

The message you're seeing in Git indicates the following:

1. **Your branch is ahead of 'origin/main' by 2 commits:**  
   This means you have made two commits locally on the main branch that have not yet been pushed to the remote repository (usually hosted on platforms like GitHub, GitLab, etc.). To resolve this and sync your local changes with the remote repository, you can use:

bash

Copy code

git push

1. **Untracked files:**  
   Git has detected a folder called 51-Microcontroller/ that is not being tracked. This means that this file or directory is in your working directory but has not yet been added to the Git staging area. Git is notifying you that there's an untracked file, but it hasn't been included in any commits yet. To include it, you need to track it by using:

bash

Copy code

git add 51-Microcontroller/

1. **Nothing added to commit but untracked files present:**  
   This means that although there are untracked files (51-Microcontroller/), they haven't been added to the staging area. You need to run the git add command mentioned above if you want to include this folder in your next commit.

**Steps to resolve:**

1. **If you want to track the untracked files and commit them:**

bash

Copy code

git add 51-Microcontroller/

git commit -m "Add 51-Microcontroller directory"

git push

1. **If you don't want to track the 51-Microcontroller/ folder:** You can ignore this folder by adding it to a .gitignore file. If there isn't already a .gitignore file in your repository, create one and add the following line:

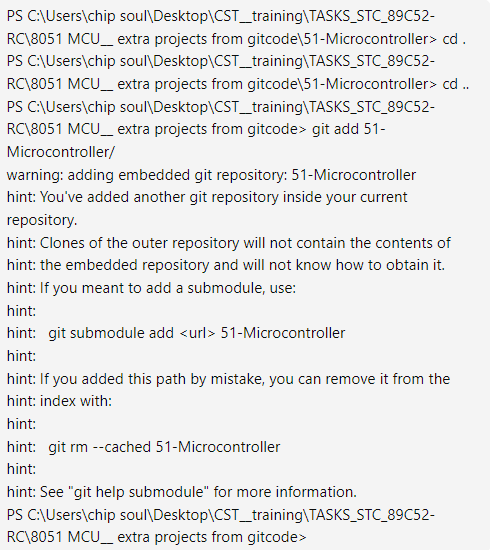
bash

Copy code

51-Microcontroller/

Then Git will stop notifying you about this untracked file.

In summary, Git is letting you know that there are local commits not yet pushed and that you have untracked files which are not part of any commits. You can either add and commit them or ignore them, depending on your preference.



The warning you're seeing is because the folder 51-Microcontroller/ contains a separate Git repository inside your current Git project. Git calls this an "embedded repository."

### What happened:

* **Embedded Git repository**: You added a folder that itself contains a Git repository (51-Microcontroller/). Git is warning you that cloning your current repository in the future will **not** clone the inner 51-Microcontroller repository, which could cause confusion for collaborators.
* If your intention was to treat the 51-Microcontroller/ directory as a regular folder, **without its own Git repository**, you will need to fix this.

### How to resolve the issue:

#### Option 1: Remove the embedded repository (if you don't need the separate Git repo inside):

If you want the 51-Microcontroller/ folder to just be part of your current repository (and **not** a separate Git repository):

1. **Remove the embedded Git repository from tracking**:

bash

Copy code

git rm --cached 51-Microcontroller

1. **Delete the .git folder inside 51-Microcontroller/**: This removes the Git repository from that subdirectory but keeps the files:

bash

Copy code

rm -rf 51-Microcontroller/.git

1. **Add the cleaned directory to your main repository**:

bash

Copy code

git add 51-Microcontroller/

git commit -m "Add 51-Microcontroller without embedded Git repository"

git push

#### Option 2: Use it as a Git submodule (if you want to keep it as a separate Git repository):

If you want to **keep** 51-Microcontroller/ as its own repository but include it in your current project as a submodule:

1. **Remove it from the index**:

bash

Copy code

git rm --cached 51-Microcontroller

1. **Add it as a submodule**:

bash

Copy code

git submodule add <url-to-51-Microcontroller-repo> 51-Microcontroller

1. **Commit the change**:

bash

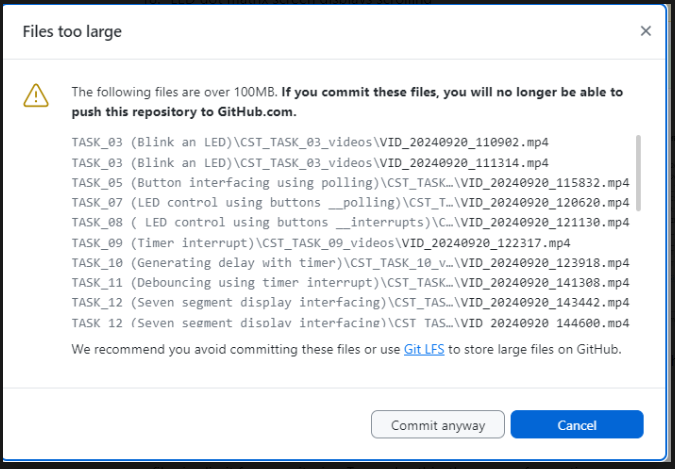
Copy code

git commit -m "Added 51-Microcontroller as a submodule"

git push

### Conclusion:

* If you don't need the separate repository, follow **Option 1** to remove the embedded Git repo.
* If the 51-Microcontroller directory is a project you plan to keep as its own repository, use **Option 2** to add it as a submodule.



How To Resolve The Issue Please Explain It And Show Me A Solution

The issue here is that you're trying to commit files larger than 100MB to GitHub, and GitHub imposes a file size limit for repositories. To resolve this, there are a few options you can consider:

### Solution 1: Use Git LFS (Large File Storage)

Git LFS is a system for storing large files. You can use it to push large files (like videos) to GitHub.

#### Steps to use Git LFS:

1. **Install Git LFS** (if you haven't already):
   * For Windows, you can download it from <https://git-lfs.github.com/>.
   * For Linux or macOS, you can use a package manager like brew install git-lfs (macOS) or sudo apt-get install git-lfs (Linux).
2. **Initialize Git LFS in your repository**: In your terminal or Git Bash, run:

bash

Copy code

git lfs install

1. **Track large files with Git LFS**: You need to tell Git LFS which file types to track. For example, to track .mp4 files, run:

bash

Copy code

git lfs track "\*.mp4"

1. **Commit the changes**: Now, you need to commit the .gitattributes file that Git LFS creates to track large files:

bash

Copy code

git add .gitattributes

git commit -m "Track MP4 files with Git LFS"

1. **Add your files and push to GitHub**: Now you can add and push your large files to GitHub as usual:

bash

Copy code

git add .

git commit -m "Add large files using Git LFS"

git push origin main

### Solution 2: Remove Large Files and Upload Separately

If you don’t need to store large files in the repository, you can remove them and upload them elsewhere (e.g., cloud storage, YouTube for videos) and reference them in your repository.

#### Steps:

1. **Remove the large files from the commit history**: If you've already committed the large files, you can use git rm to remove them:

bash

Copy code

git rm --cached <path\_to\_large\_file>

git commit -m "Removed large file"

1. **Push changes to GitHub**: Now, push the changes to GitHub without the large files:

bash

Copy code

git push origin main

### Solution 3: Split Files

If possible, you can try splitting large files into smaller parts (less than 100MB) using tools like ffmpeg for videos or file compression.

#### Steps:

1. **Split the file into smaller chunks**: For example, to split a video file into smaller parts using ffmpeg:

bash

Copy code

ffmpeg -i largefile.mp4 -c copy -map 0 -segment\_time 00:05:00 -f segment output%03d.mp4

1. **Add and commit smaller chunks**: Once you have the smaller files, you can add and commit them like regular files:

bash

Copy code

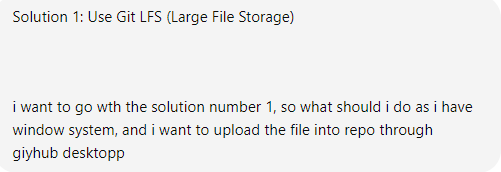
git add .

git commit -m "Add split files"

git push origin main

### Conclusion

The best option depends on your use case. If you want to keep the files in GitHub, Git LFS is the recommended solution. If you don't need them in the repository, consider removing them or hosting them elsewhere.



To implement **Solution 1: Use Git LFS (Large File Storage)** through **GitHub Desktop** on Windows, follow these steps:

### Step-by-Step Guide to Use Git LFS with GitHub Desktop:

#### 1. **Install Git LFS on Windows**

* Visit the [Git LFS website](https://git-lfs.github.com/) and download the Git LFS installer for Windows.
* Run the installer and follow the installation instructions.

#### 2. **Initialize Git LFS in your repository**

Now that Git LFS is installed, you need to initialize it in your local repository:

* Open **GitHub Desktop** and navigate to the repository where you want to upload the large files.
* Open the repository in the **Command Prompt** by clicking **Repository > Open in Command Prompt** from the top menu of GitHub Desktop.

In the Command Prompt, initialize Git LFS by typing:

bash

Copy code

git lfs install

#### 3. **Track Large Files with Git LFS**

You need to tell Git LFS which files it should manage. Since your files are video files (e.g., .mp4), you will track those specific file types.

In the Command Prompt, type:

bash

Copy code

git lfs track "\*.mp4"

This command will create a .gitattributes file, which tells Git LFS to manage all .mp4 files in your repository.

#### 4. **Commit the Changes in GitHub Desktop**

* Return to **GitHub Desktop**.
* You should now see that a new file .gitattributes has been added to the list of changes.
* **Commit** this file by adding a commit message (e.g., "Track large files using Git LFS") and click **Commit to main**.

#### 5. **Add Your Large Files**

Now you can add your large .mp4 files or any other files you want to track using Git LFS to the repository.

* Move the large files into your repository folder.
* GitHub Desktop will automatically detect these files as new changes.

#### 6. **Commit and Push the Large Files**

* In **GitHub Desktop**, enter a commit message (e.g., "Add large video files").
* Click **Commit to main** to commit the large files to your local repository.

After committing, click **Push origin** to upload your large files to the remote GitHub repository using Git LFS.

#### 7. **Verify Files are Uploaded Using Git LFS**

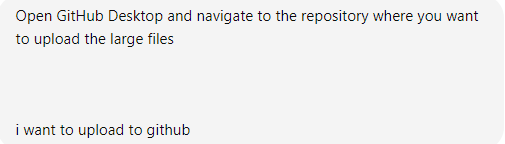
Once the files are pushed, you can verify that the files are tracked by Git LFS on GitHub:

* Go to your repository on GitHub.com.
* The large files should be listed, but instead of the file being directly uploaded, you will see that Git LFS is handling the files, and there will be a pointer file that Git LFS uses to manage them.

### Important Notes:

* **Storage Limits**: Git LFS comes with a limited amount of free storage and bandwidth (1 GB storage, 1 GB bandwidth per month). You may need to purchase additional storage if your project exceeds this limit. You can check GitHub's pricing [here](https://github.com/pricing).
* If you face any issues with Git LFS setup, you can refer to [GitHub LFS documentation](https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-large-files).

Now, you're all set to upload and manage large files through **Git LFS** in **GitHub Desktop**!



To upload large files to GitHub using **Git LFS** via **GitHub Desktop**, follow these steps:

### Step-by-Step Guide for Uploading Large Files to GitHub with Git LFS:

#### 1. **Install Git LFS**

First, you need to install Git LFS (if you haven't already).

* Download Git LFS from [Git LFS website](https://git-lfs.github.com/).
* Install it by following the on-screen instructions.

#### 2. **Initialize Git LFS in Your Repository**

Once Git LFS is installed, you can initialize it for your repository.

* Open **GitHub Desktop**.
* Navigate to the repository where you want to upload the large files.

To open the repository in **Command Prompt**:

* In **GitHub Desktop**, go to **Repository > Open in Command Prompt**.

In the Command Prompt, initialize Git LFS by running the following command:

bash

Copy code

git lfs install

#### 3. **Track Large Files**

Now you need to specify which file types you want Git LFS to track. Since your files are videos (.mp4 files), track them using this command:

bash

Copy code

git lfs track "\*.mp4"

This command creates a .gitattributes file in your repository to track .mp4 files with Git LFS.

#### 4. **Add and Commit Changes in GitHub Desktop**

* Return to **GitHub Desktop**.
* You should see that the .gitattributes file has been added.
* In **GitHub Desktop**, write a commit message (e.g., "Track large files using Git LFS") and click **Commit to main** to commit this change.

#### 5. **Add the Large Files**

Now you can add your large files (e.g., .mp4 videos) to your repository folder.

* Copy the large files into the repository folder on your local machine.
* GitHub Desktop will detect these new files as changes.

#### 6. **Commit and Push the Large Files**

* In **GitHub Desktop**, enter a commit message (e.g., "Add large video files").
* Click **Commit to main**.
* After committing, click **Push origin** to upload the files to your GitHub repository.

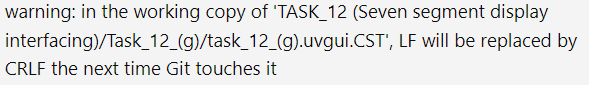
#### 7. **Verify on GitHub**

Once the files are pushed, you can verify on GitHub.com that the files have been uploaded using Git LFS.

* Go to your GitHub repository online.
* Check that the large files are being tracked by Git LFS. Instead of the raw file being uploaded, you’ll see that GitHub uses a pointer file to manage the large files through Git LFS.

### You're Done!

Your large files should now be uploaded and managed by Git LFS on GitHub via **GitHub Desktop**.



The warnings you're seeing are related to line-ending differences between Linux/Unix-style (LF) and Windows-style (CRLF) line breaks. Git is alerting you that it will automatically convert the line endings from LF to CRLF when it updates the working copy of those files.

Here’s a brief explanation and some potential solutions:

### What it means:

* **LF (Line Feed)**: Used as the line-ending character in Unix/Linux systems.
* **CRLF (Carriage Return + Line Feed)**: Used as the line-ending character in Windows systems.

Git can automatically convert line endings to match the platform you are working on. When Git detects that files have LF endings but are being used on a Windows machine, it gives this warning because the next time Git touches these files, it will replace LF with CRLF.

### To avoid these warnings:

1. **Check Git's core.autocrlf setting**:
   * You can configure Git to handle line endings automatically with the core.autocrlf setting.
   * On **Windows**, set core.autocrlf to **true**:

bash

Copy code

git config --global core.autocrlf true

This converts LF to CRLF when checking out text files and converts CRLF to LF when committing them back to the repository.

* + On **Linux/Mac**, set core.autocrlf to **input**:

bash

Copy code

git config --global core.autocrlf input

This ensures that files will be checked out with LF line endings on all platforms, and CRLFs will be converted to LF when committing.

1. **Manual Line Ending Conversion**: If you prefer to control line endings manually, you can run the following command in your repository to normalize the line endings:

bash

Copy code

git add --renormalize .

1. **Use a .gitattributes file**: You can add a .gitattributes file in your project to enforce line-ending rules for specific file types. For example, to enforce LF for all text files:

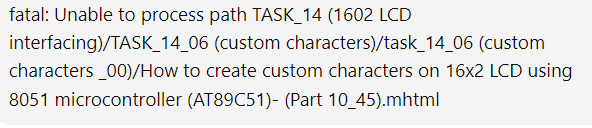
bash

Copy code

\* text=auto

This will ensure that Git automatically handles line endings based on the user's operating system.

By using one of these methods, you can suppress or prevent these warnings and ensure consistent line endings across different systems.



The error message you are seeing, fatal: Unable to process path, typically occurs when Git cannot handle certain file paths, either because they are too long, contain special characters, or use a format that's incompatible with the underlying file system.

In this specific case, the problem might be due to:

1. **Special characters**: Your file path includes spaces, parentheses (), and underscores \_. While Git can handle most of these, some combinations may cause issues on certain operating systems.
2. **File path length**: If the total path exceeds a certain length (typically 260 characters in Windows), Git may fail to process it.

### Potential Solutions

#### 1. Rename the file or folders

Try renaming the file or folders in the path to remove special characters like parentheses and spaces. For example:

* TASK\_14 (1602 LCD interfacing) to TASK\_14\_1602\_LCD\_interfacing
* task\_14\_06 (custom characters \_00) to task\_14\_06\_custom\_characters\_00

#### 2. Enable long paths (if on Windows)

If you're using Windows, enabling support for long paths might help:

* **For Git**: Open a Git Bash terminal and run the following:

bash

Copy code

git config --system core.longpaths true

* **For Windows (System-wide setting)**:
  1. Press Win + R, type gpedit.msc, and press Enter.
  2. Navigate to: Computer Configuration > Administrative Templates > System > Filesystem.
  3. Find and enable the policy: **Enable Win32 long paths**.

#### 3. Move the file to a shorter directory

Temporarily move your project to a directory with a shorter path (e.g., move it closer to the root of your drive).

#### 4. Use Git Bash with the correct encoding

If you’re on a non-Windows system, try using Git through Git Bash or your terminal and ensure the file path encoding is UTF-8.

The error message you received indicates that you don't have permission to modify the global configuration file for Git located in C:/Program Files/Git/etc/gitconfig. This is likely because you're trying to run the command without administrative privileges.

### Here are a few ways to resolve the issue:

#### 1. Run Git Bash as Administrator

To run Git Bash with elevated privileges, follow these steps:

1. **Close Git Bash** if it’s open.
2. **Right-click** on the Git Bash icon.
3. Select **Run as administrator**.
4. Once in Git Bash, try running the command again:

bash

Copy code

git config --system core.longpaths true

#### 2. Use Local Repository Configuration

If you don't want to modify the system-wide configuration, you can set this option for your specific repository:

1. Open Git Bash in your repository directory.
2. Run the following command:

bash

Copy code

git config core.longpaths true

#### 3. Check the Environment

If you're still facing issues after running as administrator, check your environment settings:

* Make sure you're not using an outdated version of Git. You can update it from [the official Git website](https://git-scm.com/downloads).
* If the problem persists, you can consider reinstalling Git, ensuring that you select the option to enable long path support during the installation process.

After applying these solutions, try running your Git commands again and see if the issue is resolved. Let me know if you need further assistance!

Changing directories in Git Bash is similar to using the command line in other terminal environments. Here’s how to do it:

### 1. Using cd Command

The cd (change directory) command is used to navigate between directories.

#### **Basic Syntax:**

bash

Copy code

cd [directory]

### 2. Examples

* **To navigate to a specific folder:** If you want to change to a folder named my\_folder in your current directory:

bash

Copy code

cd my\_folder

* **To go up one directory:** To move up to the parent directory:

bash

Copy code

cd ..

* **To go to a specific path:** If you want to change to a directory at a specific path (e.g., C:\Users\YourUsername\Documents\Projects):

bash

Copy code

cd /c/Users/YourUsername/Documents/Projects

* **To navigate to your home directory:** You can simply type:

bash

Copy code

cd ~

or just:

bash

Copy code

cd

### 3. Tab Completion

While typing the directory name, you can use the **Tab** key to auto-complete directory names, which helps avoid typing errors and speeds up navigation.

### 4. Check Current Directory

If you want to check your current directory, you can use:

bash

Copy code

pwd

This command (print working directory) will display the path of the current directory you are in.

### 5. Example Workflow

Here’s a simple example of changing directories:

1. Open Git Bash.
2. Check your current directory:

bash

Copy code

pwd

1. Change to the Desktop directory:

bash

Copy code

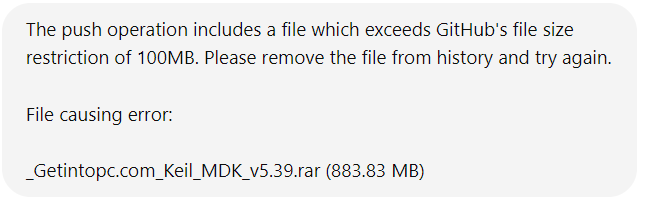
cd ~/Desktop

1. List the contents of the Desktop:

bash

Copy code

ls



To resolve the issue of pushing a file that exceeds GitHub's file size limit, you can follow these steps to remove the file from your Git history and then attempt the push again. Here's a detailed guide:

### Step 1: Remove the Large File from History

1. **Clone the repository (if needed):** Make sure you have a local clone of your repository. If not, you can clone it using:

bash

Copy code

git clone https://github.com/your-username/your-repo.git

cd your-repo

1. **Use the git filter-repo Command:** If you don’t have git filter-repo installed, you might want to install it as it's more efficient than git filter-branch.
   * To install git filter-repo, run:

bash

Copy code

pip install git-filter-repo

* + Then, navigate to your repository directory and run:

bash

Copy code

git filter-repo --invert-paths --path \_Getintopc.com\_Keil\_MDK\_v5.39.rar

1. **If you don't have git filter-repo:** Use git filter-branch instead (but this is less efficient):

bash

Copy code

git filter-branch --force --index-filter \

"git rm --cached --ignore-unmatch \_Getintopc.com\_Keil\_MDK\_v5.39.rar" \

--prune-empty --tag-name-filter cat -- --all

### Step 2: Clean Up the Repository

After removing the file, clean up any leftover references:

1. Run the following commands to clean up:

bash

Copy code

rm -rf .git/refs/original/

git reflog expire --expire=now --all

git gc --prune=now --aggressive

### Step 3: Check the File Size

Check if the file is still tracked or present in your repository:

bash

Copy code

git ls-files | grep \_Getintopc.com\_Keil\_MDK\_v5.39.rar

If nothing is returned, the file has been successfully removed.

### Step 4: Push Your Changes

Now you can attempt to push your changes to GitHub again:

bash

Copy code

git push origin main --force

(Replace main with the appropriate branch name if it’s different.)

### Step 5: Add the Large File to Git LFS (Optional)

If you still need to keep that large file in your repository, consider using Git Large File Storage (LFS):

1. **Install Git LFS:** If you haven't installed Git LFS, do so by following the instructions on the [Git LFS website](https://git-lfs.github.com/).
2. **Track the Large File:**

bash

Copy code

git lfs track "\_Getintopc.com\_Keil\_MDK\_v5.39.rar"

1. **Add and Commit the Change:**

bash

Copy code

git add .gitattributes

git add \_Getintopc.com\_Keil\_MDK\_v5.39.rar

git commit -m "Add large file with LFS"

1. **Push to GitHub:**

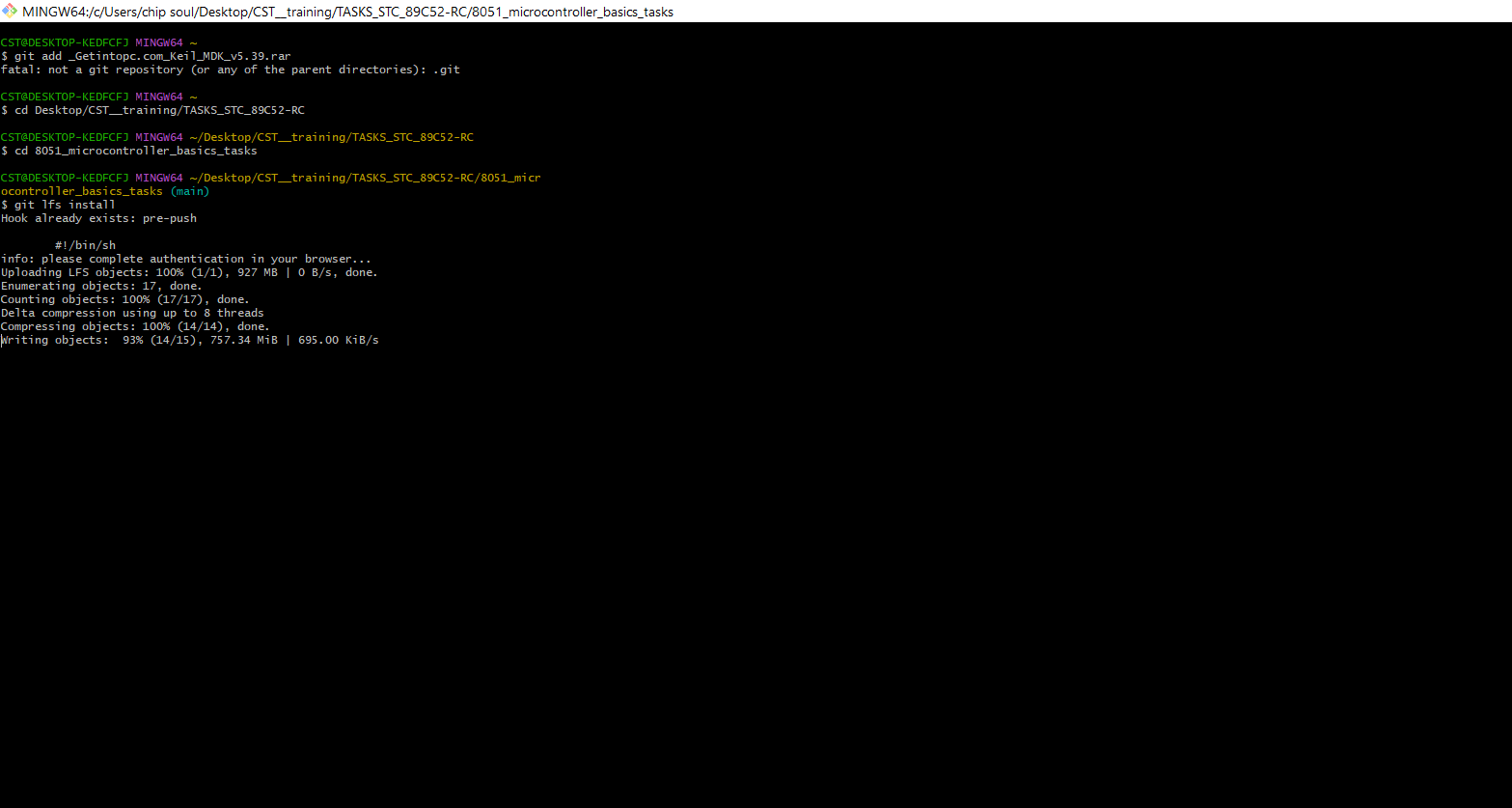
bash

Copy code

git push origin main

This should help you resolve the issue of pushing the large file while keeping your repository within GitHub's size limits. If you have any further questions, feel free to ask!

**Result:**

****