A *git* Workflow For Reproducible Research

Josh Sonstroem, DCO
jsonstro@ucsc.edu

Sponsored by: Hummingbird Zoom Clinic

[Seal of the University of California]
What Is Version Control and Why To Use It

*Version control* is a system that records changes to a file or set of files over time so that you can recall specific versions later.

Using a *Version Control System* or *VCS* is a *very wise* thing to do. It can allow you to revert files back to a previous state, reset an entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem and when that change was made, as well as providing implicit backups.*

*Not if you are gitlab, where 6 backup technologies failed in Feb 2017, but alas...
What Is Version Control and Why To Use It

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

Using a Version Control System (VCS) is a very wise thing to do. It can allow you to revert files back to a previous state, reset an entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem and when that change was made, as well as providing implicit backups.

*Not if you are gitlab, where 6 backup technologies failed in Feb 2017, but alas...
The 3 Types of Version Control

In general, all **VCS** fall into 3 broad categories:

1. **Local Version Control** - e.g. **rcs**, naive, [single] local copy
2. **Centralized Version Control** - e.g. **svn**, hierarchical control, clients checkout files, central server [as single point of failure]
3. **Distributed Version Control** - e.g. **git**, 1-to-many servers, each is a full clone, allows for decentralized models of control/distribution

The third category -- a distributed version control system or DVS for short -- is clearly ideal for projects like high-performance computing and scientific reproducibility, allowing for collaboration across diverse groups of people in multiple ways simultaneously within the same project. Many codebases for collaborative science are already available on public source code repositories such as **Github, Gitlab, Sourceforge, Bitbucket**, etc...
The 3 Types of Version Control

1. Local Version Control

**DOOM's Development: A Year of Madness**

https://www.youtube.com/watch?v=eBU34NZhW7I @ 42:24

An interview with Doom’s lead programmer John Romero from the 2018 WeAreDevelopers Conference
The 3 Types of Version Control

1. Local Version Control

DOOM's Development: A Year of Madness

https://www.youtube.com/watch?v=eBU34NZhW7I @ 42:24
The 3 Types of Version Control

1. Local Version Control

**DOOM's Development: A Year of Madness**
https://www.youtube.com/watch?v=eBU34NZhW7I @ 42:24

Developed from the way versions of files were stored by appending filenames with timestamps, e.g. `GetOTP_9Jan93.java`, `GetOTP_12Jan93.java`, etc.

Local VCS improved upon this crude approach by storing just the difference between these files in a database.

Thus, the first version would be the actual file but each successive version would correspond to the difference between the current and the last version. Difference between two such versions are called **patch-sets**. A local database was used to track changes by storing the patch-sets. Any particular version could be recreated by adding the patch-sets up to that version.
The 3 Types of Version Control

1. Local Version Control

**Benefits:**
- Less space used than many copies
- Allows for custom versions by combining non-linear patch-sets

**Drawbacks:**
- Only really useful to a single developer at once
- Susceptible to many failure-modes
The 3 Types of Version Control

1. Local Version Control

**Benefits:**
- Less space used than many copies
- Allows for custom versions by combining non-linear patch-sets

**Drawbacks:**
- Only really useful to a single developer at once
- Susceptible to many failure-modes
The 3 Types of Version Control

2. Centralized Version Control

Relies on a client/server relationship. Like FTP the repository is located on one server and provides access to many clients. All changes, users, commits and information are sent-to/received-from this central repository.

The primary benefits of a centralized approach are:
- It is easy to understand
- More control over users and access (since it is served from one place)
- More GUI & IDE clients (Subversion has been around longer)
- Simple to get started
The 3 Types of Version Control

2. Centralized Version Control

Some drawbacks:

- Dependent on access to the server
- Hard to manage a server and backups
- It can be slower because every command connects to the server
- Branching and merging tools are difficult to use
The 3 Types of Version Control

2. Centralized Version Control

Some drawbacks:

- Dependent on access to the server
- Hard to manage a server and backups
- It can be slower because every command connects to the server
- Branching and merging tools are difficult to use
The 3 Types of Version Control

3. Distributed Version Control

Distributed VCS are a newer option. In a DVS, each user has their own copy of the entire repository, not just the files but the ENTIRE history as well. Think of it as a network of individual repositories. In many cases, even though the model is distributed, services like Beanstalk are used for simplifying the technical challenges of sharing changes.

The primary benefits of a distributed approach are:

- Powerful and detailed change tracking, which means **fewer conflicts**
- No server necessary – all **actions** except sharing repositories are **local**
- Branching and merging are more **reliable**, and thus, used more often
- It’s **fast**

---

Git development began in April 2005, after many developers of the Linux kernel gave up access to BitKeeper, a proprietary source-control management (SCM) system that they had been using to maintain the project since 2002. The copyright holder of BitKeeper, Larry McVoy, had withdrawn free use of the product after claiming that Andrew Tridgell had created SourcePuller by reverse engineering the BitKeeper protocols. The same incident also spurred the creation of another version-control system, Mercurial.
The 3 Types of Version Control

3. Distributed Version Control

Some drawbacks:

- The distributed model is harder to understand
- It’s new, so not as many GUI clients
- The revisions are not incremental numbers, thus harder to reference
- Easier to make mistakes until you are familiar with the model
The 3 Types of Version Control

3. Distributed Version Control

Some drawbacks:

- The distributed model is harder to understand
- It’s new, so not as many GUI clients
- The revisions are not incremental numbers, thus harder to reference
- Easier to make mistakes until you are familiar with the model
The 3 Types of Version Control

3. Distributed Version Control

Some drawbacks:

- The distributed model is harder to understand.
- It’s new, so not as many GUI clients.
- The revisions are not incremental numbers, thus harder to reference.
- Easier to make mistakes until you are familiar with the model.
Aside: The 3 Copies of a git Checkout

Git works a bit different than other, past VCS’s. Rather than tracking changes to each file linearly, each git commit is actually a binary snapshot of the git “filesystem” in the checkout directory.

Since its distributed, each checkout of the repo is a full clone, unlike centralized VCS’ where the central repo may be the only full copy. Somewhat confusingly there are 3 copies of a repo in your git checkout: remote, local, and working.

When you make file changes you are in the working copy. When you merge those changes locally you are editing the local copy. And finally, once you have your pull request made, your commits will be merged by the release team(s) into the remote copy. This allows for multiple remotes and flexible distribution of control.
Aside: The 3 Copies of a *git* Checkout

*Git* works a bit different than other VCS's. Rather than tracking changes to each file linearly, each *git* commit is actually a binary snapshot of the *git* “filesystem” in the checkout directory.

Since its *distributed*, each checkout is a full clone, unlike centralized VCS' where the central repo may be the only full copy. Confusingly there are 3 copies of a repo in your *git* checkout:

- **Remote**
- **Local**
- **Working**

When you make file changes you are in the **working copy**. When you **merge** those changes locally you are editing the **local copy**. And when you **pull** your commits into your local copy, your **commit** will be merged by the release team(s) into the remote copy. This allows for multiple remote copies by team(s) of control.
Aside: The 3 Copies of a git Checkout

Git works a bit differently than other VCSs. Rather than tracking changes to each file linearly, each commit is a binary snapshot of the 'filesystem' in the checkout directory.

Since its distributed nature, each checkout of the repo is a full clone, unlike centralized VCSs' where there is only one full copy. Somewhat confusingly, there are 3 copies of a repository in your git checkout: remote, local, and working.

When you make file changes you are in the working copy. When you merge those changes locally you are editing the local copy. And finally, once you have your pull request made, your commits will be merged by the release team(s) into the remote copy. This allows for multiple remotes and flexible distribution of control.
What *git* service is right for me?

Although *git* is a local command line application, for ease of use and to both backup and share your repository to collaborate with others you probably want use a hosted *git* service.

**Private UCSC Gitlab service** - [https://git.ucsc.edu](https://git.ucsc.edu)
- **Benefits:** it is hosted on-campus and offers private repositories for protected research information and it has some usage limitations (non-commercial work only)
- **Drawbacks:** since it requires a UCSC email to sign-up collaborating with remote colleagues can be a challenge
- **More info available here** → [https://its.ucsc.edu/gitlab/](https://its.ucsc.edu/gitlab/)
What git service is right for me?

Although git is a local command line application, for ease of use and to both backup and share your repository to collaborate with others you probably want to use a hosted git service.

Private UCSC GitLab

- **Benefits:** it is hosted on-campus and offers private repositories for protected research information and it has some usage limitations (non-commercial work only)
- **Drawbacks:** since it requires a UCSC email to sign-up collaborating with remote colleagues can be a challenge
- **More info available here**

**UCSC GitLab**

- **Benefits:** it is hosted on-campus and offers private repositories for protected research information and it has some usage limitations (non-commercial work only)
- **Drawbacks:** since it requires a UCSC email to sign-up collaborating with remote colleagues can be a challenge
- **More info available here**

**Private UCSC GitLab**

- **Benefits:** it is hosted on-campus and offers private repositories for protected research information and it has some usage limitations (non-commercial work only)
- **Drawbacks:** since it requires a UCSC email to sign-up collaborating with remote colleagues can be a challenge
- **More info available here**
What *git* service is right for me?

Although *git* is a local command line application, for ease of use and to both backup and share your repository to collaborate with others you probably want use a hosted git service.

Private **UCSC Gitlab service** - [https://git.ucsc.edu](https://git.ucsc.edu)
- **Benefits**: it is hosted on-campus and offers private repositories for protected research information and it has some usage limitations (non-commercial work only)
- **Drawbacks**: since it requires a UCSC email to sign-up collaborating with remote colleagues can be a challenge
- **More info available here** → [https://its.ucsc.edu/gitlab/](https://its.ucsc.edu/gitlab/)

Public **Github service** - [https://github.com](https://github.com)
- **Benefits**: since the service is publicly available on the internet collaboration with external colleagues and returning forked work to original projects is very easy
- **Drawbacks**: since the free versions only allow public repos there are very real concerns around data protection and privacy and these impact research restrictions and data security
What git service is right for me?

Although git is a local command line application, for ease of use and to both backup and share your repository to collaborate with others you probably want to use a hosted git service.

Private UCSC Gitlab service - https://git.ucsc.edu

- **Benefits:** it is hosted on-campus and offers private repositories for protected research information and it has some usage limitations (non-commercial work only)
- **Drawbacks:** since it requires a UCSC email to sign-up collaborating with remote colleagues can be a challenge
- **More info available here** → https://its.ucsc.edu/gitlab/

Public Github service - https://github.com

- **Benefits:** since the service is publicly available on the internet collaboration with external colleagues and returning forked work to original projects is very easy
- **Drawbacks:** since the free versions only allow public repos there are very real concerns around data protection and privacy and these impact research restrictions and data security
What is Reproducible Research?

“An article about computational results is advertising, not scholarship. The actual scholarship is the full software environment, code and data, that produced the result.” – Claerbout and Karrenbach, 1992, “Electronic Documents Give Reproducible Research a New Meaning”

➢ paper is available
➢ code is available
➢ data is available

Your research is considered reproducible if someone with access to your raw data, your code, and your environment (hardware and software) can generate your results (tables and figures). Your code should turn raw/original data into final results.

More info: https://www.nature.com/collections/prbfkwmwvz/#/
What is Reproducible Research?

“An article about computational results is advertising, not scholarship. The actual scholarship is the full software environment, code and data, that produced the result.” – Claerbout and Karrenbach, 1992, “Electronic Documents Give Reproducible Research a New Meaning”

➢ paper is available
➢ code is available
➢ data is available

Your research is considered reproducible if someone with access to your raw data, your code, and your environment (hardware and software) can generate your results (tables and figures).

More info: https://www.nature.com/collections/prbfkwmwvz/#/
In what ways does a VCS impact my scientific research?

Version control recommendations for reproducible scientific research:

1. Encapsulate the full project into one directory
2. Document everything and use code as documentation
3. Make figures, tables, and statistics the results of scripts
4. Write code that uses relative paths
5. Always set your seed for randomization
6. Release your code and data

Source: [https://rpubs.com/marschmi/105639](https://rpubs.com/marschmi/105639)

By following these few simple guidelines you can ensure your hard-earned science workflow goes from being simply *advertising* to becoming a *breakthrough* ;^)
Organizing your research for a DVS

Organize data, code, and dependencies
- Encapsulate everything
- Separate raw data from derived data
- Separate data from code
- Use relative paths
- Write readme files (document everything!)
- Backup your derived data at multiple sites!!
- Commit often and sync to a remote VCS service

Considerations:
- being able to reproduce own results at a later date
- manage changes to data, analysis and results
- satisfy journal requirements

Source: http://kbroman.org/steps2rr/
Is your research robust?

Five recommendations for robust research:
1. Write code for humans, write data for computers
2. Make incremental changes
3. Make assertions and be loud, in code and in your methods
4. Use existing libraries (packages) whenever possible
5. Prevent catastrophe and help reproducibility by making your data read-only

→ Research science is one of the most multi-disciplinary jobs in ANY field!

Not only must you know your science forward and backwards, but you need to be able to use the necessary instrumentation, support IT systems, be knowledgeable about software design, data science and programming best practices, as well as be versed in the legal, political and social issues surrounding your field of study.

Learning what resources are available to edify and validate your own understanding of these complexities is both paramount and absolutely required for successful reproducible research in both the present science climate and for future generations.
VCS for Research checklist

1. Was as much as possible done by the computer?
2. Was any file hand-edited, or any part of the analysis done by hand?
3. Is everything documented, including the software environment?
4. Was a version control system used?
5. Have we saved any output that we cannot reconstruct from original data and the code?
6. How far back in the analysis pipeline can we go before our results are no longer automatically reproducible?

It’s always a good idea to call `sessionInfo()` in your R code!

ALWAYS BACKUP YOUR DERIVED DATA!!!
1. Was as much as possible done by the computer?
2. Was any file hand-edited, or any part of the analysis done by hand?
3. Is everything documented, including the software environment?
4. Was a version control system used?
5. Have we saved any output that we cannot reconstruct from original data and the code?
6. How far back in the analysis pipeline can we go before our results are no longer automatically reproducible?

It’s always a good idea to call `sessionInfo()` in your R code!

ALWAYS BACKUP YOUR DERIVED DATA!!!
Additional Resources for Research Best Practices:
2. “Bioinformatics Data Skills” book by Vince Buffalo (and it’s [github page](https://github.com/))
4. *Nature* Journal on reproducible research - [www.nature.com/collections/prbfkwmwz/#/17](https://www.nature.com/collections/prbfkwmwz/#/17)

Recommended Data Repositories:
[https://www.nature.com/sdata/policies/repositoriess](https://www.nature.com/sdata/policies/repositories)
Working with git

➔ **Fundamentals for source control of reproducible and robust research**
   What they are and how to use them

➔ **VCS collaboration strategies**
   What is the best way to work with my team, group, collaborators and the public?
Identify yourself
Always set your name and email address for all commits:

% git config --global user.name "Me"
% git config --global user.email me@ucsc.edu
**git Fundamentals 1**

**Tracking Changes**

*Git* will show you what bytes in a file have changed. When you *clone* a repository from a host or create one locally you specify the set of files or directories you wish to track. As you make changes they are tracked behind the scenes until you are ready to *commit* those changes.

**Committing**

As you work with the files that are under version control, each change is tracked automatically. This includes modifying a file, deleting a directory, adding a new file, moving files about; basically anything that might alter the state of the underlying filesystem. Rather than recording each change individually, *git* waits for you to submit your changes as a single collection of actions which are called a *commit*. 
Revisions and Change-Sets

When a *commit* is made, the changes are recorded as a *change-set* and given a unique revision. This revision is a unique hash (like 846eee7d5c3a1e952d34a3dff3d341e5). Knowing the revision of a *change-set* it makes it easy to view or reference it later. A change-set will includes a reference to the person who made the commit, when the change was made, the files or directories affected, a comment and the changes that happened within the files (lines of code).

When it comes to collaborating with others, viewing past revisions and change-sets is a valuable tool to see how your project has evolved and for reviewing teammates’ code. *Git* has a formatted way to view a complete history (or *log*) of each revision and change-set in the repository. The *github/gitlab* GUI has its own powerful tool -- *Pull Requests* -- which we use for reviewing/approving code changes on UCSC projects.
Getting Updates

When collaborating with a team using git it is important to keep up with all published changes. Getting the latest code from a repository is as simple as doing a pull or update from the remote. When you do a pull only the changes since your last shared commit are downloaded. A fetch on the other hand only retrieves the metadata.

Conflicts

What if the latest pull or commit results in a conflict? That is, what if your changes are so similar to someone else's changes that the VCS can't automatically determine which is the correct and authoritative change? Git provides a way to view the difference between the conflicting versions: either edit the files manually to merge the options or choose one revision over the other. It is often a good idea to collaborate with the other person to make sure you're not undoing important work!
Diffing (or, Viewing the Differences)

Since each commit is recorded as a change to a file or set of files and directories, it can be useful to view what changed between revisions. For instance, if a recent deployment of your application is broken and you’ve narrowed down the cause to a particular file, git would allow you see the who/why/when/what recently changed in that file.

By viewing a diff, you can compare two files or more files to see what lines of code changed, when it changed and who changed it. With git you can compare not only two or more sequential revisions, but also any set of revisions from anywhere in the history of the whole project.
Branching and Merging

Sometimes you want to experiment with changes to the repo that could break things elsewhere (such as adding a new feature). Instead of committing this code directly to the main set of files (usually called master), you should create a new branch. A branch is basically a copy (or snapshot) of the repository that you can modify in parallel without altering the files in master. You can continue to commit new changes to your branch, while others commit to master without the changes affecting one another.

Once you’re comfortable with the experimental code, you will want to make it part of master again. This is where merging comes in. Since git has recorded every change so far, it knows how each file has been altered. By merging the branch with master (or even another branch), git will attempt to seamlessly merge each file and line of code automatically. Once a branch is merged, a push updates the remote copy.
Resolving Conflicts

Sometimes when you merge you will get a “conflict” message, if you do you’ll have to manually edit the conflicting parts (remove "<<<<<<<<<, ========= and >>>>>>>>") , then stage the file and commit. But don’t fret, take a deep breath and use your human ingenuity...

Unstaging a staged file:
% git reset HEAD file_to_unstage

To undo changes to a file which you modified and would like to revert to how it was at the last commit:
% git checkout -- file_name
VCS Collaboration Strategies...

Every project is *different*!

Collab-styles are project specific: talk with your collaborators!

Ask: when working in *this* group, what is the best way to share my work with others?

- Forking ← *most common for scientific collaboration*
- Shared checkout
- Single branch
- Project branches
- Environment branches
- *Github-flow*
- *Gitflow*
VCS Do’s and Don’ts for Research

Do
- Regularly commit and push your changes to a remote server
- Backup your derived data!
- Use pull-requests to merge your forked changes into upstream repositories

Don’t
- Place any research data inside a git repository
- Place large files inside a git repository
- Change the history on a branch that is checked out by others
Disclaimer!

Version control is a complex landscape inhabited by many varied, equally strong and equally valid approaches. This preso does NOT intend to tell you how to use git, but rather to outline some useful concepts, strategies, and best practices for collaborative use of a VCS.

My recommendation is for each research project to develop a collaboration strategy that works for their particular scope and type of project. Requesting repeated input from other project collaborators is essential to developing/maintaining a successful model!

Good news: You can now claim to be a certified Version Control Expert!!
KEEP CALM AND GO SLAY
Overview

From here-on this preso assumes you know:
what a project and a repo are, how to create, manage, checkout, sync, clone and commit code with git

➔ Branches or forks?
What they are and why science uses forks

➔ git workflows, which?
Forking, Centralized, Feature branch, Gitflow, Github-flow, oh my...

➔ Pull requests, use ‘em!
A github web interface for discussing proposed changes with the release team
To branch or to fork?
That is the question...

Reproducible research actually forces our hand a little bit...
Branches or Forking

Most collaborative scientific projects use forking.

The choice of a branching or forking model is more about how a project’s development is managed than any other factor. Forking ensures hierarchical control; with most open-source software, a benevolent-dictator rules over the “official” repo with an iron hand, controlling what code gets integrated into the project and when. Whereas a branching model allows for more fluid integration and distributed management of a project.

A fork is a copy of a repository (a clone on the server side). Forking is done through GitHub or Gitlab.
Branches or Forking

The choice of a branching or forking model is more about how a project's development is managed than any other factor.

Forking ensures hierarchical control; with most open-source software, a benevolent-dictator rules over the "official" repo with an iron hand, controlling what code gets integrated into the project and when. Whereas a branching model allows for more fluid integration and distributed management of a project.

A fork is a copy of a repository (a clone on the server side). Forking is done through GitHub or Gitlab.
Branches or Forking

The choice of a branching or forking model is more about how a project's development is managed than any other factor.

Forking ensures hierarchical control; with most open-source software, a benevolent-dictator rules over the "official" repo with an iron hand, controlling what code gets integrated into the project and when. Whereas a branching model allows for more fluid integration and distributed management of a project.

A fork is a copy of a repository (a clone on the server side). Forking is done through GitHub or Gitlab.
Forking Workflow

Forking is fundamentally different than the other workflows. Instead of using a single server-side repository as the “central” codebase, every developer uses a server-side repository. This means that each contributor has not one, but two repos: a private local one and a public server-side one.

The main advantage of Forking is that contributions can be integrated without the need for everybody to push to a single central repo, instead developers push to their own server-side repos, and only the project maintainer can push to the official repo. Maintainer can accept commits from any developer without giving them write access to the official codebase.
The result is a distributed workflow that provides a flexible way for large, organic teams (including untrusted third-parties) to collaborate securely. This also makes it an ideal workflow for open source projects including scientific research.

If you’re coming from an SVN background, the Forking Workflow may seem like a radical paradigm shift. But don’t be afraid—all it’s really doing is introducing another level of abstraction on top of Feature Branches. Instead of sharing branches directly through a single central repository, contributions are published to a server-side repository dedicated to the originating developer.
Pull Requests

Pull Requests are amazing. Many people use them for open source work - fork a project, update the project, send a pull request to the maintainer.

However, it can also be used as an internal code review system or as a branch conversation view with your collaborators since pull requests can be sent from one branch to another in a single project.

➔ **When to make a Pull Request**
   “I need help or review on this” or “Please merge this in”

➔ **Role of Release Team**
   To merge the code to master and tag
Pull Requests help smooth out the process of merging back into *master*.

Gitlab/Github both have a nice GUI interface for managing this process.

https://www.atlassian.com/git/tutorials/making-a-pull-request
Pull requests

Collaborate and improve code quality

Branch. Discuss. Merge. With pull requests, you're in control.

Create a pull request  Learn more
Create pull request

Title

test pull request

Description

* mobile
* mobile fix
* MFA Code pieces
* green
* More MFA files
* merging MFA and mobile
* mobile updates
* Dispute queue fix
* undo mobile
* More MFA changes
* Put moira conditionals back in to limit link display
* INC0372900 - Campus Directory AOE - add text
* More duo changes. This bundle also includes INC0374677
* dev and test environments both look at idmsctst
* .htaccess fix to allow for shib

Reviewers

Rex Core

Reviewers can approve a pull request to let others know when it is good to merge

Create Cancel
Create pull request

Select source and destination

![Diagram of repository branches]

**William Woodrow** committed 9a50a589a52 25 Apr 2016

**William Woodrow** committed 43076c9a7ea8 08 Feb 2017

<table>
<thead>
<tr>
<th>Author</th>
<th>Commit</th>
<th>Message</th>
<th>Commit date</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>William Woodrow</td>
<td>9a50a589a52</td>
<td>.htaccess fix to allow for shib</td>
<td>25 Apr 2016</td>
<td></td>
</tr>
<tr>
<td>William Woodrow</td>
<td>377159bf955</td>
<td>dev and test environments both look at idmsctst</td>
<td>25 Apr 2016</td>
<td></td>
</tr>
<tr>
<td>Idil Sabbagh</td>
<td>b2a4277fb31</td>
<td>More duo changes. This bundle also includes INC0374677</td>
<td>22 Apr 2016</td>
<td></td>
</tr>
<tr>
<td>Syrma Dontcheva</td>
<td>32f5352a9fc</td>
<td>INC0372900 - Campus Directory AOE - add text</td>
<td>22 Apr 2016</td>
<td></td>
</tr>
<tr>
<td>Idil Sabbagh</td>
<td>b8e4dca154</td>
<td>Put more conditions back in to limit link display</td>
<td>19 Apr 2016</td>
<td></td>
</tr>
</tbody>
</table>
Bitbucket now bundles all your pull request activity notifications into batch emails by default. Fewer emails, less distraction.

Learn more · Change back to immediate notifications

New activity on

**test pull request**

develop  master

---

**Rex Core**

Thanks, William. I will be able to review this work no later than Thursday afternoon, 02/23/17.

Reply · Like · 10:39 AM

---

**View pull request**

You can unwatch this pull request to stop receiving email updates.

Don’t want to receive batch emails anymore? Update your notification settings.
Bitbucket now bundles all your pull request activity notifications into batch emails by default. Fewer emails, less distraction.

Learn more · Change back to immediate notifications

New activity on
test pull request

develop master

Rex Core marked the pull request as APPROVED 11:17 AM

Rex Core MERGED the pull request 11:21 AM

View pull request

You can unwatch this pull request to stop receiving email updates.

Don't want to receive batch emails anymore? Update your notification settings.
# Research VCS Best Practices

- Regularly **commit** and **push** your code to a remote repository service such as [gitlab](https://gitlab.ucsc.edu)
- **Organize** and **separate** code & read-only data; **automate**, **script**, & **document** EVERYTHING!
- Use **forks** and **pull requests** to collaborate. Release both your **code** and **data** for **reproducibility**

---

*Reproducible-Science-Curriculum* Github repo for [Reproducible Research Project Initialization](https://github.com/Reproducible-Science-Curriculum) is a great place to start your reproducible research project repo.
Research VCS Best Practices

- Use forks and pull requests to collaborate.
- Release both your code and data for reproducibility.
- Regularly commit and push your code to a remote repository service such as gitlab (git.ucsc.edu).

Organize and separate code & read-only data; automate, script, & document EVERYTHING!

Reproducible Research Project Initialization

Research project initialization and organization following reproducible research guidelines.

Overview

```
project
|-- doc/     # documentation for the study
|  |-- paper/ # manuscript(s), whether generated or not
|-- data/    # raw and primary data, are not changed once created
  |-- raw/    # raw data, will not be altered
  |-- clean/  # cleaned data, will not be altered once created
|-- code/    # any programmatic code
|-- results  # all output from workflows and analyses
  |-- figures/ # graphs, likely designated for manuscript figures
  |-- pictures/ # diagrams, images, and other non-graph graphics
|-- scratch/ # temporary files that can be safely deleted or lost
|-- README   # the top level description of content
|-- study.Rmd # executable Rmarkdown for this study, if applicable
|-- Makefile # executable Makefile for this study, if applicable
|-- study.Rproj # RStudio project for this study, if applicable
|-- datapackage.json # metadata for the (input and output) data files
```
FAQ/Additional Resources

Git Branching and Workflow Strategies

Use of Github flow (rather than the overly complex git-flow) as a branching strategy and development workflow


Use of pull requests for code review and deployment control in Stash/Bitbucket

- https://www.atlassian.com/git/tutorials/making-a-pull-request/example

Branching rather than forking... Technical requirement of AWS, since OPSWorks only allows 1 repo. If you want to review the other “workflows” review the following detailed doc about the 4 main workflow approaches and their requisite branching strategies

- https://www.atlassian.com/git/tutorials/comparing-workflows

Some projects will have environment branches in addition to master, but the nice thing is those can be added later if we decide it makes sense. If you wonder why one might need a branch per tier of deployment read more here


For a thorough review of when these different branching strategies might be useful, and when to move from one to another read this

- http://www.creativebloq.com/web-design/choose-right-git-branching-strategy-121518344

For more background on the value of continuous deployment as an approach regardless of whether a team is currently using it and the requirements for continuous integration regardless of organizational release velocity see

- http://laurathomson.com/2011/08/05/capability-for-continuous-deployment/
Thank you!
Questions? Comments?