1. **Git-HOL**

In this hands-on lab, we will learn how to

1. Setup our machine with Git Configuration
2. Integrate notepad++.exe to Git and make it a default editor
3. Add a file to source code repository

* Setting up our machine with **Git Configuration**

**A screenshot of a computer

AI-generated content may be incorrect.**

* Configuring **user ID** and **email ID**

**A computer screen with white text

AI-generated content may be incorrect.**

* Integrating **notepad++.exe** to **Git**

**A black background with colorful text

AI-generated content may be incorrect.**

* Creating an alias command for **notepad++.exe**



A screenshot of a computer

AI-generated content may be incorrect.

* Verifying **notepad++** as the default editor

A black screen with white text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* Adding a file to source code repository

A black background with white text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* Creating the **welcome.txt** file, **adding content** to it and **verifying** it

A black background with white text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A black background with white text

AI-generated content may be incorrect.

* Making the file to be **tracked** by **Git Repository**





A black screen with white text

AI-generated content may be incorrect.

* Creating the Repository named “**GitDemo**”

A screenshot of a computer

AI-generated content may be incorrect.

* Using **push** command

A screenshot of a computer program

AI-generated content may be incorrect.

* Using **pull** command

A black screen with white text

AI-generated content may be incorrect.

* Final **Repository Output**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

1. **Git-HOL**

**Objectives**

**Q1. Explain git ignore**

Git Ignore is a mechanism to explicitly exclude specific files, directories, or patterns from being tracked by Git. This is achieved through a .gitignore file, which contains rules defining what should be ignored.

**Syntax Rules**

* / → Ignores in the **root directory only**.
* \* → Wildcard for any characters.
* ! → Exception rule (negation).
* # → Comments.

**Q2. Explain how to ignore unwanted files using git ignore.**

1. **Create .gitignore**

# At the repository root:

touch .gitignore

1. **Define Rules**  
   Edit .gitignore with patterns (e.g., for a Python project):

# Ignore compiled Python files

\_\_pycache\_\_/

\*.pyc

# Ignore logs and temporary files

\*.log

temp/

1. **Validate Ignored Files**

git status *# Ignored files won’t appear*

1. **Commit .gitignore**

git add .gitignore

git commit -m "Add ignore rules for logs and binaries"

In this hands-on lab, we will learn how to:

1. Implement **git ignore** command to **ignore** unwanted files and folders

* Navigating to our **Git Repository**

A black background with colorful text

AI-generated content may be incorrect.

* Creating sample files to **ignore**

A screen shot of a computer code

AI-generated content may be incorrect.

* Creating **.gitignore** file



A screenshot of a computer

AI-generated content may be incorrect.

* Verifying local files

A screenshot of a computer

AI-generated content may be incorrect.

* Checking **git status**

A screen shot of a computer

AI-generated content may be incorrect.

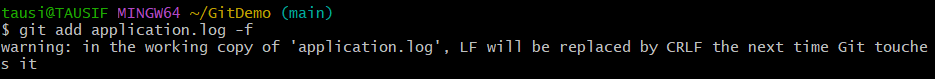
* **Adding** and **Committing .gitignore**



A screen shot of a computer code

AI-generated content may be incorrect.

* Trying to force-add a log file and checking status



A screen shot of a computer

AI-generated content may be incorrect.

* Pushing changes to **GitHub**

A screenshot of a computer program

AI-generated content may be incorrect.

* Verifying **GitHub Repository**

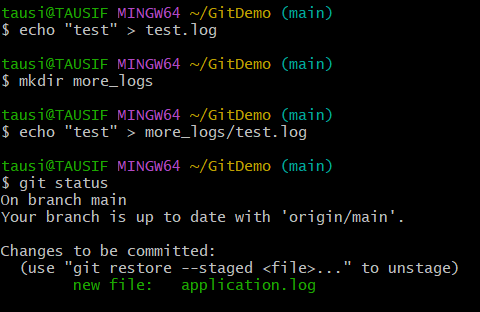
A screenshot of a chat

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* Creating new test files to verify **ignoring rules**



1. **Git-HOL**

**Objectives**

**Q1. Explain branching and merging.**

**Branching:**

Creates an independent line of development

**Syntax:**

git branch <branch\_name>

git checkout <branch\_name>

git checkout -b <branch\_name>

**Merging:**

Combines branch changes into another branch (typically main)

**Syntax:**

git checkout main

git merge feature/login

**Q2. Explain about creating a branch request in GitLab.**

**Method 1: From Git CLI**

git push origin feature/login *# Push local branch*

**Method 2: GitLab UI**

1. Navigate to Repository > Branches
2. Click "New branch"
3. Enter:
   * Branch name: feature/login
   * Create from: main (or other base branch)
4. Click "Create branch"

**Q3. Explain about creating a merge request in GitLab.**

1. **After pushing our branch:**

git push -u origin feature/login

1. **In GitLab UI:**
   * Go to Merge Requests > New Merge Request
   * Select:
     + Source: feature/login (our branch)
     + Target: main (destination branch)
2. **Configure:**
   * Title/Description
   * Assignee (team member to review)
   * Milestone/Labels (optional)
3. Click "Create merge request"

In this hands-on lab, we will learn how to:

* Construct a **branch**, do some changes in the **branch**, and **merge** it with **master (or trunk)**
* Setting up the **Repository**

A screen shot of a computer

AI-generated content may be incorrect.

* Adding the First Mile to **Master Branch**

A computer screen with white text

AI-generated content may be incorrect.

* Linking to **Remote Repository**

A screen shot of a computer

AI-generated content may be incorrect.

* **Creating** and **Switching** to a **New Branch**

A screenshot of a computer

AI-generated content may be incorrect.

* Adding changes in **GitNewBranch**

A screenshot of a computer program

AI-generated content may be incorrect.

* **Comparing** Branches – Commands :
  + **get diff master (CLI)**

A screenshot of a computer

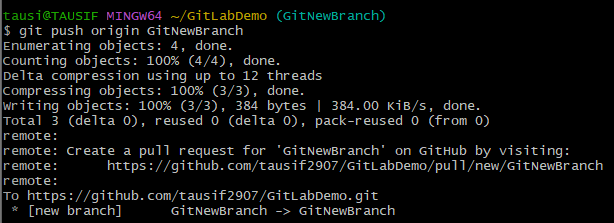
AI-generated content may be incorrect.

* + **get difftool master(Visual using P4Merge)**

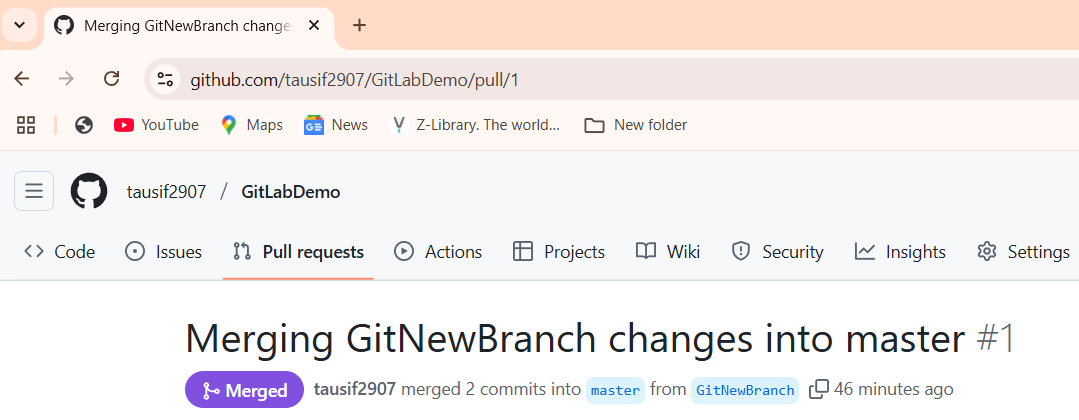
A screenshot of a computer

AI-generated content may be incorrect.

* **Pushing** the New Branch to Remote



* Creating a **Pull Request** (on GitHub)



* **Deleting** the Branch After **Merge**
  + **Local Command**

A screen shot of a computer

AI-generated content may be incorrect.

* + **Remote Command**

A screen shot of a computer code

AI-generated content may be incorrect.

* Github “**.txt”** file outputs
  + **readme.txt**

A screenshot of a computer

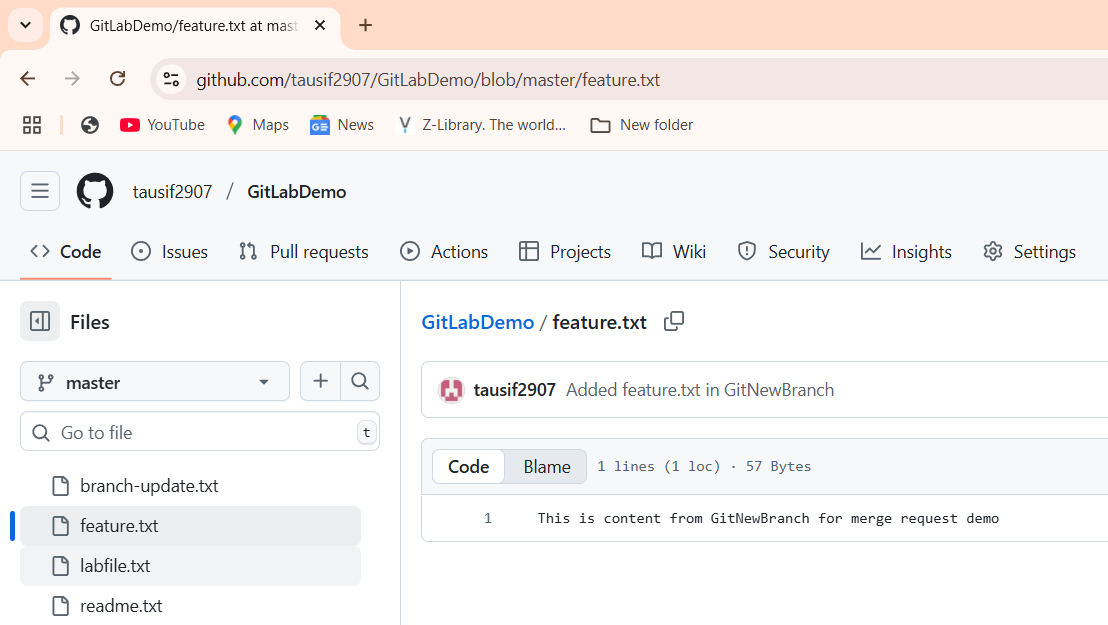
AI-generated content may be incorrect.

* + **labfile.txt**

A screenshot of a computer

AI-generated content may be incorrect.

* + **feature.txt**



* + **branch-update.txt**

A screenshot of a computer

AI-generated content may be incorrect.

1. **Git-HOL**

**Objectives**

**Q1. Explain how to resolve the conflict during merge.**

A merge conflict occurs when changes made in two different branches affect the same part of a file and Git cannot automatically merge them.  
To resolve a conflict:

1. **Identify the conflict** – Attempt a merge using git merge <branch>.
2. **View the conflicting changes** – Open the file in a text editor or use a merge tool like P4Merge (git mergetool) to compare.
3. **Edit and resolve** – Choose which changes to keep, or combine them manually to produce the correct final content.
4. **Mark as resolved** – Stage the resolved file using git add <filename>.
5. **Commit the merge** – Complete the process with git commit.
6. **Clean up** – Optionally, update .gitignore to exclude backup files (\*.orig) created by merge tools.

In this hands-on lab, we will learn how to:

* Implement conflict resolution when multiple users are updating the trunk (or master) in such a way that it results into a conflict with the branch’s modification.
* Verifying **Master** is in a **Clean State**

A black screen with white text

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

* Creating a **New Branch** and **Adding hello.xml**

A computer screen with white text

AI-generated content may be incorrect.

* Updating **hello.xml** in GitWork

A computer screen with white text

AI-generated content may be incorrect.

* Adding a **Conflicting Change in Master**

A screen shot of a computer program

AI-generated content may be incorrect.

* Viewing **Commit** History

A screenshot of a computer screen

AI-generated content may be incorrect.

* Comparing Differences between **Master** and **GitWork**

A computer screen with text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* **Merging** and **Triggering** the Conflict

A screen shot of a computer screen

AI-generated content may be incorrect.

* **Resolving** the **Conflict**

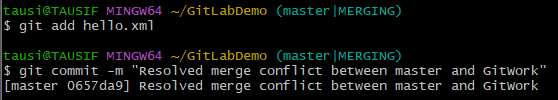
A black screen with yellow text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* Committing the **Resolved Merge**



* Ignoring **Backup Files**

A computer screen with white text

AI-generated content may be incorrect.

* **Deleting** the **Merged Branch**

A black background with white text

AI-generated content may be incorrect.

* Final **Log**

A screen shot of a computer screen

AI-generated content may be incorrect.

1. **Git-HOL**

**Objectives**

**Q1. Explain how to clean up and push back to remote Git.**

Cleaning up and pushing back to a remote Git repository involves making sure our local branch is synchronized with the remote, removing any unnecessary or untracked files, and sending your changes to the remote repository. The process includes:

1. **Verify clean working directory** – Use git status to ensure there are no uncommitted changes. If changes exist, commit or discard them.
2. **List branches** – Use git branch -a to see all local and remote branches and confirm we are on the correct one.
3. **Pull latest changes** – Run git pull origin <branch> to update our local branch with the latest commits from the remote.
4. **Remove unnecessary files** – If needed, delete unwanted files or use .gitignore to exclude them from tracking.
5. **Stage and commit changes** – Use git add <file> and git commit -m "message" to prepare our updates.
6. **Push to remote** – Use git push origin <branch> to send our committed changes to the remote repository.
7. **Verify on remote** – Check the remote repository on GitHub/GitLab to confirm that the changes have been successfully reflected.

In this hands-on lab, we will learn how to:

* Execute steps involving **clean up** and **push back** to **remote Git**.
* Creating a **New Local Repository**

A screen shot of a computer

AI-generated content may be incorrect.

* Creating the **First File** and **Committing** it

A screen shot of a computer program

AI-generated content may be incorrect.

* Linking the **Local Repository** to **Remote**

A computer screen shot of a program

AI-generated content may be incorrect.

* Adding New Changes to **Stimulate Pending Work**

A computer screen with white and blue text

AI-generated content may be incorrect.

* Verifying **Clean State**

A screen shot of a computer

AI-generated content may be incorrect.

* Listing All **Branches**

**A black screen with red and yellow text

AI-generated content may be incorrect.**

* **Pulling** Latest Changes from **Remote**

A screenshot of a computer

AI-generated content may be incorrect.

* **Pushing** Local Changes to **Remote**

**A screenshot of a computer program

AI-generated content may be incorrect.**

* Verifying Changes on **GitHub**

A screenshot of a computer

AI-generated content may be incorrect.

* **readme.txt**

A screenshot of a computer

AI-generated content may be incorrect.

* **update.txt**

**A screenshot of a computer

AI-generated content may be incorrect.**