Git & Linux

COMMANDS

$ ls -la
$ git status
$ git commit -m 'update'

git init
git pull
rm -rf

Table of Contents

Introduction *1*

Purpose 1

Scope 1

Procedure 1

Brief Problem 1

Git *2*

Working Directory, Staging Area, and Local Repository 2

1. The Working Directory 2
2. The Staging Area (Index) 2
3. The Local Repository 2

Remote Repositories: Origin and Upstream 3

- Origin 3
- Upstream 3

Core Git Data Structures: Blobs, Trees, and Commits 3

- Blobs (Binary Large Objects) 3
- Trees 3
- Commits 3

HEAD and References 4

Tags 4

- Lightweight tags: 4
- Annotated tags: 4

Navigating Commits: Tilde (~) and Caret (^) Operators 4

Git Areas and the Lifecycle of a File 5

- Untracked: 5
- Tracked: 5
- Modified: 5
- Staged: 5
- Committed: 5

Starting a New Project with Git 5

Joining an Existing Project 6

Git Commands 7

1. Repository Setup & Configuration 7
2. Staging & Committing Changes 7
3. Branching & Merging 8

4. Working with Remote Repositories 9
5. Undoing Changes & Fixing Mistakes 10
6. Viewing History & Logs 10
7. Tagging & Versioning 11
8. Stashing (Temporary Save Work) 11
9. Collaboration & Inspection 12
10. Cleaning 12
Linux *14*
Linux File System Structure 14
Linux Commands 15
1. Basic Commands 15
2. Navigation & Directory Commands 15
3. File & Directory Management 16
4. Searching and Filtering 17
5. Permissions & Ownership 17
6. Redirection & Piping 18
Conclusion *19*
Feedback & Contribution *19*
Copyright & Usage *19*
Acknowledgments *20*

Introduction

Purpose

The purpose of this document is to provide a clear and practical understanding of essential **Git** and **Linux commands**, enabling users to efficiently manage code, systems, and development workflows. It aims to help learners, developers, and IT professionals build a strong foundation in version control and command-line operations — two critical skills in software engineering and system administration.

Scope

This document covers fundamental to intermediate **Git commands** used for version control, collaboration, and repository management, as well as key **Linux commands** for navigating directories, managing files, controlling processes, and configuring systems. It focuses on commands most commonly used in professional and academic environments, ensuring that readers can apply them effectively in real-world scenarios.

Procedure

The learning process begins with an overview of Git — from initializing repositories, committing changes, and branching, to merging and resolving conflicts. Next, it explores Linux commands, explaining how to interact with the operating system through the terminal to perform essential tasks like file handling, user management, and process control. Each command is discussed with examples to reinforce understanding through practice.

Brief Problem

Many beginners struggle to manage projects efficiently due to a lack of understanding of **version control** and **command-line operations**. Without Git, code collaboration becomes error-prone and disorganized. Without Linux proficiency, automation and system management become difficult. This document addresses these challenges by equipping users with the practical knowledge needed to work confidently in both **development** and **deployment** environments.

Git

Git is more than a version control tool — it's a **complete data management system** for tracking changes to files, coordinating collaboration, and maintaining an exact history of a project.
To use Git effectively, it's essential to understand what happens *under the hood* — how it stores information, what each area of the repository does, and how commands interact with it.

Let's explore the main concepts that make up Git's internal logic and practical workflow.

Working Directory, Staging Area, and Local Repository

When you initialize or clone a Git repository, you're actually working with **three main areas**:

1. ***The Working Directory***
   This is the actual folder on your computer where your project files exist. You can open, edit, delete, or create new files here. However, Git doesn't automatically track these changes — until you tell it to. When you modify a file, Git recognizes it as "modified," but it's not yet part of the next commit.

2. ***The Staging Area (Index)***
   The staging area acts like a "preview" or "waiting room" for your next commit.
   You use git add to move specific files (or all files) into the staging area. This allows you to **control exactly what changes** will be included in the next commit.

3. ***The Local Repository***
   This is stored inside the hidden .git/ folder.
   It contains the entire history of your project — every commit, every branch, every tag, and all the relationships between them. When you use git commit, you're saving a snapshot of the current state of your staged files into this local database.

Together, these three layers create Git's core flow:

Working Directory → Staging Area → Local Repository

(edit) (add) (commit)

Remote Repositories: Origin and Upstream

Most real-world projects aren't just stored locally — they're hosted on remote servers like **GitHub**, **GitLab**, or **Bitbucket**. These are called **remote repositories**, and they allow multiple developers to collaborate.

- ***Origin***
  The term origin is simply a *name* — a short alias Git automatically gives to the remote repository you cloned from or linked to.
  When you push or pull, Git uses this origin alias to know which remote to interact with.

- ***Upstream***
  The upstream remote typically refers to the *original project* your fork came from.
  If you fork someone's project on GitHub and then clone your own fork locally, your fork is origin, and the original author's repository is upstream.
  You can fetch updates from upstream, merge them into your branch, and keep your fork in sync.

Core Git Data Structures: Blobs, Trees, and Commits

Git is built around three main types of objects, which together make up its entire storage model:

- ***Blobs (Binary Large Objects)***
  A blob represents the **content** of a file — nothing else. It doesn't store the filename, permissions, or folder path, only the actual bytes of the file's content.
  Every unique file content (even a single line difference) creates a new blob, identified by a unique SHA-1 hash.

- ***Trees***
  A tree represents a **directory**. It maps filenames to blob objects and can also contain other trees for subdirectories.
  You can think of a tree as a snapshot of how all files and folders are arranged at a particular commit.

- ***Commits***
  A commit is a **snapshot** of your entire project at a given time.
  It points to one tree (representing the root directory at that moment), includes metadata such as the author, date, and message, and references its parent commit(s).
  The entire Git history is simply a chain of commits connected by these parent relationships.

This structure allows Git to be extremely efficient — if you change one file, only that blob and its containing trees need to be updated; the rest remain untouched.

HEAD and References

In Git, **HEAD** is a symbolic pointer to the current branch or commit you are on.
When you're on the main branch, HEAD points to the latest commit in that branch.
If you checkout another branch, HEAD moves to that branch's latest commit.

Sometimes, you might enter a **detached HEAD state** — this happens when you checkout a specific commit directly instead of a branch. For example:

git checkout 4b3a1d2

In this case, HEAD points directly to that commit rather than a branch.
You can still explore, modify, or even make new commits, but unless you create a new branch from there, those commits might be lost when you switch back.

Tags

A **tag** is a pointer to a specific commit — usually used to mark important versions or releases.

There are two main types:

- ***Lightweight tags:*** simple names pointing to a commit (git tag v1.0).
- ***Annotated tags:*** include extra metadata like a message, tagger name, and date (git tag -a v1.0 -m "First stable release").

Tags are immutable and serve as permanent labels in your project's timeline — perfect for marking releases.

Navigating Commits: Tilde (~) and Caret (^) Operators

Git allows you to move around your commit history using these symbols:

- HEAD~1 means "one commit before HEAD."
- HEAD~2 means "two commits before HEAD."
- HEAD^ also means "the parent of HEAD" (same as HEAD~1).
- In merge commits (which have multiple parents), you can specify the parent number — for example, HEAD^1 and HEAD^2.

Git Areas and the Lifecycle of a File

Every file in your working directory can be in one of these states:

- ***Untracked:*** The file is new and hasn't been added to Git yet.
- ***Tracked:*** Git knows about it — it was committed at least once.
- ***Modified:*** The file has been changed since the last commit.
- ***Staged:*** The file's changes have been added to the index and are ready to commit.
- ***Committed:*** The changes are safely stored in the local repository.

This lifecycle repeats continuously as you work and commit over time.

Starting a New Project with Git

When starting a fresh project, the usual steps are:

1. Initialize Git inside your project:
2. git init
3. Add your files:
4. git add .
5. Create your first commit:
6. git commit -m "Initial commit"
7. Create a new remote repository (e.g., on GitHub).
8. Link your local repository to it:
9. git remote add origin <repo-url>
10. Push your first commit:
11. git push -u origin main

This establishes your local repository as the source for the remote, and from here on, both remain synchronized via pushes and pulls.

Joining an Existing Project

If you’re joining an already established project, your steps are slightly different:

1. Clone the repository to your machine:
2. git clone <repo-url>
3. Create a new branch for your work:
4. git switch -c feature-branch
5. Make changes, add and commit them:
6. git add .
7. git commit -m "Add new feature"
8. Pull the latest changes from the main branch to stay up to date:
9. git pull origin main
10. Push your branch:
11. git push origin feature-branch
12. Create a pull request on GitHub or GitLab to merge your work.

This workflow keeps your changes isolated and makes collaboration much smoother.

Git Commands

1. Repository Setup & Configuration

Command	Description	Example
git init	Initializes a new Git repository in the current folder.	git init my_project
git clone <url>	Clones an existing remote repository into a local directory.	git clone https://github.com/user/repo.git
git config --global user.name "<name>"	Sets your Git username globally.	git config --global user.name "Youssef"
git config --global user.email "<email>"	Sets your Git email globally.	git config --global user.email "youssef@email.com"
git config --list	Displays all current Git configuration settings.	git config --list

---

2. Staging & Committing Changes

Command	Description	Example
git status	Shows the current state of the working directory (modified, staged, etc.).	git status
git add <file>	Stages a specific file for commit.	git add main.cpp
git add .	Stages **all** modified and new files in the directory.	git add index.html
git commit -m "<message>"	Commits staged changes with a message.	git commit -m "Fixed bug in login function"
git commit -am "<message>"	Adds and commits all tracked files in one step.	git commit -am "Updated UI components"
git rm <file>	Removes a file from both the working directory and Git history.	git rm old_file.txt
git mv <old> <new>	Renames or moves a tracked file.	git mv old_name.py new_name.py

3. Branching & Merging

Command	Description	Example
git branch	Lists all local branches.	git branch
git branch <name>	Creates a new branch.	git branch feature-login
git checkout <branch>	Switches to the specified branch.	git checkout main
git switch <branch>	Alternative to checkout for switching branches (recommended).	git switch feature-login
git merge <branch>	Merges changes from the specified branch into the current branch.	git merge feature-login
git branch -d <name>	Deletes a local branch (if merged).	git branch -d feature-login
git branch -D <name>	Force deletes a branch even if unmerged.	git branch -D feature-login
git rebase <branch-name>	Reapplies commits from the current branch into the specified branch	git rebase test
git clone -b <branch-name> <repo url>	Clones a specific branch from the remote repo	git clone -b main Numerix

4. Working with Remote Repositories

Command	Description	Example
git remote show	Lists names of remote repos	git remote show
git remote -v	Displays remote connections (like origin).	git remote -v
git remote add origin <url>	Adds a remote repository connection.	git remote add origin https://github.com/user/repo.git
git remote rm origin	It disconnects your local repository from the remote repository	git remote rm origin
git remote rename local repo_name new_name	Changes name of local repo to a new name	git remote rename origin upstream
git fetch	Downloads new data from remote but doesn't merge it.	git fetch origin
git pull	Fetches and merges changes from the remote repository.	git pull origin main
git push	Uploads local commits to the remote repository.	git push origin main
git push -u origin <branch>	Sets upstream branch for tracking.	git push -u origin feature-login

5. Undoing Changes & Fixing Mistakes

Command	Description	Example
git restore <file>	Restores a modified file to its last committed state.	git restore index.html
git restore --staged <file>	Unstages a file but keeps changes in the working directory.	git restore --staged main.py
git reset <commit>	Resets to a specific commit (keeps changes).	git reset 1a2b3c4d
git reset --hard <commit>	Resets to a commit and **discards all** changes.	git reset --hard HEAD~1
git reset --soft <commit>	Only resets in Local Repo	git reset --soft HEAD~1
git reset --mixed <commit>	Resets Local Repo and Staging area	git reset --mixed HEAD~1
git revert <commit>	Creates a new commit that undoes a specific commit.	git revert 3a4b5c6d
git clean -fd	Removes untracked files and directories.	git clean -fd

---

6. Viewing History & Logs

Command	Description	Example
git log	Displays the commit history.	git log
git log -p num	Displays the commit with that number	Git log -p 1
git log --oneline	Shows a compact version of the log.	git log --oneline
git show <commit>	Shows details about a specific commit.	git show 1a2b3c4d
git diff	Shows changes between working directory and last commit.	git diff
git diff HEAD~1 HEAD	Shows the difference between the last and second-to-last commit	git diff HEAD~1 HEAD
git diff <branch1> <branch2>	Compares two branches.	git diff main feature-login
git blame <file>	Shows which commit last modified each line.	git blame app.py

7. Tagging & Versioning

Command	Description	Example
git tag	Lists all tags.	git tag
git tag <tag>	Creates a new lightweight tag.	git tag v1.0
git tag -a <tag> -m "<msg>"	Creates an annotated tag with a message.	git tag -a v1.1 -m "Release 1.1"
git push origin <tag>	Pushes a tag to the remote repository.	git push origin v1.0
git push origin --tags	Pushes all tags to the remote.	git push origin --tags

---

8. Stashing (Temporary Save Work)

Command	Description	Example
git stash	Saves uncommitted changes temporarily.	git stash
git stash list	Shows list of stashed changes.	git stash list
git stash apply	Reapplies the most recent stash.	git stash apply
git stash drop	Deletes the most recent stash.	git stash drop
git stash clear	Removes all stashes.	git stash clear

9. Collaboration & Inspection

Command	Description	Example
git shortlog	Summarizes commits by author.	git shortlog -s -n
git show-branch	Displays branches and their commits.	git show-branch
git reflog	Shows history of all branch changes (useful for recovery).	git reflog
git cherry-pick <commit>	Applies a specific commit to the current branch.	git cherry-pick a1b2c3d
git bisect	Helps find which commit introduced a bug.	git bisect start

---

10. Cleaning

Command	Description	Example
git clean -n	**Dry run** — shows which files *would* be removed, but doesn't actually delete them.	git clean -n
git clean -f	**Force clean** — removes untracked files.	git clean -f
git clean -f <path>	Clean a **specific directory or file**.	git clean -f build/
git clean -d	Removes **untracked directories** as well.	git clean -fd
git clean -x	Removes **all untracked files**, including those ignored by .gitignore.	git clean -fx
git clean -X	Removes **only ignored** files (those listed in .gitignore).	git clean -fX
git clean -i	**Interactive mode** — lets you choose which files to delete.	git clean -i

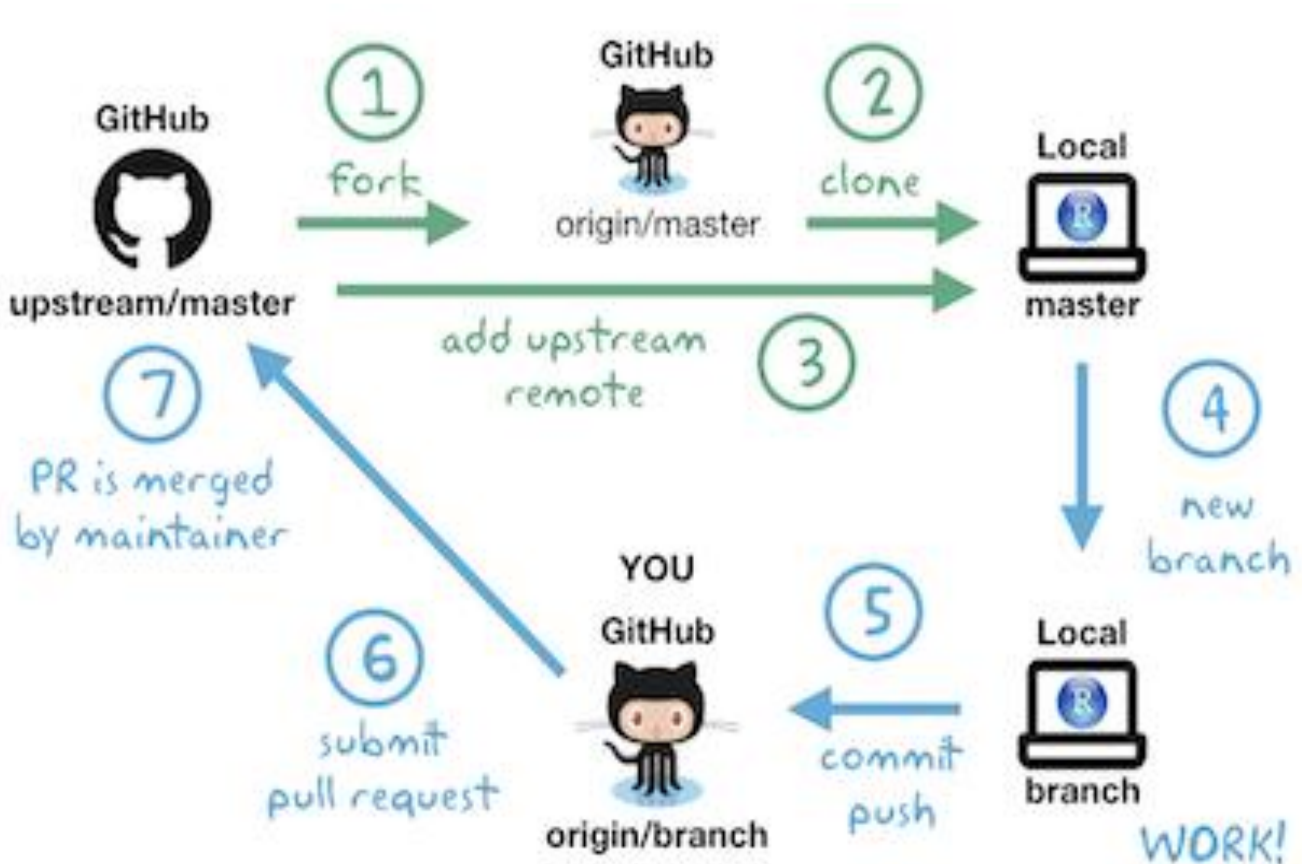
How Git Commands work

ByteByteGo.com

Working Directory
Staging Area
Local Repo
Remote Repo

git add
git commit
git push
git merge
git fetch
git pull
git checkout
git clone

Local | Remote



Made by Youssef Amgad Elkhatib

Linux

Linux is not just an operating system — it's a **powerful, modular ecosystem** that forms the foundation of nearly all servers, development environments, and even Android devices today.
Its true strength lies in the **command line interface (CLI)** — a text-based environment that allows users to control every part of the system efficiently and precisely.

The purpose of Linux is to provide a **stable, secure, open-source operating system** that gives full control over hardware and software.
Unlike graphical systems that hide details from the user, Linux exposes everything — allowing automation, scripting, networking, and system management at an expert level.

Linux File System Structure

Directory	Purpose
/	The root directory; everything starts here.
/home	User home directories (e.g., /home/youssef).
/bin	Essential binary executables (e.g., ls, cp, cat).
/etc	Configuration files for the system and applications.
/var	Variable data such as logs and cache.
/usr	User-installed applications and shared libraries.
/tmp	Temporary files (automatically cleared periodically).
/dev	Device files representing hardware (e.g., /dev/sda1).
/proc	Virtual directory containing process and kernel info.

Linux Commands

1. Basic Commands

Command	Description	Example
clear	Clears the terminal screen	clear
date	Displays current date and time	date
man <command>	Displays the manual (help) page for a command	man ls
echo <text>	Prints a message or variable value	echo "Hello Linux"
uname	Shows system and kernel information	uname -a
who	Displays currently logged-in users	who
users	Lists usernames currently logged in	users
history	Shows list of previously used commands	history
exit	Closes the terminal session	exit

---

2. Navigation & Directory Commands

Command	Description	Example
pwd	Print current working directory	pwd
ls	List files in directory	ls -la *(show hidden and long format)*
cd <dir>	Change directory	cd /home/user/Documents
cd ..	Move up one directory	cd ..
cd ~	Go to home directory	cd ~
tree	Display directory structure visually	tree /home/user

3. File & Directory Management

Command	Description	Example
touch <file>	Create empty file	touch notes.txt
mkdir <dir>	Create directory	mkdir Projects
rmdir <dir>	Remove empty directory	rmdir OldFolder
rm <file>	Remove file	rm data.csv
rm -r <dir>	Remove directory recursively	rm -r temp/
cp <src> <dest>	Copy file	cp file.txt backup.txt
cp -r <src> <dest>	Copy directory recursively	cp -r docs backup/
mv <src> <dest>	Move or rename	mv old.txt new.txt
cat <file>	Display contents	cat config.yaml
less <file>	View file page by page(scroll up-down)	less syslog.txt
more <file>	Views file content page-by-page (forward only)	more hello.py
head <file>	Show first 10 lines	head log.txt
tail <file>	Show last 10 lines	tail log.txt
tail -f <file>	Follow file in real time	tail -f /var/log/syslog

4. Searching and Filtering

Command	Description	Example
grep <pattern> <file>	Search text in files	grep "error" log.txt
grep -r <pattern> <dir>	Recursive search in the directory	grep -r "TODO" src/
find <path> -name <pattern>	Find files by pattern	find /home -name "*.py"
locate <name>	Find files using the name	locate report.pdf
sort	Sort lines in a file	sort names.txt
uniq	Filter duplicate lines	uniq log.txt
wc	Count lines, words, and characters	wc -l file.txt

---

5. Permissions & Ownership

Command	Description	Example
ls -l	Show permissions and ownership	ls -l
chmod <mode> <file>	Change permissions	chmod 755 script.sh
chown <user>:<group> <file>	Change file owner	chown root:root config.ini
umask	Set default permission mask	umask 022
id	Show current user ID and groups	id

6. Redirection & Piping

<table>
<tr><th>Symbol / Command</th><th>Description</th><th>Example</th></tr>
<tr><td>></td><td>Redirect output (overwrite)</td><td>echo "hello" > file.txt</td></tr>
<tr><td>>></td><td>Append output</td><td>echo "world" >> file.txt</td></tr>
<tr><td><</td><td>Input from file</td><td>cat < file.txt</td></tr>
<tr><td>`</td><td>`</td><td>Pipe output to another command</td></tr>
<tr><td>2></td><td>Redirect errors</td><td>python script.py 2> errors.txt</td></tr>
<tr><td>&></td><td>Redirect both output and errors</td><td>./run.sh &> log.txt</td></tr>
<tr><td>|</td><td>Redirects the output of the previous command as input to another command.</td><td>ls | grep "report"</td></tr>
</table>

Conclusion

Mastering Git and Linux commands empowers users with the ability to navigate, control, and optimize both code and systems efficiently. Git ensures structured collaboration and version tracking in software development, while Linux offers unmatched control and customization of computing environments. Together, they bridge the gap between development and deployment, promoting seamless automation, reproducibility, and system reliability. Whether used independently or in tandem, proficiency in these tools marks a critical step toward becoming a confident, capable, and resourceful technology professional.

Feedback & Contribution

This is a living document. If you have feedback, suggestions, or additional insights that could improve it, I welcome your input. Sharing ideas helps keep this material accurate, relevant, and valuable for everyone.

Copyright & Usage

© 2025 [Youssef Amgad Elkhatib].
This document is intended **for learning and reference purposes only**.
You are free to read, use, and share the content for personal or educational use, but:

- **Do not copy or republish this document as your own.**
- Always provide proper credit when referencing.
- Commercial use or redistribution without permission is not allowed.

You may not reproduce this document in whole or in part without attribution. Suggestions and feedback are welcome, and contributors will be acknowledged, but copyright remains with the author.

Acknowledgments

This document is authored and copyrighted by **[Youssef Amgad Elkhatib]**.
Special thanks to the contributors who provided valuable feedback and suggestions for improvements:

Name	Contribution