1. **GIT HOL**

**Adding and Pushing welcome.txt to GitLab  
  
Step 1: Navigate to your Git repository folder**

cd ~/GitDemo

**Step 2: Create or add the file welcome.txt in your project folder**

**Step 3: Add the file to the staging area**

git add welcome.txt

**Step 4: Commit the changes with a message**

git commit -m "Add welcome.txt with first Git message"

**Step 5: Configure your GitLab remote URL with Personal Access Token for authentication**

Replace <your\_gitlab\_username> and <your\_token> with your actual GitLab username and token:

git remote set-url origin https://<your\_gitlab\_username>:<your\_token>@gitlab.com/jayashree-group/gitdemo-project

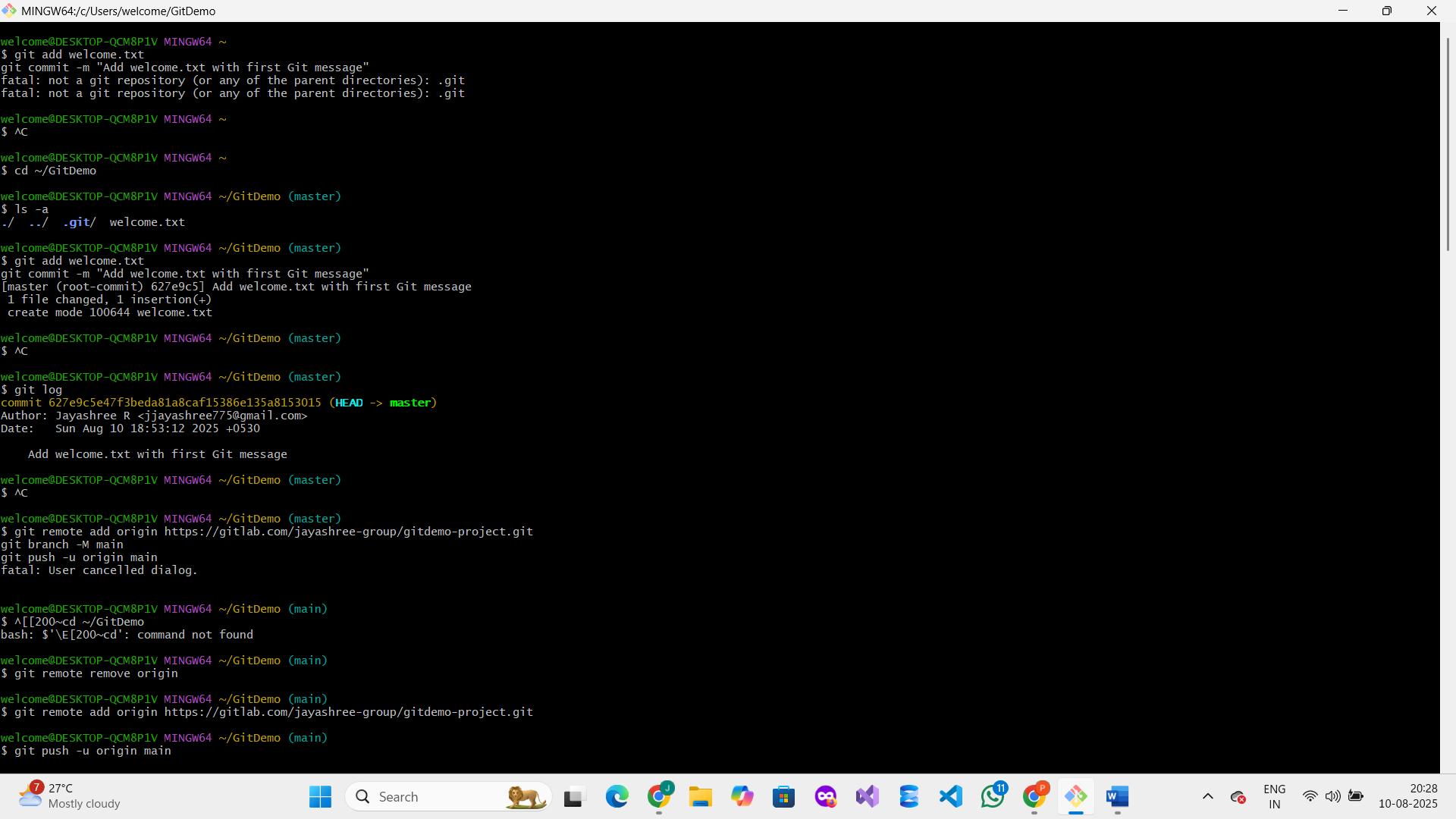
**Step 6: Pull any changes from the remote branch to avoid conflicts**

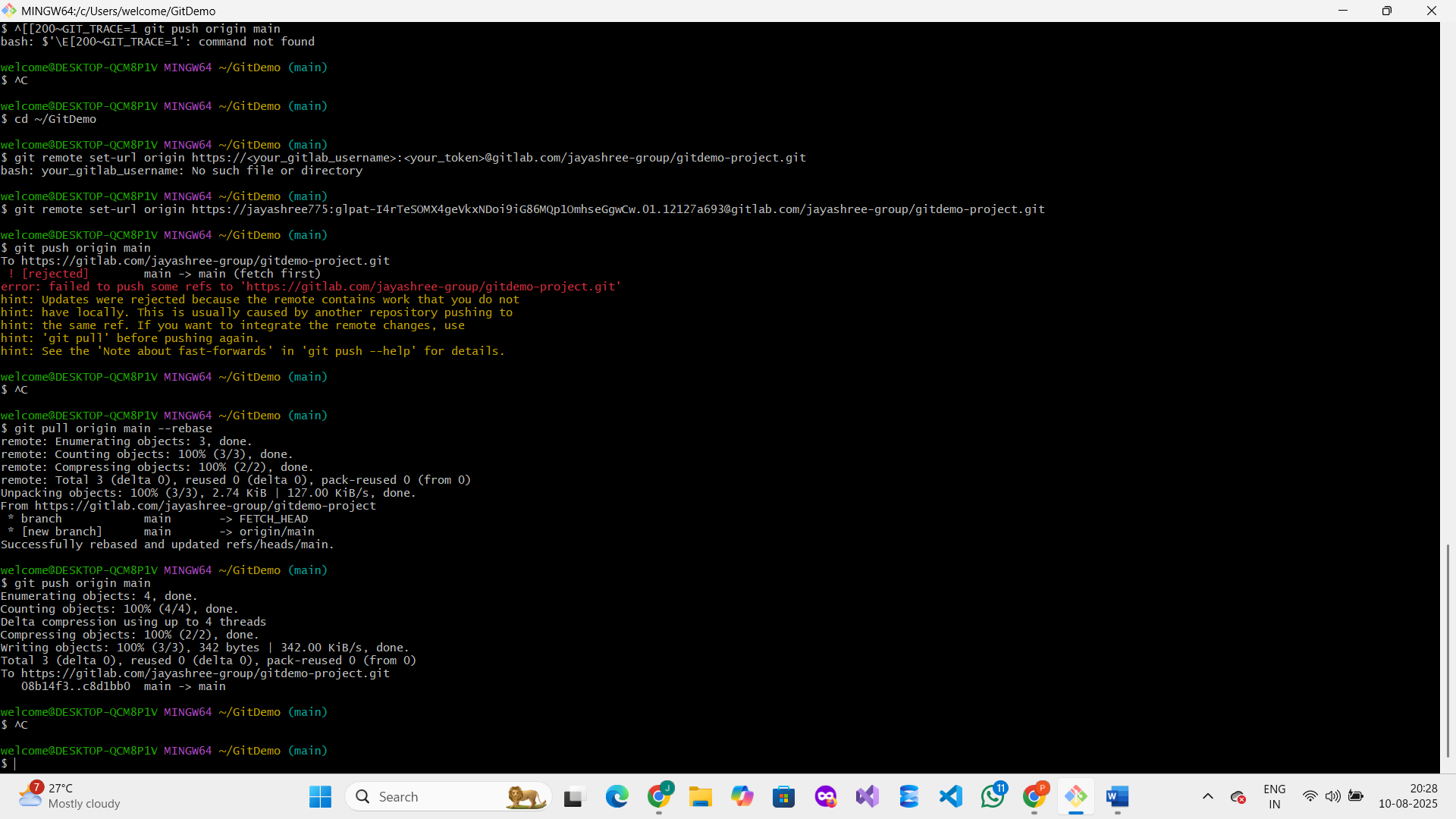
git pull origin main --rebase

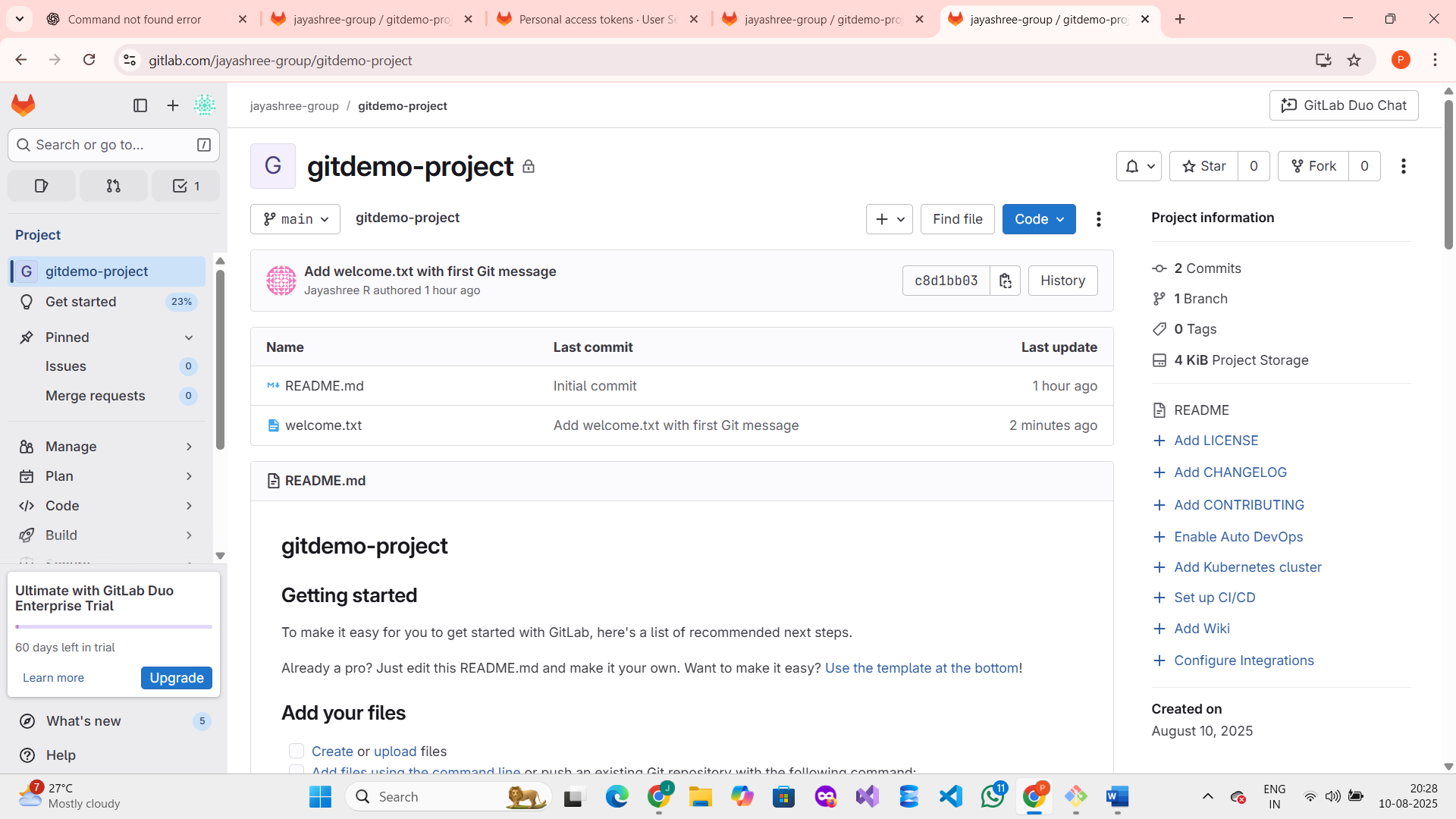
**Step 7: Push your local commits to GitLab**

git push origin main

**OUTPUT :**







**2 . GIT – HOL**

**Using .gitignore to Ignore Unwanted Files in Git**

**What is .gitignore and Its Purpose in Git**

**.gitignore is a plain text file used by Git to determine which files or directories to ignore in a project. When a file or folder is listed in .gitignore, Git will not track changes to those files, nor will it include them in commits. This is useful for excluding temporary files, logs, build outputs, or sensitive information that should not be part of the version control system. Using .gitignore helps keep the repository clean and focused only on relevant source files.**

**How to Ignore Unwanted Files and Folders Using .gitignore**

**To ignore files or folders, you add patterns to the .gitignore file placed at the root of your repository. For example:**

* **To ignore all files with the .log extension, add:**

**\*.log**

* **To ignore a specific folder and its contents (e.g., a folder named log), add:**

**log/**

**Once these rules are added, Git will automatically exclude matching files and folders from being tracked or committed.**

**Introduction**

**In this lab, we learn how to use the .gitignore file to exclude unwanted files and folders from being tracked by Git. This helps keep the repository clean by ignoring temporary or log files.**

**Prerequisites**

* **Git installed and configured**
* **Notepad++ set as the default editor for Git**
* **Local Git repository with a remote repository on GitLab**

**Steps Performed**

1. **Navigate to the local Git repository folder  
   Used the command:  
   cd ~/GitDemo**
2. **Created a sample .log file and a log folder with a file inside  
   Commands used:**

**bash**

**Copy code**

**echo "This is a log file" > sample.log**

**mkdir log**

**echo "Log folder file" > log/test.log**

1. **Checked current Git status  
   Ran: git status  
   Result showed sample.log and log/ as untracked files.**
2. **Created or updated the .gitignore file to ignore .log files and the log folder  
   Edited .gitignore to add the lines:**

**bash**

**Copy code**

**\*.log**

**log/**

1. **Verified that .log files and the log folder are ignored  
   Ran: git status  
   Result showed these files/folders were no longer listed as untracked.**
2. **Added and committed the .gitignore file  
   Commands:**

**sql**

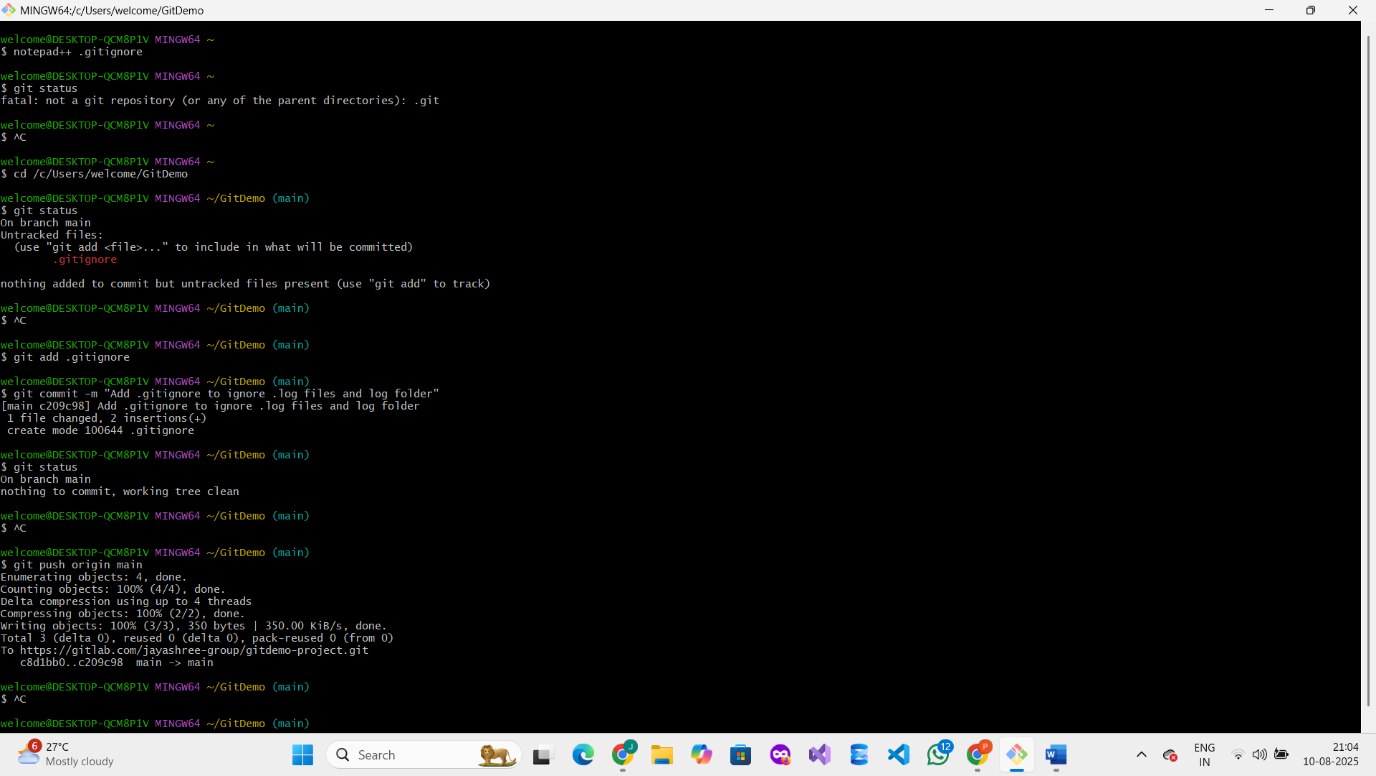
**Copy code**

**git add .gitignore**

**git commit -m "Add .gitignore to ignore .log files and log folder"**

1. **Pushed changes to remote GitLab repository  
   Command:  
   git push origin main**
2. **Result / Output**

**The .log files and the entire log folder were successfully ignored by Git as expected. The repository remains clean without tracking unwanted log files.**

****

**3 . GIT - HOL**

**Branching and Merging in Git**

**Objectives**

* **Explain branching and merging**

**Branching in Git allows you to create a separate line of development from the main codebase (usually called main or master). This enables multiple developers to work on different features or fixes independently without affecting the main project. Each branch can have its own commits and changes isolated from others.  
  
Merging is the process of integrating changes from one branch into another (commonly from a feature branch back into the main branch). When merging, Git combines the histories and changes, resolving conflicts if the same parts of files were changed differently. Merging helps to consolidate work and keep the project updated.**

* **Explain about creating a branch request in GitLab**
* **Explain about creating a merge request in GitLab**

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

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

**Prerequisites**

* **Setting up Git environment with P4Merge tool for Windows**

**Note: Please create a free GitHub account. Do not use Cognizant credentials to login to GitHub.**

**Estimated time to complete this lab: 30 minutes.**

**Branching**

**Step 0: Setup or Navigate to Git Repository**

* **Open Git Bash**
* **Navigate to your Git repository folder or create a new one:**

**mkdir GitDemo**

**cd GitDemo**

**git init**

* **If folder exists, just navigate:**

**cd GitDemo**

**git init**

**Step 1: Create a new branch named GitNewBranch**

**git branch GitNewBranch**

**Step 2: List all local and remote branches**

**git branch -a**

**Observe the \* mark which denotes the current branch.**

**Step 3: Switch to the newly created branch**

**git checkout GitNewBranch**

**Step 4: Add a file to the branch**

**echo "This is a sample file in GitNewBranch." > example.txt**

**Step 5: Stage (add) the file**

**git add example.txt**

**Step 6: Commit the changes to the branch**

**git commit -m "Add example.txt with sample content to GitNewBranch"**

**Step 7: Check the status to confirm commit**

**git status**

**Merging**

**Step 1: Switch back to the main branch**

**git checkout main**

**Step 2: View differences between main and GitNewBranch**

**git diff main GitNewBranch**

**Step 3: Visualize differences using P4Merge tool**

**Set P4Merge as your difftool if not set:**

**git config --global diff.tool p4merge**

**git config --global difftool.p4merge.cmd "p4merge \$LOCAL \$REMOTE"**

**git difftool main GitNewBranch**

**Step 4: Merge the source branch into main**

**git merge GitNewBranch**

**Step 5: View commit history with graph, one line, and decorate options**

**git log --oneline --graph --decorate**

**Step 6: Delete the branch after successful merge**

**git branch -d GitNewBranch**

**Check branches again:**

**git branch -a**

**4 . GIT - HOL**

**Git Conflict Resolution Hands-on Lab**

**Objectives**

* **Explain how to resolve conflicts during a merge.**

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

* **Implement conflict resolution when multiple users update the trunk (or master), resulting in a conflict with branch modifications.**

**Prerequisites**

* **Hands-on ID: “Git-T03-HOL\_001”**
* **Please create a free GitHub account (do not use Cognizant credentials).**
* **Estimated time: 30 minutes.**

**Step-by-step Instructions**

**Step 0: Setup or navigate to your Git repository**

* **Open Git Bash.**
* **Navigate to your repo folder or create a new one:**

**mkdir GitDemo**

**cd GitDemo**

**git init**

**Step 1: Verify if the main (or master) branch is clean**

**git status**

**Output should show:**

**On branch main**

**nothing to commit, working tree clean**

**Step 2: Create a branch GitWork and add a file hello.xml**

**git branch GitWork**

**git checkout GitWork**

**echo "<message>Hello from GitWork branch</message>" > hello.xml**

**Step 3: Update hello.xml content and check status**

**echo "<update>First update on hello.xml</update>" >> hello.xml**

**git status**

**You will see hello.xml as untracked.**

**Step 4: Stage and commit the file on GitWork branch**

**git add hello.xml**

**git commit -m "Add hello.xml with initial content on GitWork branch"**

**Step 5: Switch back to main branch**

**git checkout main**

**Step 6: Add a different hello.xml file on main**

**echo "<message>Hello from main branch</message>" > hello.xml**

**git add hello.xml**

**git commit -m "Add hello.xml with different content on main branch"**

**Step 7: View commit log for all branches**

**git log --oneline --graph --decorate --all**

**This shows commits diverged between main and GitWork.**

**Step 8: Check differences between main and GitWork**

**git diff main GitWork**

**Step 9: Use P4Merge tool to visualize differences**

**Set P4Merge as difftool:**

**git config --global diff.tool p4merge**

**git config --global difftool.p4merge.cmd "p4merge \$LOCAL \$REMOTE"**

**Run:**

**git difftool main GitWork**

**Step 10: Merge GitWork branch into main (conflict will occur)**

**git merge GitWork**

**Git will report a merge conflict on hello.xml.**

**Step 11: Resolve conflict using 3-way merge tool (P4Merge)**

**Run:**

**git mergetool**

* **P4Merge will open showing base, local (main), and incoming (GitWork) versions.**
* **Manually resolve conflicts by editing the file in P4Merge.**
* **Save and close P4Merge.**

**Step 12: Commit resolved changes to main**

**git commit -m "Resolve merge conflict for hello.xml after merging GitWork"**

**Step 13: Check status and add backup files to .gitignore**

**git status**

**echo "\*.backup" >> .gitignore**

**git add .gitignore**

**git commit -m "Add backup files to .gitignore"**

**Step 14: List all branches**

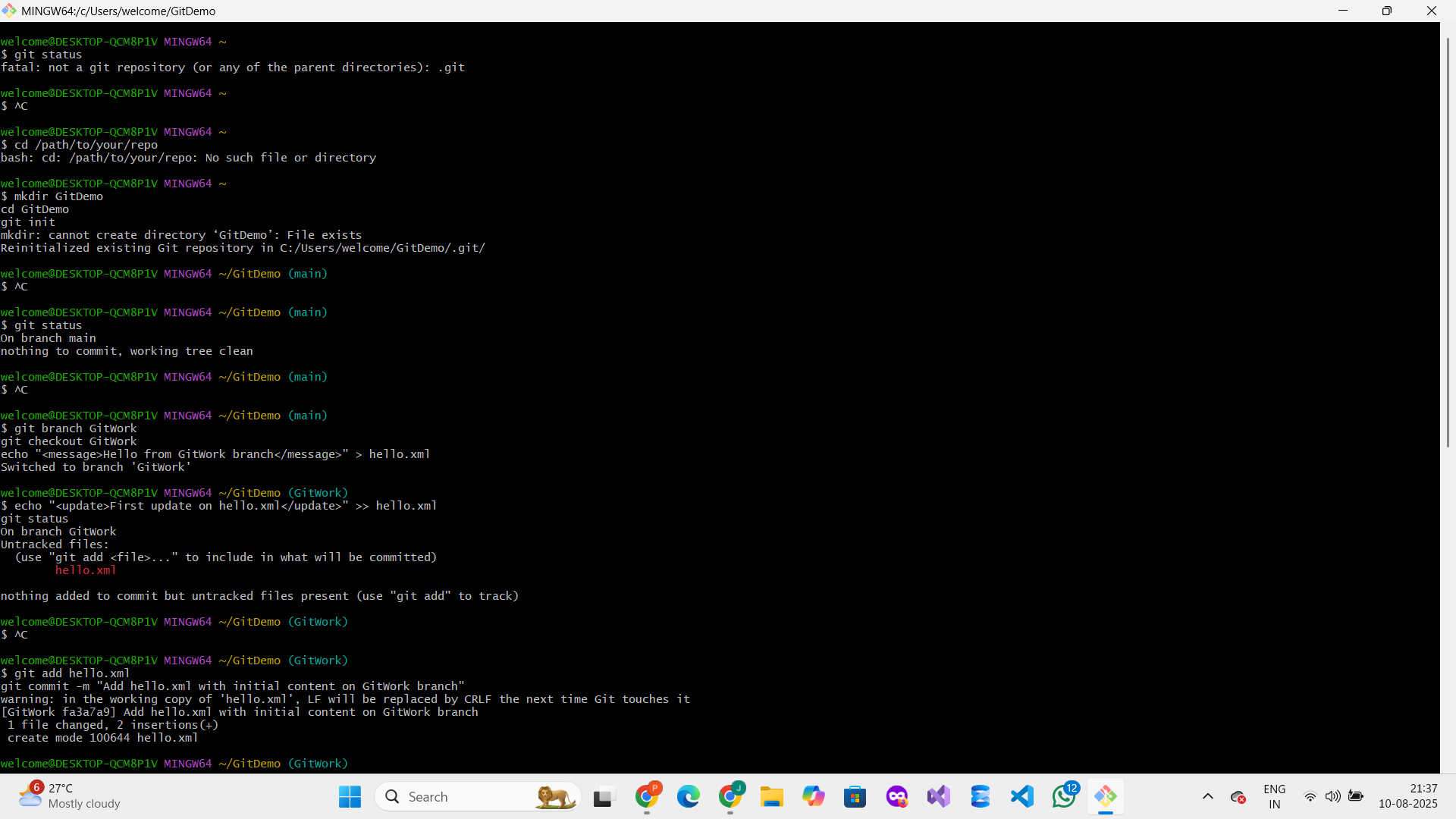
**git branch -a**

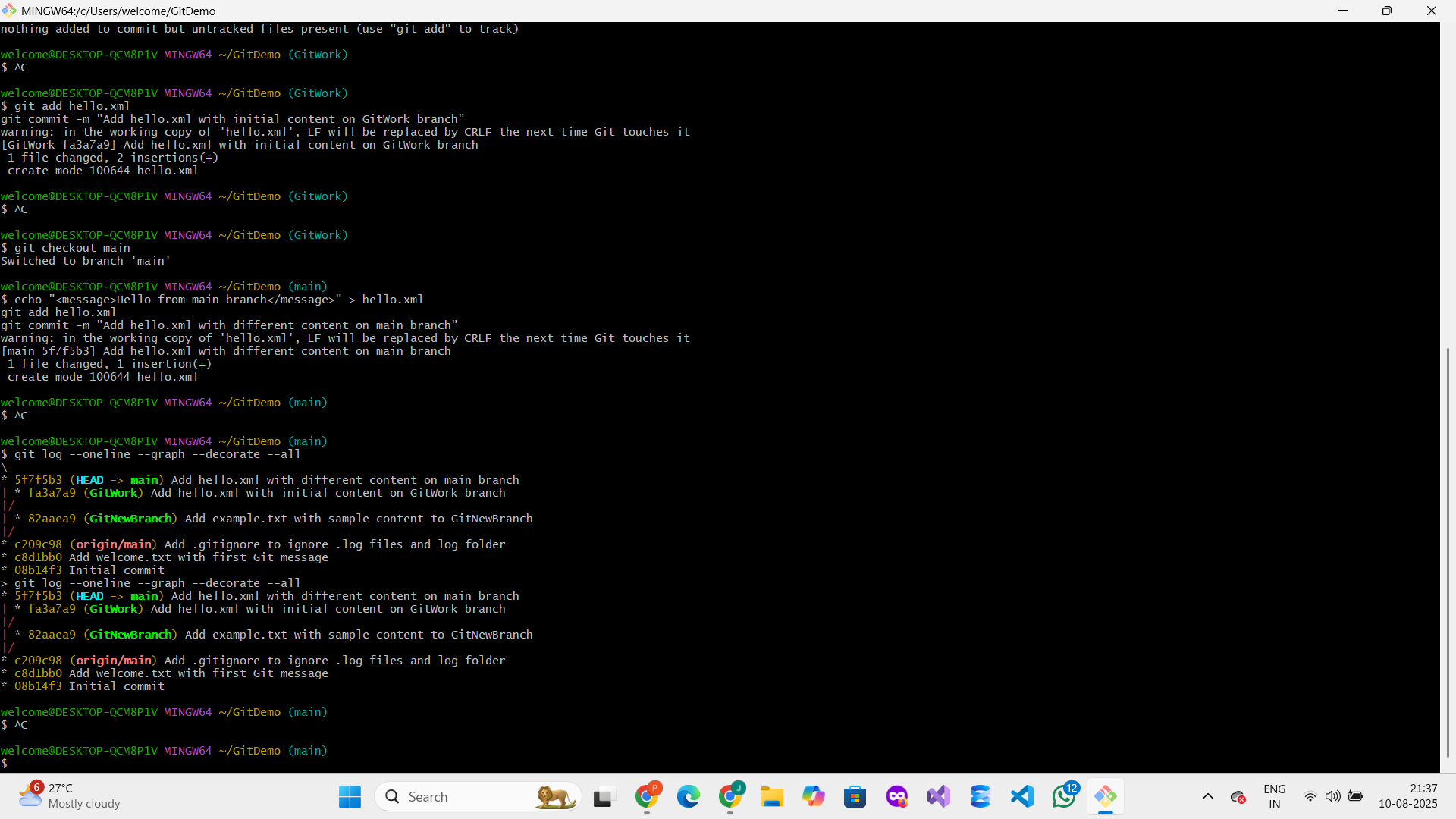
**Step 15: Delete the merged branch GitWork**

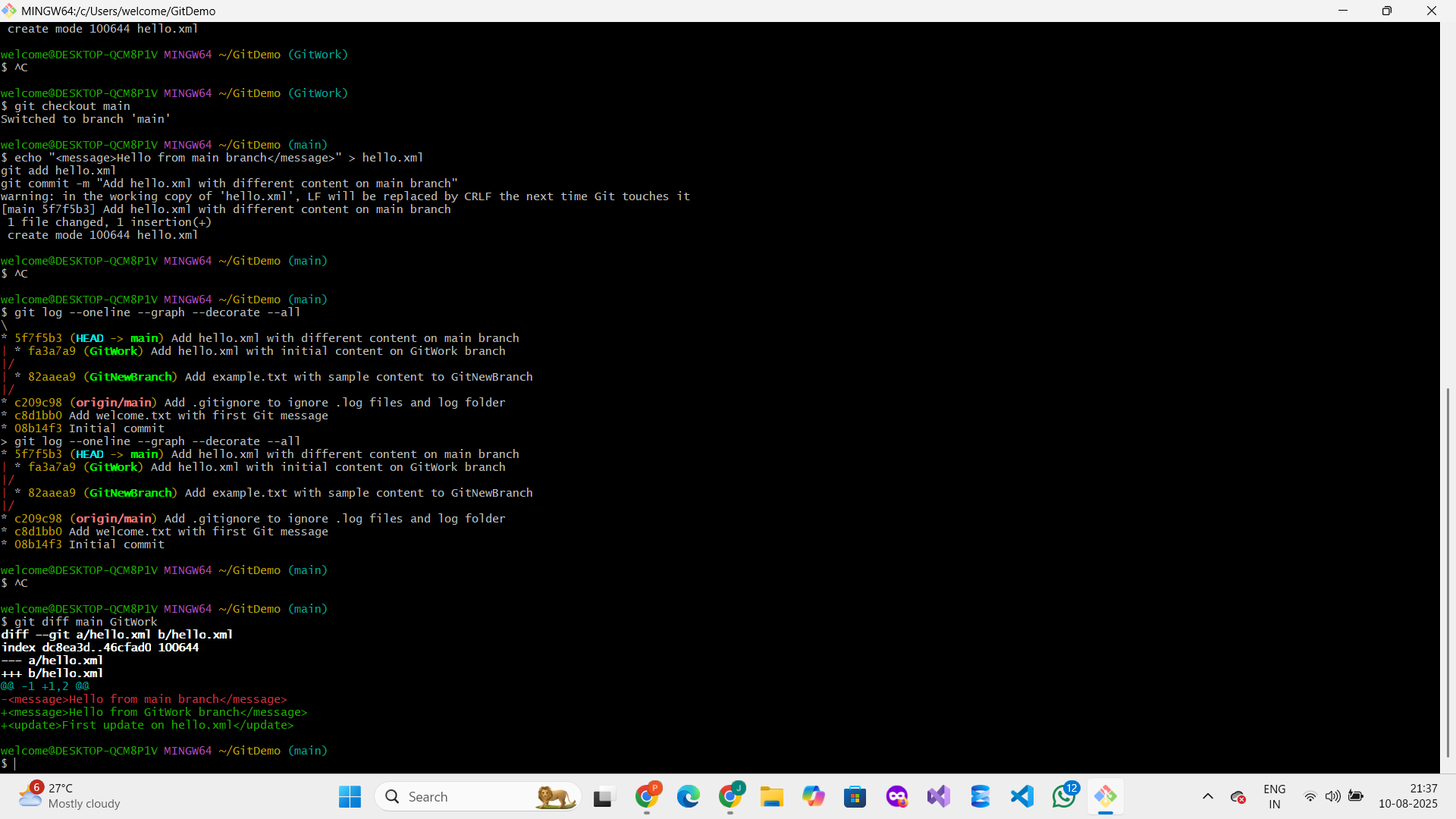
**git branch -d GitWork**

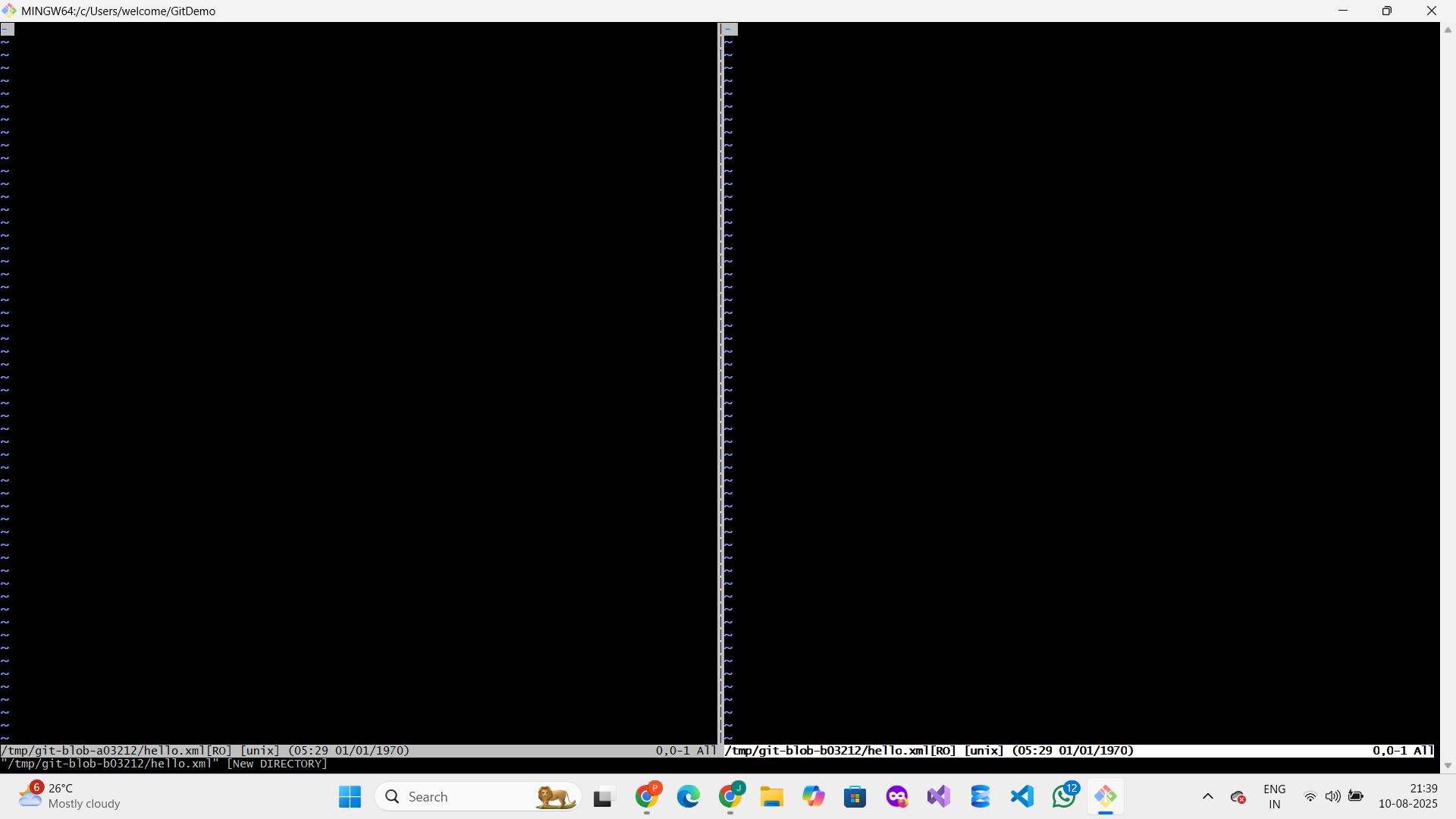
**Step 16: View final commit log graph**

**git log --oneline --graph --decorate**

**output  
**

****

****

****

**5 . GIT - HOL**

**Merge Request: Add changes from Git-T03-HOL\_002 to main**

**1. Navigate to your local Git repository folder**

**cd /c/Users/welcome/gitDemo**

**You moved into the folder where your Git repository is located.**

**2. Check if your current branch (main) is clean**

**git status**

**Output showed:**

**On branch main**

**nothing to commit, working tree clean**

**Meaning there were no uncommitted changes.**

**3. List all available branches (local and remote)**

**git branch -a**

**Output showed local branches (GitNewBranch, GitWork, main) and remote branch remotes/origin/main.**

**4. Pull the latest changes from the remote main branch**

**git pull origin main**

**Output:**

**vbnet**

**Already up to date.**

**Meaning your local main branch is synced with remote.**

**5. Create a new branch locally named Git-T03-HOL\_002**

**git checkout -b Git-T03-HOL\_002**

**You switched to a new branch for your changes.**

**6. Push the new branch and its commits to the remote repository**

**git push -u origin Git-T03-HOL\_002**

**Output showed the branch was pushed successfully and set up to track the remote branch.**

**7. Create a Merge Request on GitLab to merge Git-T03-HOL\_002 into main**

* **Open the merge request URL provided in the push output.**
* **Fill in the Title and Description.**
* **Submit the merge request for review.**