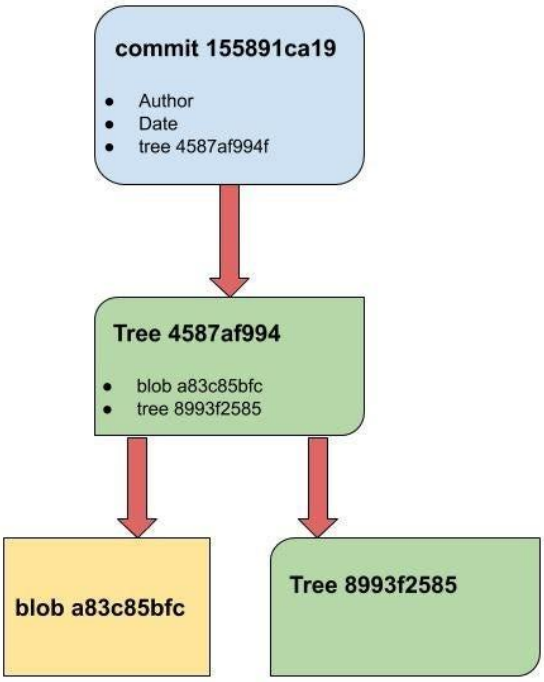
1. **Explain which type of Git object is used to store the contents of a file and how this object fits into Git's object model.**

   In Git, the contents of a file are stored in a blob object (Binary Large Object). Blobs store the file's binary data but contain no other metadata, such as the filename or directory path. This design allows Git to avoid storing duplicate files and to track content independently of its location in the directory structure.



   Git's object model is a directed acyclic graph (DAG) made up of four primary object types, each identified by a unique SHA-1 hash of its content.

   The four object types:

- Blob: Stores the actual file data. When you git add a file, Git reads its contents, calculates the SHA-1 hash, and saves the data as a blob object in the .git/objects directory. If another file with identical content is added, Git reuses the existing blob instead of creating a new one.

- Tree: Represents a directory. A tree object contains a list of entries, each with a mode (permissions), object type (blob or tree), SHA-1 hash, and filename. It links filenames and directory structure to the content stored in blob objects.
- Commit: A snapshot of your project at a specific point in time. A commit object contains the following information:
    a) A reference to the root tree object, which holds the entire directory structure for that commit.
    b) One or more parent commits, which links the commit to its history.
    c) Metadata, such as the author, committer, timestamp, and commit message.
- Annotated Tag: A special kind of object that permanently marks a specific commit, often used for marking release versions.

  Putting the pieces together

  The blob object is the foundational component of this model, but it is only made meaningful by the other objects.
- When you commit changes, Git recursively creates tree objects for each directory, which point to the appropriate blob objects for each file.
- A commit object is then created, which points to the top-level tree, effectively capturing a snapshot of the entire project at that moment.
- To restore the state of a project at any given commit, Git simply reads the commit object, traverses its tree object structure, and uses the stored blob hashes to retrieve the file contents.

2. **Git allows configuration at system, global, and local levels. Explain which level takes priority if the same setting is defined in multiple places, and why this design is useful.**

   When multiple Git configuration levels define the same setting, the local level takes the highest priority, followed by the global level, with the system level having the lowest priority. Git searches for a setting starting at the most specific level and stops once it finds a value. The search order is:

   1. Local (.git/config): Specific to a single repository.
   2. Global (~/.gitconfig): Applies to a single user on a machine.
   3. System (/etc/gitconfig): Applies to all users and all repositories on a machine.

   Design is useful for:

   This cascading priority is an intentionally flexible design that gives developers fine-grained control over their environment, from broad team standards down to individual project requirements.

3. **Compare .gitignore and .git/info/exclude. How are they similar, and in what situations would you use one instead of the other?**

   Similarities

- Purpose: Both files serve the same primary function: to specify patterns for files or directories that should be excluded from Git's tracking.
- Target: Rules in both files apply only to untracked files. They do not affect files that have already been committed to the repository.

Differences

Characteristic	.gitignore	.git/info/exclude
Scope	Applies to the entire repository and is effective for all users who clone it.	Applies only to the single, local copy of the repository and is not shared.
Version Control	Is part of the project's source tree and is committed to the repository. Changes to it are version-controlled and shared.	Is an internal Git file that is not committed or shared. It is local to your working copy.
Hierarchy	Can be placed in multiple directories throughout a repository. Rules are applied based on their location in the directory tree.	There is only one .git/info/exclude file per repository, located in the .git directory.
Precedence	Lower precedence than .git/info/exclude. An exclusion pattern in .git/info/exclude will override a pattern in .gitignore for a particular file.	Higher precedence than .gitignore.

When to use one instead of the other

Use .gitignore when:

- Ignoring project-generated files: Use this for files that all project contributors should ignore, such as compiled code, build artifacts (.o, .class files), dependency caches, or logs.
- Standardizing ignore rules for a team: By committing a .gitignore file, you ensure that every developer cloning the repository will automatically

ignore the same set of files, which helps maintain a clean and consistent project history.

- Ignoring a common file type: If you want to ignore a specific file type across the entire project (e.g., all .log files), .gitignore is the correct choice.

Use .git/info/exclude when:

- Ignoring personal or local-only files: This is ideal for files that should only be ignored in your local working copy and not shared with others. For example, a local configuration file for your IDE that isn't standardized across the team, or temporary files created by your specific workflow.
- Protecting sensitive local information: If you have temporary API keys or configuration settings that should never be committed, adding them here ensures they are ignored, without broadcasting the rule to the entire team.
- Overriding a global ignore rule for a specific repository: If you have a global ignore file (defined in your Git config), you can use .git/info/exclude to override a pattern specifically for one repository.

4. What is the difference between git diff and git diff --staged? Describe a scenario where each command would be useful.

The core difference between git diff and git diff --staged is which two versions of your files they compare.

- git diff compares your working directory with your staging area (also known as the index). It shows you the changes you have made that you have *not* yet added with git add.
- git diff --staged compares your staging area with the last commit (HEAD). It shows you exactly which changes you have marked to be included in your *next* commit. The alias git diff --cached does the same thing.

Scenario for git diff

Imagine you are working on two separate features or bug fixes at once. You have modified several files in your working directory, but you want to organize your changes into separate, logical commits.

Workflow:

1. After editing multiple files (index.html, style.css, and script.js), you run git status. You see a list of modified files under "Changes not staged for commit."
2. You decide that the changes to index.html and style.css belong together in one commit, but script.js is a separate fix.
3. You stage the files for the first commit: git add index.html style.css
4. You run git diff to confirm what is left. The output shows only the changes you made to script.js, allowing you to review and verify the unstaged changes separately from what you are about to commit.

Scenario for git diff --staged

After staging the changes for your next commit, you should perform one final review before making the commit permanent.

Workflow:

1. Continuing from the previous example, you have staged your changes for index.html and style.css.
2. You run git diff --staged. The output shows you the exact set of changes to those two files that will be included when you run git commit. This is your last line of defense against accidentally committing an unintended change, such as a debugging line you forgot to remove.
3. Once satisfied, you finalize the commit: git commit -m "Add new welcome message and style updates".

5. **If you accidentally staged a file, how would you remove it from the staging area but keep your modifications in the working directory? Explain why this might be necessary.**

To remove a file from the staging area while keeping your modifications in the working directory, you can use one of two commands: git restore --staged (the modern, recommended approach) or git reset (the traditional method).

To unstage a single file:
git restore --staged <file>

To unstage all files in the current directory:
git restore --staged .

Why this might be necessary

Unstaging a file is a crucial step in maintaining a clean and accurate commit history. Here are some common scenarios where it is necessary:

- Accidental staging: You may use git add . to stage multiple files but realize that one of them, perhaps a temporary file or a .env file containing sensitive information, was included by mistake. Unstaging it allows you to prevent it from being included in the commit without losing your work.
- Splitting commits: You might make several unrelated changes in different files and stage them all at once. Before committing, you may decide that they belong in separate, logical commits. By unstaging some of the files, you can create a smaller, focused commit for one feature, then stage and commit the other changes separately.
- Revisiting changes: After staging a file, you might realize you need to make more changes before it's ready to be committed. Unstaging it returns the file to the "Changes not staged" state, allowing you to edit it further before adding it to the staging area again.

6. **Can you directly alias git commit as git ci using Git configuration? Why or why not? If not, what alternatives exist?**

   Yes, you can directly alias git commit as git ci using Git's configuration. The alias system is one of Git's built-in features and is the standard way to create shortcuts for Git commands.

   Using the command line

   This is the simplest way to create a global alias (for all repositories) from your terminal.

   git config --global alias.ci commit

   Manually editing the configuration file

   You can also open your global Git configuration file (~/.gitconfig on Linux/macOS or %USERPROFILE%\.gitconfig on Windows) and add the alias manually under the [alias] section.

   Why this works

   The Git alias system is designed specifically for this purpose.

- When you run git ci, Git looks for ci in its list of built-in commands.
- Since ci is not a built-in command, Git checks your configuration files for a corresponding alias.
- It finds alias.ci, which tells Git to replace ci with commit and then execute the full git commit command.
- Any arguments you pass to git ci are automatically passed along to the git commit command. For example, git ci -m "My message" expands to git commit -m "My message".

   Alternatives

   Shell aliases

   For simpler command shortening, you can use your shell's alias system instead of Git's.

7. **What does the init.defaultBranch setting control in Git, and why might teams choose to set it differently?**

The init.defaultBranch setting controls the name of the initial branch created when you run git init to start a new, empty repository. Before Git version 2.28 (released in July 2020), this was hard-coded to master. Now, you can set a different default branch name, for example: git config --global init.defaultBranch main.

Why teams set it differently

1. Inclusive language
2. Consistency with remote hosting services
3. Standardizing across an organization
4. Clearer and more descriptive names

8. Every commit in Git points to at least one tree object. Explain what this means and why it is important for Git's structure.

In Git, a commit does not store a list of changes. Instead, every commit records a complete snapshot of the repository's entire file and directory structure at the moment the commit was made.

This snapshot is represented by a single root tree object. This root tree object then points to other tree objects (for subdirectories) and blob objects (for files), recursively detailing the complete state of the project's file system.

Why it's important for Git's structure

1. Data integrity and efficiency
   - Immutability: Each tree and blob object is named with a unique SHA-1 hash of its contents. This makes all objects in the Git database immutable. If the contents of a file change, a new blob is created with a new hash, and the tree object referencing it is updated to reflect this. This immutability guarantees data integrity.
   - Deduplication: Because objects are content-addressed, Git only stores each unique version of a file once. If two files have identical

content or if a file's content doesn't change across several commits, they will both point to the same blob object. This is a highly efficient way to store data and saves significant disk space.

- Renaming efficiency: When you rename a file, only the tree object that tracks its name is updated. The underlying blob object containing the file's content remains the same, which makes file renames and moves a very cheap operation in Git. Git can infer that a file was renamed by comparing tree objects.

2. Complete, self-contained history

- Full snapshots: Since each commit points to a full snapshot via its root tree, Git can retrieve the exact state of a project at any given commit without having to apply a sequence of patches or deltas. This makes operations like checking out a previous commit (git checkout <commit>) or retrieving the version of a specific file straightforward and fast.
- Simplified merging: The snapshot-based model simplifies merging. Git can compare the snapshots of two different branches and efficiently determine the necessary changes to merge them, even if there is a complex history of changes in between.

3. Foundation of the object model

- The commit object: The tree object is a fundamental part of the commit object itself. A commit object is primarily made up of a pointer to a tree object, a pointer to its parent commit(s), and metadata like the author, date, and commit message.
- Building the project hierarchy: The tree structure is what allows Git to recreate your project's directory hierarchy. Without it, the blob objects would just be a collection of file contents without any information about their filenames or directory organization.

9. If you have staged changes in main and then switch to a feature branch, what happens to those staged changes? Why does Git behave this way?

If you have staged changes on your main branch and then switch to a feature branch, Git will move the staged changes with you as long as they do not conflict with changes in the target branch.
What happens:

- Case 1: No conflicts. If the staged changes in your main branch do not affect any files that are different in the feature branch, the switch will succeed. Your staged changes will still be staged in the new feature branch, and your working directory will be updated to reflect the files from the feature branch.
- Case 2: Conflicts exist. If the staged changes affect the same files that have been modified differently in the feature branch, Git will refuse to switch. It will display an error message explaining that your local changes would be overwritten by the checkout. This is a safety mechanism to prevent you from losing your work.

Why Git behaves this way

This behavior is rooted in Git's core structure, particularly how it uses a single, global staging area (also called the "index") and how branches work.

- A single staging area: Git does not have a separate staging area for each branch. Instead, there is one staging area shared by all branches in the repository. When you stage a change, you are simply adding it to this global staging area, regardless of which branch you are on.
- Branches are just pointers: A branch is a lightweight, movable pointer to a commit. When you switch branches, you are telling Git to make your new branch the active one by moving the HEAD pointer. The staged changes, which are stored separately from the commit history, move with you because they are not part of any specific commit yet.
- Safety first: Git's primary goal is to prevent data loss. Allowing a switch that would overwrite unsaved work would be destructive. By

checking for conflicts before switching, Git protects your changes and forces you to handle them explicitly.

10. Both git switch -c feature and git checkout -b feature create a new branch. Explain the difference between these two commands and why Git introduced switch.

Characteristic	git switch -c <branch>	git checkout -b <branch>
Primary Purpose	Creating and switching branches. It is a focused command that handles only branch operations.	A multi-purpose "Swiss Army knife" command that can switch branches, check out files, and move between commits.
Associated Command	Works alongside the git restore command, which is used for restoring files.	Historically handled both branch-switching and file-restoration tasks, leading to potential confusion.
Flexibility	Less flexible. Cannot be used to restore files from the staging area or to check out a specific commit.	More flexible. Can be used to create a branch, switch branches, restore files (git checkout -- <file>), or move to a detached HEAD state (git checkout <commit>).

Safety	Safer for branch operations because it cannot accidentally overwrite files. For instance, git switch <filename> will not overwrite your file but will instead switch to a branch with that name (if it exists).	Potentially less safe. The command git checkout <filename> would restore the specified file from the staging area, which can accidentally discard your uncommitted changes.

Why git switch was introduced

git switch was introduced in Git version 2.23 (in 2019) to address the confusion surrounding the overloaded git checkout command. The reasons for this change include:

- Separation of Concerns: git checkout had too many responsibilities, performing both branch-switching and file-restoration. This ambiguity often led to user errors, especially for newcomers. The Git team split these functions into two focused, purpose-built commands: git switch for branches and git restore for files.
- Improved Clarity and Intuition: The new commands are more explicit and intuitive. A user running git switch clearly intends to change or create a branch, while a user running git restore clearly intends to restore file contents.
- Reduced Risk: By separating the commands, the risk of accidentally overwriting files is significantly reduced. You can no longer

accidentally discard your local file changes by using the wrong version of git switch, a common mistake with git checkout.

- Modern Git Practice: git switch and git restore are now considered best practice for modern Git workflows, offering a cleaner and more predictable experience. While git checkout is still fully functional for backwards compatibility, the newer commands are the recommended approach for their clarity and safety.

MCQ Questions:

1. Which of the following is NOT a benefit of using version control?
a) Collaboration among multiple developers
b) Tracking changes and history
c) Automatic bug fixing
d) Backup of source code

2. In a Centralized Version Control System (CVCS), where is the version database stored? a) On every developer's local computer
b) On a single central server
c) Only in the cloud
d) In the .git folder

3. Who developed Git and in which year?
a) Richard Stallman, 1991
b) Linus Torvalds, 2005
c) Dennis Ritchie, 1989
d) Guido van Rossum, 1995

4. Which command checks the installed version of Git?
a) git config --version
b) git --help
c) git --version
d) git show

5. Which term refers to copying an existing remote repository to your local machine?
a) Commit
b) Fork
c) Clone
d) Pull

6. Which area in Git acts as an intermediate space between the working directory and the repository?
a) Remote
b) Staging area (index)
c) Branch
d) .gitignore

7. Which command renames the current branch to main?
a) git rename main
b) git branch -M main
c) git switch main
d) git update main

8. Which command is used to download a remote repository for the first time?
a) git clone
b) git pull
c) git push
d) git fetch

9. What is the purpose of a pull request in GitHub?
a) To copy a repository from one account to another
b) To suggest merging changes from one branch into another
c) To delete a branch
d) To reset the commit history

10. If Peter wants to push changes to Daniel's repository but doesn't have permission, what must Daniel do?

a) Share his GitHub password with Peter

b) Add Peter as a collaborator

c) Run git push --force

d) Delete and recreate the repository