# Introduction

The objective of this lab was to learn to use a source control management tool and to highlight the importance of using it in a multi-developer environment. There is also a comparison with a similar tool explaining the benefits and limitations of each.

# Aims/Objectives

* Setup a repository in Azure DevOps
* Install git locally and setup the credentials for the Azure DevOps
* Create a sample project in Visual Studio (used the free community edition for this purpose)
* Add/Stage, Commit and Push code into the newly created repository
* Share and give access to the project to a colleague to make changes in a file from his side
* Explore the various features while working on the same project by two developers

# Method

1. Installed git client in the local machine. This is needed to execute git commands connecting the remote repositories
2. Logged into Azure DevOps services (<https://app.vsaex.visualstudio.com/me>)
3. Created a new organisation and a new project under it. This created the full list of services offered free including CI/CD pipeline features by the Azure DevOps suite as shown in Figure 2.
4. After cloning the repository into a local folder, created a sample .NET Web Api project using Visual Studio in the same location. This created a *.git* folder in the local location.
5. Staged the files, committed, and pushed the files back into the remote repository. While staging files and committing them are local activities, pushing gets the changes into the remote repository that is in the Azure DevOps service. It is worth noting that committing our changes needs a comment which is mandatory. This is for describing what our change is about and will appear in commit history.
6. Once the push is completed, the pushed changes/files can be seen in the repository.
7. To mimic developer collaboration, took help from a colleague to edit the same file to simulate a conflict situation.
8. Relied on the capability of Visual Studio to resolve the conflict and do a merge into the main/master branch

# Results & Analysis

* Creating of a new organisation and a project under it was straightforward. Most of the basic tasks performed can be done through the command line using git commands, few for instance:
  + git pull: to pull changes or update our local repository
  + git commit: to commit our local staged changes
  + git push: to push our changes to remote server

Figures 1, 2 and 3(clone via command line) shows the setting up of a repository

* The sample project used for saving into the repository is the default template one for Web API (Application Programming Interfaces) project which underwent minor code edits as part of testing.
* Git has the concepts of remote and local repositories. While the commits we make are reflected immediately in the local repository, it requires the push command to get our changes onto the remote server. Figures 4 to 6 shows the staging of changes commits and push to remote.
* The pushed changes once successful can be viewed from the “Files” view in the browser under the selected repository. Refer Figure 7 that shows the repository after 3 pushes including the push from my colleague. This ensures that our changes have reached the server.
* Pulling occasionally pulls any changes that are made to the remote repository by a different user or from different location/device by the same user. This should be an ideal step before start of a work/task to get the latest files before making changes to reduce the chance of a conflict.

*Conflict:*

A conflict was created deliberately to understand it better and how it can be resolved and progress on to push the changes that we are presently doing. The conflict is reported back to us from server when there is a change in the same file(s) that we edited by another person that has been pushed while we were working on the previous version of that file. There are two approaches to encountering and resolving a conflict:

Approach 1:

* A pull right before committing our changes to update our repository will result in a conflict.
* Here we will stash our local changes which will create a temporary file to move our changes and restore the files to the previous version
* Then pull again to get the remote changes applied cleanly to our local copies.
* Then apply the stash over the new copies. This might require us to use a 3rd party editor to resolve conflicts if it occurs line-by-line.
* Now we can stage, commit, and push our changes.

Approach 2:

* Without being aware of a potential conflict we would stage, commit, and attempt to push our local changes
* This will fail as the version we made changes to is different from the one in server/remote now
* We will have to merge the changes from remote into our local branch/repository first
* Then do a second commit for the merge that we just did and finally the push to remote.

*Conflict resolution*

Approach 1 is taken to resolve a deliberately created conflict. For this I have used my colleague’s help once again to commit a file that I edited before doing a pull to bring in his changes to my local. So here my local copy of that file will not have his changes.

* As shown in Figure 8, running a pull now will fail and will require us to stash our local changes and then do a clean pull again.
* Now we can apply back our stashed changes on top of the new version. This will result in a conflict which we need to resolve and merge back
* Used Visual Studio’s git window Figure 9, to resolve the conflict(s). Visual Studio presents us with 3-splitted window to easily resolve the conflict. Each window is as below:
  1. Top-left window shows the incoming changes that is the changes that we have stashed and is being applied
  2. Top-right window shows the present version that we have now
  3. The bottom one shows the result of our action that we perform on the top windows. All red/amber coloured conflicts need to be in green by the end to perform a “Accept Merge” action to perform the push. Figure 10 shows all conflicts being resolved.
* The Visual Studio git window has different options to resolve a conflicted line or a block of code:
  1. Accept the change in the top-left window. That is the incoming change from the stashed local change
  2. Accept the change in the top-right window. That is the accept the present version we pulled from remote
  3. In addition to either of the above two, we could directly edit the code by hand on the bottom results window.
* Once the merge is accepted it means the conflicts are resolved and now, we can push our changes to remote.

# Conclusion

All the client tools (Visual Studio Community Edition and git SCM) and services used are freely available ones. While setting up of repositories is free in Azure DevOps, there is a limitation set by Microsoft on the number of users that can be associated with a project under the free tier (Microsoft, n.d.). The benefits of having a source control system are evident with these activities performed. The involvement and contribution of my colleague acting as another developer further highlights its importance. Source control systems have their place in a multi-developer scenario. Without it, code maintenance and integration will be a nightmare to perform and someone has to spend hours on integrating code contributed by different members in a team on a daily basis. From VSS (Visual Source Safe (Wikepedia, n.d.)), one of the earliest source control systems in the 90s which had a client-server architecture many systems came into existence and evolved with newer features and capabilities to make not only the development easier but also the integration into CI/CD pipelines. Both the mentioned GitHub and Azure DevOps Git repository has the capability of integrating with a CI pipeline. The integration pipeline can connect with these repositories to pull the code and build the application(s) to create artifacts to deploy these onto the right environment by the deployment pipeline. There is also the possibility of executing unit test and integration test which are also codes and scripts checked into the repository hence part of the source control.

The capabilities of Azure DevOps repository are pretty much the same as of GitHub repository which is a more popular source control system and I am familiar with and have a level of hands-on experience working on few projects. Besides the basic features like pull, push, revert and commit, the advanced features too are equally available by both of them. Some of the advanced features are:

*Merge*: Merge changes from a different branch to the current branch. This helps us to fetch new changes pushed by a colleague to another branch which we might need.

*Rebase*: Combining several previous commits in a different branch into another branch. This helps us to move to different commit than the one we initially created on our new/different branch.

*Stash*: Move the local change aside into a stash so that our code will be at the last pulled version. This helps us to pull new changes without conflicts.

While the command line tool is always handy to quickly run basic and simple commands, it’s worth mentioning about the GUI tools available for working with these repositories. There are freely available tools like SourceTree and TortoiseGit that are dedicated git client tools which gives us the power to perform the aforementioned git functions with simple clicks of buttons. Most of the IDEs such as Visual Studio and Visual Studio Code too have Git integration built-in. In this exercise the Git diff and conflict resolutions were done using Visual Studio IDE. Visual Studio has a dedicated window for git like the solution explorer window which is very easy to integrate with both Azure DevOps as well as a GitHub repository. All these tools are executing respective git commands or a series of commands in the background while we perform the clicks on their UI.

Although, both systems are Microsoft owned there are many differences that are easily spotted. While GitHub is more popular and the best used as a public repository, Azure DevOps is more corporate oriented. The setting up of organisation, setting up of team members with permissions is ideal for an organisation or from a team’s perspective. GitHub is more focussed on public availability and contributions to make a service or a product more stable over time. I have also come across scenarios where GitHub is used for sharing code across to a wider audience where the formalities of an organisation is unnecessary.

Another difference worth noting is that Azure DevOps has an entire suite built-in for an enterprise which even includes easy integration with Microsoft’s cloud platform, Azure through their CI/CD pipelines and a scrum board that easily fits for an agile team, whereas GitHub lacks most of these features or is in a pre-mature stage. There is the GitHub Actions that could be used for CI/CD activities and also a scrum board that is still evolving (Klint, 2022). The pipelines of Azure DevOps can be thought of more mature since it came into existence since a long time and many companies have proven it successfully using for their production releases. A team set up with Azure DevOps subscription does not have to rely on any other third-party services for most of their tasks in the software development lifecycle.

Even though I am more familiar and comfortable using the Azure DevOps repository, I embrace the idea of openness expressed by GitHub. The power packed features along with the easiness of setting up a corporate repository is impressive and seem to be ideal for the corporate world. However, the simplicity and public availability is more encouraging for public engagements and contributions.

# References

Klint, L., 2022. *Azure DevOps vs GitHub: Which DevOps tool is better?.* [Online]   
Available at: https://acloudguru.com/blog/engineering/azure-devops-vs-github-comparing-microsofts-devops-twins  
[Accessed 12 07 2022].

Microsoft, n.d. *Azure DevOps Services Pricing | Microsoft Azure.* [Online]   
Available at: https://azure.microsoft.com/en-gb/pricing/details/devops/azure-devops-services/  
[Accessed 20 10 2022].

Wikepedia, n.d. *Microsoft Visual SourceSafe.* [Online]   
Available at: https://en.wikipedia.org/wiki/Microsoft\_Visual\_SourceSafe  
[Accessed 22 10 2022].

Figure 1: *Create repository*

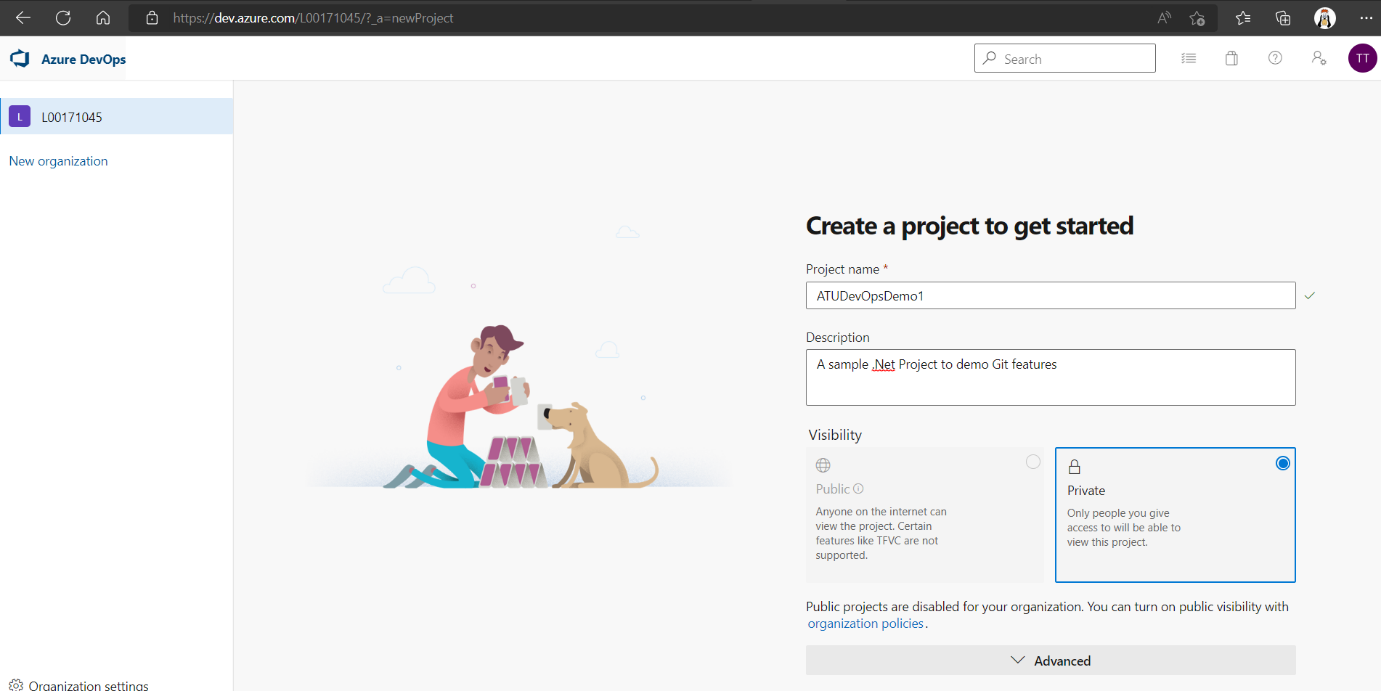


Figure 2: *URL for cloning a repository*

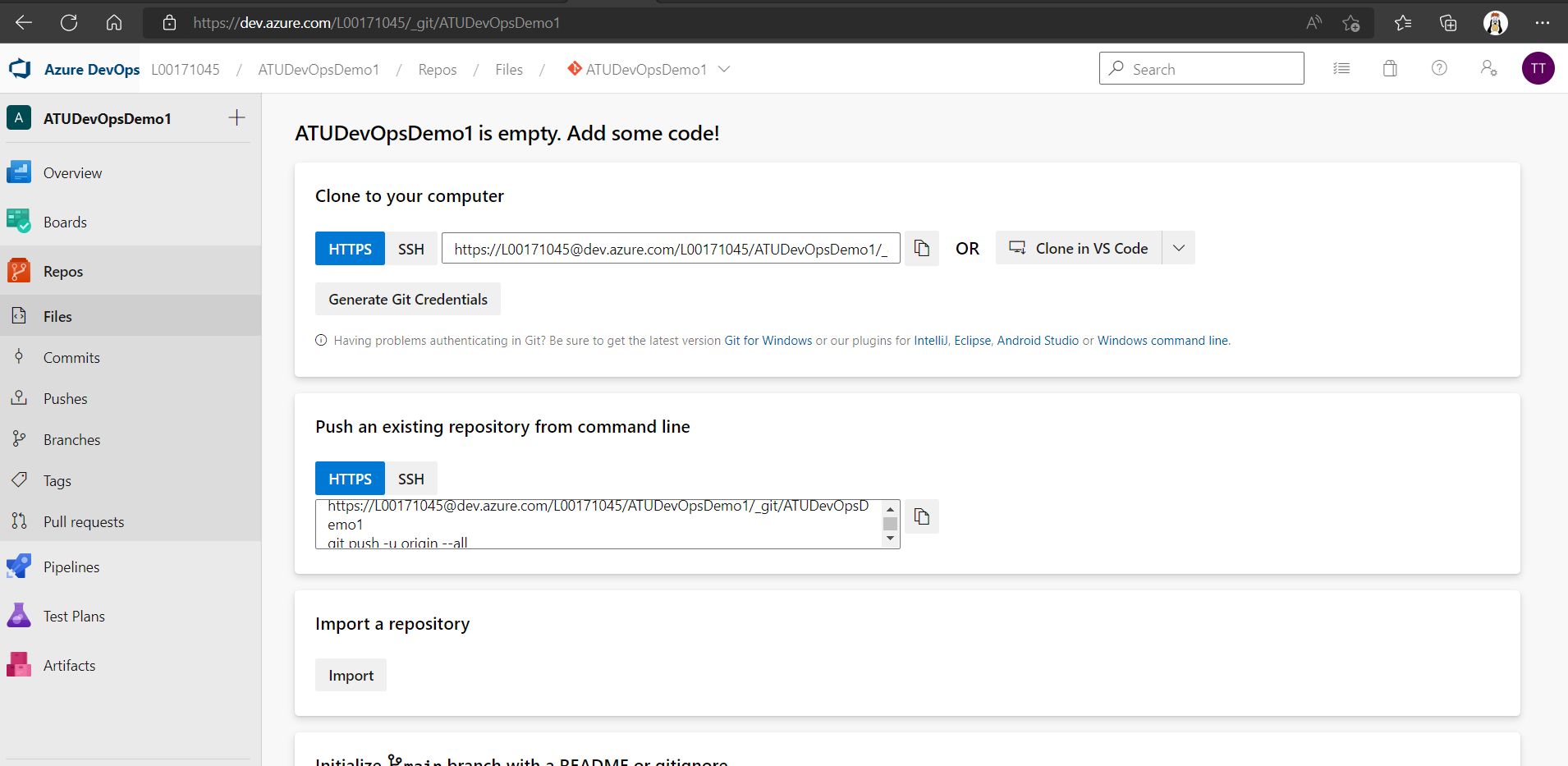


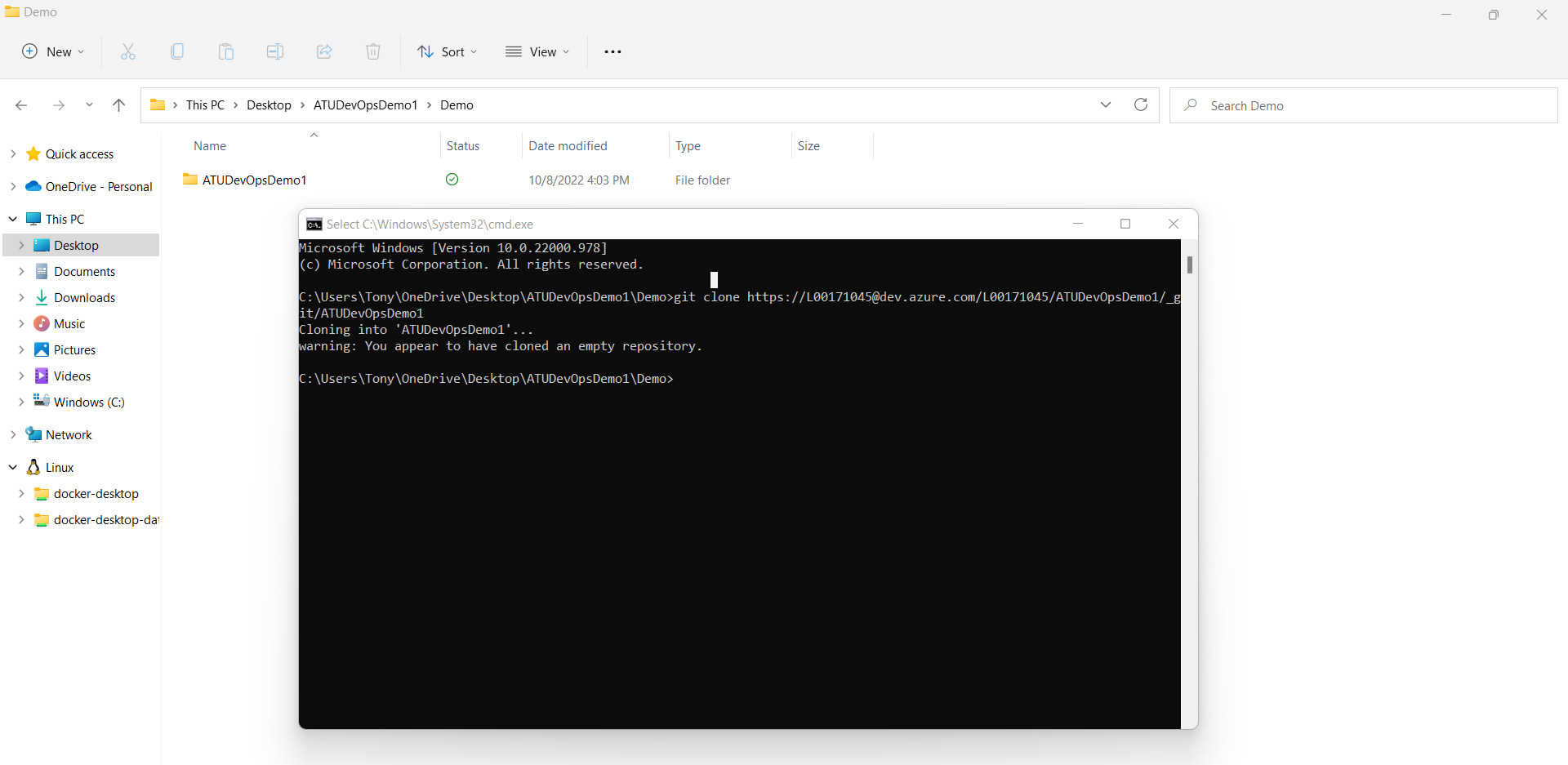
Figure 3: *Clone repository into a local folder via command line*

Figure 4: *Code files/changes added into the local folder*

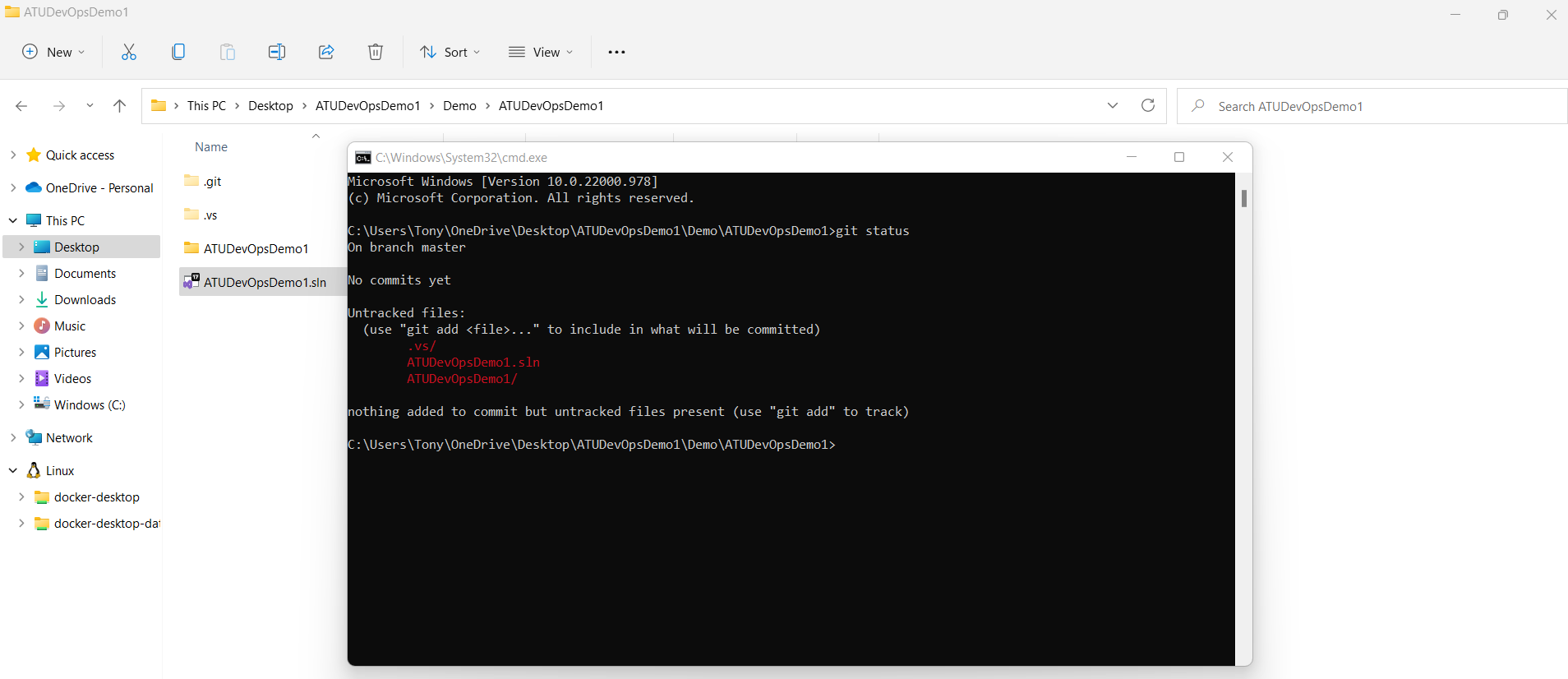


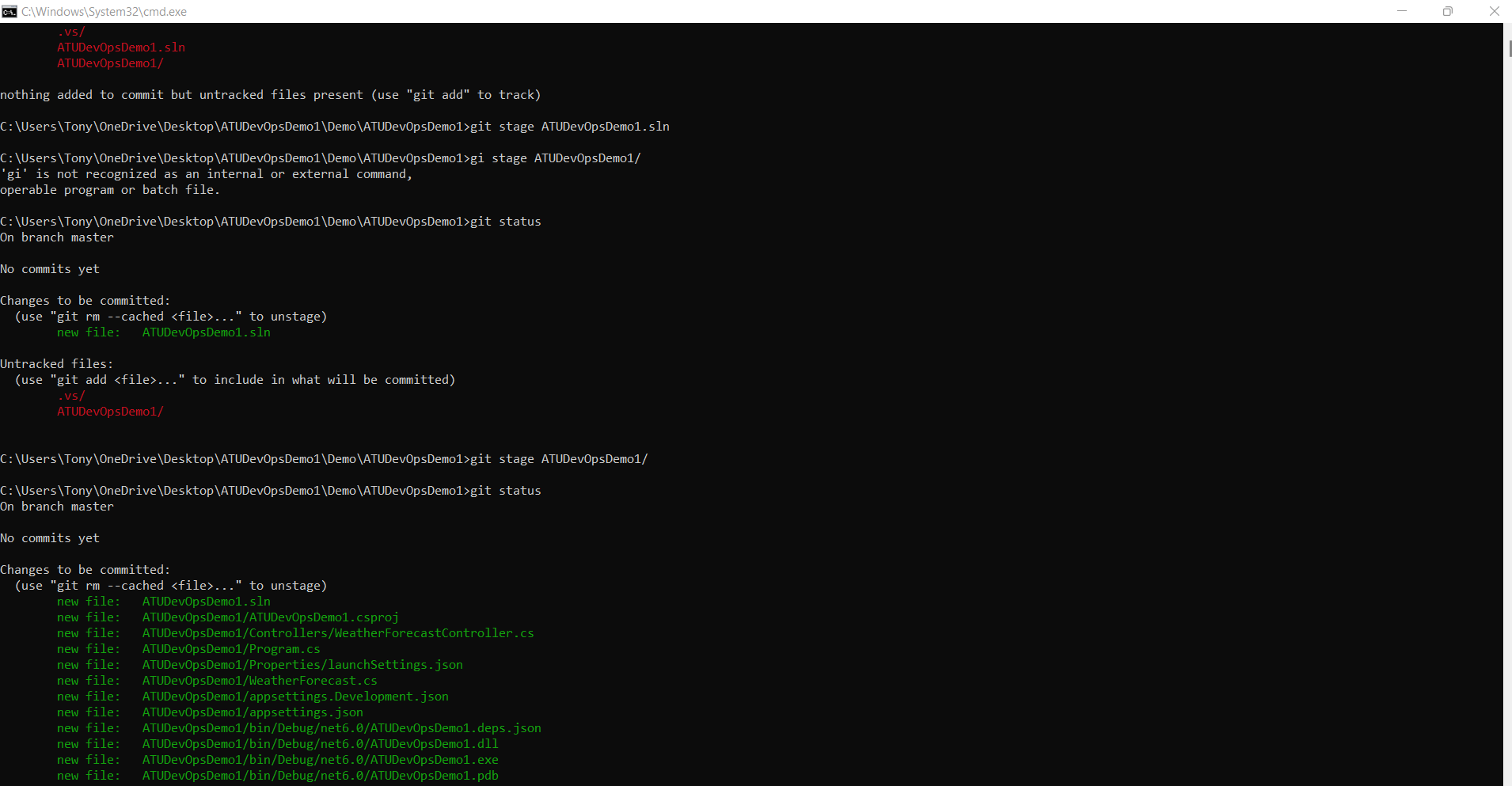
Figure 5: *Stage files via command line*

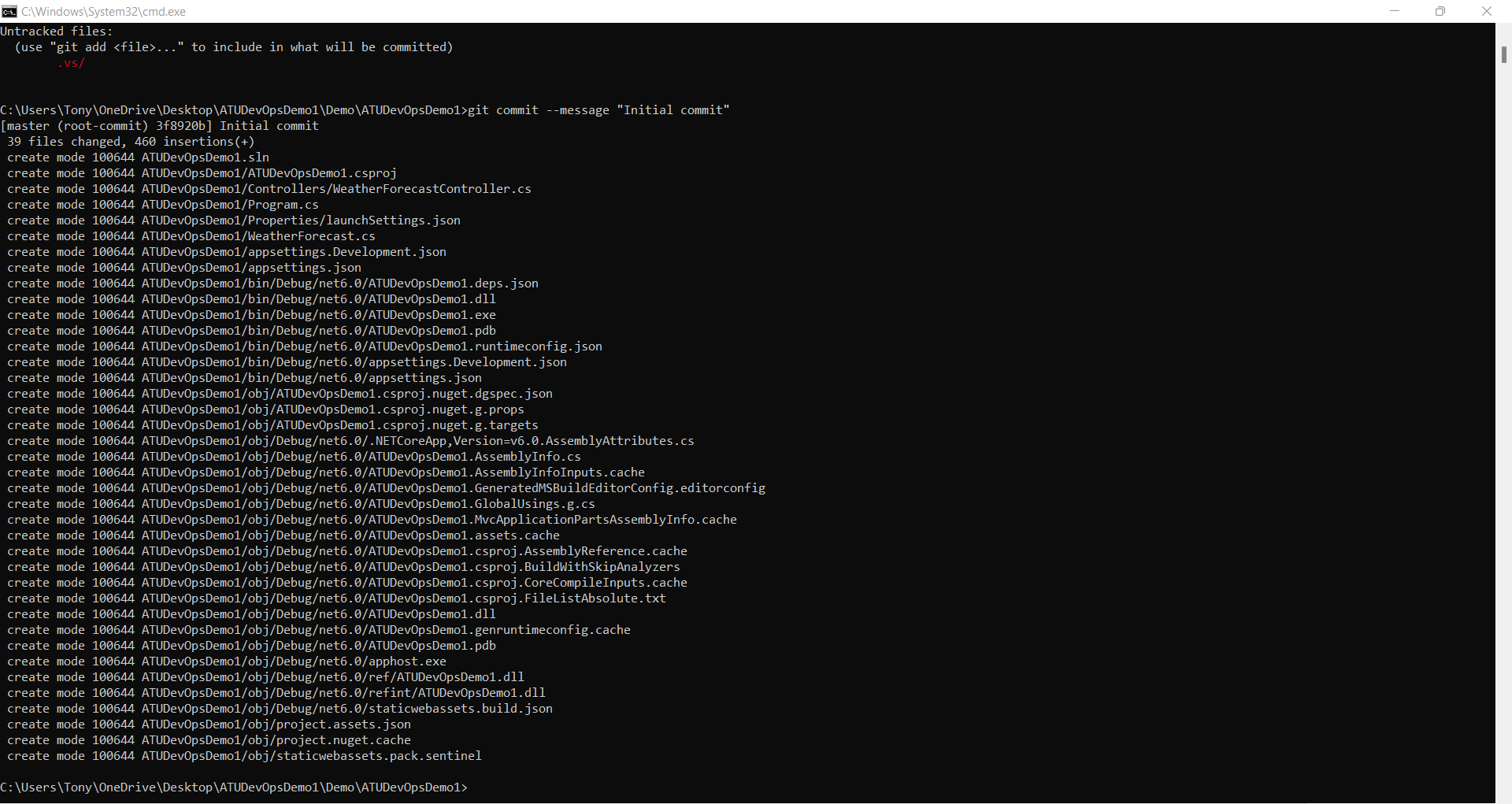
Figure 6: *Commit changes with a message/comment via command line*

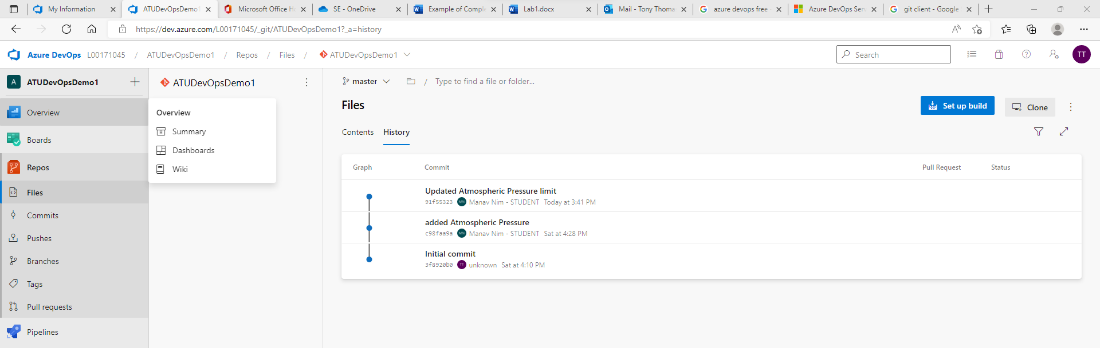
Figure 7: *Repository files view after 3 pushes*

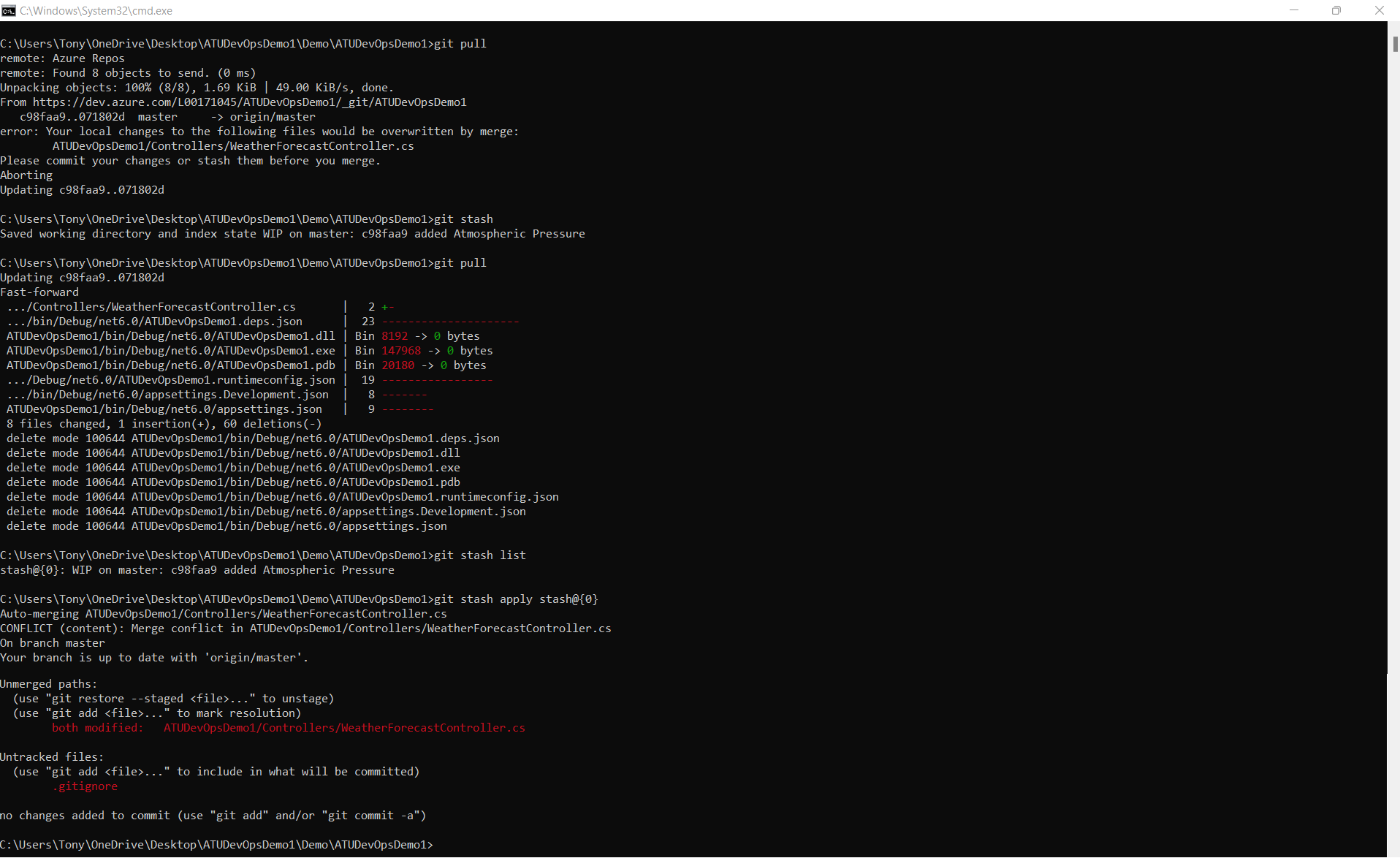
Figure 8: *a conflict during a pull*

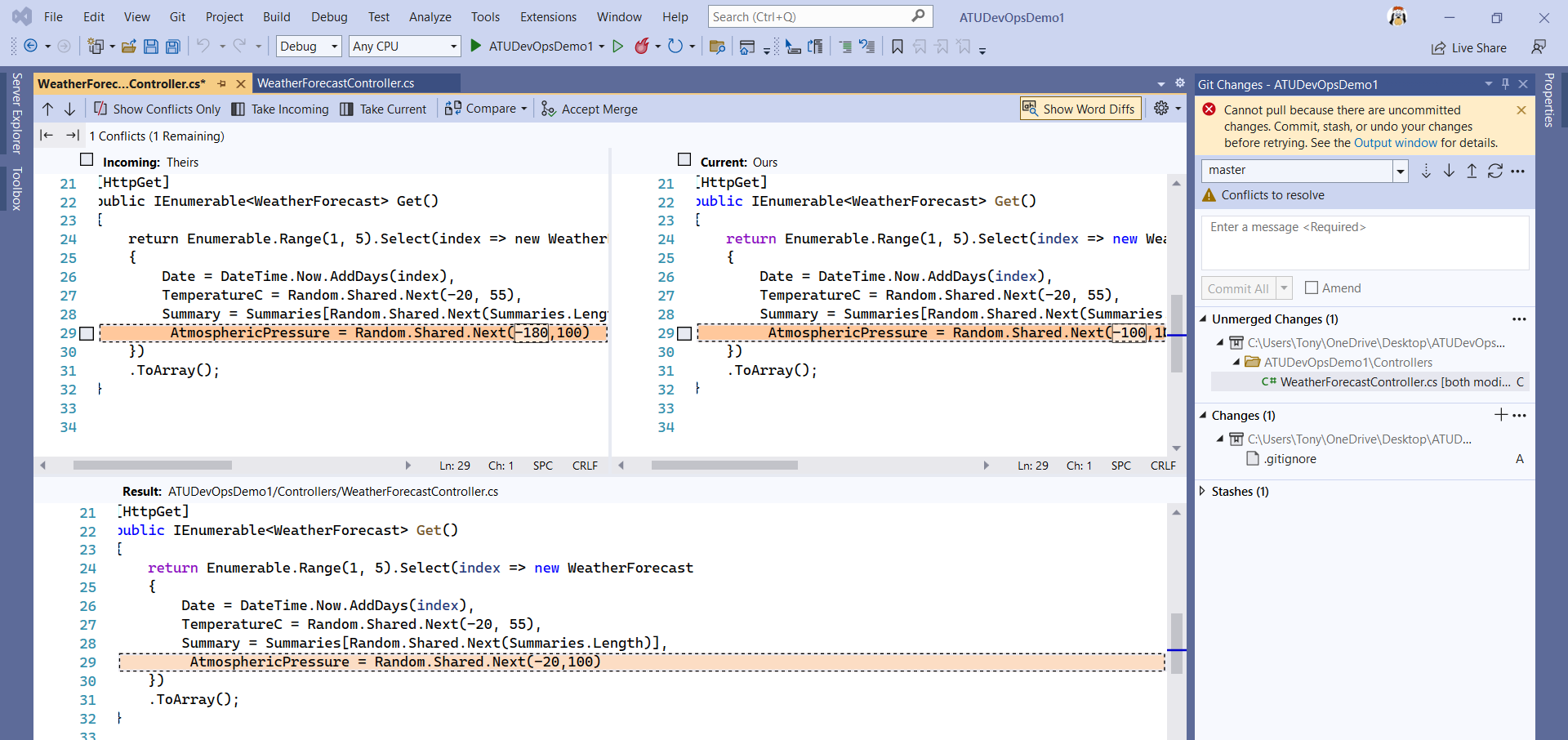
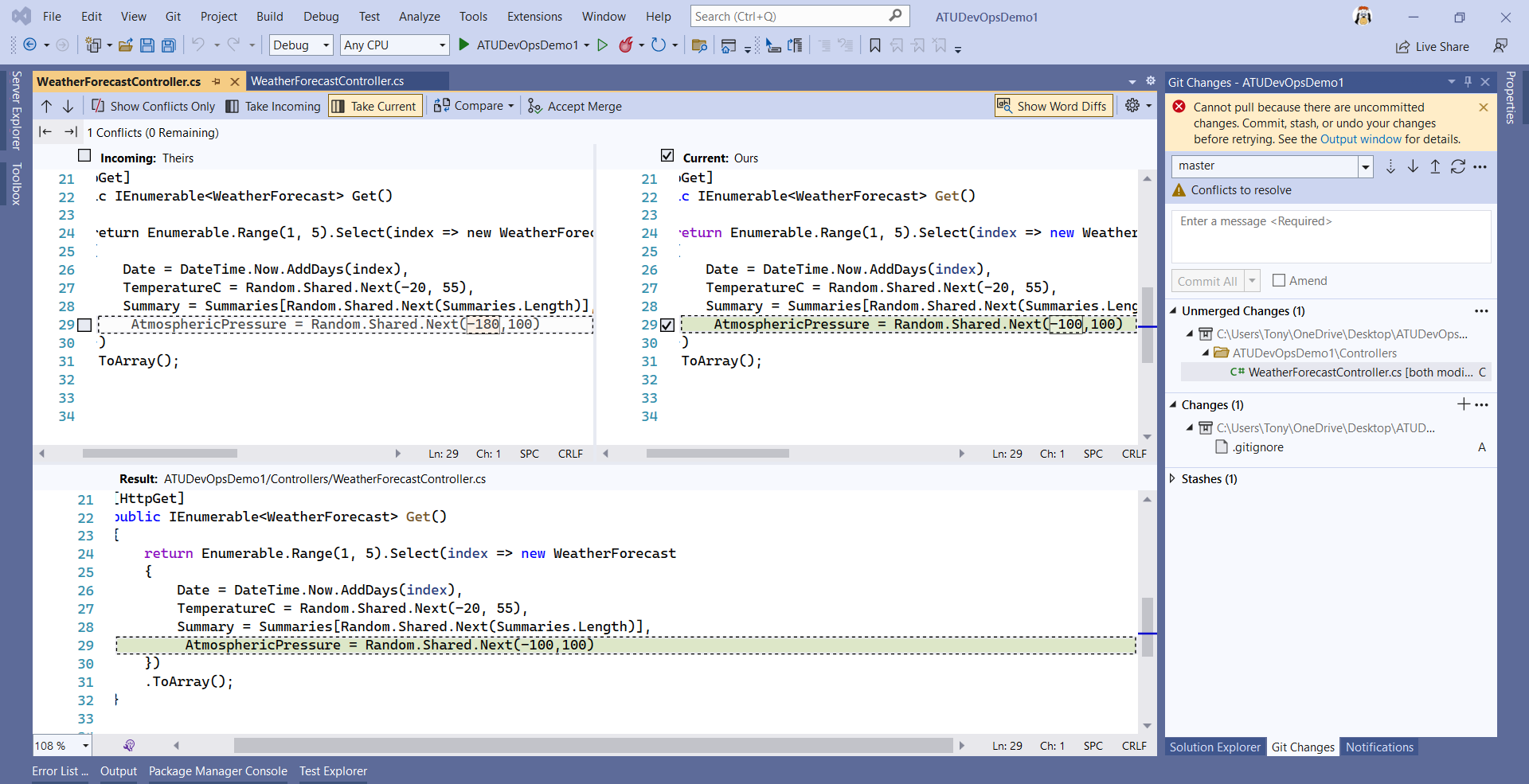
Figure 9: *Visual Studio conflict resolution merge window*

Figure 10: *Conflict resolved and to be merged*