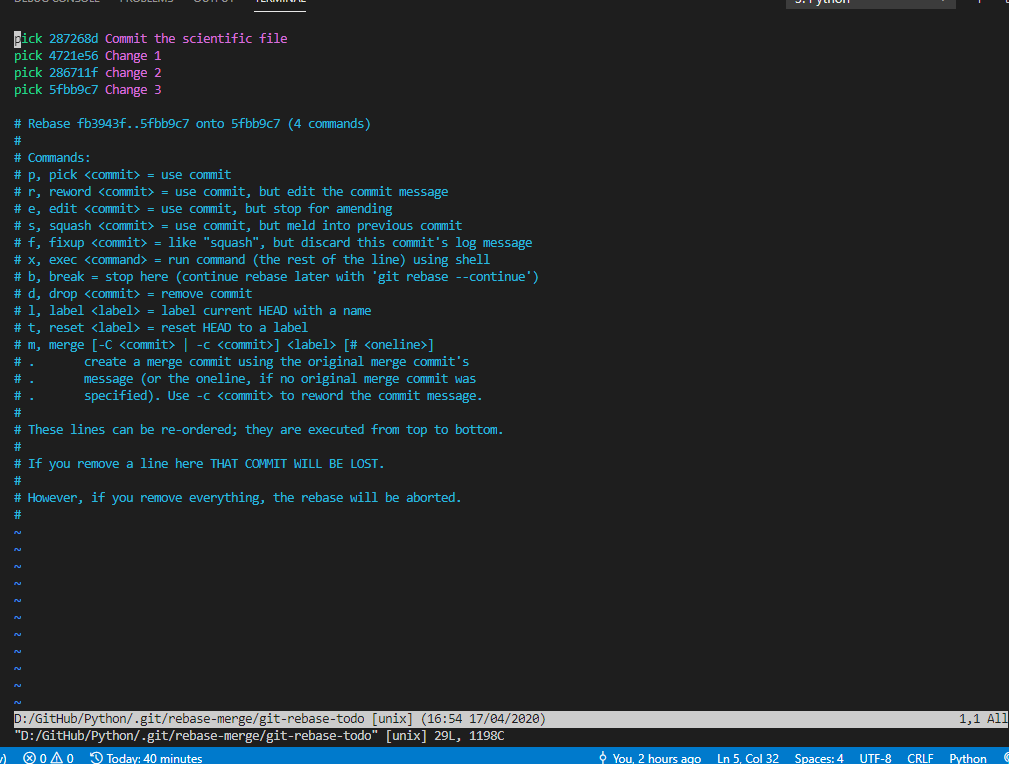
Step #1 – git rebase -I HEAD~4 (list last commits)

Step #2 – pick which one to edit, type “edit” and save and edit it.

Step #3 – Edit the file and save the changes.

Step #4 – git add . and git commit --amend , it opens the editor to update the old comment.

Step #5 – git commit --continue



Versions commits:

Upstream:

* Upstream refers to the hosted repository
* This is the repository (and the subsequent branch that is hosted on GitHub) from which contributors can clone the repository to their local environments, make changes, and publish changes to it.

Downstream:

* This refers to the repository that is situated in your local environment
* The downstream repository is obtained through the git clone command, which creates a cop of the hosted repository.

Remote:

* Remote refers to a names reference to a hosted repository
* The remote connection is configured at the local repository level in order to enable pushing and pulling information to and from the hosted repository

Origin:

* This is the default remote that’s configured on a repository
* The origin is created when a repository is cloned by executing the git clone command

Git remote:

* With the terminology having been covered, we will proceed to look at the git remote utility, to explore how we can manage the connection between the local and upstream repositories.
* The git remote utility includes commands that help with managing the remote/upstream repositories that are associated with a local repository

Git init

Git remote add origin URL

Git remote -v (verbose)

Git remote rename origin <newname>

Git push -u <newname>

Git remote show <newname>

Git fetch –depth=5 certain number of commits

**Git Fetch:**

Git fetch command downloads all the branches and tags from remote tracking from remote repository

Git fetch origin – all the branches of the orign

Git fetch -v (verbose)

Git fetch origin master

Git fetch –all it will download all the branches for this origin

Git fetch –depth=5 certain number of commits

git fetch --prune is the best utility for cleaning outdated branches. It will connect to a shared remote repository remote and fetch all remote branch refs. It will then delete remote refs that are no longer in use on the remote repository.

**Git Clone**

git clone url

**Git Pull**

Git pull : fetch from and integrate with another repository or a local branch (git fetch and git merge) With –rebase, it runs git rebase instead of git merge

**Git Revert:**

Git revert –edit <commithash#>

**Git reset:**

* The git reset command is used to roll back a file, directory, or repository to a given point in history.
* This entails updating the index, working tree, and commit history of a repository where applicable based on the options passed on the command.

**Git objects:**

Commits, trees, blobs, tags tracks as objects. Stored under .git/object folder

Git cat-file -t <commithash>

Git cat-file -p <commithash>

**Default Branch:**

By default master is the default branch. IF we want, we can change it under settings, Branches option.

We can also rules to the branch.

Git fetch –depth=5 certain number of commits

**Git Tags:**

Git tag -a v1.4 <commithash> -m “A version”

Git tag -d v1.4 (to delete a tag)

Git tag -a v1.0 -m “First release”

Git log –oneline

Git push origin v1.0

**Git Workflow**

Git checkout master

Git merge branch

**Git new pull request process:**

* Git clone <url>
* Git checkout -b newfeature
* Do all the changes to the newfeature
* Git push –set-upstream origin newfeature

**GitHub Workflow Features:**

* New pull request
* Fork and contribute to any repository

**Branches:**

Creating, Renaming, Deleting, and Listing Branches

* The commit object stores a snapshot of the directories and files that constitute a repository at a given point in time.
* The commit specifies auxiliary information, which includes the parent commit of the created commit, the author, the committer date and time, and the commit message.
* A branch is therefore a pointer to a snapshot of the repository.
* This pointer refers to the commit at the tip of the branch.
* This commit is imperative because its based on the face that git is able to navigate the history of a repository with the help of the parent-child association between commits.

Git branch newbranch – (creating a new branch)

Git branch –list (listing all the branches)

Git checkout newbranch (checkout the branch)

Git checkout -b newbranch

Git branch -d newbranch (delete a branch)

Git branch -D newBranch (deleting a branch before merge)

Git branch -m newbranch superbranch (rename a branch)

Git branch –merged (list all the branches are merged)

Git branch –no-merged (list al the branches are not merged)

Git branch – creating from particular

* Git checkout -b branchname
* Git checkout hash#
* Git checkout -b secondnewbranch

Git restore a particular file:

* Gi checkout hash# -- filename (this totally restores the file content to nothing)

Git Stash:

Git stash push -m “some detail” or just Git Stash

Git stash apply

* Git stash list stash\_id – Lists the entries in the stash list
* Git stash show [stash\_id] – Display the changes introduced by the stash identified by [stash\_id]
* Git stash apply [stash\_id] – Updates the working directory with the changes stored in [stash\_id]
* Git stash pop [stash\_id] = Updates the working directory with the stash [stash\_id] and removes it from the stash list
* Git stash drop [stash\_id] – Removes the specified stash from the stash list
* Git stash clear – Removes all stashes from the stash list

Git Merging:

* Creating a change branch off the main branch
* Building the change
* Unifying the change with the main branch

1. Compares the files in the two branches from the point where they diverged
2. Reconciles the changes from both branches and merges incoming changes into the current branch by updating the index and working directory
3. Retains changes from both branches where differences cannot be resolved for the purpose of merge conflict resolution
4. Once the merge conflict (if any) is resolved, a merge commit is created to represent the incoming changes.

If conflicts happen, open the conflict file manually and accepts the changes.

**Merge Conflicts**

* A merge conflicts is a term that depicts an issue whereby modifications made in separate branches can’t be amalgamated into one unit of change or modification.
* Modifications are made on the same line of a file
* Changes are made to a file on one branch and the same file is deleted on another branch

Git Cherrypick

* From the branch, pick any one commit and merge into master
* Git cherry-pick hashcommit# or git cherry-picks hash1 hash2 hash2
* Git add and commit

Git Pull Request (PR)

* A pull request (PR) is the culmination of a piece of work undertaken in a branch.
* A PR is an intent to merge
* Pull requests are intended to avail the forum where changes there are to be made to a repository’s main branch are accorded security
* Consensus is arrived upon the completion of a satisfactory discussion on the modifications that are borne by the branch that seeks to introduce changes.

Pull Request Templates:

* To ensure the seamless integration of changes being introduced to the primary branch of a repository, GitHub supports standardized pull requests by providing means for stating a format for pull requests.
* A template seeks to ensure that each pull request raised on the repository is clear in communicating what the potential change seeks to resolve.
* This lends itself to the appropriate discourse and comprehensive scrutiny, which in turn ensures that the result of the PR review process is a change that has been agreed upon by the relevant stakeholders.

Identifying and Fixing Merge Issues

* A PR may execute unit tests as part of a pre-merge sanity check.
* To proceed with the merging of PR, you need to ensure that the set checks pass.

Merging and Reverting Pull Requests

There are three modes available for merging pull requests:

* Create a merge commit – All commits from this branch will be added to the base branch via a merge commit.
* Squash and merge – All commit from this branch will be combined into one commits in the base branch.
* Rebase and merge – All commits from this branch will be rebased and added to the base branch.

Merge commit For example: This mode adds commits C4 and C to the branch master using a unifying commit, C6, referred to as a merge commit. This is non-fast-forward mode.

Squash and Merge: In this mode, C4 and C5 are combined into single commit, C6. The C6 commit is then merged in the fast-forward mode.

Rebase and Merge: In this mode, each commit from the branch is added to master branch without the use of a merge commit. This is a non-fast forward mode.

**Forking the workflow**

We fork from somebody’s project to our own repository.

Why do we Fork Repositories? Forks allow contributions to other repositories through pull requests in case when the individual is not added as a contributor to those repositories.

Forks also help avoid loss when the upstream repository is taken down, especially in situations where the code in the repository is a dependency to an ongoing project.

Embedding Upstream Changes

* Upstream refers to the original repository a fork is made out of.
* When working with active forked repositories, changes are going be implemented almost every hour, and within a short period, your fork will be outdated.

Git remote add upstream <sshurl>

Git push upstream (whatever copy we have in repository, it copy’s to the fork copy)

Git pull upstream master

Rebasing

* During a rebase, a feature branch is reset to the master’s most recent commit (B) then added on top.
* Think of it this way: you want to re-organize your branch’s base with the most recent changes, your work being the most recent.

Pulling the changes from the master, and make a part of the commit history to keep it up to date.

Git rebase master, keep up to date with the master. Sometimes it get some conflict. Manually fix the conflict.

Git add .

Git rebase –continue

* Git presents an option to fix up and auto squash the previous commits.
* The reason this shouldn’t be done on public branches is that Git re-writes commits, as this also a rebase operation.
* Re-writing history on a public branch would cause conflicts and data loss to other contributors.

Rebase with autosquash

Git checkout -b feature

Git add . && git commit -m “A really nice change”

Git log –oneline

Git add . && git commit –fixup hash#

Git log –online

Git rebase -I –autosquash hash#

Drop commits

* The need to get rid of unwanted changes comes up, for instance, changes that introduce bugs during a root cause analysis
* A lot of approaches can be taken towards this, including resetting HEAD to a commit just before the problematic commit or explicitly dropping the problematic commit
* Resetting HEAD may not be a viable option in a case where the commit is back in the history. To Be safe, especially with no proper rebase option, reset HEAD when the commit is most recent

Git rebase -I hash# , open the editor and select the commit and type “drop” in the front and save the editor.

Submodules

* Git submodules help organize code based on a single responsibility principle
* Submodules are tracked through a .gitmodules file and, just like any other dependency, you need to close them into their parent repositories. For instance, Node.js applications require a package manager such as npm to install dependencies to a node\_modules directory, through a package.json file
* Submodule require a .gitmodules file and install dependencies through the git submodule command
* Git submodule add <repository url> and git submodule init command

Command: git submodule add repourl build/reponame – this will create a new .gitmodules file and stores the another repo details

**Debugging and Maintenance**

* Git blame – Helps identify and present occurences from a revision that modified a block of code
* This is done line by line
* Identify who made a change, the commit used, and what line(s) were affected

Git blame <filename>

Git bisect start – enters into bisect mode

Git bisect good hash#

Git bisect bad hash#

Git bisect bad

Git bisect reset

**Git reflog**

* Git reflog is short for Git reference logs
* Reflogs keep track of changes to HEAD over a defined period of time
* These changes can be best defined as events, as they are saturated, which is basically all activities, that is, checking out branches, rebase events, and branch updates from remote URLs
* By default, reflog don’t traverse, as it consists of a list of all activities occurring on HEAD, which also includes activities from unreachable branches and lost commits through operations such as a rebase
* A default period of 30 days before the data expires
* Entries belonging to reachable branches also have an expiration period of 90 days by default

**HouseKeeping**

**Git clean**

* Git clean – recursively removes untracked files from a working tree
* Any file that is not staged to be tracked or reset is rid of, maintaining a versioned only directory
* Git clean purges files through a list defined from a .gitignore file, but is special cases, these rules can be ignored and any untracked file is cleared

**Git gc**

* Git gc is responsible for garbage collection on targeted repositories
* This process handles the deletion of staged and committed objects from unreachable branches, particularly those holding a reflog
* Depending on the repository activity, git gc helps optimize disk space and maintains a decent repository

**Git prune**

* The git prune command, similar to git gc, gets rid of all unreachable objects, that is, basically objects without references, such as ones from deleted branches
* Its best to go for git gc, which calls git prunce, killing two birds with one stone
* Git branch –merged (list all the branches are merged)
* Git branch –no-merged (list al the branches are not merged)

Pre-commit hooks

* Pre-commit hooks, as client-side Git hooks, are custom scripts that are triggered before certain actions occur, such as commits and merges, thus the name has the prefix “pre”
* They can be customized to any team’s workflow and in most scenarios, teams use pre-commit hooks to achieve the following:
  + Running language linters and standardization checks before commits
  + Verifying that sensitive keys aren’t being versioned
  + Verifying compatibility and portability of the code in play

**Test Automation**

* Continuous Integration
* Continuous Delivery
* Continuous Deployment

**Webhooks**

* Webhooks are callbacks that intercept events and implement actions
* Webhooks enable GitHub applications to subscribe to events such as pushed commits and the creation of pull requests.

**GitHub Applications**

GitHub applications are services that enable a variety of operations such as the testing, building, and deployment of applications, and are accessible from the GitHub marketplace.

* Continuous Integration
* Code Quality
* Monitoring
* Project Management
* Deployment

CircleCI – Continuous Integration from Marketplace

**Automated Pull Request**

Under setting, Branch protection rule, we can set all the rules. While pull request, all the rules are triggered.

**Release Management**

Release management can be simply put as the process of scheduling and controlling software builds across different environments

Git plays a critical role in this process through tags

**Tagging**

Tagging involves the creation of markers to correspond to software version

* Introducing and standardizing a release branch
* We should always get rid of this branch as soon as changes have been merged to the base branch and deployed to production

In a real-world scenario, the following branches would present an ideal environment and release-friendly workflow:

* **Base:** A master where each commit represents a release that’s been made, deployed and that is running in production
* **Testing:** Dictates what is running on the development, staging, and QA environments
* **Feature:** Branched from develop by all repository collaborators and merged back with new changes.
* **Release:** Branched from develop branch after feature delivery, testing, and quality assurance have been carried out

Git tag -a v1.0 -m “First release”

Git log –oneline

Git push origin v1.0

**Git Archive**

Git archive –format=tar(or zip) –prefix=tag-demo/ v1.0 | gzip > tag-demo.tar.gz