Version Control In Planetary Science: A Practical Guide to Git for Academic Software Development

William Byrne[1,2], Quentin Betton[2,3], Christian Hüttig[1], Mario D'Amore[1]

[1]Institute of Space Research, German Aerospace Center (DLR), Berlin, Germany,
[2]Institute of Geological Sciences, Freie Universität Berlin, Berlin, Germany,
[3]Laboratoire de Planétologie et Géosciences, Nantes Université, Nantes, France

Introduction

Planetary scientists tend to write their own software, and many choose to without employing version control. Git [1] was developed to help manage version control for software development, yet in academia it remains inconsistent [2, 3]. This work attempts to ease the entry of understanding Git by providing three different workflows, designed for the solo-researcher up to a large collaborative group of 5+ people.

#1 Single workflow

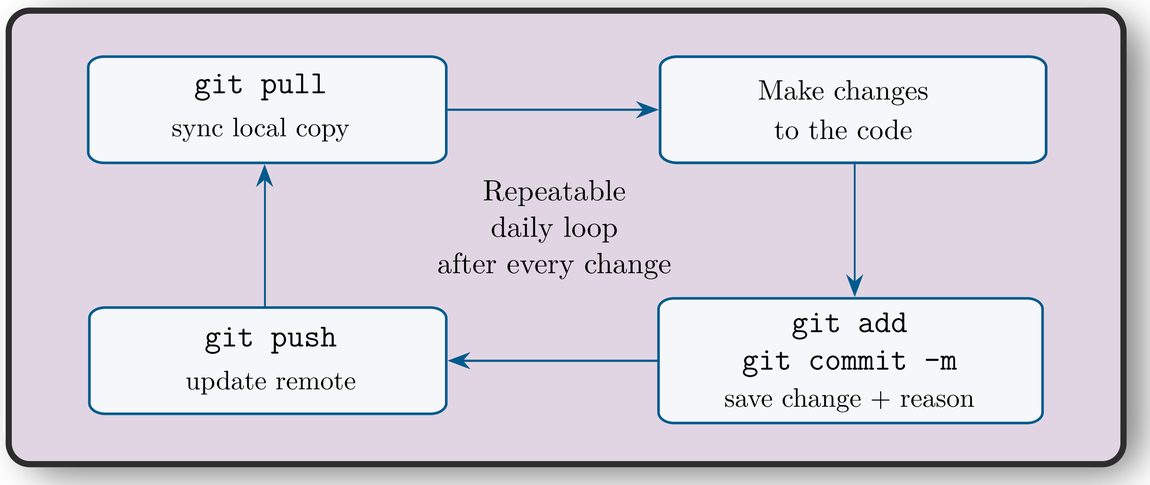
A single programmer doesn't need much to get value from Git.
A researcher working alone can initiate changes to their own work as they complete tasks. To commit a change, all you need is three commands:

```
> git add figures/poster.svg
> git commit -m "first poster design"
[master b1c0ac0] first poster design
 1 file changed, 137 insertions(+), 162 deletions(-)
> git push
```

You can now work, by yourself, on multiple computers!
To update your repository on another machine, simply type:

```
> git pull
remote: Enumerating objects: 7, done.
 figures/poster.svg | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Congratulations! You now have a complete, recoverable history of every change made to a codebase.



#2 Small team workflow

When working with more than one person, additional version control organization is required to prevent overwrite conflicts. This is when understanding *branches* in git becomes necessary. Branches are seperate timelines of changes to a repository based on a previous commit.

In this example, there are two types of branches. The **main** branch, which records your code in a usable state, and the **feature** branches, which records *temporary* code written by collaborators.

For a group between 2-5 people, we suggest a simple branch-per-feature approach, which mimics the normal academic development cycle. When a collaborator is assigned a feature to implement (e.g. making a tidal forces module), they create a new branch to develop that feature.

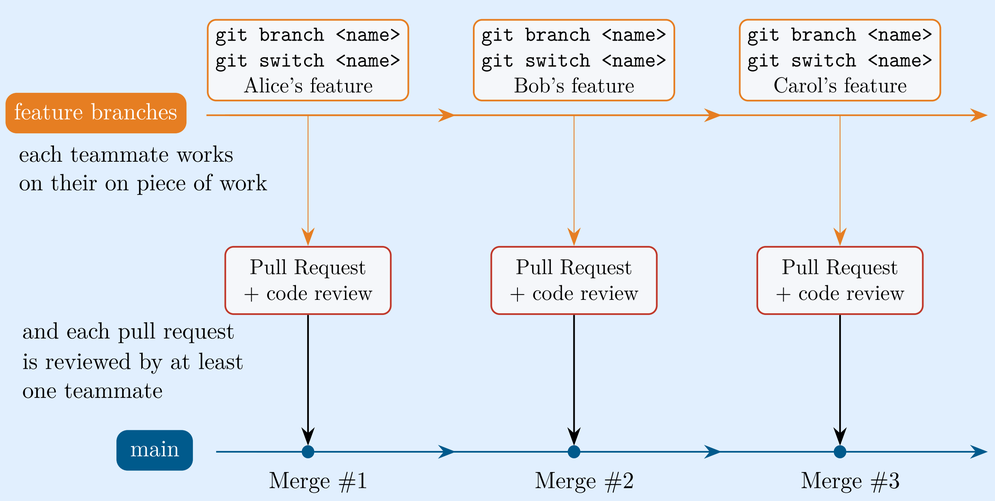
```
> git branch Alice-Tidal-Forces
> git switch Alice-Tidal-Forces
Switched to branch 'Alice-Tidal-Forces'
```

```
> touch tidal_module.py
> git add *
> git commit -m "Alice: Adding tidal module"
[Alice-Tidal-Forces 649f0c1] Alice: Adding tidal module
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 tidal_module.py
```

After a new feature is implemented, a **pull request** is required to merge two branches, which allows you to see how your changes modify the main branch. To initiate this process, push your new branch into the remote one.

```
> git push -u origin Alice-Tidal-Forces
 To https://github.com/AtWillB/EPSC2026-GIT-Workflow
  * [new branch]      Alice-Tidal-Forces -> Alice-Tidal-Forces
 branch 'Alice-Tidal-Forces' set up to track 'origin/Alice-Tidal-Forces'
```

We recommend researchers unfamilliar with git to initiate a **pull request** in either **GitHub or GitLab.**



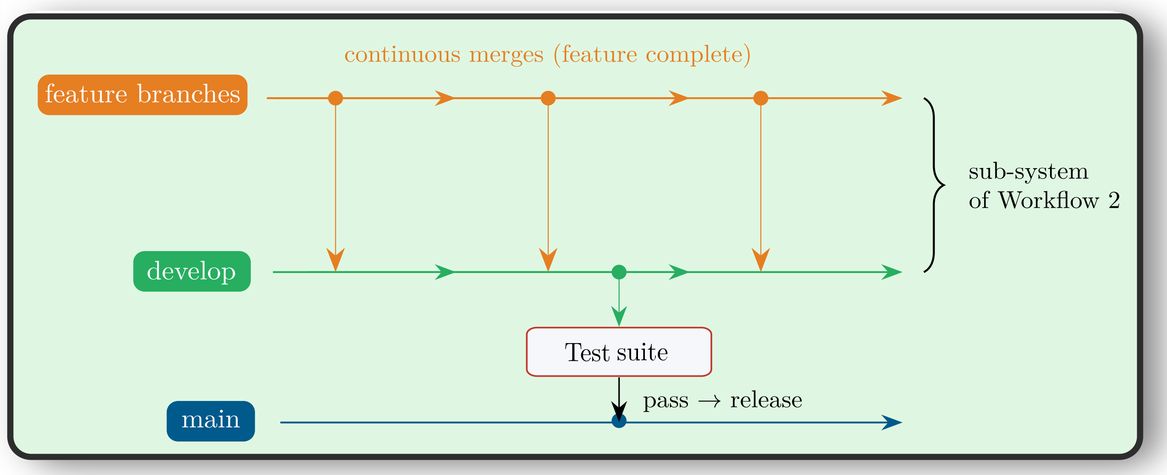
#3 Large team workflow

Workflow 3 is designed for 5+ people. For a group this large,we suggest one additional layer of organization to prevent conflicts. An intermediary **develop** branch is placed between the temporary **feature branch** and the more permanent **main** branch. Based on our previous example, your environment may look something like this:

```
> git branch
* Alice-Tidal-Forces
  develop
  main
```

When a new feature is being *developed,* a new branch is made off of **develop**. After a group of new features have been made to **develop**, it is merged into **main**. Think of software with version numbers. A version increase from **15.0** to **16.0** is equivalent to merging **develop** into **main**, while a version increase from **15.5** to **15.6** is like merging **Alice-Tidal-Forces** into **develop**.

This structure allows for groups of new features (e.g. heating module), which are made of smaller feature additions (e.g. radioactive, primordial) to be implemented with careful purpose, and facilitates intelligent software design.



Conclusion

Version control philosophy is heavily debated (especially in industry [4]), and any team, academic or otherwise, will make team-specific adjustments that fit their culture and project structure. Structured version control does not solve everything, but it is a concrete, teachable, and immediately applicable step forward that enables researchers to feel confident in their software development [5, 6].

References

[1] Git Developement Team, 2026.
[2] Blischak et al., 2016.
[3] Haaranen et al., 2015. [4] Driessen, 2010.
[5] Carver et al., 2022. [6] Ram, 2013