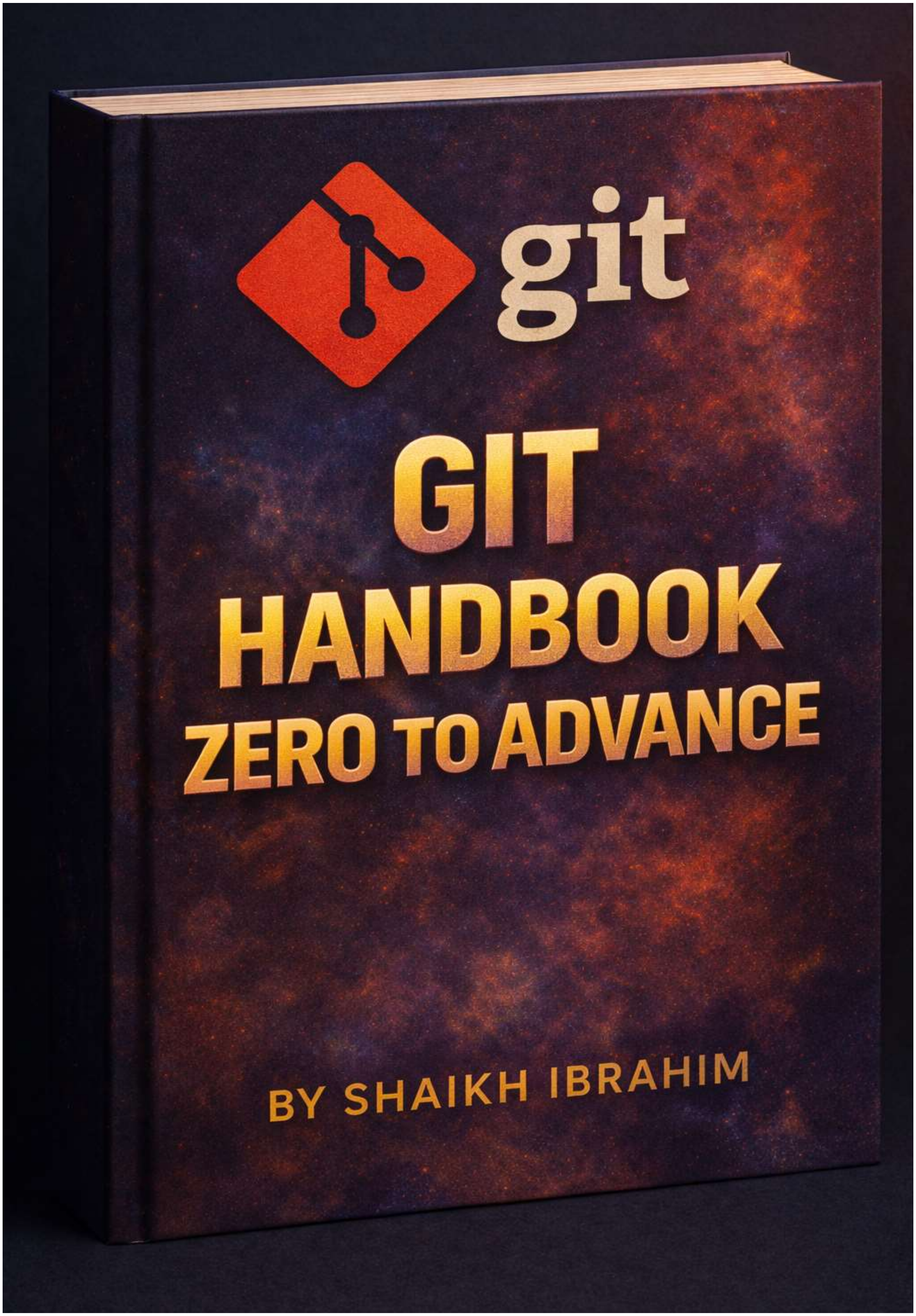


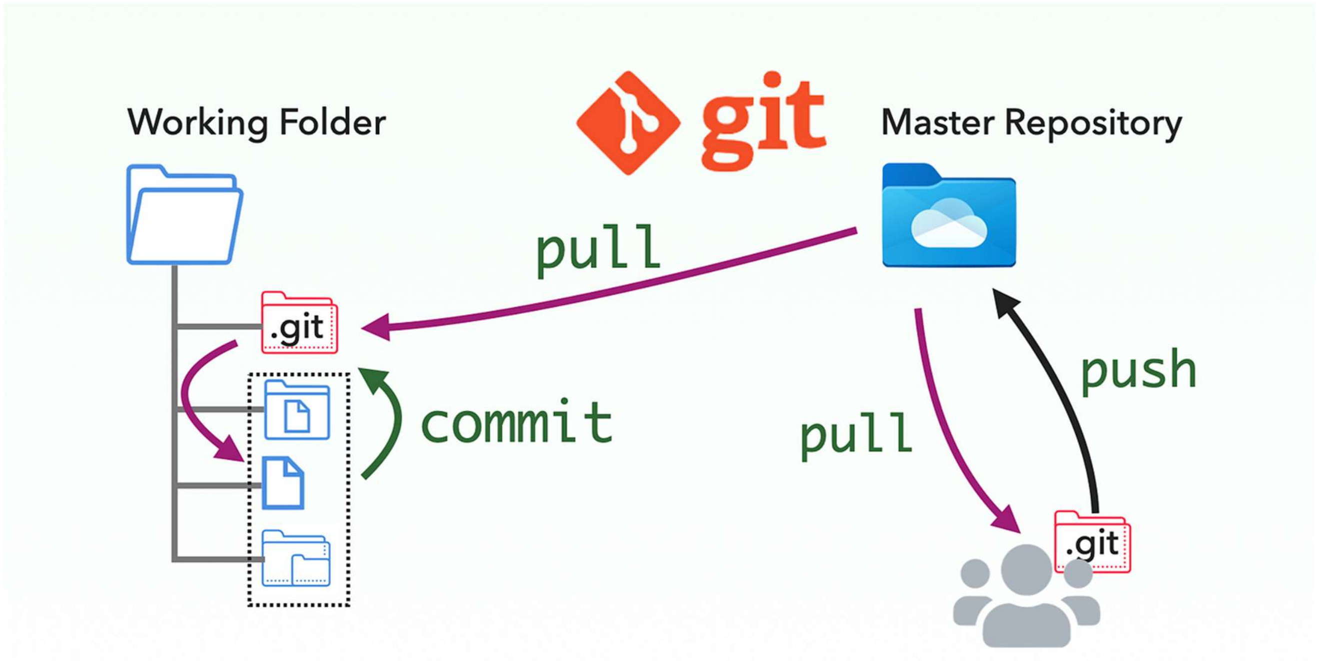
git

GIT HANDBOOK ZERO TO ADVANCE

BY SHAIKH IBRAHIM

Git Zero to Advance in Easy Language

Chapter 1 – Introduction to Git



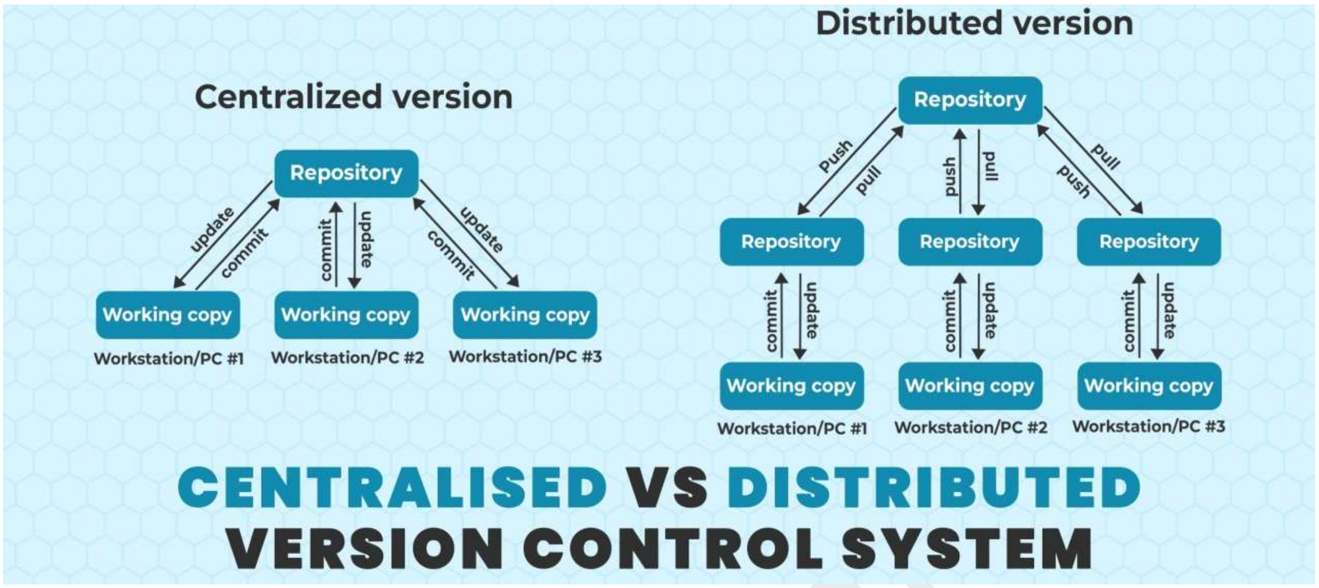
1.1 What is Git?

Git is a distributed version control system (DVCS) designed to track changes in source code during the software development lifecycle. It allows developers to record every modification made to files, maintain a complete history of changes, and collaborate efficiently with other team members.

Unlike simple file storage systems where files are overwritten, Git stores snapshots of your project at different points in time. This enables developers to move backward and forward through project history, compare versions, recover lost work, and understand exactly who changed what, when, and why.

Git was created by Linus Torvalds in 2005 to manage the Linux kernel project. Since then, it has become the industry standard for version control and is widely used in DevOps, cloud engineering, and CI/CD pipelines.

1.2 Why Version Control is Important



In real-world software projects, code is rarely written by a single person. Teams often consist of multiple developers working on the same codebase simultaneously. Without version control, this leads to problems such as:

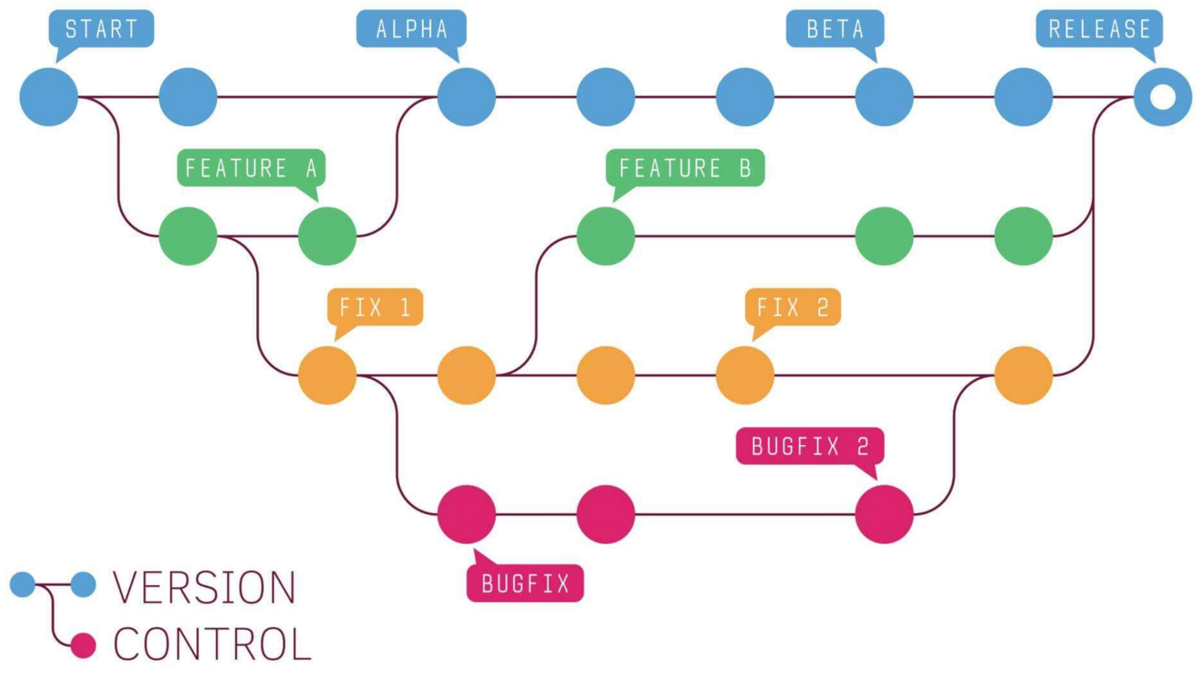
- Accidental overwriting of code
- No history of changes
- Difficulty identifying bugs
- No safe rollback mechanism

Git solves these problems by:

- Maintaining a complete change history
- Allowing multiple developers to work in parallel
- Providing tools to merge changes safely
- Enabling rollback to any previous stable version

Version control is not optional in modern development — it is a

mandatory skill for developers, DevOps engineers, and SREs



1.3 Centralized vs Distributed Version Control Systems

Before Git, most systems followed a centralized version control model.

Centralized Version Control (CVCS)

Examples: SVN, CVS

- **One central server stores the repository**
- **Developers pull code from the server**
- **If the server goes down, work stops**
- **Limited offline capability**

Distributed Version Control (DVCS – Git)

Examples: Git, Mercurial

- Every developer has a full copy of the repository
- Work can be done offline
- Faster operations
- Higher reliability and fault tolerance

Git's distributed nature is one of the biggest reasons for its global adoption.

1.4 Where Git is Used in Real Projects

Git is used everywhere in the software industry:

- Web development (Frontend & Backend)
- Mobile application development
- Cloud & DevOps automation
- Infrastructure as Code (Terraform, Ansible)
- Kubernetes deployments
- CI/CD pipelines (Jenkins, GitHub Actions, GitLab CI)

In DevOps, Git is often called the single source of truth, because all configuration, infrastructure code, and application code lives inside Git repositories.

1.5 Git in the DevOps Lifecycle

Git plays a critical role in modern DevOps workflows:

1. Developer writes code locally
2. Code is committed to Git

3. Git triggers CI pipeline
4. Automated testing runs
5. Build artifacts are generated
6. Deployment pipelines execute

This process ensures automation, consistency, and traceability across environments.

1.6 Basic Terminology (Conceptual Understanding)

Before using Git commands, it is important to understand some basic terms:

- Repository: A collection of files and folders tracked by Git
- Commit: A snapshot of changes
- Branch: An independent line of development
- Merge: Combining changes from branches
- Remote: A repository hosted on a server (GitHub, GitLab)

These concepts will be explained in detail in later chapters.

1.7 Advantages of Using Git

Git provides several advantages over traditional file management:

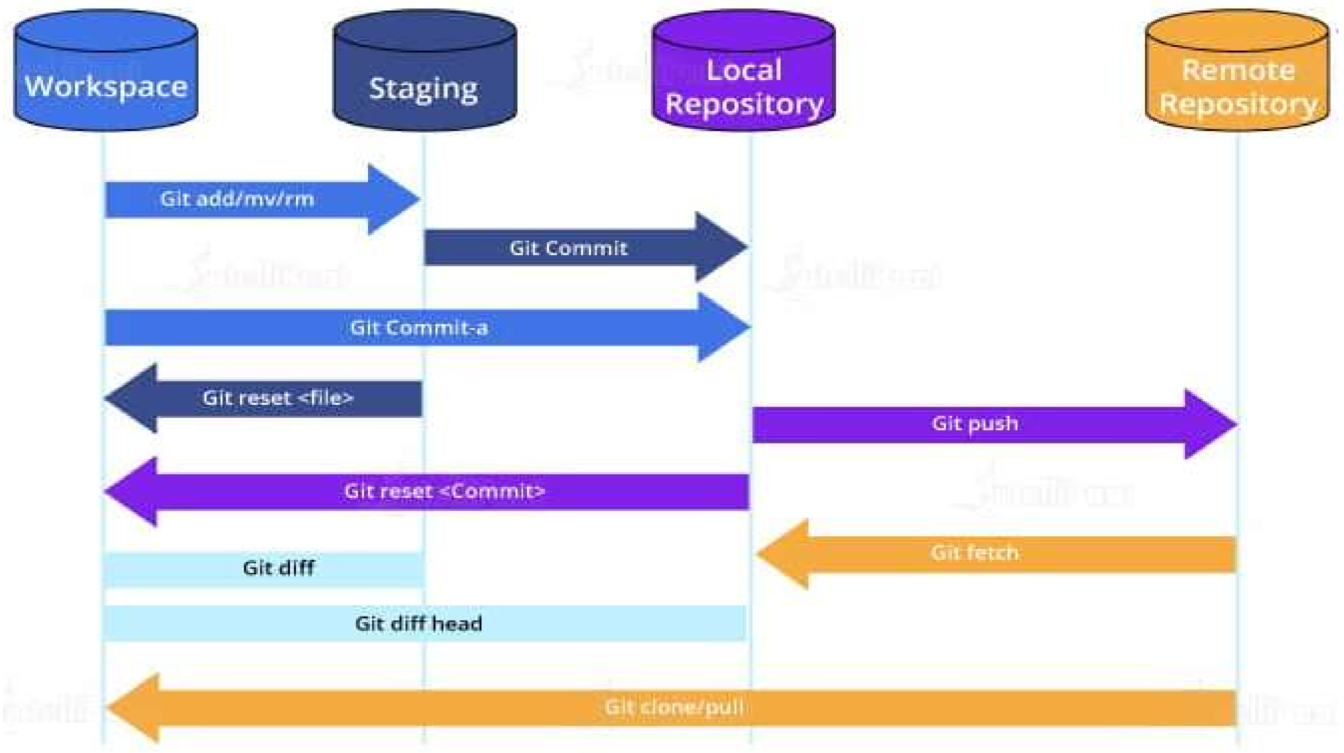
- Complete project history
- Easy collaboration
- Strong branching model
- Fast performance

- **Secure and reliable**
- **Industry-wide adoption**

Because of these advantages, Git knowledge is considered mandatory for job roles such as:

- **Software Developer**
- **DevOps Engineer**
- **Cloud Engineer**
- **Site Reliability Engineer (SRE)**

Chapter 2 – Git Architecture & Internal Working



Git works using a four-layer architecture that helps track changes safely and efficiently. Understanding this internal flow is extremely important because all Git commands operate on these layers.

2.1 Working Directory

The working directory is the actual folder on your system where project files exist. When you edit a file using any editor, the change happens in the working directory. Git does not automatically track these changes unless you explicitly tell it to.

2.2 Staging Area (Index)

The staging area acts as a temporary holding area. It allows developers to select which changes should be included in the next commit. This is useful when multiple files are modified, but only specific changes are ready.

2.3 Local Repository

The local repository is stored inside the hidden .git folder. It contains the complete commit history, branches, tags, and metadata. Once changes are committed, they are permanently saved here.

2.4 Remote Repository

A remote repository exists on platforms like GitHub, GitLab, or Bitbucket. It allows teams to collaborate by pushing and pulling code.

This layered design gives Git its power, flexibility, and safety.

Chapter 3 – Installing and Setting Up Git

Git must be installed and configured before use.

3.1 Installing Git

On Linux, Git is installed using package managers. On Windows, it comes with Git Bash, which provides a Linux-like terminal.

3.2 Verifying Installation

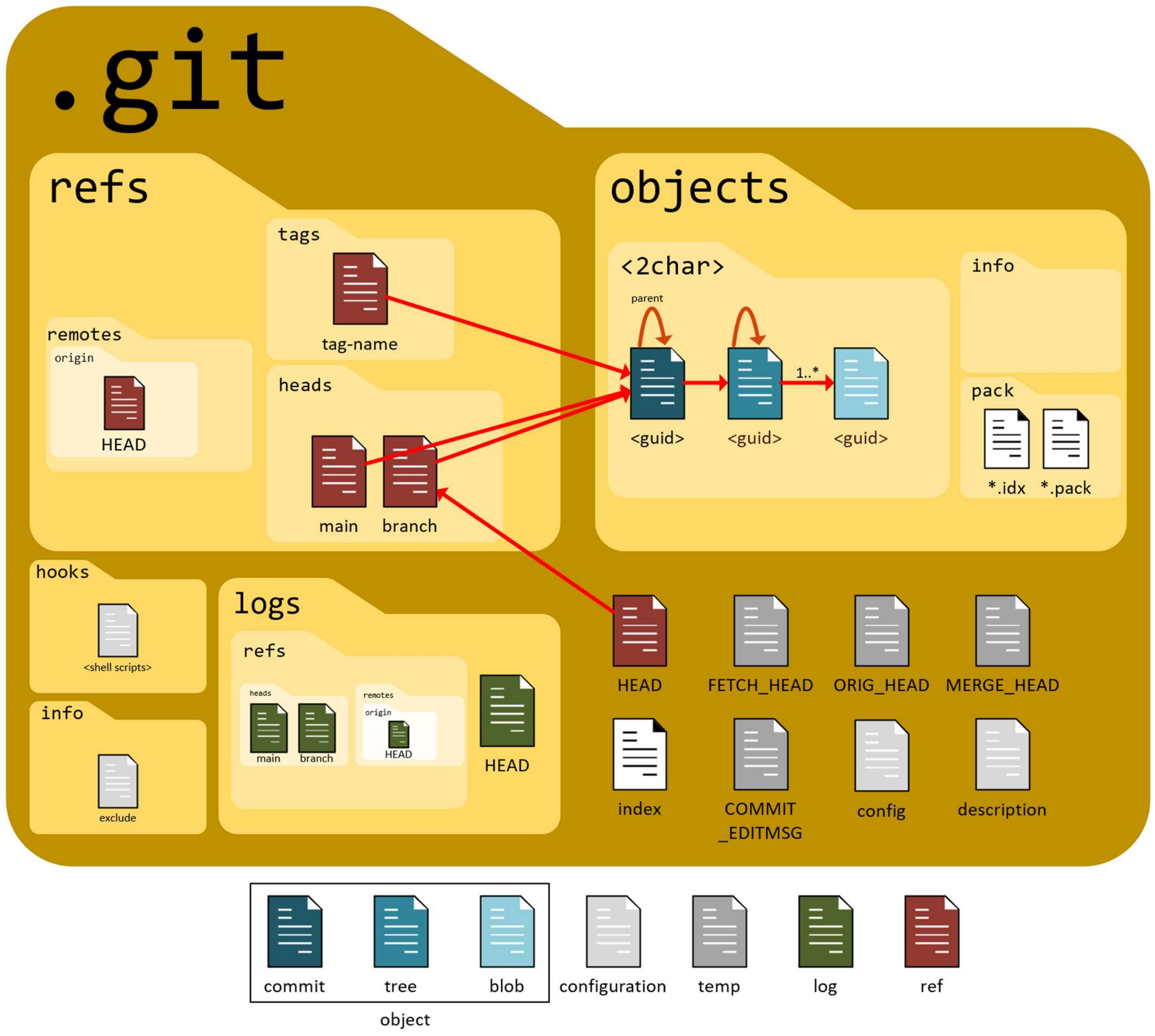
After installation, Git can be verified using the version command. This ensures Git is correctly installed and accessible.

3.3 Initial Configuration

Git requires a username and email address. These details are embedded into every commit and help identify the author of changes.

Configuration is done once and applies globally unless overridden per project.

Chapter 4 – Creating and Managing Repositories



A Git repository is where Git tracks all changes.

4.1 Initializing a Repository

When a repository is initialized, Git creates a .git directory. This directory stores all internal data required for version control.

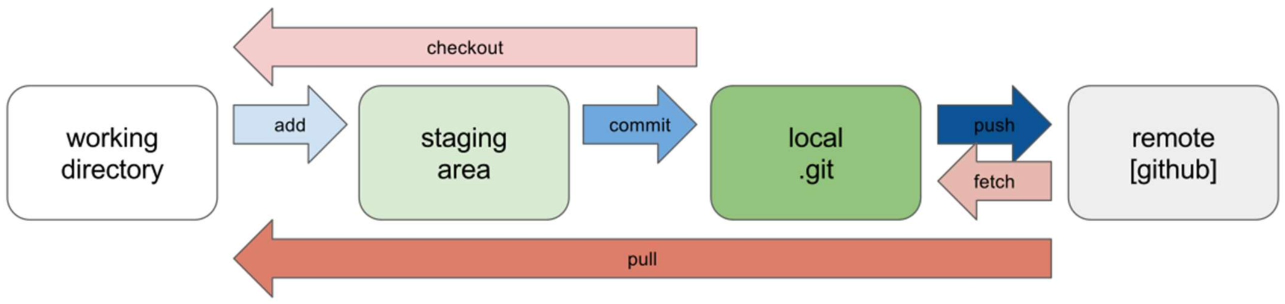
4.2 Repository Structure

The repository contains objects such as commits, trees, blobs, and references. Users do not usually interact with these directly, but they form Git's core.

4.3 Repository Lifecycle

Repositories can be local-only or connected to remote servers for collaboration.

Chapter 5 – Git Workflow Explained



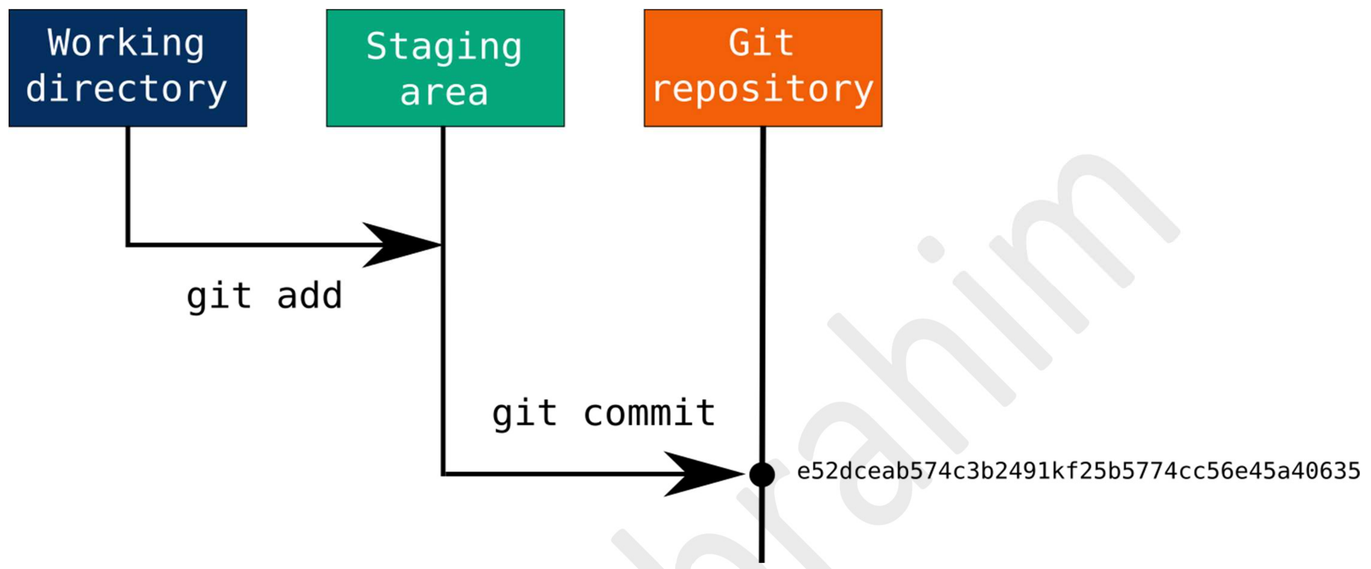
Git follows a predictable workflow:

1. **Modify files**
2. **Stage changes**
3. **Commit changes**
4. **Push to remote repository**

This workflow ensures controlled, trackable, and reversible changes.

Each step acts as a checkpoint, reducing the risk of accidental mistakes.

Chapter 6 – Understanding Git Status, Add, and Commit



6.1 Git Status

The status command shows:

- **Modified files**
- **Staged files**
- **Untracked files**

It is the most frequently used Git command.

6.2 Git Add

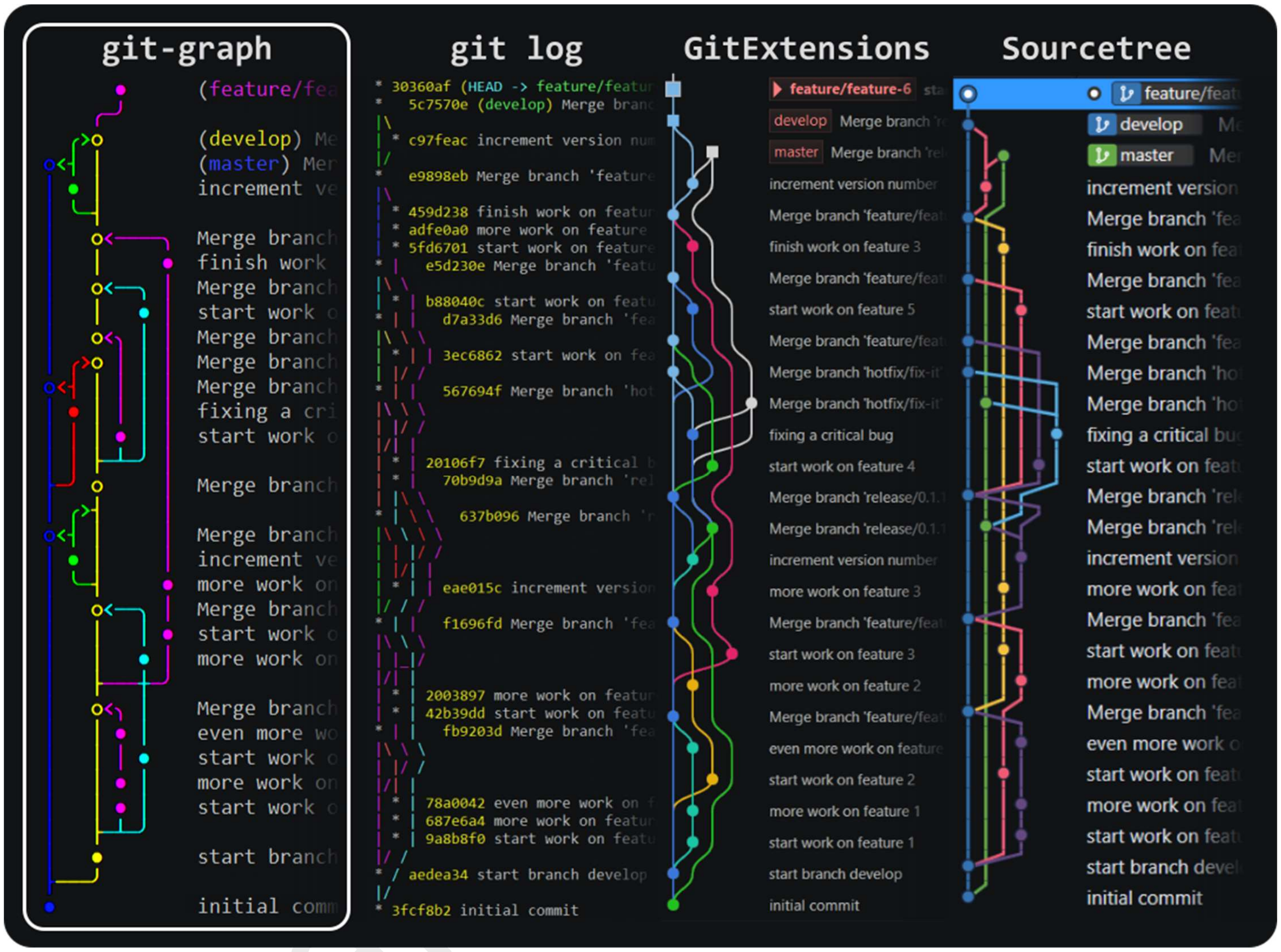
Adding files moves them from the working directory to the staging area. This allows selective commits.

6.3 Git Commit

A commit captures a snapshot of staged changes. Each commit has a unique ID (hash), author, timestamp, and message.

Commits form the project history.

Chapter 7 – Git Log and History Tracking



```
* 30360af (HEAD -> feature/featur
*   5c7570e (develop) Merge branc
|\
| * c97feac increment version num
|/
*   e9898eb Merge branch 'feature
|\
| * 459d238 finish work on featur
| * adfe0a0 more work on feature
| * 5fd6701 start work on feature
* |   e5d230e Merge branch 'featu
|\ \
| * | b88040c start work on featu
* | |   d7a33d6 Merge branch 'fea
|\ \ \
| * | | 3ec6862 start work on fea
| |/ /
* | |   567694f Merge branch 'hot
|\ \ \
| |/ /
|/| |
| * | 20106f7 fixing a critical b
| * |   70b9d9a Merge branch 'rel
| |\ \
* | \ \   637b096 Merge branch 'r
|\ \ \ \
| | |/ /
| |/| |
| * | | eae015c increment version
|/ / /
* | |   f1696fd Merge branch 'fea
|\ \ \
| |_|/
|/| |
| * | 2003897 more work on featur
| * | 42b39dd start work on featu
* | |   fb9203d Merge branch 'fea
|\ \ \
| |/ /
|/| |
| * | 78a0042 even more work on f
| * | 687e6a4 more work on featur
| * | 9a8b8f0 start work on featu
|/ /
* / aedea34 start branch develop
|/
* 3fcf8b2 initial commit
```

Git maintains a complete history of all changes.

7.1 Commit History

Every commit points to a previous commit, forming a chain. This allows Git to reconstruct any previous state.

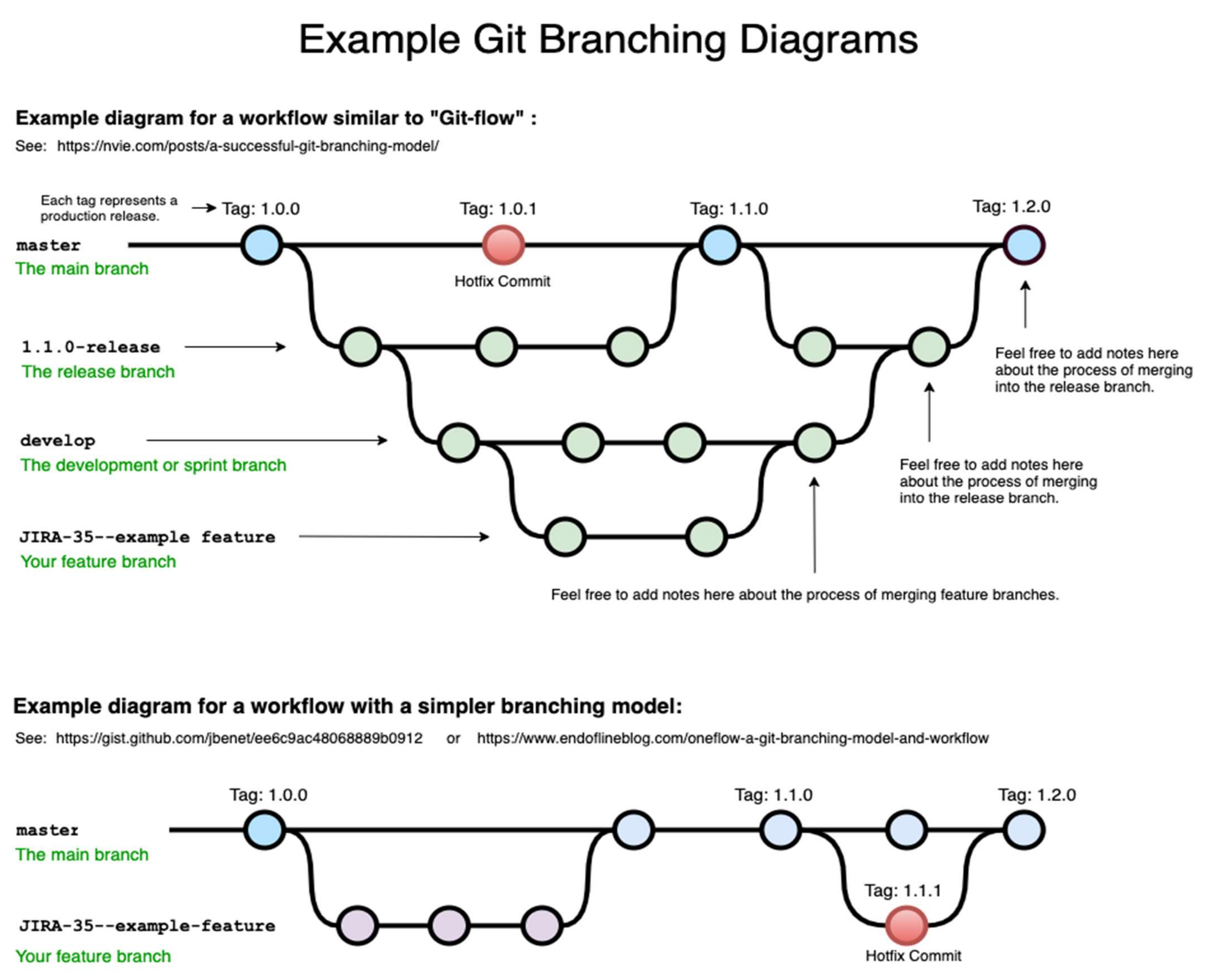
7.2 Viewing History

Git provides different log formats to visualize history, including graph-based views.

History tracking helps with:

- **Debugging**
- **Auditing**
- **Code reviews**

Chapter 8 – Branching in Git


Example Git Branching Diagrams
Example diagram for a workflow similar to "Git-flow" :
See: https://nvie.com/posts/a-successful-git-branching-model/
Each tag represents a production release.
Tag: 1.0.0
Tag: 1.0.1
Tag: 1.1.0
Tag: 1.2.0
master
The main branch
Hotfix Commit
1.1.0-release
The release branch
Feel free to add notes here about the process of merging into the release branch.
develop
The development or sprint branch
Feel free to add notes here about the process of merging into the release branch.
JIRA-35--example feature
Your feature branch
Feel free to add notes here about the process of merging feature branches.
Example diagram for a workflow with a simpler branching model:
See: https://gist.github.com/jbenet/ee6c9ac48068889b0912 or https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow
Tag: 1.0.0
Tag: 1.1.0
Tag: 1.2.0
master
The main branch
Tag: 1.1.1
JIRA-35--example-feature
Your feature branch
Hotfix Commit

Branches allow parallel development.

8.1 What is a Branch

A branch is a pointer to a commit. Creating a branch does not

duplicate code; it creates a new reference.

8.2 Why Branching is Powerful

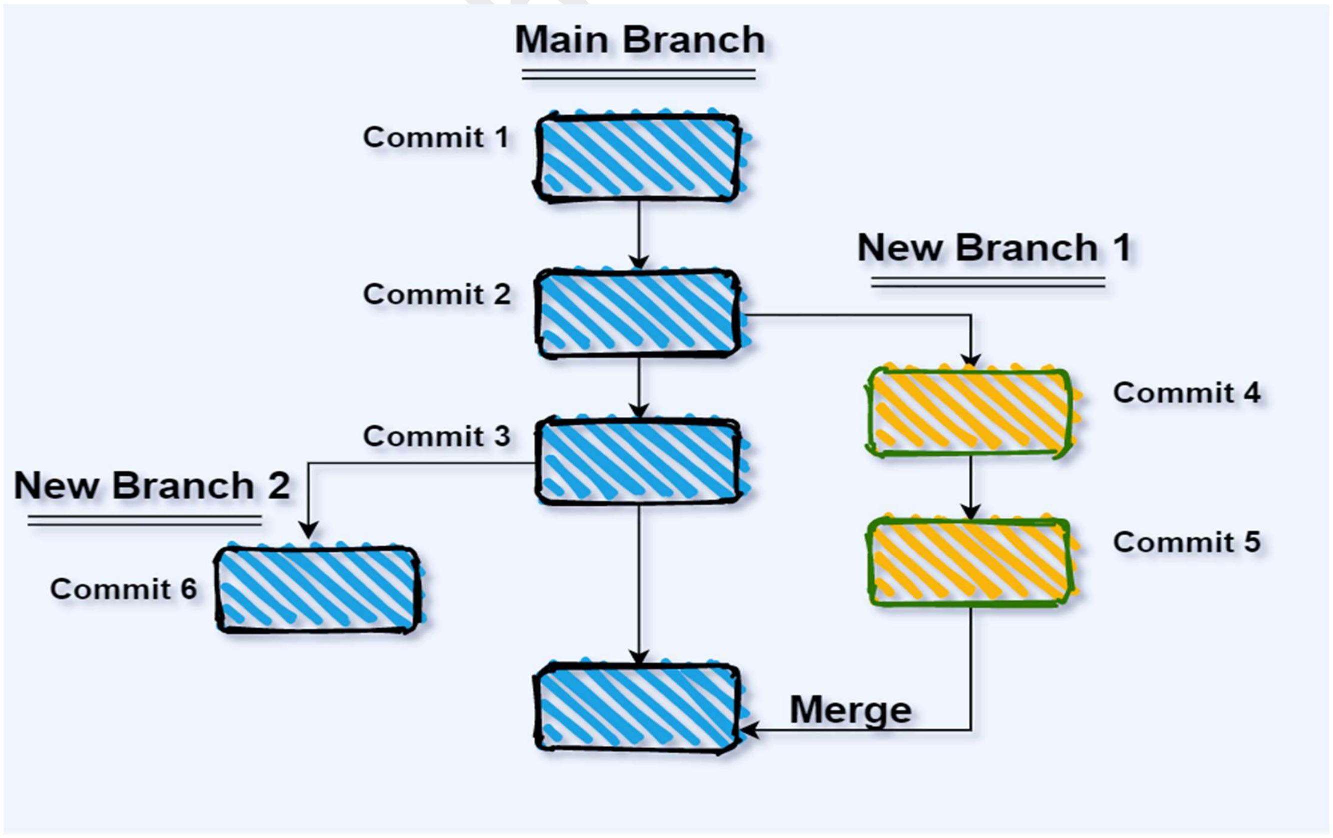
- Isolates features
- Prevents breaking main code
- Enables team collaboration

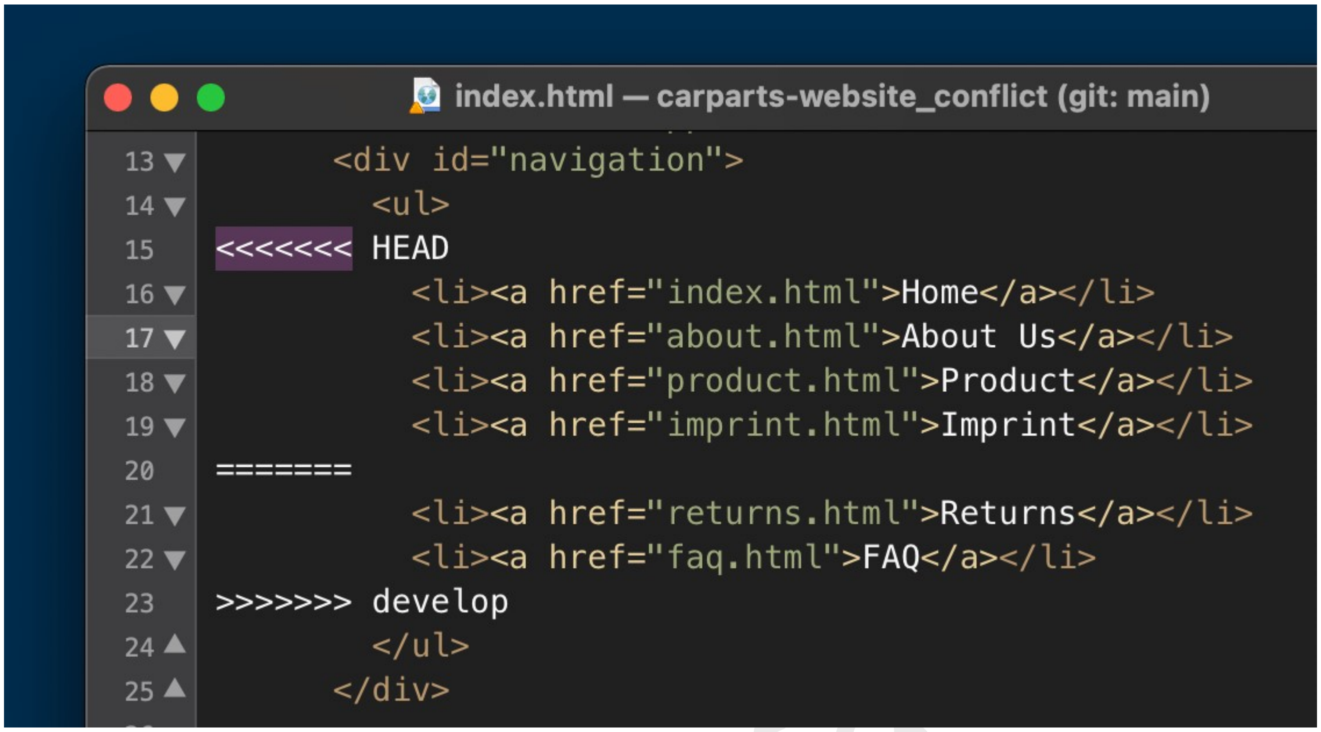
8.3 Common Branching Strategies

- Feature branches
- Release branches
- Hotfix branches

Branching is one of Git's strongest features.

Chapter 9 – Merging and Merge Conflicts





```
      <div id="navigation">
        <ul>
<<<<<<< HEAD
          <li><a href="index.html">Home</a></li>
          <li><a href="about.html">About Us</a></li>
          <li><a href="product.html">Product</a></li>
          <li><a href="imprint.html">Imprint</a></li>
=======
          <li><a href="returns.html">Returns</a></li>
          <li><a href="faq.html">FAQ</a></li>
>>>>>>> develop
        </ul>
      </div>
```

Merging combines changes from one branch into another.

9.1 Fast-Forward Merge

Occurs when the target branch has no new commits.

9.2 Three-Way Merge

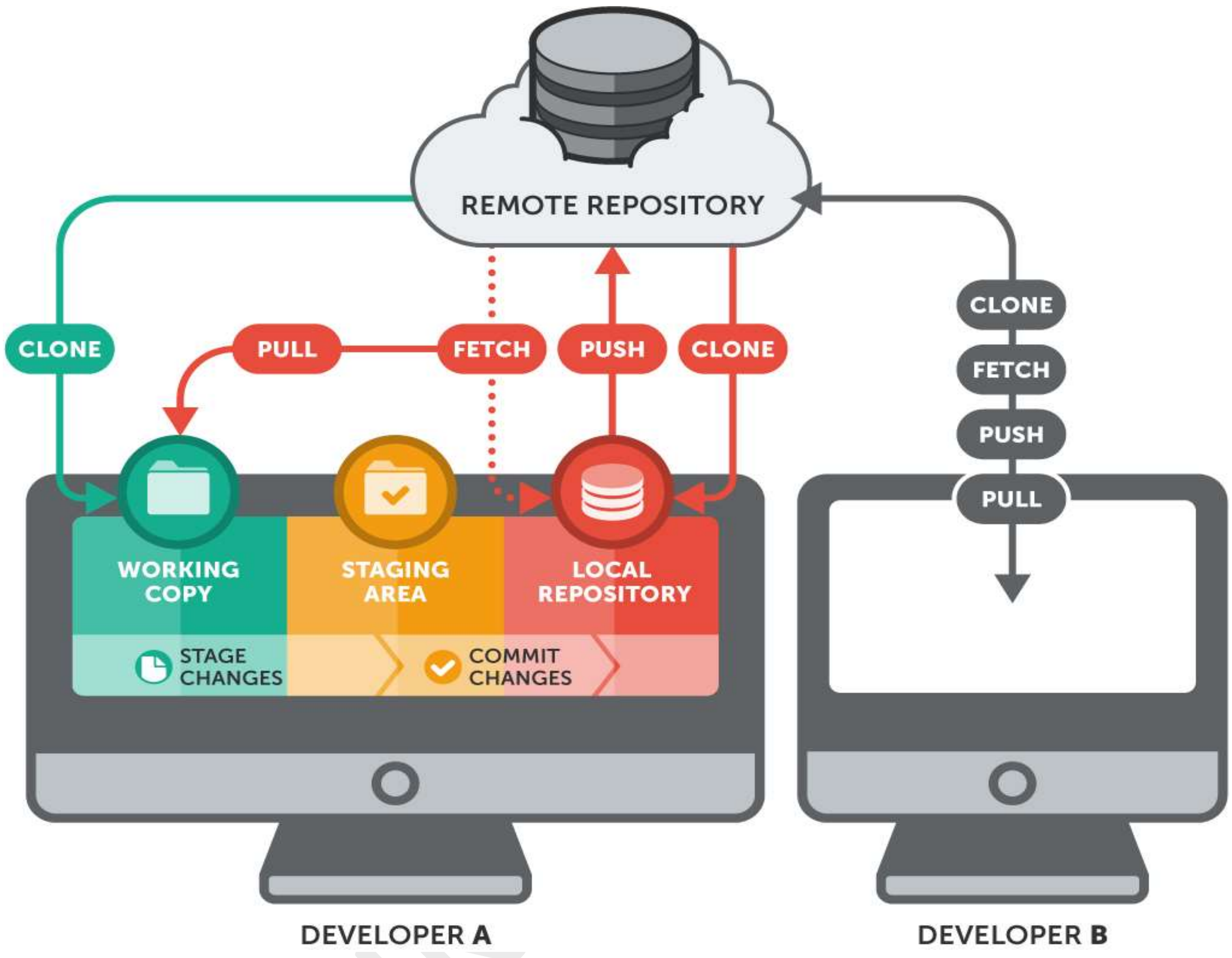
Git creates a new merge commit when branches diverge.

9.3 Merge Conflicts

Conflicts occur when the same lines are modified in different branches. Git marks conflicts and requires manual resolution.

Understanding conflict resolution is critical in real projects.

Chapter 10 – Working with Remote Repositories



Remote repositories enable collaboration.

10.1 Adding Remotes

A remote links your local repository to a shared server.

10.2 Push Operation

Pushing uploads commits to the remote repository.

10.3 Pull Operation

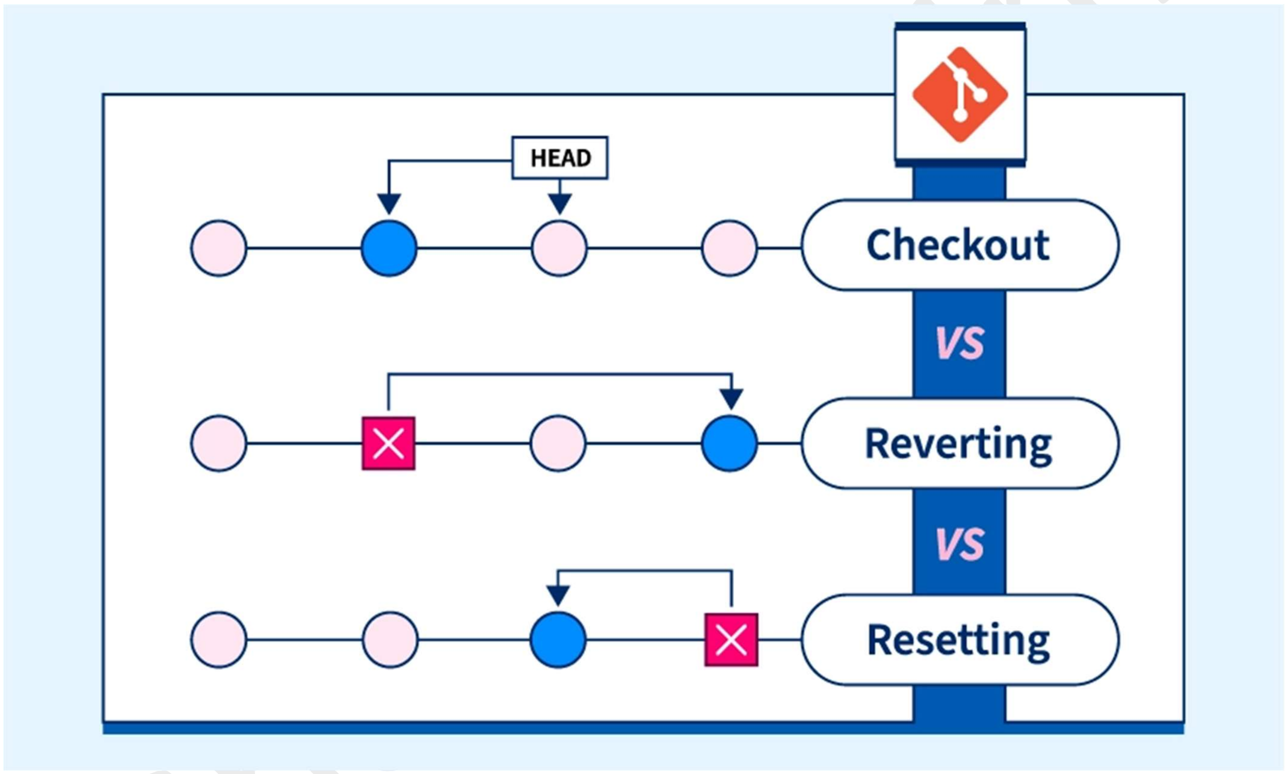
Pulling fetches and merges changes from the remote repository.

10.4 Collaboration Flow

- Developers push changes
- Others pull updates
- Conflicts are resolved collaboratively

This forms the foundation of team-based development.

Chapter 11 – Undoing Changes in Git



Mistakes are a natural part of software development. Git provides multiple mechanisms to undo changes safely, depending on where the change exists in the Git workflow.

11.1 Undoing Changes in the Working Directory

When a file is modified but not staged, Git allows reverting it back to the last committed state. This is useful when accidental edits are made or experimental changes are no longer needed.

11.2 Undoing Changes in the Staging Area

Sometimes files are staged by mistake. Git provides a way to remove files from the staging area without deleting the changes themselves.

11.3 Undoing Commits

Git allows undoing commits in different ways:

- Soft undo (keep changes)
- Hard undo (discard changes)
- Safe undo using revert

Choosing the correct method is critical, especially when working with shared repositories.

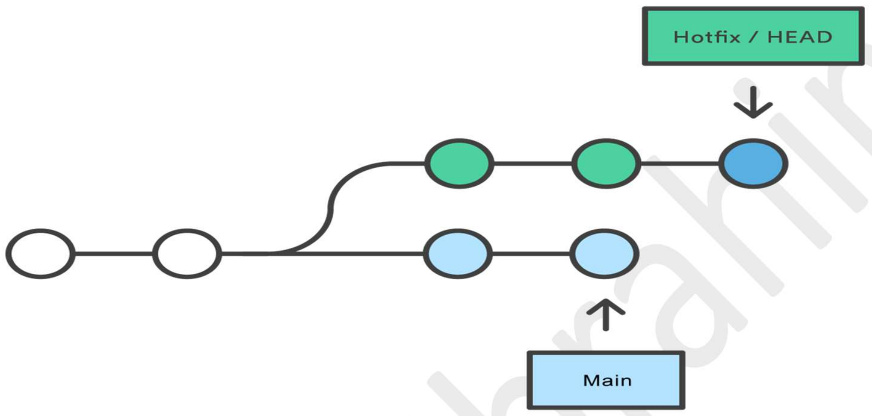
11.4 Best Practices for Undo

Undo operations should be used carefully. In collaborative environments, rewriting shared history can cause serious issues. Safer commands should be preferred for public branches.

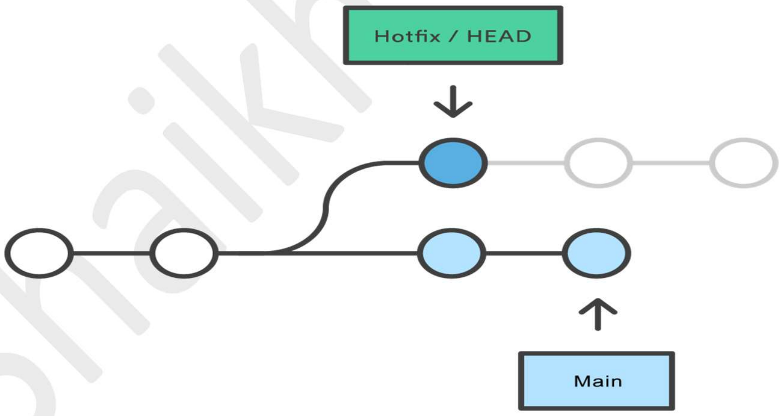
Chapter 12 – Git Reset, Revert, and Checkout (Deep Comparison)

Resetting the hotfix to HEAD

Before



After



Git provides multiple commands that appear similar but behave very differently.

12.1 Git Reset

Reset moves the branch pointer backward. It can operate in three modes:

- Soft: moves HEAD only
- Mixed: resets staging area
- Hard: resets working directory, staging, and commits

Reset is powerful but dangerous if misused.

12.2 Git Revert

Revert creates a new commit that reverses the changes of a previous commit. It does not modify history, making it safe for shared branches.

12.3 Git Checkout

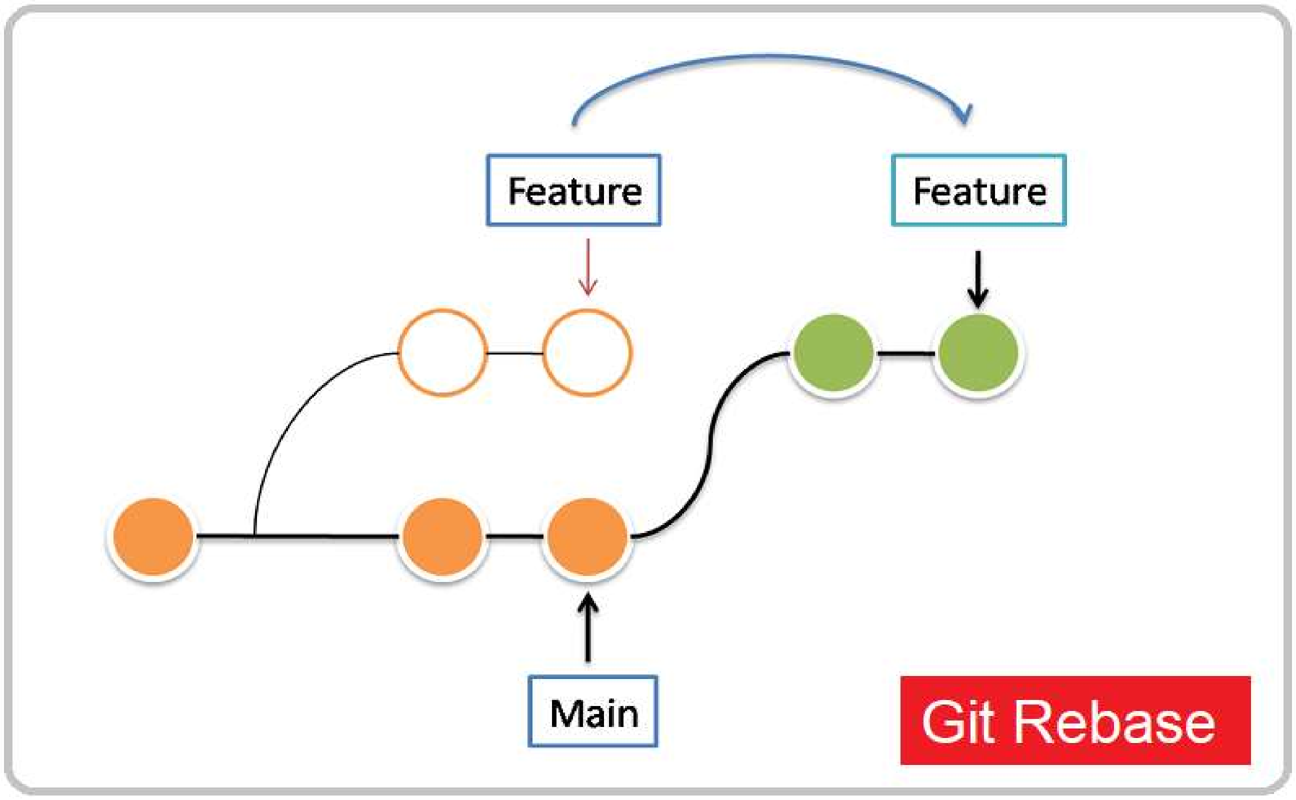
Checkout switches branches or restores files. It does not remove commit history but changes the working context.

12.4 When to Use What

- Reset: private/local corrections
- Revert: shared branch fixes
- Checkout: navigation and file recovery

Understanding these differences is essential for professional Git usage.

Chapter 13 – Git Rebase (Advanced History Management)



Rebase is an advanced Git feature used to rewrite commit history in a cleaner and more linear manner.

13.1 What is Rebase

Rebase moves a branch to a new base commit and reapplies its changes on top. This eliminates unnecessary merge commits.

13.2 Why Rebase is Used

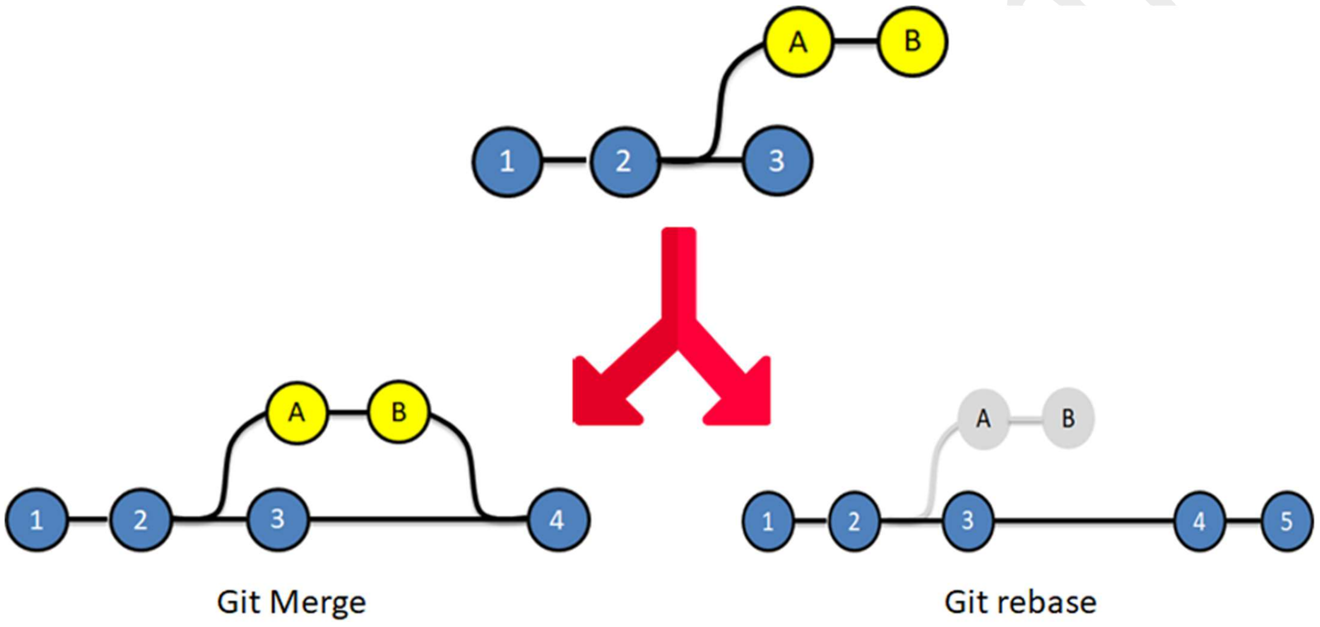
- Cleaner commit history
- Easier debugging
- Linear project timeline

13.3 Interactive Rebase

Interactive rebase allows:

- **Editing commit messages**
- **Squashing commits**
- **Reordering commits**
- **Removing commits**

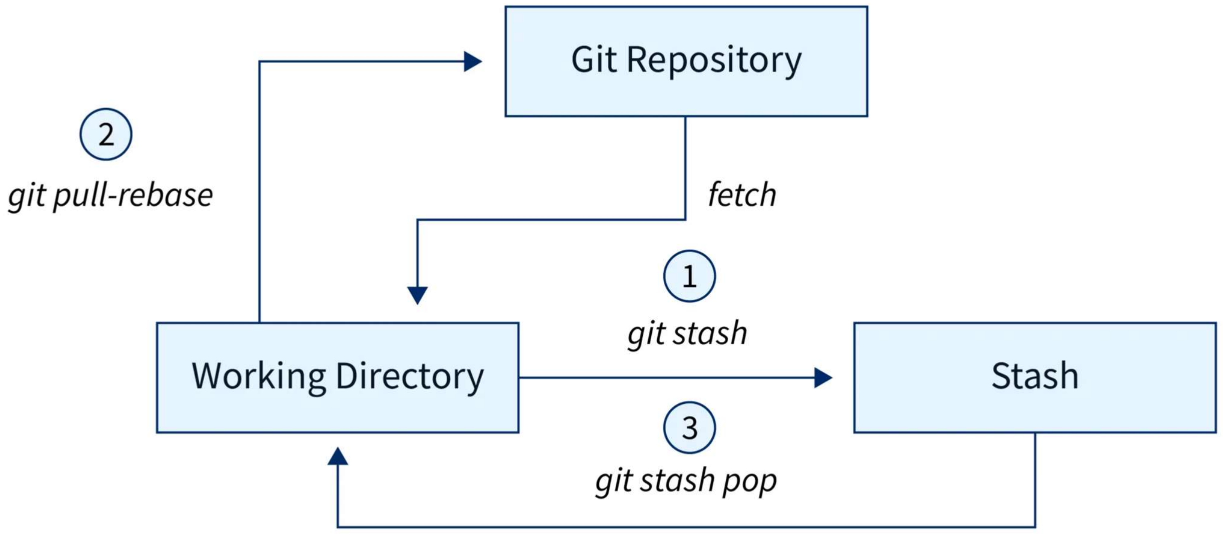
This is often used before merging feature branches.



13.4 Risks of Rebase

Rebasing public branches can rewrite shared history and cause conflicts. Rebase should be used carefully and mostly on local branches.

Chapter 14 – Git Stash (Temporary Change Management)



Git stash is used to temporarily store uncommitted changes without committing them.

14.1 Why Stash is Needed

There are times when developers must switch branches quickly without committing incomplete work. Stash solves this problem.

14.2 How Stash Works

Git saves changes into a stack-like structure and restores the working directory to the last commit.

14.3 Applying and Dropping Stashes

Stashed changes can be:

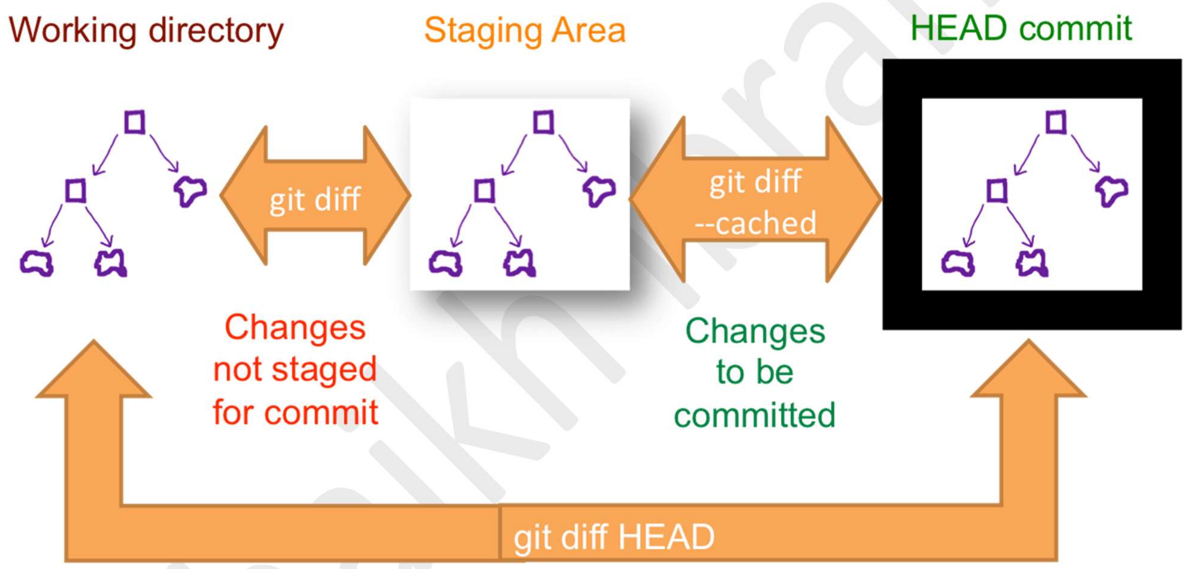
- Applied and kept
- Applied and removed
- Permanently deleted

14.4 Real-World Use Case

Stash is commonly used during:

- **Emergency bug fixes**
- **Branch switching**
- **Context switching**

Chapter 15 – Git Diff (Tracking and Reviewing Changes)



Git diff shows exact changes between files, commits, and branches.

15.1 Diff in Working Directory

Shows differences between modified files and the last commit.

15.2 Diff in Staging Area

Helps review what is staged and ready for commit.

15.3 Diff Between Commits

Useful for code reviews and debugging regressions.

15.4 Importance of Git Diff

Git diff is essential for:

- **Code review**
- **Debugging**
- **Quality control**
- **Understanding impact of changes**

Professional developers rely heavily on diff before committing or merging code.
