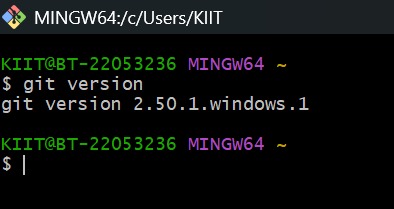
COGNIZANT DN - 4.0 DEEP SKILLING

HANDS ON WEEK-8

GIT

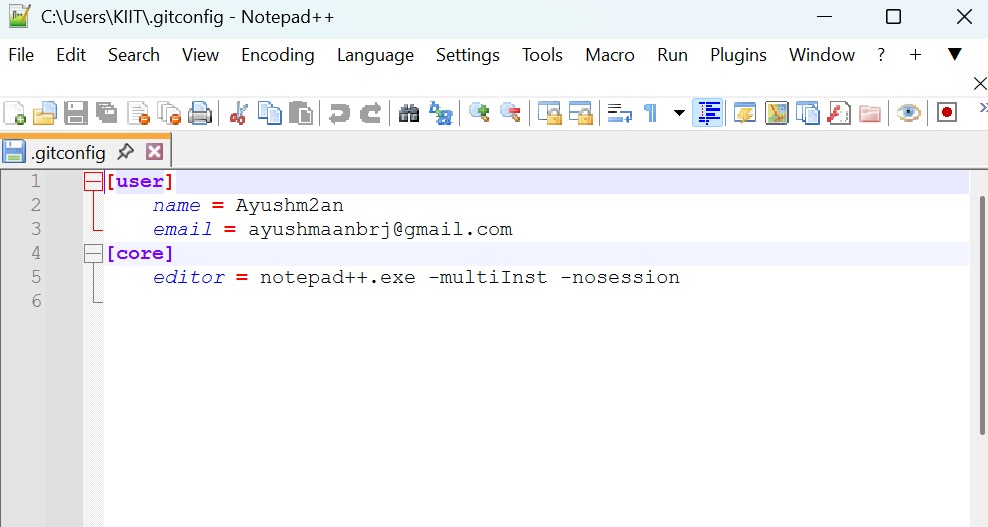
1. **Git-HOL**

**Step 1: Setup your machine with Git Configuration**

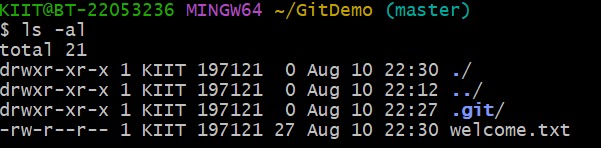
****

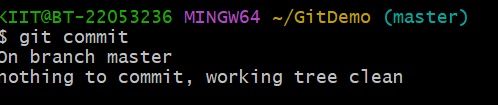


**Step 2: Integrate notepad++.exe to Git and make it a default editor**

****

**Step 3: Add a file to source code repository**





1. **Git-HOL**
2. **Explain git ignore.**

**Answer:**

The .gitignore file in Git is a simple text file that specifies which files and folders Git should ignore. When a file or directory matches a pattern listed in .gitignore, Git will not track changes to it, will not show it in git status as untracked, and will not include it in commits. This is especially useful for excluding temporary files, build artifacts, log files, system-generated files, or sensitive information such as credentials. The rules in .gitignore can use wildcards (like \*.log to ignore all log files) or specify directories (like log/ to ignore an entire folder).

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

**Answer:**

To ignore unwanted files in Git, you use a special file called .gitignore placed in your repository’s working directory. This file contains patterns that tell Git which files or folders should be excluded from tracking. For example, adding \*.log will ignore all files with the .log extension, and adding temp/ will ignore the entire temp folder. Once you define these patterns and save the .gitignore file, Git will stop listing those matching files as untracked in git status and will not include them in commits.

**Hands-on lab**

**Create a “.log” file and a log folder in the working directory of Git. Update the .gitignore file in such a way that on committing, these files (.log extensions and log folders) are ignored.**

**Verify if the git status reflects the same about working directory, local repository and git repository.**

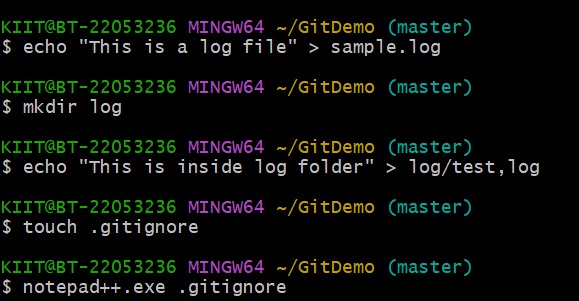
**.gitIgnore**

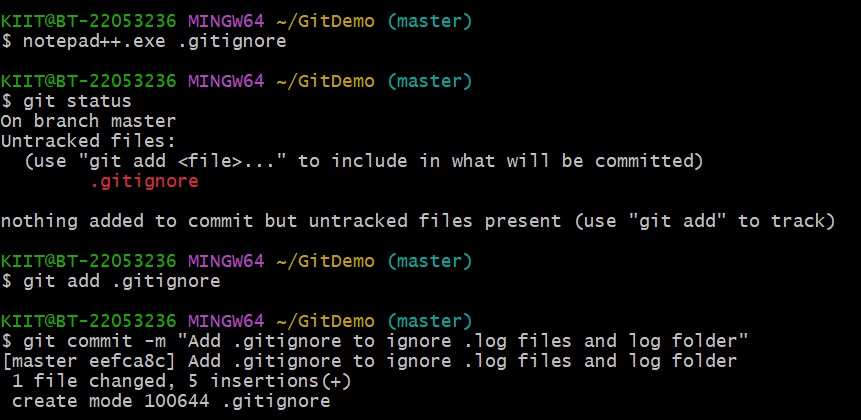
# Ignore all .log files

\*.log

# Ignore the log folder

log/

****

****

1. **Git-HOL**
2. **Explain branching and merging .**

**Answer:**

In Git**, branching** is the process of creating a separate line of development from the main codebase. Each branch is like a pointer to a specific commit, allowing you to work on new features, bug fixes, or experiments without affecting the main branch (often called main or master). For example, you might create a branch named feature-login to develop a login feature independently. This makes it possible for multiple people to work on different tasks simultaneously without interfering with each other’s code.

**Merging** is the process of combining changes from one branch into another. Once your work in a branch is complete and tested, you can merge it back into the main branch so the changes become part of the main codebase. During merging, Git integrates the commit history of both branches. If changes do not conflict, Git merges automatically; if both branches modified the same parts of a file, Git will produce a **merge conflict** that you must resolve manually.

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

**Answer:**

In GitLab, creating a branch request usually means making a **merge request,** which is a way to propose merging changes from one branch into another. First, you create a new branch in your repository, either directly in GitLab’s web interface or locally using the command git checkout -b branch-name and then push it to the remote with git push -u origin branch-name. Once the branch exists on GitLab, go to your project page, open the **Merge Requests** section, and click **New Merge Request**.

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

**Answer:**

In GitLab, a **merge request** is a way to propose changes from one branch to another, usually from a feature branch into the main branch. To create a merge request, first make sure your branch is pushed to the GitLab repository. You can do this by creating a branch locally with git checkout -b branch-name, committing your changes, and pushing it using git push -u origin branch-name.

Once your branch is on GitLab, go to your project page, open the **Merge Requests** tab, and click **New Merge Request**. Select the **source branch** (the one with your changes) and the **target branch** (often main or master). Add a descriptive title and detailed description explaining what changes you’ve made and why. You can also assign reviewers, set labels, and link related issues to help with tracking and review.

**Hands-on lab**

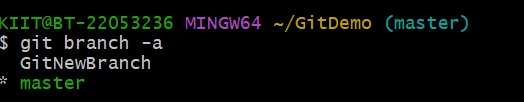
**Branching**

1. Create a new branch **“GitNewBranch”.**

**git branch GitNewBranch**

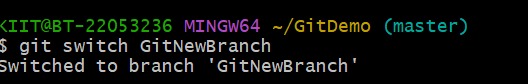
1. List all local and remote branches

**git branch –a**

****

1. Switch to the new branch.

**git checkout GitNewBranch**

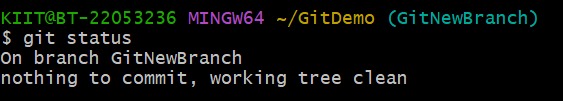
****

1. Add some files and content.

$ echo "This is a file in GitNewBranch" > branchfile.txt

1. Check the status.

git status



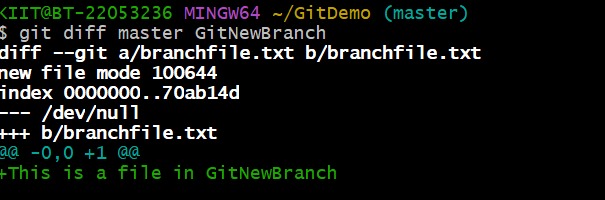
**Branching**

1. Switch back to master.

$ git checkout master

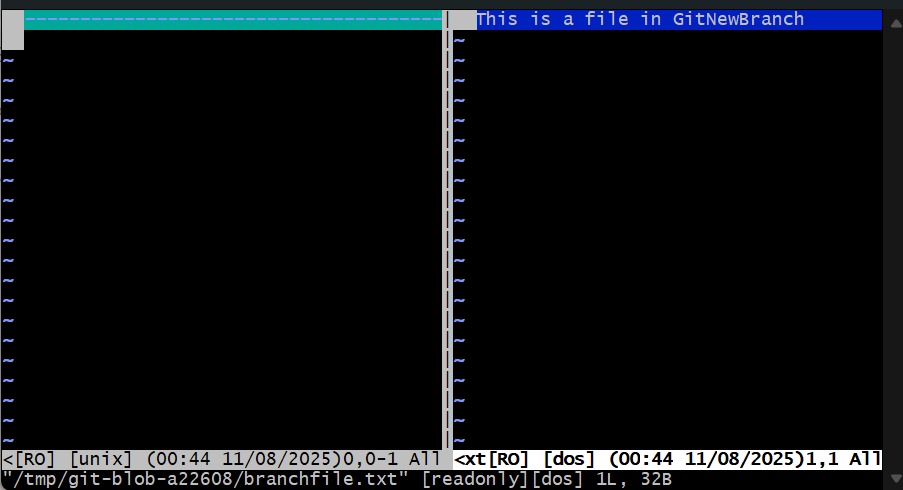
1. Show command-line differences between master and branch.

$ git diff master GitNewBranch



1. Show visual differences using P4Merge

**git difftool master GitNewBranch**

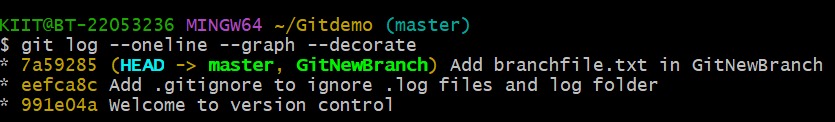


1. Merge the branch into master.

**git merge GitNewBranch**

1. View merge history

$ git log --oneline --graph --decorate

****

1. Delete the merged branch.

$ git branch -d GitNewBranch

1. **Git-HOL**
2. **Explain how to resolve the conflict during merge.**

**Answer:**

When Git tries to merge two branches, it automatically combines changes if they affect different parts of the files. However, if both branches modify the same lines or make changes that can’t be merged automatically, a **merge conflict** occurs. Git stops the merge and marks the files with conflicts so you can fix them manually.

To resolve a merge conflict:

1. **Identify conflicted files** – Run git status after the failed merge. Conflicted files will be listed as *unmerged*.
2. **Open the conflicted file** – Inside, Git adds special markers:

<<<<<<< HEAD

code from the current branch

=======

code from the branch being merged

>>>>>>> branch-name

* + Everything between <<<<<<< HEAD and ======= is from your current branch.
  + Everything between ======= and >>>>>>> branch-name is from the branch you’re merging.

1. **Edit the file** – Decide which changes to keep, or combine them, and remove the conflict markers (<<<<<<<, =======, >>>>>>>).
2. **Mark the conflict as resolved** – After editing:

git add filename

1. **Complete the merge** – Once all conflicts are resolved and staged:

git commit

(If Git already created a merge commit message, you just need to save and close the editor.)

1. **Verify** – Use git status to confirm there are no remaining conflicts.

**Hands On Lab**

1. **Verify if master is in a clean state.**

$ git checkout master

$ git status

1. **Create a branch GitWork and add hello.xml.**

$ git branch GitWork

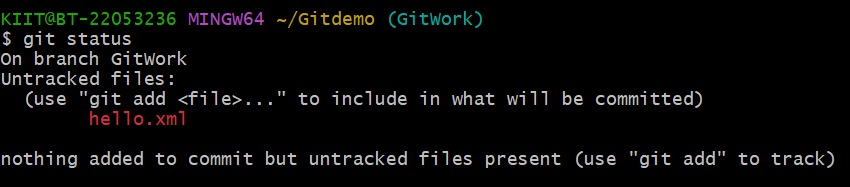
$ git checkout GitWork

$ echo "<greeting>Hello from GitWork branch</greeting>" > hello.xml

1. **Update the content of hello.xml and observe status.**

$ echo "<greeting>Hello again from GitWork</greeting>" > hello.xml

$ git status



1. **Commit the changes in the branch.**

$ git add hello.xml

$ git commit -m "Add and update hello.xml in GitWork branch"



1. **Switch to master.**

$ git checkout master

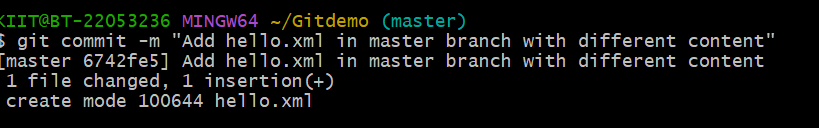
1. **Add hello.xml in master with different content.**

$ echo "<greeting>Hello from master branch</greeting>" > hello.xml

1. **Commit the changes in master.**

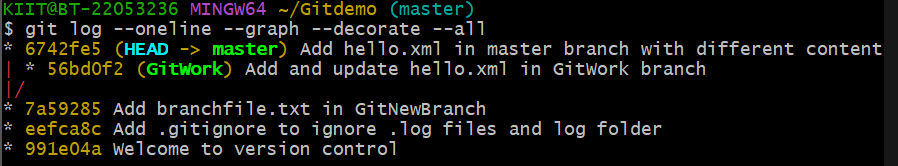
$ git add hello.xml

$ git commit -m "Add hello.xml in master branch with different content"



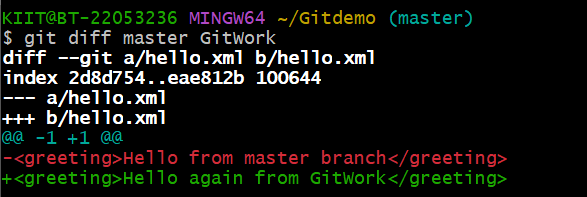
1. **Observe the log (all branches)**

$ git log --oneline --graph --decorate –all

****

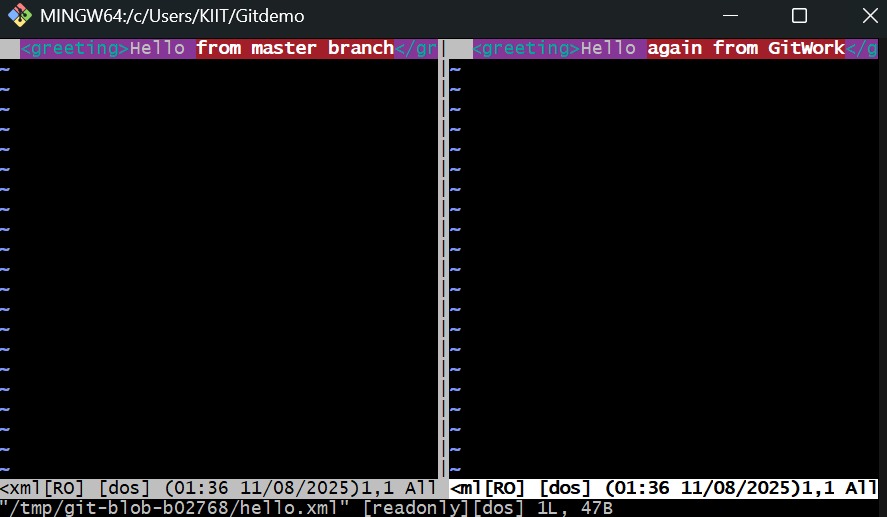
1. **Check text-based differences.**

$ git diff master GitWork

****

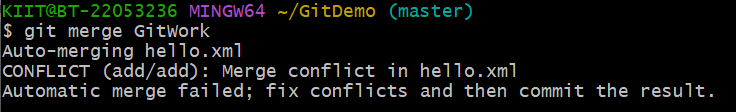
1. **For better visualization, use P4Merge tool to list out all the differences between master and branch.**

git difftool master GitWork

****

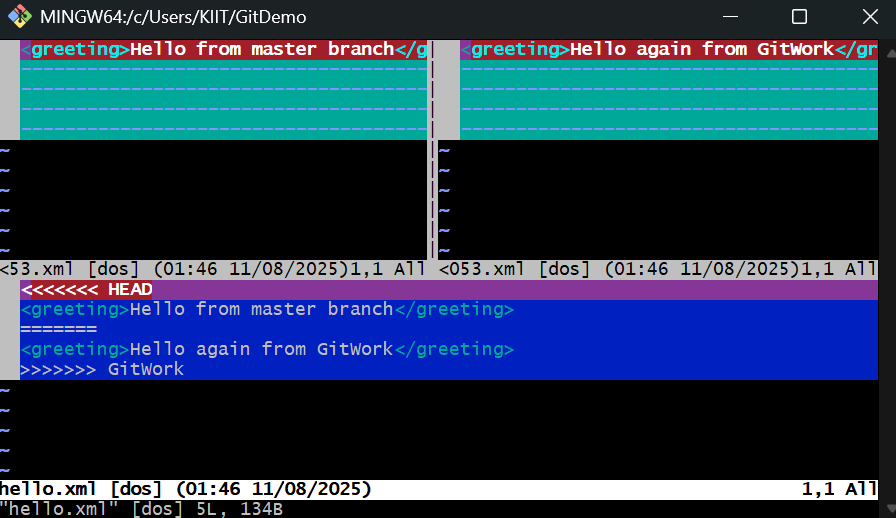
1. **Merge the bran to the master.**

$ git merge GitWork

****

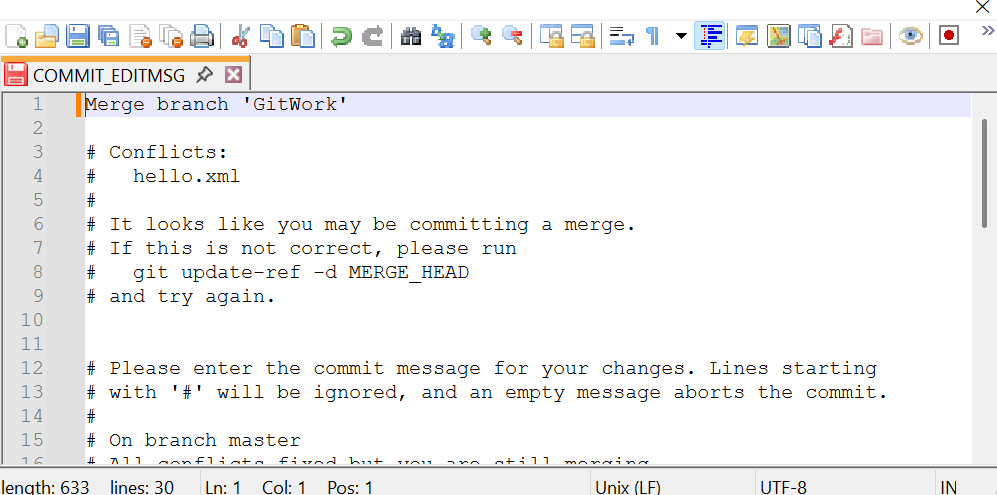
1. **Observe the git mark up.**
2. **Use 3-way merge tool to resolve the conflict.**

git mergetool

****

1. **Commit the changes to the master, once done with conflict.**

$ git commit

****

1. **Observe the git status and add backup file to the .gitignore file.**

$ echo "\*.orig" >> .gitignore

1. **Commit the changes to the .gitignore**

$ git commit -m "Ignore merge bakup files"

1. **List out all the available branches**

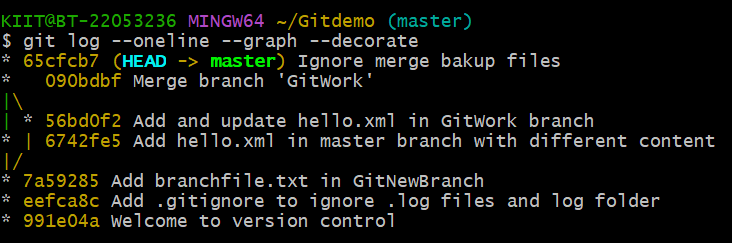
$ git branch

1. **Delete merged branch.**

$ git branch -d GitWork

1. **Observe final log.**

$ git log --oneline --graph --decorate

****

1. **Git-HOL**
2. **Explain how to clean up and push back to remote Git.**

**Answer:**

Cleaning up and pushing back to a remote Git repository means making sure your local repository is tidy (no unnecessary branches, files, or commits) and then syncing your cleaned-up work with the remote repository (like GitHub or GitLab).

First, **clean up your local repository.** Remove untracked or temporary files that you don’t want in Git by running git status to see them and git clean -f to delete untracked files (use -fd to also remove untracked folders). If you have local branches you no longer need and that have already been merged, delete them with git branch -d branch-name. For branches that are not merged and you still want to remove, use git branch -D branch-name.

Next, **sync with the remote**. Start by making sure your local branch is up to date by running git pull origin branch-name. Resolve any merge conflicts if they appear. Then commit all the changes you want to keep using git add . followed by git commit -m "Your message". Finally, push your branch to the remote repository using:

git push origin branch-name

**Hands On Lab**

1. **Verify if master is in clean state.**

$ git status

1. **List out all the available branches.**

$ git branch -a

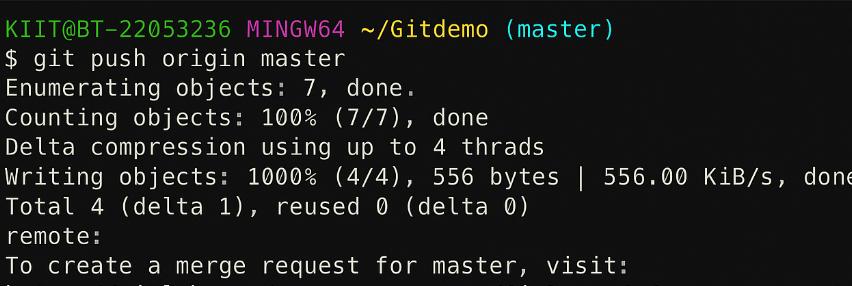
1. **Pull the remote git repository to the master.**

$ git pull origin master

1. **Push pending changes from “Git-T03-HOL\_002”.**

$ git push origin gitignore

1. **Observe if the changes are reflected in the remote repository.**

****