Git Workflow for EngagePoint app team:

1. Cloning the repository

Each team member should clone the repo to their local machine:

```
git clone https://github.com/ChristiaanHuisman/EngagePoint_ITMDA_GroupS11.git
cd EngagePoint_ITMDA_GroupS11
```

---

2. Making sure your local repo is up-to-date

Before starting work **every day** or **before starting a new task**:

```
git checkout main      # switch to main branch
git pull origin main   # pull latest changes from GitHub
```

- **If changes affect your task:**
  - Decide whether to start your feature branch now or after changes are merged.
  - If your current branch has conflicting changes, you may need to **merge** or **rebase** (explained below).

---

3. Branches vs. working on main

3.1 Feature Branch Workflow (recommended for new features / big changes)

1. Create a branch for your task:

```
git checkout -b feature/<task-name>
```

Example:

```
git checkout -b feature/review-sentiment
```

2. Do your work, then stage changes:

```
git add .
git commit -m "Add sentiment analysis model and test data"
```

3. Push branch to GitHub:

```
git push origin feature/review-sentiment
```

4. Create a Pull Request on GitHub to merge into **main** (or **dev** if you use a dev branch).
5. After merge, delete the branch locally and on GitHub:

```
git branch -d feature/review-sentiment       # delete local branch
git push origin --delete feature/review-sentiment
```

3.2 Direct commits to main (for minor fixes / documentation)

1. Make sure main is up-to-date:

```
git checkout main
git pull origin main
```

2. Stage and commit your changes:

```
git add .
git commit -m "Fix typo in README"
git push origin main
```

Tip: Prefer feature branches for code changes to avoid breaking main.

---

4. Committing & pushing only a specific folder

Sometimes you only want to commit the **Frontend** or a **specific microservice**.

1. Stage the folder only:

```
git add Frontend/flutter_app/
```

or

```
git add Backend/ReviewSentiment_Service/
```

2. Commit your changes:

```
git commit -m "Implement basic sentiment analysis endpoint"
```

3. Push to a branch:

```
git push origin feature/review-sentiment
```

---

5. Handling conflicts / changes from other team members

If someone else pushed changes while you were working:

1. Pull the latest changes:

```
git checkout main
git pull origin main
```

2. If your branch is behind, merge main into your branch:

```
git checkout feature/review-sentiment
git merge main
```

- Resolve conflicts if any (Git will mark conflicted files).
- Stage resolved files and commit:

```
git add <resolved-file>
git commit -m "Resolve merge conflicts with main"
```

3. Then push your branch again:

```
git push origin feature/review-sentiment
```

---

6. Useful Git Tips for the Team

- **Check status**:

```
git status
```

- **Check logs**:

```
git log --oneline --graph –all
```

- **Discard changes in a file** (careful!):

```
git restore <file>
```

- **Switch branches**:

```
git checkout <branch-name>
```

Summary Recommendations

- Always pull main before starting work.
- Use **feature branches** for code changes, main only for minor fixes.
- Commit frequently, ideally one commit per logical change.
- Push branches to GitHub early if collaboration is needed.
- Stage only the folders/files you want to commit if working on a single microservice.

Collaboration Scenarios to know

- **Someone deletes a branch you're working on:** Git will warn you; switch to a safe branch and merge changes before deleting.
- **Someone force pushes:** Rare, but can overwrite history — only admins should do this.
- **Your changes break others' work:** Always pull and test before pushing.

Resolving Conflicts:

The **resolution itself is done in the code** (or text) files. The terminal is only used to:

- **See which files have conflicts** (git status)
- **Stage the resolved files** (git add <file>)
- **Commit the resolution** (git commit)

GitHub can **show conflicts visually** during a pull/merge request, but it's best to fix the conflict **in your code editor**, then commit via terminal or your Git GUI.

Think of it like this:

1. Git tells you where the conflicts are.
2. You fix them **in the actual code files**.
3. Git finalizes the fix with add + commit.

---

1. Identify the conflicts

When you try to merge or pull and there’s a conflict, Git will tell you:

```
Auto-merging Backend/ReviewSentiment_Service/app/main.py
CONFLICT (content): Merge conflict in
Backend/ReviewSentiment_Service/app/main.py
Automatic merge failed; fix conflicts and then commit the result.
```

You can check all conflicted files with:

```
git status
```

They will appear under **"Unmerged paths"**.

---

2. Open the conflicted file

Git marks the conflict inside the file like this:

```
def example_function():
<<<<<<< HEAD
   print("This is my local version")
=======
   print("This is the version from the branch being merged")
>>>>>>> feature/other-branch
```

- **HEAD** → your local version (the branch you were on).
- **=======** → separator between versions.
- **>>>>>>> branch-name** → the version from the branch being merged.

---

3. Resolve the conflict

You have three options:

1. **Keep your version:** delete the other branch’s changes and markers.
2. **Keep the incoming version:** delete your local changes and markers.
3. **Combine both:** manually merge changes in a way that makes sense.

Example of combining both:

```
def example_function():
    print("This is my local version")
    print("This is the version from the branch being merged")
```

After editing, **make sure you remove all** <<<<<<<, =======, and >>>>>>> markers.

---

4. Stage resolved files

Once you fix the conflicts in all files:

```
git add Backend/ReviewSentiment_Service/app/main.py
```

---

5. Commit the merge

After staging:

```
git commit -m "Resolved merge conflict in main.py"
```

- If you were in the middle of a merge, Git will automatically suggest a commit message like "Merge branch 'feature/other-branch'". You can keep it or edit it.

---

6. Push changes

Finally, push your branch:

```
git push origin feature/your-branch
```

Tips

- Use a **code editor** like VS Code — it highlights conflicts and offers "Accept Current / Incoming / Both" buttons.
- Don't rush: carefully read both versions to make sure you're not overwriting important code.
- Communicate with the team member if you're unsure which change to keep.