**P3 Development Workflow Protocol**

Last updated: 2025-01-22

The purpose of this document is to describe workflow for the development and maintenance of the code for the P3 microphysics scheme and supporting materials.

**Git Workflow**

Repositories

All P3 code and supporting materials are hosted on two repositories on GitHub:

github.com://P3-microphysics/P3-microphysics (public)

github.com://jason-milbrandt/P3-microphysics (private)

The public repository contains the latest “official” release of the P3 package. There is only a single branch (‘main’) with commits tagged by the package version number.

The private repository is for organizing and archiving all development. It is accessible by the core development team and collaborators. The ‘main’ branch on this repo is identical that on the public repo. As soon as a commit is merged to ‘main’ on the development repo, it is pushed to the public repo. This redundancy serves simply to keep all development private.

Branch Organization

The P3 Git workflow protocol is based on the “Gitflow” workflow, with some slight differences. The Gitflow workflow is illustrated in Fig. 1. The short YouTube video indicated in the caption provides a nice visual overview of how this works. The ‘development’ branch is similar to that in Gitflow; ‘dev-x’ branches are basically like ‘feature’ branches in Gitflow. In the P3 workflow hot-fixes are not done on ‘main’; rather, all bug-fixes are done as ‘bugfix-x’ branches off of ‘development’ and eventually merged back to ‘main’ (and active ‘dev-x’ branches) as described below. There can also be sub-branches off of development sub-branches (i.e. ‘dev-x’) (described below).

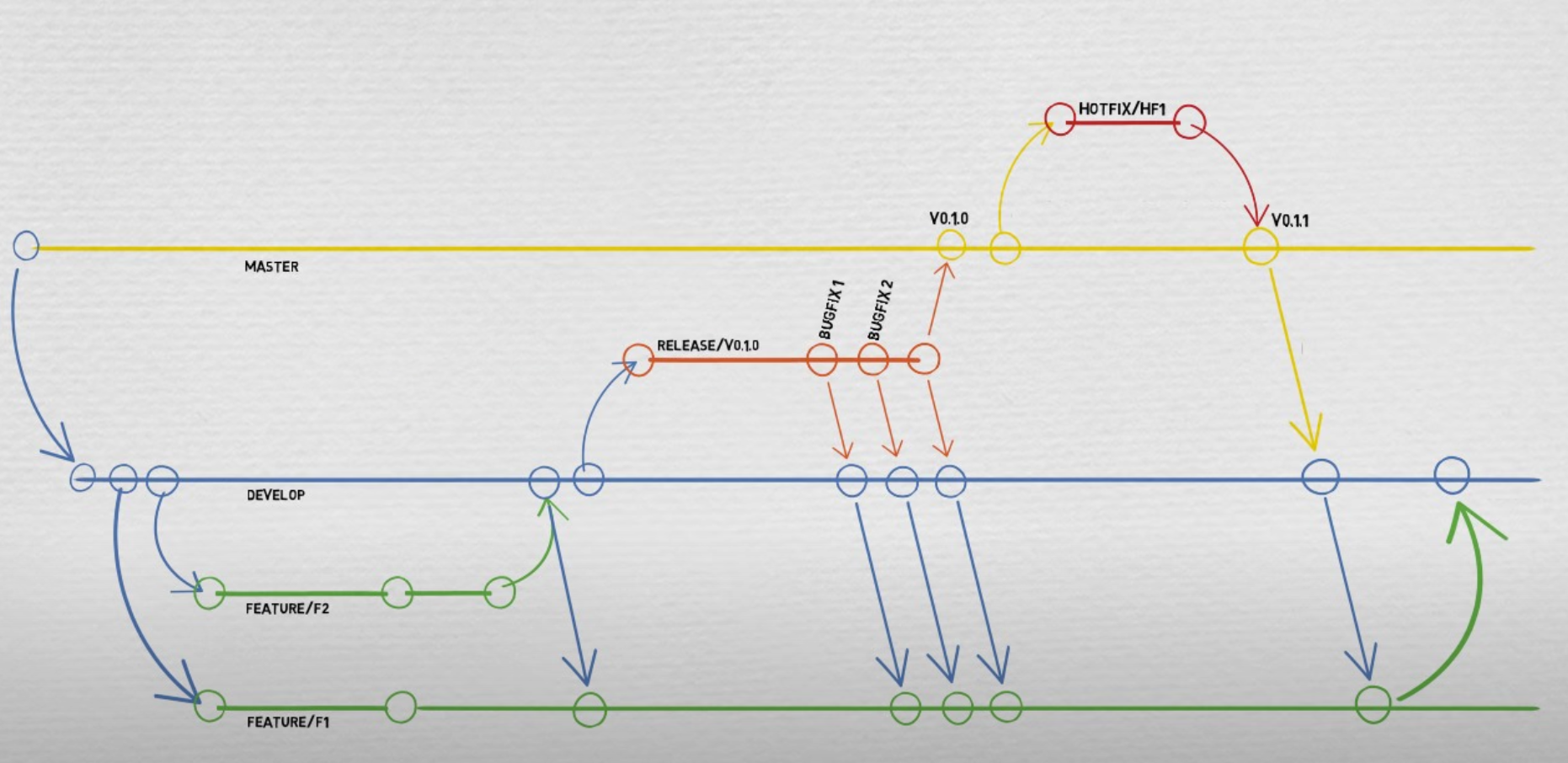


Fig. 1 Schematic of the “Gitflow” workflow. (Source: <https://www.youtube.com/watch?v=1SXpE08hvGs>)

P3 Workflow

The P3 workflow is organized as follows:

* At the highest level, “official” release versions of the P3 package are merged to the ‘main’ branch (from ‘development’) once they have been thoroughly tested according to the testing protocols (see **P3 Code** below). The tags from ‘development’ are also copied. All commits and tags on ‘main’ are then pushed to the public repo (‘main’).
* The ‘development’ branch is immediately below ‘main’. No development or bugfixes are done directly on this branch. Rather, they are done in branches off of ‘release’ and then merged back following sufficient testing. At the time of merging to ‘development’, the package version number is updated and the commit is tagged with the version number. The exception to this flow is if changes are made to some component of the repo that does not, in any way, affect the 3 pieces of P3 code or lookup tables but for which an update to the repository is made (e.g. to the model interface files, the kinematic model, updated documentation, etc.). In this case, the version tag gets relocated to the latest commit on ‘development’.
* ‘dev-x’ branches are branched directly from ‘development’ and are for a major or minor development (and are like ‘feature’ branches in the Gitflow workflow). Upon completion and sufficient testing, the package version is then updated (in the code comments) with one final commit and then merged to ‘development’and then tagged. If a different developer does work on a given ‘dev-x’ branch, he/she creates a sub-branch ‘dev-x-y’ and works on that.
* ‘bugfix-x’ branches are branched directly from ‘development’ are work similarly to ‘dev-x’ branches except that their purpose is to track changes for a specific bug fix before testing and merging back to ‘development’.
* ‘cleanup-x’ branches are branched directly from ‘development’. Their purpose is for general code clean-up, refactoring, or other modifications to the repository (e.g. changes to kin1D).
* There are also ‘proj-x’ branches for specific objectives or groups. These “project” branches are not necessarily intended for development that is to be merged back to the ‘development’ branch; rather, some exist for bookkeeping/archiving purposes such that the repository contains a complete history of scheme versions. However, collaborators (external to the core team) that “own” a ‘proj-x’ branch may make merge requests; the core team will decide if the proposed changes are to be merged to ‘development’ and made part of the “official” P3 code.

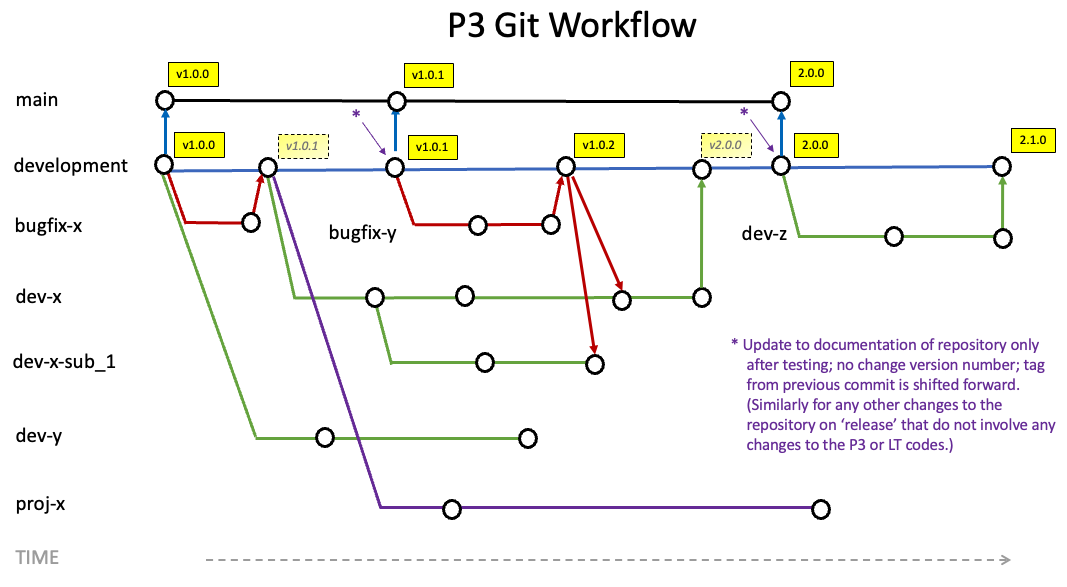


Fig. 2 Schematic of the P3 workflow on the private (development) repository. The circles denote commits. The public P3 repository has the identical ‘main’ branch but no others.

*Branch nomenclature*:

For any given ‘dev-x’, ‘dev-x-y’, ‘bugfix-x’, or ‘project-x’ branch, meaningful (but preferably short) names for ‘x’ and ‘y’ and given by the person who created the branch (using underscores for spaces). Examples include ‘dev-aerosols’, ‘dev-aerosols-ice\_nucleation’, ‘bugfix-initialization\_LT1’, ‘proj-gem\_520’.

All commits on ‘development’ and ‘main’ are tagged with the version number of the P3 package (see below).

Only one developer should ever commit changes to a given branch. Upon starting a new branch off of given commit from ‘development’ (i.e. starting with a closed version), usually the most recent, the developer names the branch following the nomenclature described above – e.g. ‘dev-aerosols’. The developer proceeds to make and commit code changes.

At the first commit, the developer should change the version number of the piece(s) of code he/she is working on in the comment section at the top of the code. For example, if ‘dev-aerosols’ was branched off of ‘release [v5.3.2]’ and only ‘microphy\_p3.f90’ is being modified, then at the first commit the commented version number should be modified to ‘! Version: 5.3.2 + aerosols’. (The final version number will be determined upon being merged back to ‘development’.

Similarly, if another developer makes changes to someone else’s development branch, he/she should follow the same procedure. For example, if someone wants to make changes off of ‘dev-aerosols’, he/she should create a branch such as ‘dev-aerosols-ice\_nucleation’ and change the commented version number to ‘! Version: 5.3.2 + aerosols + ice-nucleation’. Once this sub-branch is merged back to ‘dev-aerosols’, the commented version number returns to ‘! Version: 5.3.2 + aerosols’.

Only the creator of any given branch should ever commit changes to that branch. Once a developer of a sub-branch is finished with a set of changes, he/she should make a (verbal) merge request. For example, if Jim is done with ‘dev-aerosols-ice\_nucleation’, he tells Jane (developer of ‘dev-aerosols’) who then checks the code and merges this to her ‘dev-aerosols’ branch. This approach avoids the possibility of two people trying to commit changes starting from the same hash, thereby creating conflicts and confusion. Before making the merge request, the developer of the sub-branch should first ensure that there are no merge conflicts. This is done by first updating the parent branch on his/her local repository (by fetching or pulling from the private GitHub repository) and then either rebasing the sub-branch to start from the last commit on the parent branch or merging the updated parent branch to the sub-branch and correcting all merge conflicts.

Similarly, only the repository lead merges code to ‘development’ or ‘main’. Once a development or bugfix is complete, the developer of that branch informs the lead that the code is ready to be merged. After sufficient code review and possible additional modifications, the repo lead will then make one final commit to that feature or bugfix branch with (only) a change to the commented version number (e.g. change ‘! Version 5.3.2 + aerosols’ to ‘! Version: 6.0.0’, merge to ‘development’, and add a tag ‘v6.0.0’.

Commits should be done frequently and pushed to the private GitHub repository frequently (after each coding session). The commit messages should use the following format, with each line no longer than 80 characters (for readability using ‘git log’):

Short, one-line description of the changes made in the commit.

*blank line*

- Elaboration detail 1

- Elaboration detail 2

Description of test(s) done and the impact of the commit.

Ideally, every commit should be preceded by at least one test in kin1D (minimum) and noted in the commit message.

*Bugfixes for older versions*:

Occasionally it may be necessary to add a bug fix to an older version on ‘development’ even though the ‘development’ branch has advanced. For example, there may be a recent version on ‘development’ but it is necessary to incorporate a bugfix to the older version in order to maintain support for it since a particular user/system has not yet upgraded to the latest version and has a need to continue with the older one.

In this case, the hash (or tag) of the old version should be checked out, the bugfix implemented, and the branch be named ‘bugfix-vX.Y.Z’. This branch will simply remain, and not be merged back to ‘development’. The bugfixes themselves will, if relevant, can also be ported to the latest version on ‘development’ (and then to ‘main’) through cherry-pick.

*Documenting tests*:

Summary documents for tests (including figures, summaries, etc. that are beyond the scope of commit messages) are to be added in the directory ‘./test-summaries’. There is no template for the documents, but the idea is to summarize the impacts on simulations with code versions between commits or, for more thorough documentation, between two closed versions on the ‘release’ branch. The file itself can be as short/simple or as details as is appropriate. For example, for a minor bug-fix, the test document could contain just two screenshots of a given field, illustrating the impacts; for a large upgrade, it the document could contain several images of scores, field comparisons, etc. between sets of (GEM) runs comparing two different P3 versions. In principle, every commit on the ‘development’ branch that includes a change of P3 version (and these should always be merges from ‘dev-x’ or ‘bugfix-x’ branches) should have a test summary file, along with a short description of the test results in the commit message. With this approach to documenting tests, it is possible to readily go back and identify points of which simulation results with P3 changed (and why).

The file names should conform to the pattern: ‘*YYYYMMDD-[VERSION/TAG](or [BRANCH])-short\_description.pdf*’. This way, the relevant commit or set of commits can be easily deduced.

e.g. ./test-summaries/20231107-v5.3.2-gem520b3-v3\_vs\_v5.pdf

./test-summaries/20240506-v5.3.4-lambdalimiterfix.pdf

./test-summaries/20240915-dev-aerosols-new\_nucleation.pdf

**P3 Code**

This section describes the protocol/guidelines for maintaining the 3 pieces of code for the P3 scheme and the 3 lookup tables.

Version Numbering:

The version number of the P3 package follows the convention ‘vX.Y.Z’, where X, Y, and Z denote major changes/developments, minor changes, and bug fixes (and or minor re-factoring).

Testing Protocol:

*TBA*

Coding Conventional:

*TBA*

**Git Cheat-Sheet**

Setting up repositories

Comparing files on different commits

git diff HEAD~2 HEAD -- file1 # diffs file 1 on HEAD (current commit) and 2 commits back

git diff HASH1 HASH2 -- file1 # diffs file 1 between that on commit HASH1 and HASH2

git difftool HASH1 HASH2 -- file1 # as above, but using set difftool (e.g. xxdiff)

Branch re-naming:

git branch -m new-name # if you are on the branch you want to rename

git branch -m old-name new-name # if you are on a different branch

git push origin :old-name new-name # “rename” on remote (actually, deletes old and pushes new

git push origin -u new-name # resets the upstream branch for the new-name local branch

git branch --delete mybranch # deletes mybranch off of local repository

git push origin --delete mybranch # deletes mybranch off of remote repository

Cherry-picking:

git log --oneline source-branch # lists the hashes of the commits you want to cherry-pick from

git checkout destination-branch # switch to the branch onto which you want to add the changes

git cherry-pick [HASH1] [HASH2] # successively merges the specific commits to the destination branch in the order in which they are listed (which may be different from the order in the source branch)

Miscellaneous

git fetch --prune # removes remote branches from local cache that are visible from ‘git branch -a’ but which have been removed remote