Repository and Development

Documentation about the ACME source code repository and how it should be used for development.

Development Big Picture

The Big Picture of E3SM Model Development

All E3SM developers should be familiar with the basics of development with git including cloning the E3SM repo, making a branch, committing changes and submitting a Pull Request on Github. Also writing good commit messages and pull request descriptions. For more info see, the Git Tutorial, Development Getting Started Guide, Development Reference and Commit message template.

Group Leads and Epic leads coordinate and schedule development relevant to their group and/or epic. They should have developers assigned to work on features in the order necessary to meet deliverables in the Roadmaps (the leads might also be the developers). Epic leads should make sure developers are finishing their work and getting it merged to the E3SM "master" branch.

Integrators: every group has 2 or 3 Integrators who take completed feature branches and, if they meet the requirements, merge them to master. See Integrator Guide for more info.

The Coupled Model Group established the timeline for major new features (especially from two or more groups) that are needed for the model, relevant testing criteria, and when the master should be tagged (using the Branch, Tag, and Version name conventions) all timed to meet E3SM deliverables as defined in the Roadmaps. Integrators should be aware of the Roadmap for their component.

A branch for every feature: Group and task leads and developers should define units of code development that are big enough to be relevant to others, but simple enough that it can be called one feature (or perhaps a couple of highly-interdependent features) and make a branch for each unit of work. These branches are called "feature branches" in our documentation. The developer should make commits that are minimal atomic units of work to complete the feature that the branch seeks to implement. This separation makes it easier to review and debug later. New development should generally start as a branch from "master". Avoid creating large branches that contain many features and touch dozens of files. If a large branch is needed for some major development, the Software Engineering hub should be involved in planning.

> **Bug fix branches:** When a bug is found on master, the branch for developing a fix should start from the commit that introduced the bug. This is so that any feature branch which also contains the bug can easily merge in the bug fix without having to merge in many other changes on that may have occurred while the feature branch was being developed.

Focus on your branch: While working on their branch, developers should focus on the work necessary to complete the branch and not worry about what else is going on. In particular, there is almost never a need in git to "merge from master" or "stay current" with development going on in master or other branches to make the eventual merge easier. If a developer thinks a merge from master or another branch is necessary to complete their work, they should check with an Integrator first.

Developing with externals: some code in E3SM has its primary development in another repo. See Externals in the E3SM code

"master" is always stable: Our code development workflow ensure that master is always stable and can be used to start new development.

Finishing development by Testing and Merging: When the feature branch is done, *the developer makes sure the acme_developer test suite passes* (or the failures are expected). When the test results are in order, the developer issues a "pull request (PR)", and designates an Integrator. The Integrator conducts the code review, merges the feature branch into the "next" integration branch and tests the feature within the latest version of E3SM using the acme_integration suite. The integrator and developer may need to work together to resolve any issues that arise from the code review or testing in the "next" branch. The integrator will then merge the branch into master. Once merged to master, the branch is complete and the developer can move on to other topics. See the Development Reference for more information. In summary:

- One feature per git branch (ideally)
- One pull request (PR) per branch (absolutely)
- Nothing gets added to "next" or "master" except by merging a branch through a PR (absolutely)

Coding with performance in mind: Fortran Performance Best Practices

Development Getting Started Guide

How to work with the E3SM code base.

1. Read the Development Big Picture page. Note in particular the recommended philosophy of working with branches is different than what you may be used to.
2. **github account for source code**
   a. Sign up for a free account at http://github.com
   b. Add your github user name to the "About Me" section in your Confluence profile.
   c. Email James Foucar (jgfouca@sandia.gov) and ask to be added to the E3SM private organization on github. Include your Confluence account name so he can verify you're a member of E3SM. **OR** "fork" the model to your own github account.
3. Optional: If you are also developing input data files see E3SM Input Data Servers

START: Steps 1-3 are done once per person

---

4. Log on to the platform on which you want to do E3SM development. See Computational Resources for a description of available machines. Our supported platforms will have all the necessary software available. NOTE: Make sure the python executable is in your path. This might be automatic or might require adding a module depending on the platform.
5. Install an ssh key from that platform to your github account. See these instructions and note you can skip step 2 if you already have an ssh key on your machine.
   (**NOTE:** you can skip the next four steps below by running this script https://gist.github.com/douglasjacobsen/3ea4421c0d8ae8c460b0)
6. Set up your git environment with these commands:
   a. git config --global user.name "First Last"
   b. git config --global user.email "user@lab.gov" *(NOTE: use the email you registered on github.com)*
   c. git config --global push.default nothing
   d. git config --global core.editor ${EDITOR} *(vim, emacs, etc.)*
   e. git config --global color.ui true
   f. git config --global core.autocrlf false
   g. git config --global core.safecrlf false
   h. git config --global pull.ff only
7. Clone the repository to your development machine. This will create a directory called "E3SM" with the code.
   git clone git@github.com:E3SM-Project/E3SM.git (Or clone your fork to you development machine).
8. Get the source code for submodules needed by E3SM. change directory to E3SM and do:
   git submodule update --init
9. Clone the repository hooks: while still in the E3SM directory, do:
   rm -rf .git/hooks; git clone git@github.com:E3SM-Project/E3SM-Hooks.git .git/hooks
10. Setup the commit message template: In directory "E3SM", do:
    git config commit.template ${PWD}/.git/hooks/commit.template
11. (Optional but recommended for bash users)
    a. Download and source git-prompt to add branch information to your bash prompt.
    b. Download and source git-completion.bash to enable tab completion of git commands.
12. (Optional when developing with ocean or coupled cases) This is done automatically when building, but is useful to do when using ssh passphrases. Download cvmix and BGC prior to building with the following commands:
    cd components/mpas-source/src/core_ocean
    ./get_cvmix.sh
    ./get_BGC.sh
    If you don't and you use passphrases, you'll have to babysit the build and enter your passphrase when prompted. Note these commands should probably be run whenever MPAS-O is updated in case of new cvmix or BGC versions

    You are ready to start developing!

SETUP: Steps 4-12 are done once per platform.

---

13. Make sure there is a JIRA task associated with your code development. This will be referenced in the commit message.
14. By default, you will be on the "master" branch. For new development you may want to either

BASIC DEVELOPMENT: Steps

(1) create your own branch off of "master" or a "maint" branch or (2) work on a branch that someone else has already created.

a. To switch your working tree (local copy) to a pre-existing branch:
    i. look for remote branch names: *git branch -r* or see Existing and Planned Tags and Branches
    ii. Run: *git checkout --track <remote branchname>*
    iii. Ensure your checkout is up to date: *git pull --ff-only*
    iv. Update MPAS components and other submodules: *git submodule update --init*

b. To create a new topic branch for your development: **The following command assumes you are on the master branch, change 'master' to the branch name you're using if it's different**
    i. Ensure your clone is up to date: *git pull --ff-only origin master*
        1. If an error is encountered when pulling, talk to an integrator about fixing it
    ii. Create your topic branch for your development: FOLLOW THE E3SM BRANCH NAME CONVENTIONS!
    iii. create the branch in your repository, branching from master: *git branch <branchname> master*
    iv. switch your working tree (local copy) to this branch: git *checkout <branchname>*
    v. Update MPAS Components and other submodules: *git submodule update --init*
        1. *Note:* The URL for submodules may change in the future. If this happens, git does not automatically update the URL of the submodule for you. As long as the URL that your submodule refers to contains the commit being requested for the submodule, this is fine. However, if you get an error saying the commit was not found, you may need to update the URL used by your local submodule to the one updated and stored in .gitmodules. This can be done with the command: *git submodule sync* If this is needed, it is a one-time operation that is only needed when the URL to which a submodule refers to changes.
    vi. To ensure you are working on the correct branch, run "git branch" which will show all your local branches and list a "*" by the branch you are currently on.

13-21 are done for each "chunk" of code development.

15. Before doing any development, create baselines for any tests you will be using to verify your changes, and place these baselines in a directory you will use to test your branch. At a minimum, this should be baselines for the e3sm_developer test suite. This is a good idea even if you are not making answer-changing code modifications, since the baseline files for the master branch can be changed by other merges while you are working. For information on generating baselines, see Testing.
16. Start editing your code and commit changes to your local repository. (e.g., after making changes, do: *git add <files> ; git commit*). See Commit message template for how to make a commit message. We no longer use ChangeLog so commit messages are important.
    a. If you have any issues compiling or running the code, check with the POC for the platform you are using: Configuration Management. The POC is responsible for ensuring E3SM tests can run on supported platforms and will be knowledgeable about system specific quirks and features.
17. Once you have committed at least one change locally, push your branch to the main repository. You may not be done but this will allow others to follow and contribute to your work.
    a. *git push origin <branchname>* ***Anytime you run "git push" you need to be extremely careful that you know exactly what git is going to do (use --dry-run if you're not sure).***
    b. ***Never run git push without specifying a branchname. This ensures you push the correct branch (and not the master branch)***
18. Add tests for your new feature following E3SM unit test framework (NOT YET IMPLEMENTED)
19. Periodically run *"Developers Test Suite"* including your new tests and confirm **All Tests Pass.** Basic instructions for running the Developers Test Suite are in Testing

FINISHING UP: Steps 22-25 are done for each "chunk" of code development.

20. When you are finished with your development, push your branch to github (see step 18)
21. Run the "Developers Test Suite" if it is supported on your platform. See Configuration Management table and Testing instructions. Make sure all tests pass.
22. Go to our github site and make a Pull Request. See Development Reference#Submittingapullrequest for details.
    NOTE: Do NOT merge your own branch. DO NOT PUSH THE MERGE PULL REQUEST button.

Remaining steps will be led by an E3SM Integrator. Your feature is not done until the Integrator moves your changes to the "master" branch. Pay attention to the comments on the PR. The Integrator may ask you for help or to make more changes in response to testing or reviews.

Further and more detailed instructions can be found in the Development Reference.

If you're new to git, see the Git Tutorial page

Development Reference

This document provides details on E3SM development conventions and practices.

NOTE:

This document does not discuss the use of forks. If a developer understands and prefers to use a fork, it is recommended that they make use of forks.

Introduction

All E3SM developers should consider themselves stewards of the repository history. Our goal with defining a workflow is to improve the utility of the repository and create a useful history that can provide a tangible benefit to other developers.

As always, it's a good idea to understand what you're doing before you do it. This document should not take the place of understanding, but can be used to learn and remember.

How to use this document

This document expands on portions of the Development Getting Started Guide providing more detail on E3SM development workflow. You should read the Quick Guide first, and use this document as needed.

Specific pieces of information will be colored as follows:

important -- Items colored in red mean they are important, and should not be ignored.

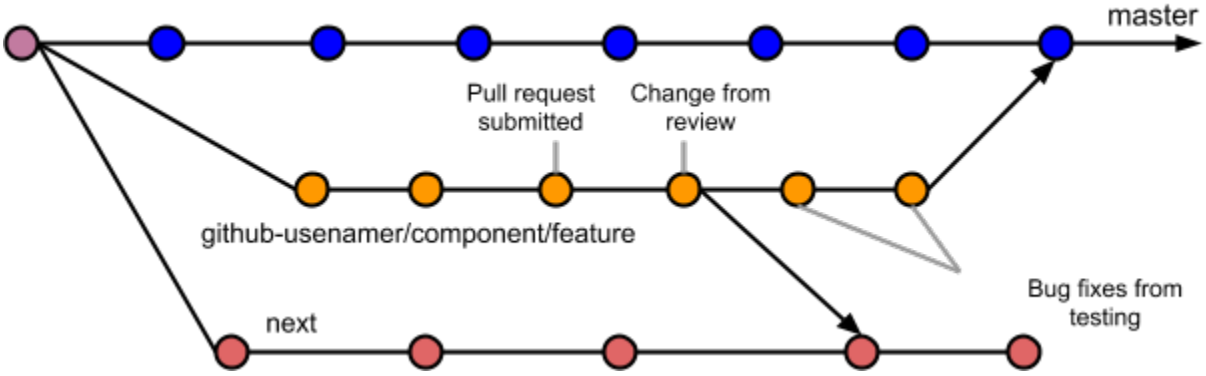
one time -- Items colored in green are commands to be issued once per machine.

repo once -- Items colored in orange are commands to be issued once per local repository.

common -- Items colored in bold black are commands that will be commonly used.

Project Life Cycle

Before getting into the git commands related to our workflow, here is an overview of the life cycle of a feature. This can be used to get a big picture that will be broken up in the following steps.



In the above image, red dots represent merge commits of features into an integration branch. Orange dots represent commits a developer makes as part of a feature branch. Blue commits represent merge commits incorporating a completed feature into the master branch. Purple commits represent tagged versions of the full model.

NOTE:

Within this diagram, blue dots have associated feature branches that are omitted for the sake of readability. However, each of them followed the same development cycle as the feature that is being focused on in the diagram. Also, red commits are made to next. Both next and master should only be modified by integrators or gatekeepers.

Basic Development

Creating a new branch: new feature

New development should be carried out on a branch. New developments include the addition of a new feature (**github-username/component/feature**) or fixing a bug (**github-username/component/bug-fix**).

Your branch will typically start from master. It may start from another developers branch or a maintenance branch. Never start a branch from "next".

When creating a branch, please name it based on guidelines contained in Branch, Tag, and Version name conventions. You should also create a set of baselines for tests you plan to use to verify your code changes, and store these baselines in a location specific to your new branch. This procedure is described in Testing.

NOTE:

As seen above, E3SM utilizes a naming convention for branches. Branches have compound names separated by a "/" to describe what large scale part of E3SM the branch changes will modify. The / does not denote a directory, it just is used in naming the branch. This lets other developers know what these branches are developing and what parts of the model they should be modifying. This practice will be referred to as "namespacing a branch"

When creating a branch, multiple levels of namespacing are allowed. In E3SM, a branch should always be namespaced first by github user in charge of the branch. Next comes the main component the feature is being developed for. Finally, a description of the new development. For example, a branch that is implementing a new parameterization within EAM would be named:

joeuser/eam/new-parameterization

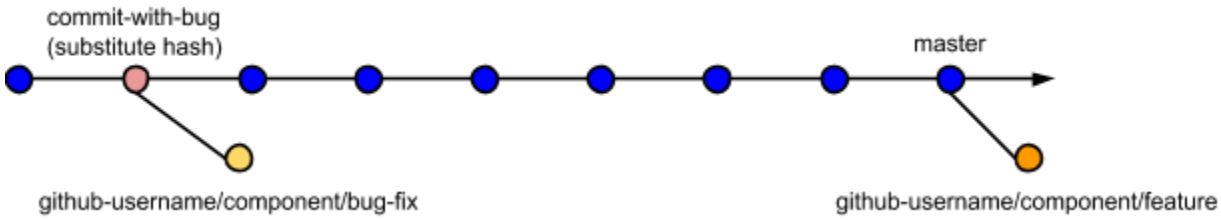
And a branch that is modifying the HOMME dynamics would be:

janeuser/homme/new-se-feature

Branch names should be as long as needed to clearly convey what the developer is working on, but they shouldn't be overly long and descriptive. The following example is a branch name that is too long:

joeuser/cam/this-is-my-awesome-new-feature-that-does-something-cool-and-we-might-not-use-in-the-future

The following diagram depicts two branches that are created. One to fix a bug (**github-usernam/component/bug-fix**) and another to master a new feature (**github-username/component/feature**).



The following commands can be used to create the feature branches:

git branch github-username/component/feature

The above command assumes you are on the head of the master branch and that is where you want to start the new development.

If you want to start development from a specific tag, include the tag name as an additional argument.

git branch github-username/component/feature v1.0

When creating a new feature branch, best practice suggests you should branch from a tested version of the code. HOWEVER, while the model is in "version 0" mode, always start development from the HEAD of master.

Creating a new branch: bug fix

Typically, a bug fix is applied to the commit that introduced the bug. In that case, the branch creation has an additional argument:

git branch github-username/component/bug-fix commit-with-bug

The last argument is the hash of the commit that first introduced the bug. If that was a tagged version, you can use the tag name as for the feature branch above.

It is ok to fix multiple bugs on a single bug fix branch if it makes sense to do so (e.g. one related set of code changes can fix multiple bugs.)

In order to find the hash of the commit that introduced the bug, a developer can use:

git blame file-with-bug

This will open the file with the bug in a text editor, and annotate each line with the commit that last touched it. This can be traced back to find the commit that introduced the bug.

EXCEPTIONS: You can start a bug-fix branch from the HEAD of master if any of the following are true:

- The bug existed in code imported from CESM such as in the initial version added to the repo.
- The bug was introduced prior to v0.3 (the version where the testing is working)

Creating a new branch: maintenance

The E3SM repository will have one or more active maintenance branches usually to keep adding bug fixes and machine updates to old versions. See Existing and Planned Tags and Branches for the list of branches, their purpose and what kind of changes they will accept.

If the feature, machine update or bug fix you are working on needs to go on a maintenance branch, switch to that branch before making your feature branch. For example:

git checkout maint-1.0

This will put your working directory at the head of the maintenance branch. You can then start a new branch as you do for master. When you

make a pull request, specify the maintenance branch as the "base".

If you need to fix a bug on a maintenance branch and the bug was introduced by a commit on the branch, you can follow the same directions as for a bug-fix branch.

If you are developing something for both a maintenance branch AND master, do the maintenance development first. You can then ask the integrator to merge maint to master. Or you can make 2 pull requests, one for maint and one for master.

If your development depends on something that was added to master AFTER the maintenance branch started, it can not be added to the maintenance branch. Master "contains" the content on the maintenance branches (if they are not too old) and one can merge from maint to master. We can not merge from master to maint.

IMPORTANT: We do not have "next" branches for maintenance branches to test multiple features against each other. To compensate, all development on a maintenance branch should be done sequentially. That is, one developer makes their branch, completes the feature, **tests it** a nd then has it integrated to the branch before the next development starts. This should not be a problem because maintenance branches are not supposed to have heavy development. You should always test feature branches but this is more important for maintenance features since there is no "next" testing and you don't want to break the maintenance branch by integrating a broken feature.

Changing Branches

Creating a new branch will not change the current branch that is being worked on in the current working directory.

Because the git branch command does not modify the current branch in the current working directory, new commits will not be made to the new branch. In order to change the branch that is being worked on, the git checkout command is used. For example:

git checkout github-username/component/feature

will change the current branch to be github-username/component/feature.

NOTE:
Creating a new branch and swapping to it can be done in a single command via:

git checkout -b github-username/component/feature master

Utilizing the repository history

Seeing as all of the E3SM developers are stewards of the repository history, it would be useful for developers to understand how to make use of the history they are maintaining.

The most useful command for making use of the history is the **git log** command. This command has an abundance of optional arguments and this second can be used to get an idea of what you can do with **git log**.

The basic usage of this command gives the equivalent of an **svn log**. Where you get a list of all commits and their commit messages. By default, **git log** only shows commits that are reachable in the history of the HEAD. Optional arguments can be used to look at different (or all) branches.

One extremely useful version of git log gives a command line graph of the history.

git log --oneline --graph --decorate

git log can also only show local branches that match a certain naming convention. i.e.

git log --branches=*pattern*

The --branches option can be replaced with **--remotes** to only show remote tracking branches that match a pattern.

A specific commit has three pieces of information. The first is the commit's tree (i.e. the files and directories contained in the commit), the second is the commit's message (the commit message issued when creating the commit), and the third is a list of the commit's parents. There can be multiple parents for a single commit, which would imply the commit was a merge commit. Git stores the order of the parents, with the first parent always being the commit the merge was initiated from. This is useful to note, because within our workflow first parent commits on master should always be merge commits bringing in features or bug fixes. git log can be used to narrow your view to only see first parent commits as well, via:

git log --first-parent

As you develop a new feature, you might migrate files from one place to another, or even rename the file. In order to have git show you all renames of a file, you can use the following command:

git log --name-only --follow -- path/to/file

Any of these options can be combined to give more flexibility to exploring the history of a git repository and make the history more useful.

Committing new files and changes as you go

While working on a feature or a bug fix, you will be editing source code files. After editing a file, you might want to commit those changes to your local repository in order to checkpoint your work, or track the change you made. By default, git doesn't track any files in the repository unless they are explicitly added to the repository. In order to add a file to the repository, you can use the following command:

git add path/to/file/in/repo

In addition to causing git to track the file, this will also stage all changes in the file. The next commit that is created will contain all changes in the stage by default. In order to commit only the changes in the staging area, the following command can be used:

git commit

NOTE: The git add command will only stage the changes in the file at the time of the git add. If the file is subsequently modified after a git add, and then a git commit is issued, the commit will not contain changes between the git add and the git commit. Use **git status** frequently to see the status of staged and un-staged files.

git commit will open up an editor where a commit message can be written. Please refer to the Commit message template for more instructions on the E3SM commit message guidelines.

An alternative way of committing all changes to any files that are currently tracked by the repository is:

git commit -a

This will again open an editor for a commit message to be entered and the Commit message template should again be followed.

Advanced: Incorporating another branch on to your branch.

When developing a new feature, another developer might have created a feature you require for your work, or another developer might have made large changes to the interfaces you make use of. This section will describe how to incorporate those changes into your branch.

NOTE:
This action can have negative consequences and cause issues in the future, so it is important to know what you're doing prior to doing any of these.

Cherry-pick method

The first option for incorporating changes from another development line into your development history is via a cherry-pick. A cherry-pick will copy individual commits (or a range of commits) from one point in history onto the HEAD of your current development line. It should be used when the number of commits your future development efforts depend on are small (order 1-10). It can be used as follows:

git cherry-pick <commit-ish>

This method is the easiest to use and one of the least likely to create issues in future development, and allows the most flexible review modifications. One downside to this method, however, is that the commits that are cherry-picked will now occur twice in the history.

Merge Method

The second option for incorporating changes into your development history is via a merge. A merge will attempt to merge an arbitrary

number of other commits (or branches) **into** your current branch. It can be used as follows:

git merge [commit-ish1] [commit-is2] …

In this case, you can list how ever many commits you want on this line, and git will attempt to merge them all into the commit you are currently working on (into your working directory).

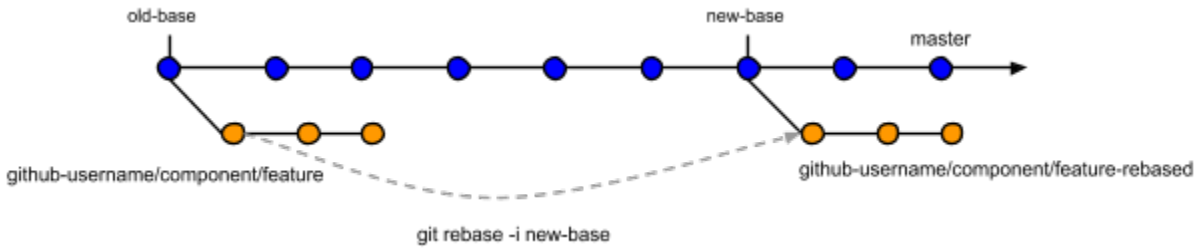
The merge will allow you to enter a commit message. The commit message should be very descriptive. It should explain what you're merging, and why you're merging it.

NOTE:

A merge creates a fixed point in history. The merge commit fixes all history before it, which limits possibilities for cleaning up history during a code review. A merge should be avoided if at all possible, but in some cases it is necessary. If it is necessary, try to clean up all history prior to the merge before beginning the merge.

Rebase Method

The third and final option for incorporating changes into your development history is via a rebase. A rebase allows you to modify commits. This includes operation such as squashing, re-ordering, deleting, etc. When you create a branch, the base of the branch is the commit you branched off of. A rebase allows you to modify what the base of your branch is. The following diagram can be used to visualize a rebase:



When performing a rebase, you specify the new base for your work. Rebase will then replay your work **onto** the new base. For example:

git rebase -i new-base

Will replay the history from the current branch on top of the new-base. The -i flag in this case allows interactive modification of the commits to replay. It should be used to verify what is going to happen after the rebase.

NOTE:

Commits that occur prior to a merge cannot be modified, or rebased. In general, if you previously merged within your branch (i.e. to incorporate an "external feature") or if your branch was previously merged into another branch, you should not rebase anymore, because it will cause conflicts during future work.

Finishing Up

Submitting a pull request (PR)

Once you think development is finished on your branch, you should push the branch to the shared repository (if you haven't already), and submit a **pull request** (PR) for the integration of your feature into master.

A pull request initiates a process to merge your branch into master. The process will be documented with the subsequent code review via discussion and comments on the PR.

1. Commit all your changes and push the branch to the shared repository,
2. go to https://github.com/E3SM-Project/E3SM/branches and find your branch. Click on "New pull request".
3. You should verify **base** and **compare** of your pull request are correct. The "base" is master and "compare" is your branch.
4. Enter a PR Title: Make sure the title is 70 characters or less and explains the PR in a "verb noun" format like "add cool new feature". Do not just use the branch name or what is filled in by default. (If your branch has a single commit, the title of that commit will be the default, but if your branch has multiple commits, the branch name will be the default.) You should edit the title to make sure it is descriptive

enough. **DO NOT** include github issue #'s or JIRA tasks in the title. Hyperlinking does not work from the title.

5. Write a PR Description.
    a. In the "Write" field, provide a descriptive message of what all the commits in the PR will do together. This description will be used as the commit message when the branch is merged so follow the Commit message template guidelines.
    b. If a bug fix: after the description, close any Github issue numbers for the bugs this commit fixes using keywords like "Fixes #123". See https://help.github.com/articles/closing-issues-using-keywords/
    c. After the description and any issue numbers, add a keyword indicating how the PR might change answers.
        i. [BFB] or [non-BFB] or [CC]: Add ONE of these keys to indicate if this commit will affect testing results to roundoff [non-BFB] or climate changing [CC]. Use [BFB] if-and-only-f commit is bit-for-bit and you know all the tests will pass without regenerating baselines.
        ii. [FCC]: Add [FCC] if the commit will change climate if a flag is activated
        iii. [NML]: Add [NML] if the commit introduces changes to the namelist.
    d. The reviewer will review the description as part of the pull request. The description will be used in the commit message used when the PR is merged to next and master. The integrator may work with you to edit the PR description. Do not include Confluence URL's in the PR description.
6. Give PR label(s) ('Label' pull down menu). Add a label for the component this PR involves. Add the "bug fix" label if this is a bug fix. Add the label(s) for BFB, non-BFB, CC, FCC, NML as appropriate.
7. Assign a single Integrator to manage the PR (use the 'Assignee' pull down menu).
8. If you want additional reviewers, add them using the Reviewer pull-down menu. Anyone can be asked to review.
9. When complete, click on "Create pull request". This will start the code review and the process of moving this feature to master.
10. After pull request is created, add the following in the Comments section:
    a. **In the first comment**, provide a link to the Design Document governing this PR. See Code Review Process.
    b. **In subsequent comments**.
        i. Provide information to aid the Integrator in running, testing, and validating the feature (but that is too specific to be included in the general PR description).
        ii. Say how the feature was tested. Example: "e3sm_developer on Titan passed". If a test is expected to fail and required redoing baselines, state that and list the tests that fail.

Your PR is not finished until it has been merged to master by the Integrator.

This document can be used to help with pull request related issues.

Integrator Code Review (Phase 3 of Code Review Process)

Issuing a PR and review by an integrator is done in Phase 3 of ACME's Code Review Process.

Phase 3 code reviews are conducted online on GitHub using comments on the pull request.

Integrator code review steps

1. Check the github entry for the PR and make sure it has a good title and description, correct labels and a comment with a link to the Design Document. A PR can not be merged to next or master unless it has a Design Document with Phase 1 and 2 completed. See Code Review Process.
2. Look at the code changes either on github or using: git log --reverse -p master.. on your checked out copy of the branch.
    a. Does new code hold up to visual inspection for code quality? Look over code changes for glaring mistakes or code style issues (e.g. useful comments, reasonable subroutine lengths, new code in an existing file follows conventions of that file).
    b. Check to see if the description of the code changes in the PR match the actual changes. Make sure nothing unrelated to the PR was committed accidentally.
    c. Although they can't be changed, see if commit messages on the branch follow the Commit message template and let the developer know if they can not. Consider asking the developer to squash commits to clean up the history.
    d. Have tests been added or suggested that exercise this feature?
    e. Does code run on all platforms after integration into next?

If there are any problems, work with the developer to correct them. All correspondence should be done as comments to the PR in github.

An E3SM group may define additional review procedures for code changes affecting their component.

Handling reviewer comments

During a code review there are several steps. The first step is for a reviewer to review your pull request description, along with the code and commits from the pull request. The reviewer is allowed to request the following from a developer:

1. Modify the pull request description
2. Modify commit history (Including removal, reordering, squashing, etc)
3. Modify code. This could be requested for stylistic reasons, or functional reasons.

Modifying the pull request description is as simple as editing the wording of the pull request description via github's website.

Modifying the commit history requires rebasing. In this case, you want to figure out what the base of your current branch is. It is likely that you branched off of master, in which case you can use the merge-base command to determine your base. For example:

git merge-base github-username/component/my-feature origin/master

will tell you the commit that is the base of your branch. If you perform an **interactive rebase onto** the merge base, you can modify the history of your branch without introducing merge conflicts associated with changing your base. For example:

git rebase -i $(git merge-base HEAD origin/master)

will rebase your branch using the tip of the E3SM master branch as its base. Finally, modifying code might require introducing new commits, or editing previous commits.

Editing existing commits

Occasionally, you may need to change one of the commits you've made. For example, if you added a file that shouldn't have been added to the repository, you may want to eliminate it from the commit to prevent it from being tracked in the repository's history.

There are several ways to edit an existing commit. In general, the commands you use are case specific, so general commands cannot be provided. Instead, general command usage and guidelines will be provided, but the integrator helping with the pull request should be able to give more information about specific commands.

One way to edit existing commits is to interactively rebase a branch onto it's own base, which is found using the **merge-base** command in the above section. After typing the **rebase** command, a text editor will open with several lines. Each line represents a commit that git will allow modification for. The lines can be broken down into the following information:

action hash subject

The hash and subject provide information about each commit so you know which is which. The listed on each line is an abbreviated hash prior to the rebase. The subject is the first line from the commit message for that commit. On each line, action tells git what it should attempt to do with the specific commit. Available actions depend on your git version, but all of them start with a default value of pick. Pick means git will "cherry-pick" the commit and leave it as is. In this case, we want to change pick to edit on the commit we need to modify. After selecting this, we can save and exit the text editor. Git will then attempt to rebuild the branch with the selected actions.

When git encounters the commit we selected edit for, it will stop and allow modifications to the commit. The edits can be made by amending the HEAD commit (i.e. make changes and `git commit --amend`). After the commit is left in the desired state, the rebase can continue using `git rebase --continue`.

It would be nice to have a worked example here to make things a bit less abstract.

Testing a feature

After a reviewer is satisfied with all parts of a pull request, both the reviewer and the developer will perform testing on the pull request. The developer may be required to point a reviewer to a specific test case or namelist that can help the reviewer test the feature.

In addition to testing the feature, it is the reviewers responsibility to verify the pull request does not introduce issues related to the previous functionality of master. To ensure that the model still passes basic tests in multiple configurations, developers will be asked to run the e3sm_**developer** test suite. This can either be performed on the branch itself, or in an integration branch. The reviewer will inform the developer where testing should take place.

Basic instructions for running the e3sm_developer test suite can be found at Testing. Some additional information on testing procedures can be found at ACME test suites / categories and Testing issues, strategies, and technologies. Results from nightly test suites run on the "next" and "master" branches can be viewed on the ACME CDash website.

Fixing bugs in a new feature

During the testing phase, bugs may be found related to the feature. Bug fixes can be committed inline to the feature branch. This allows a bug fix to show up as an individual commit but remain part of the development history for that feature. Don't fill out a bug report for a feature that has not

made it to master. Just mention the bug in the comments on the PR.

Final merge and conflict resolution

Finally, after testing is complete the reviewer is ready to merge your feature into master. When they are doing this, they might run into unexpected merge conflicts they are unable to resolve.

If they do happen to run into this situation, you may be requested to help with the process. The easiest way to help is to create a new branch at the HEAD of the branch your pull request was submitted from. This branch should have the same name as the other branch with -resolved appended to the end. After the branch is created, you can merge E3SM-Climate/E3SM/master **into** it, and push the resolved version onto the shared repository. This gives the reviewer a version of the code that is merged and resolved, but allows the reviewer to ensure the history maintains the standards.

Never use the github automerge! **DO NOT PRESS THE GREEN BUTTON!**

Merges to master will be done locally by integrators. See Integrator Guide for more info.

Additional Topics

This section will be used to house additional information for people who are interested. It will attempt to clarify some of the comments made in the development reference by expanding a bit more on them. Developers can safely ignore this content, it's only provided for educational purposes.

The list of "nevers":

- **Never routinely merge from master to your feature branch. See below for more info.**
- **Never use "cherry-pick" on "next**"
- **Never do production simulations using "next"**
- **Never commit directly to "master" or "next". Only use merge commits of branches**.
- **Never rebase commits on your branch after it is merged to next.**
- **Never start new development from "next".**

Changing the url for your remote

After the rename of ACME-Climate/ACME to E3SM-Project/E3SM you should change url or "origin" in each clone.

First verify that "origin" is still pointing to ACME-Climate/ACME. It will unless you changed it yourself but you can check by running "git remote -v".

You'll see output like:

```
origin git@github.com:ACME-Climate/ACME.git (fetch)
origin git@github.com:ACME-Climate/ACME.git (push)
```

Change the URL for origin with this command:

```
git remote set-url origin git@github.com:E3SM-Project/E3SM.git
```

```
verify with "git remote -v"

origin git@github.com:E3SM-Project/E3SM.git (fetch)
origin git@github.com:E3SM-Project/E3SM.git (push)

If you have clones of other repositories that were on ACME-Climate, you'll need to update their URL's as
well with similar command:

git remote set-url origin git@github.com:E3SM-Project/<repo name>.git
```

Working with Remote repositories

Git is a distributed content versioning system (DCVS). When you clone a repository, you make a local repository that is essentially a backup of whatever you cloned. When you commit, the commit only occurs in your local repository. In order to allow other people to make use of the commits you make, or to get your commits incorporated into master they need to be “pushed” to a remote.

A remote is a repository that you would like to communicate with, that is not in the local working directory. When you clone a repository, a default remote is created named “origin” that points to whatever you cloned. A remote is really just an alias to the location of another repository you either want to read or write with.

Before communicating with remotes, you might want to add or remove remotes. In order to add and remove remotes the git remote command can be used. The two uses are as follows:

git remote add remote-name protocol:address/to/repo # Creates a remote

git remote remove remote-name # Removes a remote

In order to communicate with remotes, there are three actions. push, pull, and fetch.

The push command can be used to write some commits from your local repository to a remote repository. For example:

git push <remote-name> <local-branch>:<remote-branch>

can be used to write a local branch to a remote branch on the remote pointed to by the name provided. An explicit example would be:

git push E3SM-Project/E3SM feature:github-username/component/feature

In this case, we’re writing the branch feature to the remote origin and changing its name to core/feature.

git push is the command used when writing history to remotes. In order to read history from remotes, the git pull and git fetch commands can be used.

The git fetch command will update the local repository with the most recent history from a specific remote, without modifying any local branches in the repository. It can be used as follows:

git fetch <remote-name>

The git pull command is a combination of two operations. The first is a git fetch in order to update the history in the local repository of the branches that exist on the remote repository. The second is a git merge which merges the changes into the current working directory. **It is only recommended to use this if you know what you’re doing. It can easily cause unexpected problems**. It can be used as follows:

git pull <remote-name> <branch-name>

Avoid Routine Merges From Master

You never need to "merge from master" to make the eventual merge of your feature branch "easier" or to "keep up". Unlike svn, git is smart enough to ignore changes that have occurred in files you aren't working on when you finally merge your branch to master. Also, this practice can cause more problems.

This policy is to make it easier on developers - developers don't have to concern themselves with a hundred other things going on in E3SM and can focus solely on their feature. The integrator will evaluate how this feature works with the latest version of E3SM when they merge the feature

to 'next', before it is merged to 'master'. We also don't want unnecessary merges from upstream since they just clutter and confuse the repository history and provide more opportunity to break things in the branch.

There was a discussion and several links on this topic here. To summarize some of key points:

- "frequent pulling of the \[master] into a development branch will add a certain amount of randomness to that branch; this randomness is not particularly helpful for somebody who is trying to get a feature working. It also increases the chances that another developer who ends up in the middle of the series while running a bisect operation will encounter unrelated bugs."
- A branch has a specific purpose. A topic branch 'add-frotz' would be about adding a new 'frotz' feature and **shouldn't do anything else**. When you merge from master, you declare that all the other unrelated changes done on 'master' in preparation for the next release somehow bring 'add-frotz' closer to the goal of the 'frotz' topic. That is usually *not* true
- Unnecessary merges and similar repository clutter reduces the ability to summarize, audit, notice bugs in review, and find bugs after the fact. Keeping clean history is not difficult. It requires a little bit of discipline on the part of integrators and developers, but it's a small price to pay for the time saved and improved quality/reliability.

There are some cases where merging from master (or better, a tagged version of master) can make sense. **The rule of thumb for any merge is that if you can clearly describe *what* you are merging and *why* that merge is necessary for your branch to be *completed*, then it's fine to merge**. For example, you might need a feature on master (some crucial functionality, not just a build system updates) to complete the feature on your branch. Integrators may ask you to merge from master to help resolve conflicts during a PR integration. But before merging from master, try one of the other ways to get features from other peoples development described in Incorporatinganotherbranchontoyourbranch.

If you want to see how your feature works with the latest version of master, do a test merge with a throwaway branch. Update your local version of master, make a new branch called something like "testmaster" and check it out. Merge your feature branch to this new branch (which is just a copy of master) and run your tests. When you're done, you can delete the "testmaster" branch. Repeat as necessary.

If you want to know how an Integrator would merge a branch with conflicts they can't resolve, see Integrator Guide#Resolvingmergeconflicts

On conflicts: While git is better at handing merges and branches and resulting conflicts, do not expect that you (or the Integrator) will never encounter a merge conflict. Merge conflicts still exist, and have to be dealt with. No version control tool will be able to automatically resolve a conflict when two people change the same section of code. If two or more developers need to work on the same piece of code, or code that touches several routines, the way to avoid conflicts is to serialize the development or find the common part and do that first, then make new branches after those changes are committed.

Git branching model and avoiding conflicts:

One of the main differences between git and svn is how history and branches are stored. In svn branches are virtual directories (i.e. *svn copy ^/trunk branch_name*) and the history is stored as a stack of revisions (providing the monotonically increasing revision number). The problem with this is that a branch isn't actually a branch. It doesn't contain the information about where it was created from or what has been "merged" into it. The same is true when a branch is merged back into trunk; there isn't a clear historical description of what has already been integrated into trunk. (NOTE: svn has put some effort into improving their branching and merging capabilities, but they are still not up-to-par with DVCS tools)

However, in git the history is stored as a Directed Acyclic Graph (DAG), and branches are first-class citizens. The DAG structure allows any commit to determine which other commits it includes changes from (or which commits have been integrated into it). Using this history structure, *git is able to ignore changes from commits that have already been integrated,* reducing the number of merge conflicts that occur when bringing in a large set of changes.

Additional References on git merges

http://stackoverflow.com/a/2692999

http://stackoverflow.com/questions/2475831/merging-hg-git-vs-svn/2477089#2477089

http://stackoverflow.com/questions/2471606/how-and-or-why-is-merging-in-git-better-than-in-svn/2472251#2472251

Integrator Guide

Integrator Mailing List

All integrators should be subscribed to the integration email list: e3sm-integrators@lists.mcs.anl.gov (Contact Robert Jacob to be added). This list has a daily posting as to the status of the 'next' and 'master' branches with regards to scheduling pull requests.

Integrator Roles

The Integrator's role is to take a finished feature branch and integrate it into master, with some care. The developer of the feature branch assists the integrator.

E3SM Integrators (and github username)

Atmosphere: Balwinder Singh (singhbalwinder), Wuyin Lin (wlin7), Aaron Donahue (AaronDonahue, Livermore Computing POC)

HOMME: Mark Taylor (mt5555)

Land: Gautam Bisht (bishtgautam), Junqi Yin (jqyin)

MPAS-O/LI/SI: Jon Wolfe (jonbob), Matt Hoffman (matthewhoffman)

(For MPAS bug reporting, assign bugs as follows: MPAS-O: Mark Petersen, MPAS-Sea ice: Adrian Turner, MPAS-Land Ice: Matt Hoffman, Any other MPAS: Jon Wolfe)

Drv/share/utils/dead: Robert Jacob (rljacob), Jayesh Krishna (jayeshkrishna), Azamat Mametjanov (amametjanov)

CIME: James Foucar (jgfouca), Robert Jacob

Free agents (can take on any integration task as they see necessary): James Foucar (jgfouca), Robert Jacob (rljacob)

Machine file integrators (Machine POCs) not already listed above: Min Xu, Noel Keen, Jason Sarich (Can integrate any machine-specific changes even if they are the author.)

Integrators know...

1. The overall development goals for the component and where the proposed check-in fits within those plans and the E3SM development timeline.
2. How to run the test suite on a platform.
3. How to conduct a code review.
4. The E3SM Development Reference
5. This Integrator Guide.

Integrators can...

1. Merge code to the master, next or maint permanent branches as appropriate. (Only through pull requests)
2. Reset next branches.
3. Create maint branches.
4. Make tags on the master or maint branches following Branch, Tag, and Version name conventions.
5. Verify bugs and, working with group leads, assign developers to fix them.

"Clear to merge"

A branch is clear to merge to next or master if:

1. The test dashboard is green or the daily status email says it is OPEN
2. No other non-BFB branches from any other components have already been merged that day (only important if your branch is non-BFB)

Integrating a Feature Branch.

> **Before Integrating a Feature Branch**
> Configure your local git to never perform fast forward merges (From within your local ACME repo): *git config merge.ff false*
>
> To keep the testing from confusing results, only ONE non-BFB merge to next or master can be done per day.

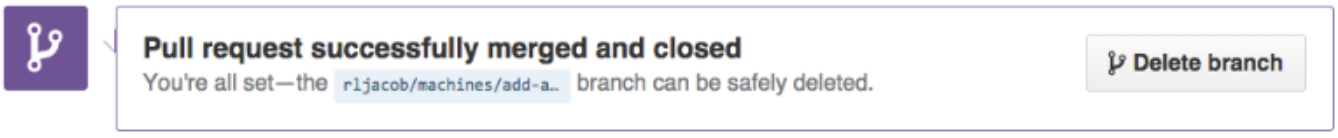
NOTE: Later in this guide will be a section about using forks. If you're integrating a branch from a fork, please refer to both this section and the later section.

You will be the "assignee" on a Pull Request from a feature branch Developer (see last step of Development Getting Started Guide) and oversee the testing and code review. You may re-assign the lead to one of the other Integrators or ask for additional help. There should only be **one** assi gnee.

1. Make sure the PR title and description follows the standard described in Development Reference#Submittingapullrequest(PR)
    a. EDIT the PR title and description in github as needed or ask the developer to fix it. Work with developer to craft the PR description.
    b. If the developer has not done so, label the pull request with appropriate component labels on GitHub (i.e. scripts, machinefiles, land, atmosphere, etc...).
    c. If the branch is fixing a bug, make sure the issue # for the bug is mentioned in the PR description as "Fixes #NN" and the "bug fix" label has also been applied to the PR
    d. If the developer did not state what testing was run in the PR, ask them in the comments. They should at minimum run the e3sm_developer test suite on their branch OR run the 2 test cases used for porting if e3sm_developer is not available AND add new tests for new features if appropriate. You can run the developer test on their branch for them if you wish.
    e. Make sure the BFB (non-BFB, CC) status of the PR is noted in the description and the same github label is applied.
2. If you are satisfied the branch has been tested, conduct Phase 3 of the code review. See Development Reference#IntegratorCodeReview(Phase3of) Your group may have additional requirements for the code to pass review. You may ask specific people to help review by adding them to the PR with the "Reviewers" pull down menu.
3. Merge feature branch into "next" branch: (If the target for the branch is maintenance branch, skip to "Maintenance Branch Integration" below)
    a. Start with your clone of E3SM on a machine you are comfortable developing on.
    b. Checkout the branch you are going to merge.
        i. git checkout author/branch-name (note that you don't need to include "origin" in the branch-name. Git will automatically set up a tracking branch.)
    c. Look at the code changes either on github or using: git log --reverse -p master..
    d. Assuming tests pass, do "git checkout next"
    e. Update your version of origin/next with "git fetch origin"
    f. Update your local next with "git reset --hard origin/next"
    g. Do the merge with: "git merge --no-ff author/branch-name".
    h. If there were no conflicts, edit the commit message to follow the Commit message template.
    i. If there were conflicts, try to resolve them: **If you are unsure how to proceed, ask for help!**
        i. List the problem files with 'git status', edit to fix, commit the changes with "git commit -a" which should complete the merge.
        ii. If you can't fix the problems yourself, try working with the developer as outlined in Resolvingmergeconflicts
    j. Verify the merge was performed correctly: DAG is as expected, compare the new merge to what was on next before to make sure you are creating the history you think you are, etc... Options include:
        i. *git log --graph --oneline --decorate next (add "--no-color" for a dumb terminal)*
        ii. *git log --graph --oneline --decorate next origin/next*
        iii. *git diff origin/next..next*
    k. Run the e3sm_developer test suite on your local merged next before pushing, to verify the merge was done correctly.
    l. If you are Cleartomerge, push the changes to the main repo with "git push origin next"

4. Wait for automated testing harness to run the Integration Test Suite (nightly) on next and confirm that all tests still pass. Results are on http://my.cdash.org/index.php?project=ACME_Climate.
    a. If the integration tests **DO NOT** pass, determine if the fails/diffs are due to expected baseline differences OR unexpected differences and/or test failures. Information about the test suites is located on the pages Using the ACME CDash Dashboard and Interpreting test results.
        i. IF ALL DIFFS ARE EXPECTED (because answers changed or you added/changed a test)
            1. proceed to **step 5 and merge to master.**
            2. **On the same day you merge to master**, file a request to bless the tests at https://acme-climate.atlassian.net/servicedesk/customer/portal/2.
                a. You must merge to master the same day that the next-testing diffs were blessed. This is because we now use the same baselines for next and master. If you don't merge to master on the same day the baselines are updated, master will report a fail for tests run that night because its using the old code with the new baseline
                b. *merge and then request a bless* in that order because it removes the possibility of a bless happening without a merge and for namelist blesses it's helpful to have the PR on master.
        ii. For unexpected diffs or test fails.
            1. If a bug is found quickly, fix the bug, commit the fix to the feature-branch, re-merge the branch to next and repeat step 4
            2. If you can not fix the problem quickly, revert the merge commit and work on a fix. You can then start again at step 3. **NOTE ABOUT REVERTING**: Care needs to be taken with any branch that has had a merge reverted on next (or any integration branch). In a subsequent merge of the same branch (e.g., you merge to next, find a problem, revert the merge to next, add new commits to the branch to fix the issue, merge to next again), git will exclude from the new merge any commits it already sees in the history of next (even though they were subsequently reverted), which can lead to a very confusing 'partial' merge of the updated branch. There are two possible solutions to deal with this. 1. Revert the revert on next before re-merging the branch to 'reactivate' the reverted commits. See https://git-scm.com/blog/2010/03/02/undoing-merges.html, "Reverting the revert" for details. 2. rebase the branch so *all* the hashes change. This prevents git from matching the commits on the branch to the otherwise identical commits that were previously merged. (An interactive rebase where you simply change the commit message by a character in the first commit would also satisfy this criterion.)
            3. If the branch is fundamentally flawed, revert the merge commit to next, close the PR (on github) and work with the developer to decide if the branch should be deleted.
    b. If tests **DO** pass, proceed to step 5.
    c. Testing can only be skipped for a "hot fix" or for code outside the component models IF the code can be verified by inspection to be correct and have no side effects. Add "Skipping Integration Testing" as a comment in the PR on github.
5. Merge the pull request to master.
    a. checkout master with "git checkout master"
    b. Update your version of origin/master with "git fetch origin"
    c. Update your local master with "git reset --hard origin/master"
    d. Do the merge with "git merge --no-ff author/branch-name"
    e. If there were no conflicts, edit the commit message to follow the Commit message template.
    f. If there were conflicts, try to resolve them: **If you are unsure how to proceed, ask for help!**
        i. List the problem files with 'git status', edit to fix, commit the changes with "git commit -a" which should complete the merge.
        ii. If you can't fix the problems yourself, try working with the developer as outlined in Resolvingmergeconflicts
    g. Verify the merge was performed correctly (in terms of the DAG): *git log --graph --oneline --decorate master* (or *'git log --graph --oneline --decorate master --no-color')*
    h. If you are Cleartomerge, push the changes to the main repo with "git push origin master"
        i. **You may use the github automerge button (the green button) only for the merge to master.** Edit the the title and commit message to match the Commit message template. The default title does not follow ACME conventions.

6. If the PR changed answers, either climate changing or roundoff, record details of the PR in the table on Answer-changing commits.
7. Delete the feature branch after the merge has been completed. This does not have to be done immediately. You **can** use the "Delete branch" button on github.



Maintenance Branch Integration:

If the target of the feature branch is a maintenance branch (the "base" is set to a maintenance branch) you should not merge the branch to next. Instead, make sure the developer has tested the branch and integrate it directly to the indicated maintenance branch.

NOTE: Integrators should not bring their own development branches to master. Designate one of the other integrators for your branches. An exception is made for machine files which must be tested on a specific platform integrators may not have access too.

NOTE: Integrators should never force push to master or next, unless they are rewinding next purposefully and approval has been given to do so. Force pushing to either branch requires project wide emails notifying people what what is going on, and how to ensure they don't run into any issues as a side effect.

Post-cime merging

Moved to Post-cime merging for Integrators

Processing a bug

If you've been assigned to a bug, follow these steps:

1. If the description is not complete, work with the bug reporter, through issue comments, to understand/reproduce bug.
2. Make sure the issue has the "bug" label and at least one component label (atmosphere, land, ocean, etc.).
3. Add ONE of the following additional labels: duplicate, won't fix, invalid, confirmed
    a. duplicate: If the bug is a duplicate of another bug, mention the issue number and close the issue.
    b. won't fix: If the bug is really a user error or not a bug in the ACME source.
    c. invalid: For issues that shouldn't be in our issues section.
    d. confirmed: The bug is real.
4. Optional: Add additional label "Critical" if the bug needs extra attention. Add label "Minor" if bug fix can be delayed.
5. Assign the bug to a component developer to fix, using their github account name and the "Assignee" menu on the github issue. Work with group leads if necessary.
6. Optional: After consulting with group leads, create a JIRA task in your group for fixing the bug. Use the task id in related commit messages. See Commit message template and JIRA and Github linking.

Reseting Next

When a new tag is created on master, the next branch should be reset. In order to reset the next branch, an integrator should perform the following steps.

1. Ensure they have the new tag: *git fetch origin -p*
2. Once they have the new tag, they can reset their local copy of next to the tag
    a. *git checkout next*
    b. *git reset --hard <tag> (e.g. git reset --hard v0.3)*
3. After next has locally been reset. They can force push next to origin: *git push origin +next*

Making a Maintenance Branch

When a new tag is created on master, a new maint branch should be created for that tag. An integrator should perform the following steps to create a new maint branch.

1. Create a new local maint branch off of that tag: *git checkout -b maint-<tag> <tag>* (e.g. *git checkout -b maint-0.3 v0.3*)
2. Push the maint branch: *git push origin maint-<tag>*

Merging Maint to Master

If a maintenance branch received a new feature that also needs to be on master (such as a bug fix), you can merge the maintenance branch to master after the feature has been merged to the maintenance branch. If this is not possible because the master branch has had to many changes, have the developer issue a second PR to master.

Creating a Tag on Master or Maint branches

Tags that are created on master or a maint branch should always be annotate tags. Tags are created on specific commits and don't track branches. In order to create a new tag, an integrator should perform the following steps:

1. Determine the hash of the commit that should be tagged (this could be a branch name, if the HEAD of the branch should be tagged)
2. Create a new annotated tag on that commit: *git tag -a <tagname> <hash>* (again, <hash> can be replaced with a branch name to tag the HEAD of the branch)
3. An editor will come up to write an annotation for the tag. The tag annotation should describe what the tag is for, and preferably a short changelog of what went into the tag relative to the last tag.
    a. In order to summarize the work that went into the tag, an integrator can use the following command: *git log --oneline --decorate --first-parent --graph <last_tag>...<hash>* (e.g. *git log --oneline --decorate --first-parent --graph v0.2...origin/master*)
    b. The summary created with the above command shouldn't be placed directly in the annotation. Instead it should be paraphrased to give an idea of the modifications that went into the model.
4. Push the newly created tag: *git push origin <tagname>*

Additional Information

Role of integration branches.

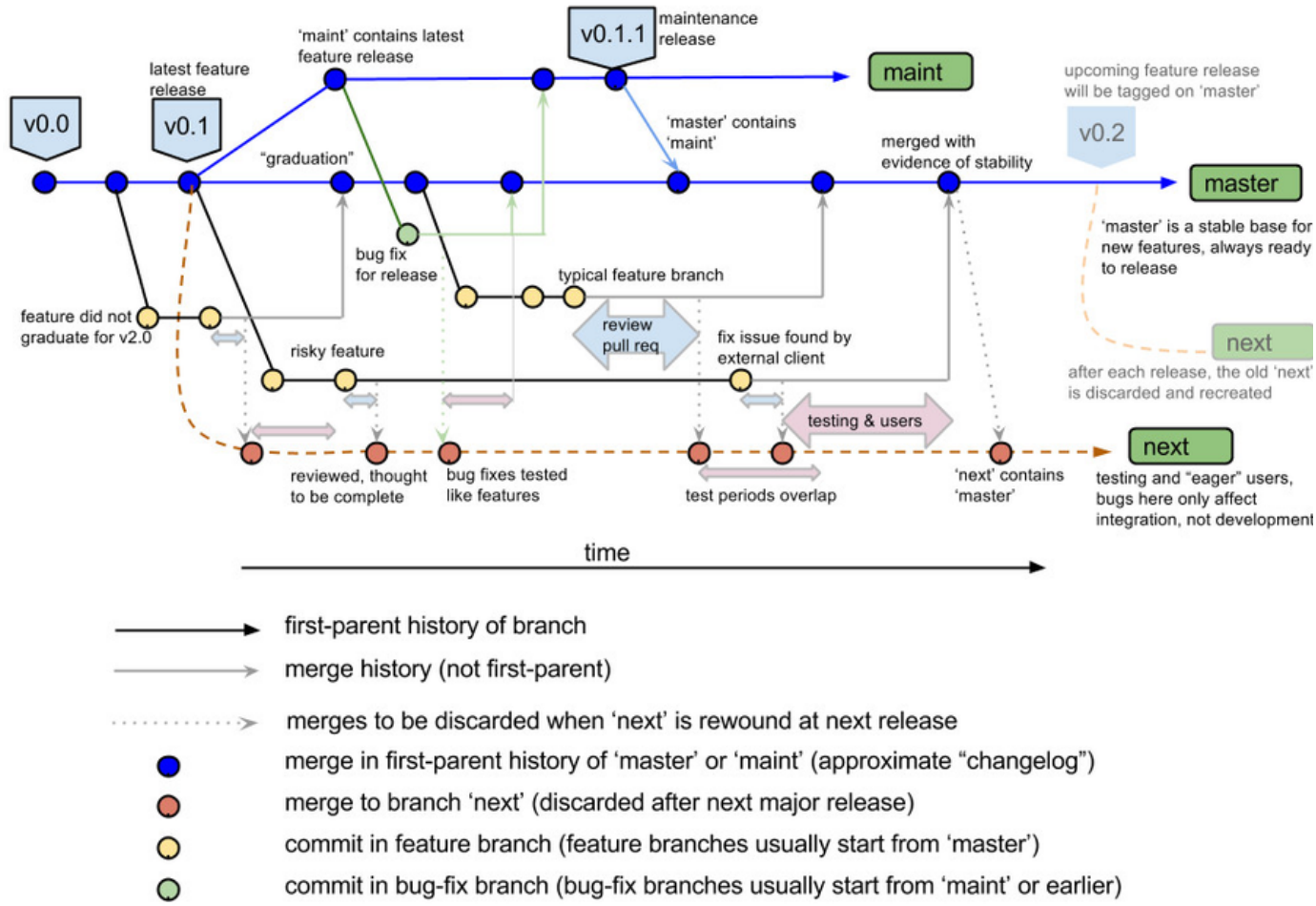
It is important for integrators to understand the role of the integration branches:

- maint - maintenance branch for bug fixes and portability updates for ACME releases
- master - stable platform for new development and integration of features for upcoming release
- next - latest changes for testing, may be unstable and is never merged to 'master' or 'maint'

An additional purpose of next is to help weed out features and bugs before they land on the master branch. The master branch is intended to be as stable and bug free as possible. As a result, we use the next branch to try and help enforce this. Some use cases the emphasize the benefit next provides:

1. A developer implements a new feature that passes the developer test suite, but serious issues are brought to light when the feature is pushed onto next. In the end, it is decided that the feature needs to be removed and re-designed from scratch.
    a. In this case, the merge onto next would be reverted, the feature branch can be deleted, and all can be forgotten.
    b. A benefit this workflow provides, is that the feature that causes major problems never lands on master, so master is unaffected by these major issues.
2. A developer implements a new feature that bases the developer test suite, and additional bugs are showcased when the integration test suite is run. In this case, the branch can have additional commits which fix the bugs added onto it, and the branch can be merged again into next (note, the first merge in next is not reverted in this case).
    a. Here, a feature branch can have multiple merges into next, to make sure we are comfortable calling the feature "stable"
    b. Note, the feature still only has a single merge into master.

The process is summarized in this diagram:



In addition to understanding the typical workflow, an integrator should understand the tagging policies defined in Branch, Tag, and Version name conventions

Integrating from a fork

As developers can choose to work in a fork of ACME-Climate/ACME it's important for all integrators to know what a fork is, and understand how to integrate from a fork.

A fork is nothing more than a developers personal clone of ACME-Climate/ACME that is hosted on github. A fork will be created under a user account, as opposed to an organization. A fork of ACME-Climate/ACME will result in a repository named username/ACME (e.g. douglasjacobsen/ACME).

When a developer works in a fork, their branches exist in their fork, and not in ACME-Climate/ACME.

To integrate a branch from a fork to the ACME master, you need to gain access to the fork's branches. To do this you need to add a new remote to your local repository that points to the developer's fork. For example:

```
git remote add username/ACME git@github.com:username/ACME.git
```

While replacing username with the developers actual github username will add a new remote to your local repository named username/ACME. Once the remote is added, you can update that remote's remote tracking branches using:

```
git fetch username/ACME -p
```

The -p flag will prune any removed branches in addition to update other branches.

After all remote branches are fetched, they will show up when using `git branch -a` or `git branch -r` namespaced under username/ACME.

Typical integration can now continue as listed above, using the remote pointing to the developers fork in places of *origin*.

Example of a successful merge and push to master

This is what you might see if you there are no conflicts.

```
blogin3[100]: git checkout master

Switched to branch 'master'


blogin3[101]: git branch

* master

  next

  rljacob/machines/add-anl-cluster


blogin3[102]: git pull

Already up-to-date.


blogin3[103]: git merge rljacob/machines/add-anl-cluster

Auto-merging scripts/ccsm_utils/Machines/config_machines.xml

Merge made by the 'recursive' strategy.

 scripts/ccsm_utils/Machines/config_compilers.xml    | 24 ++++++++++++++++++++++

 scripts/ccsm_utils/Machines/config_machines.xml     | 20 ++++++++++++++++++-

 scripts/ccsm_utils/Machines/config_pes.xml          | 16 ++++++++++-----

 scripts/ccsm_utils/Machines/env_mach_specific.blues | 25 ++++++++++++++++++++++

 scripts/ccsm_utils/Machines/mkbatch.blues           | 85
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

 5 files changed, 164 insertions(+), 6 deletions(-)

 create mode 100755 scripts/ccsm_utils/Machines/env_mach_specific.blues

 create mode 100755 scripts/ccsm_utils/Machines/mkbatch.blues


blogin3[104]: git push origin master
```

```
Counting objects: 16, done.
Delta compression using up to 16 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 934 bytes, done.
Total 6 (delta 5), reused 0 (delta 0)
To git@github.com:ACME-Climate/ACME.git
   c355863..8fdf993  master -> master
```

Resolving merge conflicts

Sometimes when attempting to integrate a feature, an integrator might encounter several merge conflicts they don't know how to deal with. In order to make the job of an integrator easier, they can employ the help of the developer who submitted the pull request using the following steps:

The integrator can request the developer:

- Fetch the history of ACME-Climate/ACME: *git fetch origin*
- Create a new branch at the head of their current branch with the same name but -resolved appended to the end:
  - *git branch user/component/branch-resolved*
- Merge the target branch into this new branch:
  - *git checkout user/component/branch-resolved;*
  - *git merge master* (If master is the target. Other targets could be next, maint, or other branches)
- Resolve the merge conflicts: *git status; vim file; git add file; git commit*)
  - Another method of resolving merge conflicts is to use *git mergetool*
  - The commit message for this resolved version is not required to have any specific format.
- Push the resolved merge commit onto github: *git push origin user/component/branch-resolved*

After the resolved version is pushed onto github, the developers job is complete. Now the integrator can attempt the merge again, using the resolved branch to fix any merge conflicts.

- Fetch the history of ACME-Climate/ACME: *git fetch origin*
- Checkout the target branch: *git checkout master* (If master is the target. Other targets could be next, maint, or other branches)
- Merge topic branch into target branch: *git merge origin/user/component/branch* (**NOTE:** Don't merge the -resolved version)
- Resolve merge conflicts using the -resolved branch: *git checkout origin/user/component/branch-resolved -- .*
- Finish merge commit, and edit to follow the format described in Commit message template: *git commit*
- Push merge commit onto target branch in ACME-Climate/ACME: *git push origin master*

Bypassing hooks

Sometimes when integrating a branch, the branch name is really long which triggers an error using our ACME-Hooks. In this case, the hook can be bypassed after ensuring that your commit message looks correct by issuing the following command:

```
git commit --no-verify
```

Integrating machine file changes

If the changes are only to machine files:

```
Depends.$machine
env_mach_specific.$machine
mkbatch.$machine
syslog.$machine
```

or the machine-specific blocks **config_machines.xml, config_compiler.xml** or **testlist.xml** then the POC of the machine (see Configuration Management) can integrate them even if they are the author of the branch.

Existing and Planned Tags and Branches

See Answer-changing commits for a detailed summary of non-BFB changes on ACME master between tags.

Conventions for the use of Tags can be found here: Branch, Tag, and Version name conventions

See also https://github.com/ACME-Climate/ACME/tags

Planned future tags are in red.

E3SM Tags: master and maint* branches

Master tag or 'git describe'.	Maintenance branch tags	Release date	Git hash	Main features (most are off by default. Must turn on with namelist).
v0.0		03 Jul 2014		Equivalent to cesm1_2_rel06. Scientific ancestor to all of ACME
	v0.0.1	31 Jul 2014		Version of v0.0 used by DOE UHR Project. cesm1_2_rel06 + CAM5_3_07 + code mods from DOE UHR project documented here: ACME V0 Run Analysis/Diagnostics
	v0.0.2	10 Aug 2014		Updates needed to run "High res" version after the 2014 Titan system upgrade
	v0.0.3	13 Sep 2014		Remove complaint from provenance recorder about non-svn repo.
v0.0-1-ga2d719c				Import of CESM 1_3_beta10. Includes CAM5_3_36. ACME development started here.
v0.1		01 Oct 2014		Version of Oct 1 2014, being used for coupled model runs. CESM 1_3_beta10 + DOE UHR mods from v0.0.1 + many machine file updates for ACME supported platforms.
v0.2		26 Nov 2014		Bring in CLM 4.5.1r85 from CESM dev and fix restart bug.
v0.3		20 Apr 2015		Add: atm polar-mods, RRM capability, new PIO runtime options, MAM4 package. Testing framework functional **Code used for AMIP "v0" baseline simulations**

v0.4		02 Oct 2015		Add: MPAS-O, MPASLI, MCT 2.9, 2 I cases, titan testing, COSP v1.4, prescribed aerosol with CAM5 or 4, CLM single point sims, crop harvest and yield, 3 E compsets, Bug fix: MG1, zenith angle, COSP interface **Last version before CIME merge**
v0.5		06 Nov 2015	a47f4120	Adoption of CIME, add generic BGC interface, add MPAS-CICE, link with Albany and PETSc, new python create_test, add coupling between GLC and LND, add dynamic roots, CLUBBe, UNICON, BeTR Many bug fixes.
v0.6		23 Nov 2015	660c01f3	Bring in full HOMME, add new DCS, fix CLUBBe and polar, add VSFM, define ACME B-case, allow P with denitrif, Many bug fixes including CLUBB-Mira, campp.
v0.7		04 Dec 2015	ba5010ffc6	Add P with fire, new ice nucleation, P fluxes with land use changes, improve MAMx and light-absorbing particles, add CLM UQ, add soil nutrient competition, Fix link between CLUBB and polar (again).
v0.8		28 Dec 2015	27aab2638	Add P deposition stream, accelerated decomp spinup, update MPAS components, add topounits, PIO options, Albany linking, redefine/add ACME B-cases, add ATM compsets, linux generic support, complete Albany support, add BC and POM in coarse mode, link new ice and MAM4, Fix ice nucleate bug, dynroot bug, bgc configs, fix liquid cloud, fix COSP
v0.9		13 Jan 2016	5fd1d04	Add ability to run on BGC case grid (ne30_oEC60to30), add interactive ozone chemistry (off by default), add marine organic sea spray emissions (off by default), add new CAM I/O infrastructure.
v1.0.0-alpha.1		18 Jan 2016	1d9f9086f8	Add Mosart, Update edison batch, add RRS15-5km mesh, update CLUBB, COSP to work with new I/O Fix: bugs for land timers, script regression test, hertfrz arrays on Mira with L72, non-threaded builds, Albany build on cetus **All v1 features integrated but not tuned.**
v1.0.0-alpha.2		26 Jan 2016	d174014fa	Updates to atmosphere compsets for BC deposition, marine organics; Update RSF file, add atm tests, add year 2000 land surf data for ne120, add ne16 surf data, add HOMME timers, CLUBB tunable params, new A_B2000 compsets. Bug fixes: linoz restart, typo in ATMMODCOSP, resus printing CIME fixes: better return code handling, bless_test test, NERSC machine updates
v1.0.0-alpha.3		03 Mar 2016	e565bfbeb	Additions: v1 WCYCL and CRYO compsets, 72L F compsets, MOSART tests, low-res MPAS grids, land SPBC compset. AV1F tests, ATM -00 compsets coupler: Change default fully coupled sequencing option from CESM1_MOD_TIGHT to CESM1_MOD MPAS fixes: critical ones for coupling and for long runs, watercycle defaults, preformance, cpl time avg, restart writing, output alarms, fix ERS tests for mpasli, freezing temp, latent heat. ATM Fixes: restart in CLUBB, PIO in cam, hetfrz_classnuc, check_energy, tevolve, qneg4, divide-by-zero in MG, GNU build, COSP lidar simulator, hetfrz uninit vars, marine organics and amicphys, Linoz radiative active and spinup, Lnd fixes: use correct r2o mapping file for ne30_oEC, correct ic for 1850, ne30_oEC and ne30_g16, read some params from file, ic for SP, ne30np4 and two ocean masks, argument intent Machine fixes: pnetcdf and mpi-serial on redsky and skybridge, edison modules, cetus testing, cori mpi-serial, constance pgi, PBS environment, WCYCLE pe-layouts for cori, edison, titan, mira. ENV variable handline on cori/edison, fix RESUBMIT_QUEUED, add Lawrencium. Performance fixes: update performance archiving **Low-res v1 WCYCLE case confirmed to run on: edison, cori, titan, mira, blues**
v1.0.0-alpha.4		01 Apr 2016	e413cc34df	Additions: Add AV1C-01 and AV1F-01 compsets. add ne120 L72 atm ic, add mom mixing state param, land BGC compsets, land ne120 surface data, PET and PEM for WCYCL. atm: fix a bug in the FCAV1C-01 compset, fix aero timers, fix clubb namelist settings coupler: driver threading on by default scripts: PET, PMT, ERP now working. lnd: fix running with debugging, timers, fix ch4 zero bug, build script, log messages, machines: XLF macros, Albany and cetus build, module commands an csh, testing timing, **Redefine WCYCLE case to use AV1C-01 atmosphere compset. Change coupling sequencing from CESM1_MOD to RASM_OPTION1 for ocean/ice stability.** **Enable high-res ne120_oRRS15 grid.**
v1.0.0-alpha.5		28 Apr 2016	0b1048d1946	**Major fixes:** (ATM) turn on atmosphere part of BC/dust deposition to snow/ice, fix ice nucleation, sync CLUBB tunables, fix marine aerosol and BC, fix sea spray (OCN) limit pressure from sea ice Other fixes: (ATM) cmvix with debugging, remove cosp timer, COPS restart bug, loop in sea salt (LND) BeTR with Pgi and debugging, workaround for PGI array indexing algorithm limitation, BFB with nested threading (OCN) CVMix debug FPE (SCRIPTS) fix sierra, make gnu default on Blues, update intel on eos, (TESTS) mpas tests on edison, mpas in threading tests, -j control, New: (ATM) add FC5AV1C-02, F1850C5AV1C-01, F1850C5AV1C-02 and -L compsets, ERS test for COSP, "FC5AV1C-01 minus marine organics" option, (LND) ic for land for 2000 and 1850 for ne120_oRRS15 and ne120_g16 (SCRIPTS) timers with prefixes **Redefine WCYCLE case to use AV1C-02 atmosphere compset.**

v1.0.0-alpha.6		20 May 2016	4842dc64643	**Major fixes:** (OCN) vertical mixing fixes, new high res mapping files, Other fixes: (ATM) compute tms bug fix, fix namelist for RK MP, high-res config on mira (OCN) fix badly formatted write, (SCRIPTS) error code in jenkins, turn off hyperthreading on cori, remove PEA test, fix HOMME tests on NERSC. New: (ATM) add FC5AV1C-03, F1850C5AV1C-03, add zmconv_alfa to namelist (OCN) add Redi mixing in GM (disabled), transect transport AM, more info on timesteps, output attributes on history (SCRIPTS) high-res builds on cori (ALL) PIO2 support **Redefine WCYCLE case to use AV1C-03 atmosphere compset. High res WCYCLE can build (but not run) on cori**
v1.0.0-alpha.7		04 Aug 2016 (note: only 2 commits to master between May 20 and June 20)	b7e7dacf77ec	**Major fixes:** (ATM) Restart bug with qneq4, one element per MPI task bug (SEA ICE) Use subcycling for high-res, fix conservation analysis (LND) ECA fixes (OCN) limit salt fluxes to prevent negative salinities (MOSART) add and fix 1/8th degree grid and all runoff mapping files (SCRIPTS) initial ne120 layouts for corip1/titan/mira/edison Other fixes: (ATM) improved openmp performance (LND) avoid VSFM crash with PGI, change SMS to ERS, move pointclm, CLM40 urban bug (OCN) cvmix fixes (MOSART) uninitialized var (MPAS) tmp redirect bug (SCRIPTS) update titan compiler, cmake modules, corip1 modules, increase timer level, fix sierra (ESMF) remove warnings New: (ATM) Add FC5AV1C-04 2000 and 1850 compset, clubb tunable params by namelist, HOMME velocity interp, Conus RMM with land, HOMME on KNL (LND) initial conditions for ne30/16 and mpas grids, IC for conus4v1, I compsets for BC (OCN) GMPAS-IAF compset, separate barotroic timestep (SCRIPTS) cime5 scripts, add deakin at SNL, ANL workstation, Minnesota machines, no more B-case tests with POP (ALL) ne16/ne11/ne4 grid files and ICs, cime_config directories. **Redefine WCYCLE to use AV1C-04 atmosphere compset. High res WCYCLE case can build and run on edison, mira. High res now uses 1/8th degree MOSART grid. Support for ultra-low res (ne16, ne11, oQU240)**
v1.0.0-alpha.8		14 Sep 2016	6f5d9d8adbb0011	**Major fixes:** (ATM) new qqflx fixer for water convservation, qflx restart problem (OCN) bug fixes to CVMIX and GM/OpenMP, speed-up for high-res, salt content of frazil (MOSART) allow faster equillibrium (LND) add two missing runoff fields, speed up clm_init Other fixes: (MPASLI) new framework changes (ATM) Mira depends and other build updates (MPAS) remove reference time from all streams. (MPASO) fix alias names for oQU240 grid (CLM) update ic for 1850 ne30_g16, crop model tests and bugfixes, VSFM bug fixes, xlf compiler fixes (CPL) update frazil terms in budgets (UTILS) more PIO rearranger options (SCRIPTS) update Titan environment New: (ATM) Add FC5AV1C-04P compsets for 1850, 2000, 20TR., ne11L30 ncdata, v0 compsets, wind gustiness, more CLUBB tuneable params, F20TRC5AV1C-03, F20TRC5AV1C-04, F20TRC5AV1C-L compsets, (OCN) introduce the oRRS18to6 grid (LND) diagnostics for total water, params for ECA, landuse datasets for ne11, ne16 (SCRIPTS) add new sandia machines, cron script **Redefine WCYCLE to use AV1C-04p atmosphere compset. Allow MPAS ocean and sea-ice to start from spun-up conditions. Better conserve water. Support ne11-oQU240**
v1.0.0-alpha.9		07 Nov 2016	490e9395d523aa8	**Major fixes:** (OCN-ROF) Fix 3 bugs preventing long simulations. (OCN) MPAS timekeeping bugs (ICE) MPAS-CICE restart with WYCL case (ATM) Fix RRTMG bugs Other fixes: (ATM) Added non-negativity constraints for CB aerosols, change threshold for QNEG3, read adiabatic and ideal from atm namelist (LND) VSFM density fix, correct land IC for 1850 and 2000 for ne30_oQU120 (CPL) coupling frequency for ne4, ne11 grids. (SCRIPTS) update melvin, OPENACC depends on Titan. New: (ATM) support FC5AV1C04-ne4ne4 (SCRIPTS) Add support for Anvil, ERS ne11-oQU240 coupled test, support for CIME5. **Fix for ocn-rof problems preventing long runs. RRTMG bugs. Anvil support**.
v1.0.0-beta aka "beta0"		17 Nov 2016	7a17edbe59324	**Major fixes:** (OCN) Fix date handling in MPAS drivers. Change frazil settings back to alpha7 version. Other fixes: (ATM) Many updates to HOMME: simplify code base, remove leapfrog, semi-implicit and filter code. (OCN). Various bfb changes (analysis members, testing suite, land ice, forward model options). (Sea Ice) improve numerics for principal stress calculation New: (ATM) new compset AV1C-04P2 (OCN) new grids with cavities below ice shelves. **Update WCYCL and CRYO compsets to use new AV1C-O4P2 atm compset.**
v1.0.0-beta.1		02 Mar 2017	00a38722dbce8e	**Major fixes/updates:** (OCN) Use Ri smoothing, Filter high-freq modes from surface, BFB with threading, v3 grids (CIME) update to 5.1 Other fixes: (CIME) updated configs on many machines: Anvil (pe layouts, timer dir, ifort version,), Lawrencium, LANL IC, Mira/Cetus, Cori haswell and KNL, Edison (faster layouts), Redsky, Titan, Melvin. Also HOMME testlog, performance archiving, fix compset names, suite-specific walltimes (MPAS) all cores updated, better SCRATCH file behavior (ATM) Allow SCM to work with CIME5, speed up CLUBB, Disable MPI_RSEND, fix non-BFB with threading issue (CICE) fix PIO error (HOMME) dycore cleanup, remove FVM **Fixes for ocean (Ri smoothing, surface tilt filtering, BFB with threading, v3 grids), CIME5.1**

v1.0.0-beta.2		27 Sep 2017	8ea5147238fa13c15	**Major fixes**: (OCN) The critical path fixes: correct del2 scaling, new BL smoothing for KP, rayleigh damping in bottom layer, horizontal flux limiting, (ATM) add IEFLX fixer, correct MG2 rain number convection limiter, correct MG2 ice number sub tend, avoid ice sublimation divzero in MG2, limiter for MG2 precip frac, static frozen energy bug. (LND) Add and update FATES, add BETR_v2, N imbalance and fixation, organic pools and radiocarbon, fire and p mass budget bug, urban albedo bug Other science fixes: (ATM) add i_ieflx_fix namelist, Switch RTM to SROF in F-cases, surface fluxes in SCM, CAM4 albedo, momentum flux in gustiness, fix HOMME areas, SCM idealizations for relaxation and precipitation, fix floating invalid in MG2, provide aerosol init for SCM, fix CALYPSO simulator and MG2 precip fluxes, gamma terms in mo_strata (LND) add interface with NGEE-Actic CLM-PFLOTRAN, init unset vars in CNAllocationMOD, forest N fert. exp, soil P pools, varsoil in ZD, make sure ci is positive after photoshythesismod (OCN) ocean/tracer heat budget, bottom drag option, CPL_ALBAV fix, budget terms post-cime5, double the EC60to30v3 timesetep (CICE) fix snicar path bug, history write bug (COUPLER) update coszrs calc with jday, fix coupling intervals and add salinity restoring for C,G cases, BFBFLAG is true Other fixes (ATM) reduce h0 size, shr_repo_sum NAN check, HOMME I/O init, zoltan in HOMME, HOMME build, add dyn_npes, add aerosol and cloud history, update nosmp config, allow physics load balancing by default and fix twin init, fix dyn_npes < npes bug, HOMME performance optimizations (LND) ALM v2 pieces, build script and BGC, memleak in landunit, threading and clm-bgcint, fix threads and CN, unalloc pointer in ALM, add EMI to VSFM, change units of water flux btw root, soil (OCN) Threading optimizations, Don't always write ts restarts, new high freq stats, PIO_TYPENAME control, Fix analysis, new log system, fix block partitioning, update regionalStatistics analysis defaults Compsets: many high-res added. I-Compsets for ECA with CRUNCEP and 20th transient, spun up with v0atm, E and I for BGC, ne120 F and WCYCL compsets for F1850 and 20TR. Tests: COSP and cosplite in developer, hi-res and extended suites, land 20th trans, FATES, low-res in atm, SCM test for ARM97, add PEM_Ld3.ne30_oECv3_ICG.A_WCYCL1850S and other tests for that compset, shorten crop model run length, ECA tests New grids: TWP RRM, EC60to30v3wLI, ENA RRM, ne120_oRRS18v3_ICG, New datasets: Many land ic updates, P dataset and 1850,2000 ic for land ne240np4, include phosphorous in land surface datasets, update IC files for oRRS18to6v3_ICG and oEC60to30v3_ICG, update ic for land ne120_oRRS18v3 CIME additions: update to 5.3 (alpha06 then alpha10), charge to subaccount, allow jenkins to run mutlple compilers, check tput, build with MOAB, allow running short-term archiving during a run. CIME fixes: handle corrupted TestStatus, don't move to many files in short term archiving dashboard resubmit, Fix SEQ on skybridge/chama, archive Macros, Fix TIMER_DETAIL, TIMER_LEVEL, build_env.txt file, highwater memory stats, st_archive template, $ENV resolutions, remove SAVE_TIMING from non-prod machines, make SVN check less strict, remove submodule hacks, add Pat's performance fixes in MCT, use MPI_Isends in PIO to avoid buggy mpi_irsends, don't build hdf5 if using mpi-serial, always save interim restarts, archive mpas files, refactor perf provenance saving, man fixes to jenkins_generic_job, PIO2 bug in restart physics, interaction with HOMME run_acme updates: add to model repo, big cleanup, add script for nightly testing, restart and stop check, submission cleanup, fix branch runs, improve functionality Config updates: too many to list. Every machine was updated since beta1. Many pe layouts added. **Critical path fixes to MPAS-o, MG2 fixes in atm, CIME5.3.0**

v1.0.0-beta.3		25 Jan 2018	1521d38	**Major fixes/additions:** (OCN) split FCT advection, many updates for BGC (ATM) fix bug in FSUTOAC, code to read CMIP6 strat aerosols (LND) P balance error, fix soil decomposition when N,P are co-limited, fix ne30 land surface datasets, reading of soilorder, many CN updates (ICE) several bugs fixed, many updates for BGC Known problems: Runs with Intel17 can possibly have data corruption. CIME query_config script broken. Other science fixes: (ATM) new mass gradient option in MG2, SCM with prescribed aerosol, new limiter option (LND) Fix phosphatase activity parameter dimension, update to FATES 3.0.0 (OCN) correct mpas_forcing, add eddy analysis product (ICE) D-compsets now work (COUPLER) allow different maps for liquid and ice glc2ocn, add convergence criteria in flux calculation Other fixes (ATM) add dyn_npes_stride option, remove norm2 use, error checking in nested threading, nan check in HOMME (LND) longer htape name, IO of text var, remove unnecessary compilation in MPP, performance improvement to VSFM (OCN) multiple forcing streams, allow users to point to different graph files, add wallclock output, reduce halo barriers, reduce default output (ICE) add namelist entries Compsets: several BGC compsets, add aqua-planet compsets for FC5AV1C-L, add CMIP6 compsets A_WCYCL20TRS_CMIP6, F20TRC5-CMIP6, A_WCYCL1850S_CMIP6, F1850C5-CMIP6. Add MPAS-CICE BGC compset, GSWP3 and cpl_bypass compsets for ECA and RD Tests: new standalone HOMME tests, add high-res tests for B and G, Add threading test to developer suite, Get Theta ready for CI, change anvil to run integration grids: Remove many unused grids New datasets: ne240 land init, map for high res phosphorous data sets, new map for high res runoff to ocean CIME additions: query_config, case.qstatus, skip-preview-namelist option, script for ESCMI merges/splits, document time spent before timing lib init, save pre-run provenance, batch email options,b allow different drivers, bless logs, unit tests on skybridge, update to 5.4 (alpha17), hide the run and archive main scripts, add python3 support, encapsulate XML, many more. CIME fixes: cprnc log handling, refactor queue selection, rename PES_PER_NODEpio root 0, refactor baseline handling, fix bless_test_results, Limit performance archiving to specific projects, Reduce compiler warnings from Fortran code. Many more. run_acme updates: fix branch runs, cosp, add to model repo, big cleanup, add script for nightly testing, restart and stop check, submission cleanup, fix branch runs, improve functionality Config updates: Theta updates, LLNL updates, work around PGI bugs on Summitdev, updates to SNL machines (toss3), many updates to NERSC machines some to keep up with environment, others to speed up simulation. KNL now default on cori and intel18 default compiler, hyperthreading on Theta Many more. **MPAS split FCT, many land fixes, add BGC compsets, BFB with beta3rc10 which was run for 200 years.**
v1.0.0-beta.4		09 Mar 2018	18388d835c	**Major fixes/additions**: (COUPLED) and 1950 compsets (LND) update to BeTRv2, ECA soil P fixes (SEA ICE) improved graph files (ATM) Fix Intel17 bug with HOMME Other fixes/additions (ATM) add RELVARC diag, regional output with SE, update fsnowoptics for high res (LND) 14C pool bug fix, update MPP, dynamic land units for crops, update EMI, remove autogenerated files, change ELM dynamic root option, sBetr and gnu, nag, (OCN) salinity restoring under sea ice for G cases, 2 new mid resolution ocean-ice meshes (RIVER) new high res routing file CIME additions: update to 5.4.0-alpha.23, rename acme to e3sm, auto-regen batch files, add multiple sources of input data, fix query_config, performance boost, data model updates, reduce Fortran warnings Config updates: Update Titan software, High res layouts for Theta, cori-knl, sandiatoss3 intel to 17.0, Jenkins test name length, edison layouts for wcycle low res **Fix Intel17 bug in HOMME on edison. Last version before removing v0 components (POP was removed immediately after this)**
v1.0.0-beta.5		05 Apr 2018	c133017ece33	**Major fixes/additions:** (COUPLED) add -DAPPLY_POST_DECK_BUGFIXES for 1950 compsets. Bugfixes includes: (ATM) cosp1.4 modis fixes, RRTMG limiters (LND) surface water and lakes bug, Also: 1950 compset fixes, add BGC compsets, remove POP, CISM, WW3. Other fixes/additions (ATM) add RRM tests, init cosp arrays correctly, fewer QNEG3, QNEG4 warnings, perturbation growth tests (SEA ICE) add abort streams (LND) mods for BGC spinup, update param files for RD, fix dynamic roots for tropics, crop bug fixes, I cases with GSW, pe layout descriptor, change ocean soil order values (LND ICE) fix coupler indicies, update calving and stats (RIVER) use MOSART everywhere (MPAS) handle hybrid cases better. CIME fixes: Update to 5.4.0-alpha.25, better handling of non-core jobs, allow custom batch directives, better email support for LSF, add A_WCYCL1850S_CMIP6 test Config updates: remove compiler warnings on Theta, adjust Theta optimization settings, Add Summit, add stampede2, add fp exception checking for KNL, cleanup titan, **Used for v1.0 prototype 1950 high-resolution water cycle runs**
v1.0.0		20 Apr 2018	a0ef6f6df7951	**Major fixes/additions:** (COUPLED) change CMIP6 water cycle to be hybrid out of box (OCN) single MPAS source dir, update to v6 public release (LND) update to public versions of sBeTR, FATES, MPP (CIME) add new wget server Other fixes/additions: (ATM) remove unsupported compsets, SCM and run with 1 task and full mpi library, new ne120 comsets, change compilation on PGI/Titan for some files (LND) remove unsupported compsets, fix C init for ECA, new ECA param file, init a bgc variable to zero (OCN) rename to mpas-ocean (LND ICE) rename to maps-albany-landice. Remove CLM40, aquap CIME: update to cime 5.5.0 (aka 5.4.0-alpha.28), change location of e3sm tests, fix aprun command, move test_mods dirs, fix gptl warnings, order tests by runtime, update MCT to 2.10rc2, add shell batch scripts Config updates: Change Titan baseline dir, Make Cori layouts so tasks < elements, change all submodules to use https for release. **First public release**

E3SM Permanent Branches

Branch	Purpose	Development requested	Who to ask about development if you're not sure.
master	The "trunk" or latest version of the model.	most development. Part of a roadmap. Bug fixes	Your group lead.
next	Special branch used for testing multiple new features before merging to master	N/A	
maint-0.0	Branch used for maintenance of v0.0. The code used by the DOE "Ultra High Resolution" project lives on this branch.	none. No longer in development.	David C. Bader
maint-1.0	Branch used for maintenance of v1.0. The code used for CMIP6 DECK runs.	Only changes to keep the model running on edison. Other machine updates as requested.	Chris Golaz
maint-1.1	Branch use for maintenance v1.1. The codes used for v1 BGC simulations.	Machine file updates for currently used machines. Some bug fixes. Some BGC features.	Susannah Burrows

Branch, Tag, and Version name conventions

Background

E3SM keeps nearly all of its code in a single repository.

To help track development, branch and tag names in the E3SM source must follow conventions given here.

For background, see discussions at Tag and release branch name conventions page, Referencing un-tagged commits, Additional tag name conventions, and Council 2015-03-05 Meeting notes.

E3SM Version Names

E3SM will generally follow Semantic Versions 2.0.0 http://semver.org/spec/v2.0.0.html

In general, versions are labeled vMAJOR.MINOR.PATCH

First release of major versions are labeled as "vX.0.0".

First release of minor versions will be labeled as "vX.Y.0" with Y>0. (Note: Y is not limited to 1-9, so we can have v1.27.0 before v2.0.0)

Bug-fixes and other changes made to a released version vX.Y will be done on a "maint-X.Y" branch and tagged "vX.Y.Z" with Z>0.

Major version zero (v0.Y) is for initial development. Anything may change at any time.

We increment the numbers as follows:

1. MAJOR version when significant new science capabilities are added and old ones possibly deleted.
2. MINOR version when E3SM adds minor new science functionality in a backwards-compatible manner, and
3. PATCH version when E3SM makes backwards-compatible bug fixes.

alpha, beta, rc

Prior to a vX.0 (X >1), E3SM may add alpha, beta and rc lables to versions as follows:

alpha = after all new features have been integrated. This version is run for a decade or more to look for instabilities not detected by short running development/integration testing. The first alpha tag is made after all the features from the feature freeze have been integrated to master. Additional alpha tags are made as needed.

beta = after alpha. Versions used for coupled tuning. The first beta tag is made after the coupled simulation group says alpha testing is finished.

rc = release candidate. Near-final version. Checked for documentation, portability. The first rc tag is made after the coupled simulation group determines beta testing is finished AND after a code freeze. Typically, the "rc" versions and released versions are nearly identical.

Tag sequence before v1.0 may look like: v0.1, v0.2, ... v0.14, v0.15.....v1.0.0-alpha.1, v1.0.0-alpha.2, ..., v1.0.0-beta.1, v1.0.0-beta.2, ..., v1.0.0-rc.1,v1.0.0-rc.2, ..., v1.0.0

Development (aka Topic or Feature) Branch Names

What: A Development or "Topic" or "Feature" branch is made for any code changes needed for E3SM. This can be new development, bug fixes, merges from CESM, or maintenance updates.

Who: Any E3SM developer can create a topic branch. They can be for development or also to temporarily mark a place in the code.

Convention: E3SM branch names will have the form:

```
<Github username>/<source code area or component>/<feature-description>
```

- **Use lower-case for everything. Even your username.**
- Use hyphens instead of underscores.
- **one** name for the github username.
    - This is the person in charge of the branch and not necessarily the only person working on it.
    - The E3SM SE team needs one person to act as point-of-contact to answer questions about the branch and take requests for merging.
- Group leads can and should impose additional conventions for <source code area or component> and <feature-description>.

Examples

`maltrud/machinefiles/mustang`

`singhbalwinder/atm/polar-mods`

`tangq/cam/rrm`

`douglasjacobsen/clm/4.5.72`

Permanent Branch Names: *master, next, maint-x.y*

What: The E3SM source code will have permanent branches used for development and maintenance of the code.

Who: Only Integrators can change or create new permanent branches.

Convention:

master: The "trunk" or main branch of the code. Contains latest stable code. Always in a releasable state. Nearly all development should start as a branch from master.

next: A branch used for merging several topic branches for testing. Used to test several features at once. Is destroyed and re-created after the target group of features is merged to master. Also called an "Integration" branch.

maint-x.y: A branch started when a vX.Y release (internal or external) is made and used to provide updates and bug fixes for that release. A.k.a "release branch". After the maintenance branch is made, master can proceed to the next version.

Examples:

maint-1.0 - contains updates and bug fixes to v1.0

maint-1.1 - contains updates and bug fixes to v1.1

maint-2.0 - contains updates and bug fixes to v2.0

There is no such thing as the "maint-0.0.1" branch.

Developers should avoid using "maint" in their topic branch names.

Tag Names on *master* and *maint-x.y*

Who: Only integrators can create release tags of these branches. **It is OK for users to make archival tags on master.**

What: E3SM will not tag every single merge into the E3SM master. E3SM will use tags on master to mark significant development steps, and may coincide with versions where the Coupled Model and/or Performance teams devote significant resources to evaluate climate or code performance.

The recommended method for users to mark specific points on the git master branch is for them to create a topic branch or fork.

For current and upcoming tags on master and maint-x.y see Existing and Planned Tags and Branches

Tag Names on feature branches: Archive Tags

Who: Users and teams can tag their development branches in certain cases.

What: Tags can be used to archive versions of the code. This is the case where

1. exploratory runs were made (for climate or performance) and that commit was not already tagged on a permanent branch (see below).
2. where speculative code development was performed, not merged into master, but needs to be retained
3. Some journals requires the precise code used in the published simulations be available publicly. Simulations made during the development process may be using code that doesn't have an official tag like "acmev1-beta01", and one can create an archive tag to

identify this version. Archival tags will publicly accessible in E3SM public releases.

(See Referencing un-tagged commits for discussion that articulated this use case.

Naming:

archive/<User or Group Name>/<description>

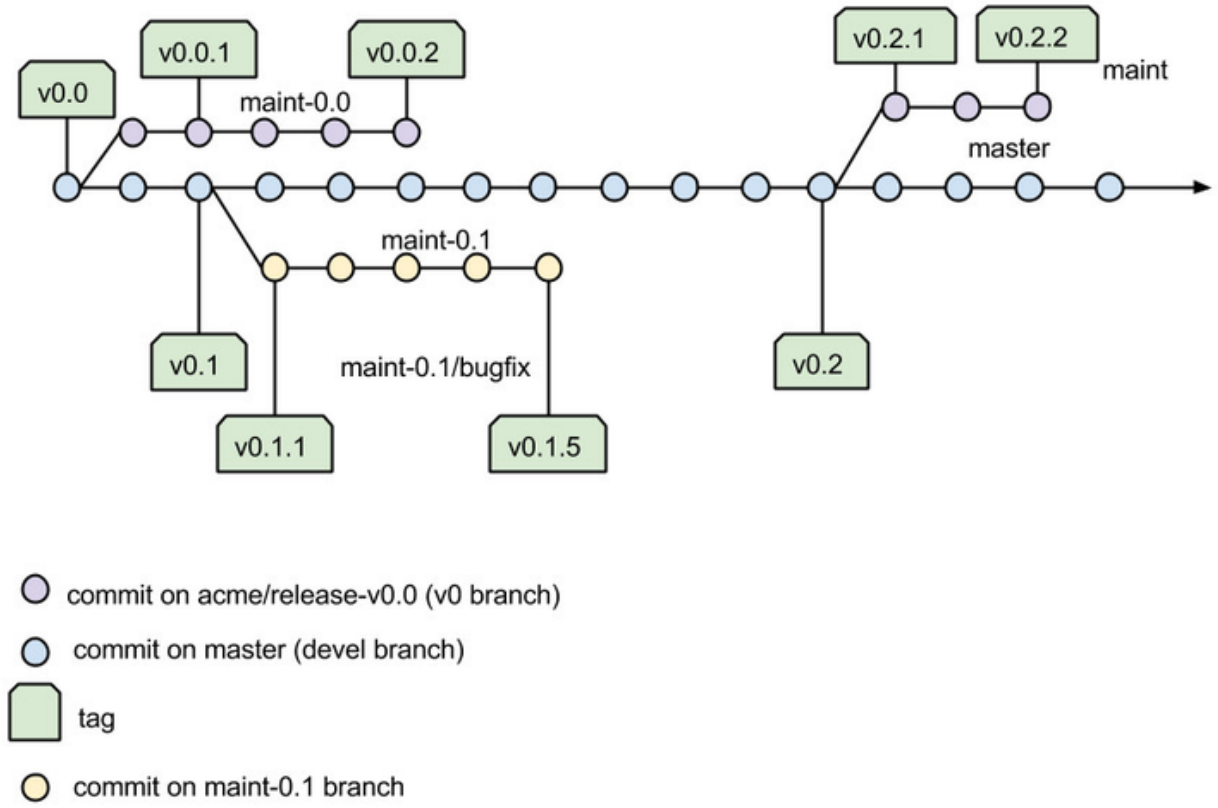
Note that once a commit has been tagged, it will be preserved even if the branch it's on gets deleted. We recommend that unused branches be deleted and we can rely on the archive tag to keep needed commits reachable. Component teams may want to adopt a consistent versioning system within their tag namespace, such as

- archive/cam/5.3.1
- archive/cam/5.3.2
- archive/cam/5.3.3

Figure: Tags and development graph

The figure below illustrates the relationship between tag names and code development history.

Every circle represents a commit of some code.



Git Tutorial

Questions? Use the Help: Git and GitHub page.

Teach Yourself git:

For those new to git, there is a wealth of information on the web.

Here are a few of the ACME SE team's favorite sites to start with:

Basic Git

- Official Git Documentation - the manual and cheat sheets **<- Best place to start!**
- Collection of Tutorials- from the official site.
- Git: the simple guide - includes pdf cheat sheet.
- Responsive cheat sheet
- try.github.io - interactive tutorial on basic operations

- Git for computer scientists - graphical intro to the Git data model
- Think Like a Git
- Visual Git Guide
- Git: Revision Control Perfected

Specific git features

- Interactive tutorial on branching
- Git Pretty what to do if you have a mess of development in your repo.

Here are a few sites that provide information on how to properly format commit messages. This step is very important because several git commands rely on this formatting.

- A note about commit messages
- Git commit messages (OpenStack)
- ACME Commit message template

ACME-related Lessons

The Interactive tutorial on branching is an excellent resource. We have created some exercises for workflow steps seen in ACME

- Develop a new feature
- Integrate and graduate features

Hands-on Tutorials:

These are being organized on a volunteer basis by individuals at the labs.

LANL - Doug Jacobsen (Deactivated) *(1: 24 Jul 2014* / 12:00pm - 3:00pm / Building 200 - Access Grid Room) (2: 05 Aug 2014 / 11:00am - 2:00pm / Building 200 - Access Grid Room) (3: 06 Oct 2014 / 9:00am - 12:00pm / Building 200 - Access Grid Room)

Sildes: https://docs.google.com/presentation/d/1syVrLb-6F-DFcrlRKOVCfSm8-LfqEsF8e9gu-I8_6tE/edit?usp=sharing

LBNL - Jeffrey Johnson *date/time/place*

ANL - Jed Brown 06 Aug 2014 / 3:30pm-5pm / Building 241, A323 (PDF slides)

The SE-team highly recommends using the hands-on tutorial material prepared by the Software Carpentry organization.

Git Cheat Sheet

This page provides a cheat sheet for people to use as a quick reference for the commands used in this document.

NOTE:

The cheat sheet should not be used in place of understanding why you use specific commands. Misunderstanding of the concepts described in this document can easily cause issues within the workflow.

The cheat sheet is colored as follows:

important -- Items colored in red mean they are important, and should not be ignored.

one time -- Items colored in green are commands to be issued once per machine.

repo once -- Items colored in orange are commands to be issued once per local repository.

common -- Items colored in bold black are commands that will be commonly used.

Setup Commands:

Command	Use
git config --global user.name "First Last"	Setup the first and last name that will be used in commits
git config --global user.email "user@domain.com"	Setup the email that will be used in commits
git config --global core.editor ${EDITOR}	Setup the default editor for editing messages
git config --global color.ui true	Enable coloring of output from git commands
git config --global http.proxy http://proxy/server:port	Enable a proxy for git commands

Repository Setup Commands:

Command	Use
git clone protocol://path/to/repo.git	Clone a remote repository into a local repository
git clone git@github.com:ACME-Climate/ACME.git	Specific clone command for ACME

Commit Commands:

Command	Use
git add path/to/file/in/repo	Stage a new file for the next commit. This file will be tracked in future commits
git commit	Commit all staged files
git commit -a	Commit all changes to tracked files

Branch Related Commands:

Command	Use
git branch github-username/component/feature acme-vXX	Create a new branch from a tested tag named github-username/component/feature

git branch github-username/component/bug-fix commit-with-bug	Create a new branch from a specific commit named github-username/component/bug-fix
git checkout github-username/component/feature	Change the current working branch to github-username/component/feature
git checkout -b github-username/component/feature master	Create a branch from master named github-username/component/feature and change the current working branch to it
git branch	List all local branches
git branch -r	List all remote tracking branches
git branch -a	List all local and remote tracking branches
git merge-base github-username/component/my-feature ACME-Climate/ACME/master	Determine the last shared commit between github-username/component/my-feature and ACME-Climate/ACME/master

Working with remotes:

Command	Use
git remote add remote-name protocol:address/to/repo	Create an alias “remote-name” to communicate with a remote repo at the address specified
git remote remove remote-name	Delete an alias named “remote-name”
git push <remote-name> <local-branch>:<remote-branch>	Push the local branch named <local-branch> (and all associated commit) to the remote named <remote-name>. :<remote-name> will rename the branch on the remote.
git push origin feature:github-username/component/feature	Pushing a local branch named feature to the branch github-username/component/feature on the remote origin
git fetch <remote-name>	Fetch the history from the remote named <remote-name>. Don’t store this history in any local branches
git fetch --all -p	Fetch the history from all remotes, and prune any branches that were deleted on the remote.
git pull <remote-name> <branch-name>	Fetch the history of <branch-name> from the remote named <remote-name> and merge it into the current working branch. **This command is for advanced developers.**

History Manipulation Commands:

Command	Use
git cherry-pick <commit-ish>	Apply a commit or a range of commits **onto** the current working branch
git merge [commit-ish1] [commit-is2] …	Merge a branch or set of branches **into** the current working branch
git rebase -i new-base	Apply all commits from the current working branch that do not occur in the history of new-base **onto** new-base **This command is for advanced developers.**
git rebase -i $(git merge-base HEAD ACME-Climate/ACME/master)	Apply all commits in the current working branch **onto** the base of the current working branch, but allow modification. **This command is for advanced developers.**

History Viewing Commands:

Command	Use
git log --oneline --graph --decorate <commit-ish>	Display the history of <commit-ish> as a graph with one line per commit.
git log --branches=*pattern*	Display the history of all local branches that match pattern
git log --remotes=*pattern*	Display the history of all remote tracking branches that match pattern
git log --first-parent <commit-ish>	Display all of the history of <commit-ish> only showing first parent commits
git log --name-only --follow -- path/to/file	Display all of the history of a specific file, including renames.
git log --pretty=format:"%h%x09%an%x09%ad%x09%s"	Display one line summary of commits with date. Use with *---first-parent* on master to see merges to master.

Commit message template

In order to have a clear, readable record of changes made to the E3SM source, its essential for all commit messages to have a similar style and content. We do not use a "ChangeLog" file. The "git log" command and good commit messages replaces the ChangeLog file.

The description of a github Pull Request will be used in the merge commit message for the branch so all PR descriptions should follow these rules.

All commit messages should have a title AND a body. Just repeat the title for the body if you can't think of anything to add.

The title should completely describe the change within its character limits. Do not continue the sentence in to the body.

The first sentence of the body should be passive and not contain a subject. It is understood you are talking about the commit or PR. Do not start with "This commit...." or "This PR..."

Do not substitute github issue numbers for descriptions of what you are fixing or doing. Imagine that GitHub goes away someday. The log messages in our git repo should still make sense. Reference issue numbers at the end (the "FIxes" lines) or parenthetically.

In addition to JIRA issue numbers at the bottom of the commit, you may, but are not required to, use other commands to manipulate the associated JIRA issue. See JIRA and Github linking.

Commit message in a branch

```
Describe changes in less than 70 characters in title.

Be sure to add a blank between title above and this text.  In the body
of the message provide more detail on what these changes do.
It should be enough information for someone not following this
development to understand.  DO NOT LEAVE THE BODY BLANK.  Just repeat
the title if you have to.

[BFB] or [non-BFB] or [CC]    !! Add at least ONE of these keys to
indicate how this commit will affect testing against baselines.
                              !! [BFB] means all output from tests will
be [BFB] with the baselines.
                              !! [non-BFB] one more more tests will not
be BFB with baselines.  You can specify the tests.  You can also
                              !! use [non-BFB] and follow it with a
description of what cases will be non-bfb if those arent under test.
                              !! [CC] the commit will change the climate
of one or more cases under test.

[FCC]                         !!  Add [FCC] if the commit will change
climate if a flag is activated.
[NML]                         !!  Add [NML] if the commit introduces
changes to the namelist.

AG-67, AG-56     !! On the last line, add JIRA issue numbers for issues
this commit is satisfying if available.
```

Merge commit messages (For Integrators when merging to "next" OR "master")

```
Merge branch <branchname> (PR #10)      !! Add the PR # to the
automatically generated title.  Leave "into next" in title when merging
to next.
                                        !! Don't worry about going over
the 70 char limit.

(Be sure to add a blank after above title)
(copy and paste the original description from the PR.  It may already
have the content below.)

(copy and paste the testing description in the PR and add info on any
more testing done prior to merging)

Fixes #XY, Fixes #MN  (For commit to master only, include Github issue
numbers for the bugs this commit fixes.  Be sure to use the word
"Fixes"
         before each # to close the associated bugs)

[BFB] or [non-BFB] or [CC]    !! Add ONE of these keys to indicate if
this commit will affect testing results to roundoff [non-BFB]
               !!  or climate changing [CC].  Use [BFB] if-and-only-f
commit is bit-for-bit and
                              !!  you know all the tests will pass
without regenerating baselines.
[FCC]                         !!  Add [FCC] if the commit will change
climate if a flag is activated.
[NML]                         !!  Add [NML] if the commit introduces
changes to the namelist.

LG-92 (if there are also JIRA issue numbers, add those below github
issues and BFB keys.)

(If the merge produces merge conflicts,the list of files that were in
conflict should be left in the message.)
```

Answer-changing commits

For a higher level view of ACME master, see **Existing and Planned Tags and Branches**

This is a record in Confluence of all the commits to ACME master that "changed answers", either round-off or climate-changing, and required testing baselines to be regenerated. Rebaselining should only be done following: **Guidelines for rebaselining ACME tests**

Date = date the change "graduated" to master. Use Confluence date format.

Brief description = write a description someone in another ACME science group could understand. Do not bother explaining "why" just focus on the "what". If the change is bug fix, just link to the Github issue describing the bug.

hash (on master) = link to the hash of the merge commit on master for the change. It will have the form "https://github.com/ACME-Climate/ACME/commit/<hash>"

Code Review Page = a link to the specific page in Confluence documenting the Code Review for this change (if available)

POC = the developer or task lead responsible for the Code Review. Integrator if no code review available.

Date	Brief description of change	hash/tag (on master)	Code Review Page	POC
20 Apr 2015	ACME v0.3 tag created	v0.3		
27 Jul 2015	Replace fsurdat in default namelist.	d1d2563	Generating new surface data including soil order for phosphorus cycle	Xiaoying Shi
27 Jul 2015	NCAR reported bugfix for micro_mg_cam.F90 [Climate Changing]	3ecf3b5	NCAR reported bugfix for micro_mg_cam.F90	Balwinder Singh
28 Aug 2015	Switch 6 integration tests from FV to SE	566d63	(none)	Balwinder Singh
02 Oct 2015	Introduce CIME with latest coupler	186b37	Incorporation of CIME Infrastructure into ACME	Robert Jacob
21 Jan 2016	Updated RSF input file for MAM packages	e61c667	(none)	Susannah Burrows
28 Jan 2016	A bug fix for hetfrz_classnuc from NCAR	5081c4c	(none)	Balwinder Singh
09 Feb 2016	Change default coupling sequencing to CESM1_MOD	938cb76	(none)	Patrick Worley

Guidelines for rebaselining ACME tests

Draft version - open for comments 2015-10-1

The ACME testing system includes several types of functional tests, such as testing if the restart capability is working. It also has "comparison" tests, which compares simulation output with baseline output, where the baseline output was produced by a "blessed" version of the code. If the new code produces output which is not BFB identical with the baseline output, the test will fail. There are many issues which can cause these failures, and this confluence page gives the guidelines for what is needed to accept the new code, and "rebaseline" our tests.

1. **Test failures where we are confident the climate is unchanged**
    a. Example: adding additional fields to the output or removing no longer needed fields, and the unchanged fields remain BFB with the baseline
    b. Example: removing tests, adding new tests, changing the resolution of existing tests
    c. integrator should follow the "**rebaseline - obvious"** protocal
2. **Roundoff level changes**
    a. Many types of changing only effect and roundoff error levels: loop ordering, compiler options, order of arithmetic. These changes are not expected to change the climate, but because of the chaotic nature of the climate system, will cause all comparison tests to fail.
    b. The developer should have a good reason to believe that their code changes are roundoff level only. For example, they believe they have not changed the algorithm, only restructured it for reasons such as better performance or modularity.
    c. If the effects are limited to a single component, the developer should follow procedures developed by the component group and have the results approved by the component group leads. This will serve as a sanity check to make sure they did not introduce a bug by accident.
    d. If the impact effects all components, such as changes in the coupler, the developer should follow procedures developed by the coupled simulation group and have the results approved by the coupled simulation group leads.
    e. The integrator should follow the "**rebasline - roundoff"** protocol.
        i. Example: Recent changes in the atmosphere dycore, where the monotone limiter was rewritten for better vectorization,resulted in roundoff level differences. Standalone dycore tests were used to verify that the changes were consistent with roundoff level changes: The L2 error in idealized test cases was not changed, and the plots of the fields looked identical. Then two FC5 (AMIP w/cyclic year 2000 conditions) were performed, with the old and new code. AMWG diagnostics were computed and the atmosphere group leads looked at the differences and determined, through expert judgement, that the differences were consistent with roundoff level changes.
        ii. Example: the CIME merge. CIME standalone tests suggests the coupler is working correctly and all code changes have resulted in at most roundoff level answer changes. But because this will change the results of all ACME simulations and effect all components, and we have no tests that verify that CIME is correctly interfaced to ACME, the coupled group will develop a protocol to verify that the changes in the coupled model are consistent with roundoff level changes in the coupler. An example could be to perform a B20TRC5 simulation with the old and new code and then use expert judgement to determine if the differences between these simulations are acceptable.
        iii. ACME is developing more sophisticated tests that may automate this process: such as growth tests and statistical ensemble tests.
3. **Bug fixes**
    a. Bug fixes will often result in small changes in the simulated climate, and similar to roundoff level changes, they will cause all comparison tests to fail.
    b. Minor bug fixes: meaning no additional changes are needed
    c. Medium bug fixes: Re-tuning will be necessary, by the climate is still expected to be acceptable
    d. Major bug fixes: Large changes in the climate are expected and other processes may need to be modified.
    e. The developer will get approval from the component group leads before issuing the pull request.
    f. The integrator should follow the "**rebasline - bugfix**" protocol
    g. For Major bugs: this will impact all other component models and could make it so the ACME master branch is not usable. These changes should be approved by the council + group leads telecon.
4. **New components**
    a. Bringing in a new component, if done in a way which does not impact existing components, will not impact existing tests and thus rebaselining is not necessary.
    b. A new component should include a new test, in which case the baseline tests do need to be augmented with the new results - the integrated should follow the "**rebaseline - obvious**" protocol.
5. **New features**
    a. Bringing in a new feature that is disabled by default will not impact existing tests and rebaselining is not necessary
    b. If the feature is enabled be default, then it must have gone through the ACME code review policy and the integrator should follow the "**rebaseline - new feature**" protocol.

Rebaseline Protocols

1. Rebase-Obvious:
    a. The SE group will generate new baselines based on the integrator's request.
2. Rebase-Roundoff:
    a. The developer will update: Answer-changing commits and make a note to indicate that the change is roundoff only and not

expected to change the climate
   b. The integrator will confirm with the developer that the necessary tests to very roundoff level only changes have been performed, and that the component group leads have approved this request. (as described above in "2. Roundoff level changes")
   c. The SE group will verify that Answer-changing commits has been updated and then generate new baselines based on the integators request
3. Rebase-Bugfix
   a. The developer will update: Answer-changing commits and make a note to indicate that the change is a bug fix and note if is is minor, medium or major.
   b. The integrator will confirm with the developer that the necessary group lead approvals have been obtained (as described above in "3. Bug fixes")
   c. The SE group will verify that Answer-changing commits has been updated, and then generate new baselines based on the integrators request. For major bugs, the SE group will wait for the developer to notify ACME-all before rebaselining.
4. Rebase-NewFeature
   a. The developer will update: Answer-changing commits including the link to the **approved** design document Code Review Process Implementation
   b. The SE group will verify that Answer-changing commits has been updated, and then generate new baselines based on the integrators request.

Externals in the ACME code

Some of the subdirectories in the ACME code contain code that was brought in from another repository. List them all.

Terms

- **external** any subdirectory in ACME whose entire content is from another repository
- **upstream master** the repository everyone is sending their changes to. Only one so that everyone pulling from the same source is working with all the same changes. The source of an external.

Methods for including externals

- **submodule** a link that points to another git repository. See Sharing External Code using Git Submodules . Submodules require an extra step after cloning to get the code.
- **subtree** a set of files and commits from another git repo, brought in to your repo. See Sharing External Code using Git Subtree. Subtrees are included in ACME when you "git clone".
- **monorepo** adding multiple projects, related or not, in the same repo. Code is included when you clone ACME.
- **buildtime** the repo is checked out to the $CIME_OUTPUT_ROOT/$CASE/bld when compiling a case for the first time. Code does not exist in ACME repo.

Rules for adding externals

1. Group leads must approve adding a new external. The group needing the external can decide on the method of inclusion. A POC for the upstream master must be identified and the SE group leads notified.
2. The upstream master for the external must be build-able and testable on its own.
3. For submodules, the linked repo must be readable by all ACME developers. If the upstream master can not be made readable, a fork must be maintained in ACME-Climate github and ACME then links to that fork.

Development in externals

In general, there are several ways to develop with external code in ACME

- strict 1-way: all development is merged to the upstream master first and flows down to ACME. No exceptions.
- mostly 1-way: prefer strict 1-way but some exceptions can be made for emergency fixes
- 2-way: developers can make changes in ACME or the upstream master. POC keeps them in sync.

More specific instructions for externals are below. **NOTE: a commit should never mix external and non-external code.**

Table of Externals currently in ACME

Name	Subdirectory in ACME	external method	upstream master	development method	POC	I
CIME	cime	subtree	https://github.com/ESMCI/cime	2-way	James Foucar	
MCT	cime/externals/MCT	subtree	https://github.com/MCSclimate/MCT	mostly 1-way	Robert Jacob	u
PIO	cime/externals/pio*	subtree	https://github.com/NCAR/ParallelIO	mostly 1-way	Jayesh Krishna	u
MPAS-Ocean	components/mpas-o/model	submodule	https://github.com/MPAS-Dev/MPAS forked to https://github.com/ACME-Climate/MPAS	strict 1-way	Mark Petersen	
MPAS-Land Ice	components/mpasli/model	submodule	https://github.com/MPAS-Dev/MPAS forked to https://github.com/ACME-Climate/MPAS	strict 1-way	Matt Hoffman	
MPAS-Sea Ice	components/mpas-cice/model	submodule	https://github.com/MPAS-Dev/MPAS forked to https://github.com/ACME-Climate/MPAS branch cice/develop	strict 1-way	Adrian Turner	
FATES	components/clm/src/external_models/fates	submodule	https://github.com/NGEET/fates forked to https://github.com/ACME-Climate/fates	strict 1-way	Gautam Bisht	

MPP	components/clm/src/external_models/mpp	submodule	https://github.com/ACME-Climate/mpp branch alm/develop	strict 1-way	Gautam Bisht	
CVMix	NA	buildtime	https://github.com/CVMix/CVMix-src	strict 1-way	?	A $
ocean BGC	NA	buildtime	https://github.com/ACME-Climate/Ocean-BGC	strict 1-way	?	A $

Externals planned for ACME

Name	Subdirectory in ACME	external method	upstream master	development method	POC	Notes
CICE column physics	TBD	TBD	?	?	?	
sBeTR for land model	components/clm/src/external_models/sbetr	submodule	https://github.com/ACME-Climate/sbetr	strict 1-way	Gautam Bisht	
GCAM	TBD	submodule?	https://github.com/JGCRI/gcam-core	strict 1-way	Katherine Calvin	

Specific external development instructions

CIME:

`Code location:  ACME/cime`

Any code development in ACME/cime/config/acme can be committed directly to ACME through the normal PR process.

Code development in other parts of CIME should be committed first with a PR in https://github.com/ESMCI/cime. It will be then brought to ACME by a subtree merge as in Maintaining the CIME subtree in ACME.

ACME staff can report any bugs in CIME at https://github.com/ACME-Climate/ACME/issues. SE/CPL staff may open a companion report at https://github.com/ESMCI/cime/issues. ACME cime bugs which have been fixed in ESMCI and are waiting a subtree merge will have the github label "Fixed in ESMCI" attached to them.

ACME Code Development Process for Collaborators

The goal of the ACME Code Development Process is to achieve high developer productivity towards the improvement of the ACME earth system model. A degree of formality is needed because of the large size of the ACME team, the geographic distribution of the team, and large scope of the ACME project, particularly when including collaborations.

The process is meant to promote the quality of the ACME model along many dimensions: high-quality science, verified implementations, high-performing implementations, portability to DOE computer architectures, and maintainability/extensibility by those other then the original developers. The process intends to give individuals and small teams the ability to independently develop new features and algorithms, yet with a clear path to incorporation into the ACME model. The ACME Software Engineering (SE) team will help orient and guide new collaborators, but the expectation is that collaborators (as well as ACME developer team members) achieve a level of expertise in code practice that can be sustained and maintained across the team without requiring SE-team intervention.

Main steps of the ACME code development process:

1. Code development should begin from the head of the master branch or most recent release branch of ACME from the github repository
2. Code should be of high enough quality to be readable, modifiable, and maintainable by others.
3. Code development should follow ACME standards for using git, *e.g.* preparing a separate commit for each stand-alone contribution along with a well-formed commit message.
4. Developers must run the ACME test suite to verify that their development has not unintentionally changed the code behavior, and all new development must be accompanied by tests to protect the new feature against future developments.
5. Code development should be incorporated into, or rebased to, the ACME code base at least every few months (since ACME has no resources to reconcile code that has been on diverging development paths).
6. For eventual inclusion of a new feature or algorithm into the ACME Model, documentation of the main steps of the ACME Code Review process should be created:
    a. A design document (or paper) detailing the equations/algorithms that are being implemented
    b. Verification evidence that supports that the implementation is correct
    c. Performance analysis and data showing the expected and measured performance impact of the new feature
    d. Validation evidence that the feature matches observational data

Freezes and tags

Definitions

Before a freeze: science runs (to be done with a tagged release) are planned, features needed for those runs are identified, roadmaps are made to develop those features, people are tasked to develop the features. Developers will ensure new features work with existing master.

Feature freeze:

Date at which all work on writing new features for the next release is halted and "pull requests" are made to merge those features to master. Effort shifts towards integrating the completed features in to master and fixing bugs. A feature freeze improves stability by preventing the addition of new features which may not be complete or sufficiently tested and/or may have unexpected interactions; thus, a feature freeze helps improve the program's stability.

Configuration freeze:

Date at which all work to define the science configurations (compsets and resolutions) available from the new features is suspended, shifting the effort towards tuning the compsets for production runs. Changing science configurations that are being actively tuned may have disruptive effect on tuning, due to the introduction of new interactions between frozen features. Boundary and initial condition files are also frozen. Parameter values in code or scripts may continue to be changed as tuning requires.

Code freeze:

Date at which no changes whatsoever are permitted to a portion or the entirety of the program's source code. Particularly in large software systems, any change to the source code may have unintended consequences, potentially introducing new bugs; thus, a code freeze helps ensure that a portion of the program that is known to work correctly will continue to do so. Code freezes are often employed in the final stages of development, when a particular release or iteration has finished testing and production runs have begun, but may also be used to prevent changes to one portion of a program while another is undergoing development.

- Science code freeze: No changes at all to the science code including compsets and parameter values. May still have changes to script system (machine files, test lists) and internal documentation.
- Full code freeze: No changes whatsoever to any code.

Freeze schedule

Explanation of what is frozen when. Written for v1.0.0 but applicable to any ACME version. **Focused on science code.**

Development stage	What can change or What happens	What cannot change
Before Feature Freeze	Any code that implements a new feature	NA
After feature freeze and during integration stage	Bug fixes. BFB, non-BFB or CC changes to completed features. Script system changes.	Code that implements new science features.
Feature integration completed	First alpha tag	
After first alpha tag and before configuration freeze	Bug fixes, BFB changes. non-BFB or CC changes IF approved by coupled system tuning group.	Code that implements new science features.
After configuration freeze and start of tuning.	BFB changes. Bug fixes and other changes IF approved by coupled system tuning group.	Code for compsets being tuned.
Alpha testing completed. Model does not blow up for all science configurations	First beta tag.	
After first beta tag.	BFB changes. Bug fixes in science code IF AND ONLY IF approved by coupled simulation group.	Anything else.
Beta testing completed	First release candidate (rc) tag. Science code freeze.	
Production runs with release candidates.	Only ***critical*** bugs found in production sims AND approved by coupled simulation group. BFB script changes.	Anything else.

release candidate after production runs	Nothing. Full code freeze.	Everything.
release	Make vX.0.0 tag.	

E3SM Input Data Servers

This page provides information on the E3SM Input Data Servers including how to add data.

Background

The CIME build/configure scripts scan the requested model configuration after a case is created and, during the case.submit stage, download any necessary data (initial condition files, boundary condition data sets, mapping files, etc.) from mutiple servers, each with the same directory structure. The current collection of this data for all possible configurations is ~5TB and is split across several servers as explained below This automatic download feature means that each user does not need to mirror the entire 5TB input dataset. CIME will only download exactly what is needed for the simulations run on that machine. Typically all the users on a machine will use one shared input data directory, such as /projects/ccsm/inputdata, and all files will be downloaded to this directory. This has the additional advantage that each file is only downloaded once per machine even if needed by multiple users. If the file already exists on a machine, it will not be downloaded again.

We refer to the shared directory as the "inputdata directory" below.

Data servers in order of precedence

The servers to search are listed in https://github.com/E3SM-Project/E3SM/blob/master/cime/config/e3sm/config_inputdata.xml. Several methods are supported including svn, ftp, wget and gridftp.

In order, the servers for E3SM input data are:

https://web.lcrc.anl.gov/public/e3sm/inputdata/

https://svn-ccsm-inputdata.cgd.ucar.edu/trunk/inputdata

They are both world readable and require no authentication.

E3SM input data server policy

1. **When you have a new input data file:**
    a. it should be uploaded to Blues in the desired subdirectory of /lcrc/group/acme/public_html/inputdata. From this location it will be immediately available by the http server. web.lcrc.anl.gov . In addition to uploading the file, file permissions need to be updated (via chmod go+r <filename>). Once this is done, the new files will be available to all E3SM staff and the public. If you created a new directory, make sure it has world read/execute permission: chmod go+rx <dirname>
    b. If necessary, make code modifications (on a branch, like any development) to read the new files. The files should be uploaded before a Pull Request is issued that will need the new files.
2. **If you are a creator of input data files**: get an account on blues to avoid unnecessary roadblocks in adding files to the http server. See AccesstoBluesserver below.
3. **Never replace or overwrite existing input data files on the server or locally**. Files usually contain a creation date in the filename. When adding a new file similar to a previously existing file, even if only metadata is changed, update this date stamp and make a new file so that both files remain on the server.

We do not add data to the CESM server. Its included in our list for some test cases that still read files from that server.

E3SM local inputdata directory policy

As explained above, each supported machine has one shared input data directory that all users read from. All users should ideally also have write permission so they can download new files as needed.

Known locations of local inputdata directory

The current default locations where E3SM will search for inputdata directories can be found in: https://github.com/E3SM-Project/E3SM/blob/master/cime/config/e3sm/machines/config_machines.xml in a variable called DIN_LOC_ROOT

Below is a table of known inputdata directory locations on machines being used within E3SM. If the directory location is unknown, contact machine POC for assistance (please refer to Configuration Management page for authoritative list of machine POCs).

Location	Machine name	Inputdata Administrator (not necessarily the same as machine POC)	Location of inputdata directory	Write Access (needed for automatic downloading of missing files)	Updated for E3SM in config_machines.xml ?
NERSC	edison, cori	Noel Keen	/project/projectdirs/acme/inputdata	All E3SM developers	Y
ALCF	mira, cetus	Jason Sarich	/projects/ccsm/inputdata	Any E3SM developer, by request to Mira POC	Y
OLCF	titan and eos	Min Xu	/lustre/atlas/world-shared/cli900/cesm/inputdata	Restricted - see below on how to get files added	Y
ANL	blues (anvil), bebop	Jason Sarich	/home/ccsm-data	All E3SM developers; make sure you're in the "climate" unix group.	Y
PNNL	cascade	Balwinder Singh	/dtemp/sing201/acme/inputdata/	PNNL E3SM developers	Y

Policies for local inputdata dir on open systems (e.g., Mira, many institutional clusters)

1. On many systems, especially institutional clusters, the "inputdata directory" is write accessable by one or more unix groups. This is the only way the CIME automatic download scripts will function, since the user needs to be able to add files to this directory.
2. Thus E3SM developers have write access to this shared directory, and they should only ever add files to this shared directory through CIME's automatic scripting feature which is done during case.submit. On these systems, it is assumed that E3SM developers are knowledgeable enough to know not to modify files in the inputdata directory.
3. This also applies to CESM users, on systems used by both CESM and E3SM.
4. If developers want to test different versions of input data files without making them officially part of E3SM, they should use CIME's "user_nl_*" feature, which allows one to specify alternative files for any of the inputdata files read through a namelist variable. Developers can thus have local versions of these files that they are free to modify without impacting other users.

Policies for local inputdata dir on locked-down systems (e.g., Titan)

1. On Titan, and perhaps other systems, inputdata directory write access is controlled by a limited number of users.
2. On these machines, the CIME automatic download capability will not function, since the user cannot add files to inputdata
3. On Titan, the inputdata now mirrors the entire E3SM server at LCRC. (08/01/2018)
4. If an E3SM user wants to use a file not present in "inputdata", they have two options:
    a. Add it to the E3SM server at LCRC and wait for "inputdata" to be updated automatically. It usually takes one day but could be longer if the systems and/or network are unavailable.
    b. Put it someplace locally, and specify it in E3SM's "user_nl_*" capability for each case.

Adding a new default inputdata directory location to E3SM's machine-specific settings

To provide a new default inputdata location on a specific machine, edit the file

`cime/config/e3sm/machines/config_machines.xml`

Inside of the entry for your machine, edit the variables

`<DIN_LOC_ROOT>`

and

`<DIN_LOC_ROOT_CLMFORC>`

to contain the appropriate path to the default inputdata directory location and the default CLM datm (data atmosphere) location.

Getting access to the data servers

Our 2 E3SM servers have different methods for access.

Access to Blues server

The Blues http server is world readable. For write access, you need an account on Blues. Follow the same instructions given for accessing Blues for computing.

https://acme-climate.atlassian.net/wiki/spaces/Docs/pages/98992379/Anvil+-+ACME+s+dedicated+nodes+hosted+on+Blues.#Anvil-ACME'sdedicatednodeshostedonBlues.-GettinganAccount

Once you have access to blues, you will also have read/write access to the directory files are served from: /lcrc/group/acme/public_html/inputdata/. This is just a normal filesystem and directory structure and you can copy files to it or remotely copy files using scp or globus online.