A quick introduction to version control with Git and GitHub

John D. Blischak(1), Emily R. Davenport(2), Greg Wilson(3)
(1) Committee on Genetics, Genomics, and Systems Biology, University of Chicago, Chicago, IL, USA
(2) Department of Molecular Biology & Genetics, Cornell University, Ithaca, NY, USA
(3) Software Carpentry Foundation, Toronto, Ontario, Canada

0.1 Introduction to version control

Many scientists write code as part of their research. Just as experiments are logged in laboratory notebooks, it is important to document the code you use for analysis. However, a few key problems can arise when iteratively developing code that make it difficult to document and track which code version was used to create each result. First, you often need to experiment with new ideas, such as adding new features to a script or increasing the speed of a slow step, but you do not want to risk breaking the currently working code. The simplest solution is to make a copy of the script before making new edits. However, this can quickly become a problem because it clutters your filesystem with uniformative filenames, e.g. `analysis.sh`, `analysis_02.sh`, `analysis_03.sh`, etc. It is difficult to remember the differences between the versions of the files, and more importantly which version you used to produce specific results, especially if you return to the code months later. Second, you will likely share your code with multiple lab mates or collaborators and they may have suggestions on how to improve it. If you email the code to multiple people, you will have to manually incorporate all the changes each of them sends.

Fortunately, software engineers have already developed software to manage these issues: version control. Version control software allows you to track the iterative changes you make to your code. Thus you can experiment with new ideas but always have the option to revert to a specific past version of the code you used to generate particular results. Furthermore, you can record messages as you save each successive version so that you (or anyone else) reviewing the development history of the code is able to understand the rationale for the given edits. Also, it facilitates collaboration. Using version control software, your collaborators can make and save changes to the code, and you can automatically incorporate these changes to the main code base. The collaborative aspect is enhanced with the emergence of websites that host version controlled code.

In this quick guide, we introduce you to one version control system (VCS), Git (git-scm.com), and one online hosting site, GitHub (github.com), both of which are currently popular among scientists and programmers in general. More importantly, we hope to convince you that although mastering a given VCS takes time, you can already achieve great benefits by getting started using a few simple commands. Furthermore, not only does using a VCS solve many common problems when writing code, it can also improve the scientific process. By tracking your code development with a VCS and hosting it online, you are performing science that is more transparent, reproducible, and open to collaboration [1, 2]. There is no reason this framework needs to be limited only to code; a VCS is well-suited for tracking any plain-text files: manuscripts, electronic lab notebooks, protocols, etc.

0.2 Version your code

The first step is to learn how to version your own code. In this tutorial, we will run Git from the command line of the Unix shell. Thus we expect readers are already comfortable with navigating a filesystem and running basic commands in such an environment. You can find directions for installing Git for the operating system running on your computer by following one of the links provided in Table 1. There are many graphical user interfaces (GUIs) available for running Git (Table 1), which we encourage you to explore, but learning to use Git on the command line is necessary for performing more advanced

operations and using Git on a remote machine.

For the purposes of this tutorial, imagine you have a folder in your home directory named `thesis`, which contains three files. `process.sh` runs some common bioinformatics tools on your raw data, `clean.py` removes bad samples and combines the data into a matrix, and `analyze.R` runs a statistical test and plots the result.

If you have just installed Git, the first thing you need to do is provide some information about yourself, since it records who makes each change to the file(s). Set your name and email by running the following lines, but replacing "First Last" and "user@domain" with your full name and email address, respectively.

```
$ git config --global user.name "First Last"
$ git config --global user.email "user@domain"
```

To start versioning your code with Git, navigate to your newly created or existing project directory (in this case, `~/thesis`). Start tracking your code by running the command `git init`, which initializes a new Git repository in the current folder. A repository refers to the current version of the tracked files as well as all the previously saved versions (Box 1).

```
$ cd ~/thesis
$ ls
analyze.R clean.py process.sh
$ git init
Initialized empty Git repository in ~/thesis/.git/
```

Now you are ready to start tracking your code (Figure 1). Conceptually, Git saves snapshots of the changes you make to your files whenever you instruct it to. For instance, after you edit a script in your text editor, you save the updated script to your thesis folder. If you tell Git to save a shapshot of the updated document, then you will have a permanent record of the file in that exact version even if you make subsequent edits to the file. In the Git framework, any changes you have made to a script, but have not yet recorded as a snapshot with Git, reside in the working diretory (Figure 1). To follow what Git is doing as you record the initial version of your files, use the informative command `git status`.

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        analyze.R
        clean.py
        process.sh

nothing added to commit but untracked files present (use "git add" to track)
```

There are a few key things to notice from this output. First, the three scripts are recognized as untracked files because you have not told git to take snapshots of anything yet. Second, the word "commit" means "a version of the code", e.g. "the figure was generated using the commit from yesterday" (Box 1). This word can also be used as a verb, in which case it means "to save", e.g. "to commit a change." Lastly, it explains how you can start tracking your files. You need to use the command `git add`. Add the file `process.sh`.

```
$ git add process.sh
```

And check its new status.

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   process.sh

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        analyze.R
        clean.py
```

Now the file `process.sh` has been added to the staging area, while both `clean.py` and `analyze.R` remain unstaged. Adding a file to the staging area will result in the changes to that file being included in the next commit, or snapshot of the code (Figure 1). As an analogy, adding files to the staging area is like putting things in a box to mail off, and committing is like putting the box in the mail.

Since this will be the first commit, or first version of the code, add the other two files to the staging area as well. Then create the first commit using the command `git commit`.

```
$ git add clean.py analyze.R
$ git commit -m "Add initial version of thesis code."
[master (root-commit) 660213b] Add initial version of thesis code.
 3 files changed, 25 insertions(+)
 create mode 100644 analyze.R
 create mode 100644 clean.py
 create mode 100644 process.sh
```

Notice the flag `-m` was used to pass a message for the commit. This message describes the changes that have been made to the code and is required. If you do not pass a message at the command line, the default text editor for your system will open so you can enter the message. You have just performed the typical development cycle with Git: make some changes, add updated files to the staging area, and commit the changes as a snapshot once you are satisfied with them.

Since Git records all of the commits, you can always look through the complete history of a project. To view the record of your commits, use the command `git log`. For each commit, it lists the the unique identifier for that revision, author, date, and commit message.

```
$ git log
commit 660213b91af167d992885e45ab19f585f02d4661
Author: First Last <user@domain>
Date:   Sun Mar 29 14:52:05 2015 -0500

    Add initial version of thesis code.
```

The commit identifier can be used to compare two different versions of a file, restore a file to a previous version from a past commit, and even retrieve tracked files if you accidentally delete them.

Now you are free to make changes to the files knowing that you can always revert them to the state of this commit by referencing its identifier. As an example, edit `clean.py`. Here is the current top of the

file.

```
$ head clean.py
#!/usr/bin/env python

# Thesis project:
# Remove bad samples.
# Export clean data as a matrix.

# Usage: clean.py input [input ...] > data_clean.py

import sys
import os
```

Update the description to explicitly define which samples are removed.

```
$ head clean.py
#!/usr/bin/env python

# Thesis project:
# Remove samples with more than 5% missing data.
# Export clean data as a matrix.

# Usage: clean.py input [input ...] > data_clean.py

import sys
import os
```

You can view all the differences between the current version and last committed version of the file by running the command `git diff`.

```
$ git diff
diff --git a/clean.py b/clean.py
index c1fcad7..c0bfe5c 100644
--- a/clean.py
+++ b/clean.py
@@ -1,7 +1,7 @@
 #!/usr/bin/env python

 # Thesis project:
-# Remove bad samples.
+# Remove samples with more than 5% missing data.
 # Export clean data as a matrix.

 # Usage: clean.py input [input ...] > data_clean.py
```

The new line starts with `+` and the previous line starts with `-`. You can ignore the first five lines of output because they are directions for other software programs that can merge changes to files. If you wanted to save this edit, you could add `clean.py` to the staging area using `git add` and then commit the change using `git commit`, as you did above. Instead, this time restore the last committed version of the file using the command `git checkout`.

```
$ git checkout -- clean.py
$ git diff
```

Now `git diff` returns no output because `git checkout` reverted `clean.py` to the version in the last commit. And this ability to revert to past versions of a file is not limited to just the last commit. If you had committed multiple changes to the file `clean.py` and then decided you wanted the original version, you could replace the argument `--` with the commit identifier (we only need to specify the first few characters for it to be unique). The `--` used above was simply a placeholder for the first argument because by default `git checkout` restores the most recent version of the file.

```
$ git checkout 660213b clean.py
```

There are also more advanced options for reverting history that we will not cover in this quick introduction.

At this point, you have learned the commands needed to version your code with Git. Thus you already have the benefits of being able to make edits to files without copying them first, to create a record of your changes with accompanying messages, and to revert to previous versions of the files if needed. Now you will always be able to recreate past results that were generated with previous versions of the code and see the exact changes you have made over the course of a project.

0.3 Share your code

Once you have your files saved in a Git repository, you can share it with your collaborators and the wider scientific community by putting your code online. This also has the added benefit of creating a backup of your work and provides a mechanism for syncing your files across multiple computers. Sharing a repository is made easier if you use one of the many online services that host Git repositories (Table 1), e.g. GitHub. Note, however, that any files that have not been tracked with at least one commit are not included in the Git repository, even if they are located within the same directory on your local computer (see Box 3 for advice on the types of files that should not be versioned with Git).

To begin using GitHub, you will first need to sign up for an account. For the examples in this tutorial, we will use the fake username "scientist123". Next choose the option to "Create a new repository". Call it "thesis" because that is the directory name containing the files, but this is not a requirement. Also, now that the code will be existing in multiple places, you need to learn some more terminology (Box 1). A local repository refers to code that is stored on the machine you are using, e.g. your laptop; whereas, a remote repository refers to the code that is hosted online. Thus, you have just created a remote repository.

Now you need to send the code on your computer to GitHub. The key to this is the URL that GitHub assigns your newly created remote repository. It will have the form `https://github.com/username/reponame.git`, e.g. `https://github.com/scientist123/thesis.git`. In order to link the local thesis repository on your computer to the remote repository you just created, in your local repository you need to tell Git the URL of the remote repository using the command `git remote add`.

```
$ git remote add origin https://github.com/scientist123/thesis.git
```

The name "origin" is a bookmark for the remote repository so that you do not have to type out the full URL every time you sync your changes (this is the default name for a remote repository, but you could use another name if you liked).

Send your code to GitHub using the command `git push` (Figure 2).

```
$ git push origin master
```

You first specify the remote repository, "origin". Second, you tell Git to push to the "master" copy of the repository - we won't go into other options in this tutorial, but Box 2 discusses them briefly.

Pushing to GitHub also has the added benefit of backing up your code in case anything were to happen to your computer. Also, it can be used to sync your code across multiple machines, similar to a service like Dropbox, but with the added capabilities of Git. For example, what if you wanted to work on your code on your computer at home? You can download the Git repository using the command `git clone`.

```
$ git clone https://github.com/scientist123/thesis.git
```

By default, this will download the Git repository into a local directory named "thesis". Furthermore, the remote "origin" will automatically be added so that you can easily push your changes back to GitHub. You now have copies of your repository on your work computer, your GitHub account online, and your home computer. You can make changes, commit them on your home computer, and send those commits to the remote repository with `git push`, just as you did on your work computer.

Then the next day back at your work computer, you could update the code with the changes you made the previous evening using the command `git pull`.

```
$ git pull origin master
```

This pulls in all the commits that you had previously pushed to the GitHub remote repository from your home computer. In this workflow, you are essentially collaborating with yourself as you work from multiple computers. If you are working on a project with just one or two other collaborators, you could extend this workflow so that they could edit the code in the same way. You can do this by adding them as Collaborators on your repository (Settings ->Collaborators ->Add collaborator). However, with projects with lots of contributors, GitHub provides a workflow for finer-grained control of the code development.

With the addition of a GitHub account and a few commands for sending and receiving code, you can now share your code with others, sync your code across multiple machines, and setup simple collaborative workflows.

0.4 Contribute to other projects

Lots of scientific software is hosted online in Git repositories. Now that you know the basics of Git, you can directly contribute to developing the scientific software you use for your research. From a small contribution like fixing a typo in the documentation to a larger change such as fixing a bug, it is empowering to be able to improve the software used by you and many other scientists.

When contributing to a larger project with many contributors, you will not be able to push your changes with `git push` directly to the project's remote repository. Instead you will first need to create your own remote copy of the repository, called a fork. You can fork any repository on GitHub by clicking the button "Fork" on the top right of the page.

Once you have a fork of a project's repository, you can clone it to your computer and make changes just like a repository you created yourself. Let's say you created a fork of the hypothetical repository, "cool_project", so that you could fix a typo you found in the directions in the README file. In order to make changes, you first download it with `git clone`.

```
$ git clone https://github.com/scientist123/cool_project.git
```

After making the edits you want, you can add, commit, and push the changes back to your remote repository on GitHub (Figure 3).

```
$ git add README
$ git commit -m "Fix typo in documentation."
$ git push origin master
```

Currently the typo is fixed in your fork of cool_project. To merge this change into the main repository that is owned by the creator of the software, send a pull request using the GitHub interface (Pull request ->New pull request ->Create pull request). After the pull request is created, the owner of the original repository can review your change. If she approves of the change, she can merge it into the main repository. Although this process of forking a project's repository and issuing a pull request seems like a lot of work to contribute changes, this workflow gives the owner of the project control over what changes get incorporated into the code. You can have others contribute to your projects using the same workflow (Figure 4).

The ability to use Git to contribute changes is very powerful because it allows you to improve the software that is used by many other scientists and also potentially shape the future direction of its development.

0.5 Conclusion

Git, albeit complicated at first, is a powerful tool that can improve code development and documentation. Ultimately the complexity of a VCS not only gives users a well-documented "undo" button for their analyses, but it also allows for collaboration and sharing of code on a massive scale. Furthermore, it does not need to be learned in its entirety to be useful. Instead, you can derive tangible benefits from adopting version control in stages. With a few commands (`git init`, `git add`, `git commit`), you can start tracking your code development and avoid a filesystem full of copied files. Adding a few additional commands (`git push`, `git clone`, `git pull`) and a GitHub account, you can share your code online, sync your changes across machines, and collaborate in small groups. Lastly, by forking public repositories and sending pull requests, you can directly improve scientific software.

0.6 Methods

We collaboratively wrote the article in LaTeX (latex-project.org) using the online authoring platform Authorea (authorea.com). Furthermore, we tracked the development of the document using Git and GitHub. The Git repo is available at github.com/jdblischak/git-for-science, and the rendered LaTeX article is available at authorea.com/users/5990/articles/17489.

0.7 Box 1: Definitions

- **Version Control System (VCS)**: *(noun)* a program that tracks changes to specified files over time and maintains a library of all past versions of those files
- **Git**: *(noun)* a version control system
- **repository (repo)**: *(noun)* folder containing all tracked files as well as the version control history
- **commit**: *(noun)* a snapshot of changes made to the staged file(s); *(verb)* to save a snapshot of changes made to the staged file(s)
- **branch**: *(noun)* a parallel version of the files in a repository (Box 2)
- **local**: *(noun)* the version of your repository that is stored on your personal computer
- **remote**: *(noun)* the version of your repository that is stored on the internet, for instance on GitHub
- **clone**: *(verb)* to create a local copy of a remote repository on your personal computer
- **fork**: *(noun)* a copy of a repository; *(verb)* to copy a repository, for instance from one user's Github account to your own
- **merge**: *(verb)* to update files by incorporating the changes introduced in new commits
- **pull**: *(verb)* to retrieve commits from a remote repository and merge them into a local repository
- **push**: *(verb)* to send commits from a local repository to a remote repository
- **pull request**: *(noun)* a message sent by one user to merge the commits in their remote repository into another user's remote repository

0.8 Table 1: Resources

Resource	Options
Distibuted VCS	Git (git-scm.com)
	Mercurial (mercurial.selenic.com)
	Bazaar (bazaar.canonical.com)
Online hosting site	GitHub (github.com)
	Bitbucket (bitbucket.org)
	GitLab (gitlab.com)
	Source Forge (sourceforge.net)
Git installation	git-scm.com/downloads
Git Tutorials	Software Carpentry(swcarpentry.github.io/git-novice)
	Pro Git (git-scm.com/book)
	A Visual Git Reference (marklodato.github.io/visual-git-guide)
	tryGit (try.github.io)
Graphical User Interface for Git	git-scm.com/downloads/guis

0.9 Box 2: Branching

Do you ever make changes to your code, but are not sure you will want to keep those changes for your final analysis? Using Git, you can maintain parallel versions of your code that you can easily bounce between while you are working on your changes. You can think of it like making a copy of the folder you keep your scripts in, so that you have your original scripts intact but also have the new folder where you make changes. Using Git, this is called branching and it is better than separate folders because 1) it uses a fraction of the space on your computer, 2) keeps a record of when you made the parallel copy (branch) and what you have done on the branch, and 3) there is a way to incorporate those changes back into your main code if you decide to keep your changes (and a way to deal with conflicts). By default, your repository will start with one branch, usually called "master". To create a new branch in your repository, type `git branch new_branch_name`. You can see what branches a current repository has by typing `git branch`, with the branch you are currently in being marked by a star. To move between branches, type `git checkout branch_to_move_to`. You can edit files and commit them on each branch separately. If you want combine the changes in your new branch with the master branch, you can merge the branches by typing `git merge new_branch_name` while in the master branch.

0.10 Box 3: What *not* to version control

You *can* version control any file that you put in a Git repository, whether it is text-based, an image, or giant data files. However, just because you *can* version control something, does not mean you *should*. Git works best for plain text based documents such as your scripts or your manuscript if written in LaTeX or Markdown. This is because for text files, Git saves the entire file only the first time you commit it and then saves just your changes with each commit. This takes up very little space and Git has the capability to compare between versions (using `git diff`). You can commit a non-text file, but a full copy of the file will be saved with each commit. Over time, you may find the size of your repository growing very quickly. A good rule of thumb is to version control anything text based: your scripts or manuscripts if they are written in plain text. Things *not* to version control are large data files that never change, binary files (including Word and Excel documents), and the output of your code.

0.11 Figure Legends

Figure 1. The git add/commit process. To store a snapshot of changes in your repository, first `git add` any files to the staging area you wish to commit (for example, you've updated the `process.sh`

file). Second, type `git commit` with a message. Only files added to the staging area will be committed. All past commits are located in the hidden `.git` directory in your repository.

Figure 2. Working with both a local and remote repository as a single user. A) On your computer you commit to a Git repository (commit d75es). B) On GitHub, you create a new repository called `thesis`. This repository is currently empty and not linked to the repo on your local machine. C) The command `git remote add` connects your local repository to your remote repository. The remote repository is still empty, however, because you have not pushed any content to it. D) You send all the local commits to the remote repository using the command `git push`. Only files that have been committed will appear in the remote repository. E) You repeat several more rounds of updating scripts and committing on your local computer (commit f658t and then commit xv871). You have not yet pushed these commits to the remote repository, so only the previously pushed commit is in the remote repo (commit d75es). F) To sync the local and remote repositories, you `git push` the two new commits to the remote repository. The local and remote repositories now contain the same files and commit histories.

Figure 3. Contributing to Open Source Projects. You found an error in the README for a cool project hosted on GitHub and you would like to fix it. A) The cool_project repository exists on GitHub (along with the commit history of the project), but you would like to edit the README on your computer. B) First, you fork the cool_project repository into your account on GitHub, bringing the entire commit history of the project along with the code. C) To create a copy of the repository on your computer, you `git clone` the repository from your GitHub account. D) You make a change to the README, save the file, add the file to the staging area, and commit (creating commit 09pr4). This commit is only on your local computer. Your remote cool_project repository and the original remote cool_project repository do not have this commit. E) You sync your local and remote repositories using `git push`. The original cool_project still does not have your update to the README file. F) To suggest the change in the README to the original cool_project team, submit a pull request via GitHub. If the owner(s) of the cool_project repository like your change, they will accept the pull request and your changes will be incorporated into the project.

Figure 4. Collaboration using GitHub. A) You have a repository on your computer and on GitHub that you have already connected using `git remote add`, but you have not pushed any content to the remote repo yet. B) You push your local commits to your remote repository using `git push` (both your local and your remote have commit d75es). C) Your collaborator would like to help you analyze your data. They make a copy of your remote repository into their GitHub account using the "fork" option in GitHub. All three repositories have the same version history (commit d75es). D) Your collaborator wants a copy of the repository on their local computer so they can make edits. They `git clone` the repository from their remote GitHub account. E) Your collaborator edits a file and commits a change to their local repository (commit t957s). Changes are not automatically shared across all repositories, therefore your local, your remote, and your collaborator's remote only have commit d75es. F) Your collaborator syncs their local repo with their remote repo by pushing. Now both your collaborator's local and remote repositories have two commits (commit d75es and commit t957s), while your local and remote only have one (commit d75es). G) Unaware of what your collaborator is doing, you continue to work on your code and you make a new commit locally (commit f658t). H) Your collaborator wants you to incorporate their changes into your code, so from GitHub they issue a Pull Request. The changes are acceptable, so you confirm the Pull Request and merge the changes into your remote repository. Your remote repository is ahead of your local repository by one commit: the commit your collaborator made (commit t957s) In addition your local repository is ahead of your remote repository by one commit that was made before the Pull Request was issued by your collaborator (commit f658t). I) To incorporate the new commit from your collaborator into your local repository, you `git pull` commits from your remote to your local repository. Your local now has the commit from your collaborator incorporated, but is still ahead of your remote by one commit. You could `git push` your changes to sync your remote and local repositories.

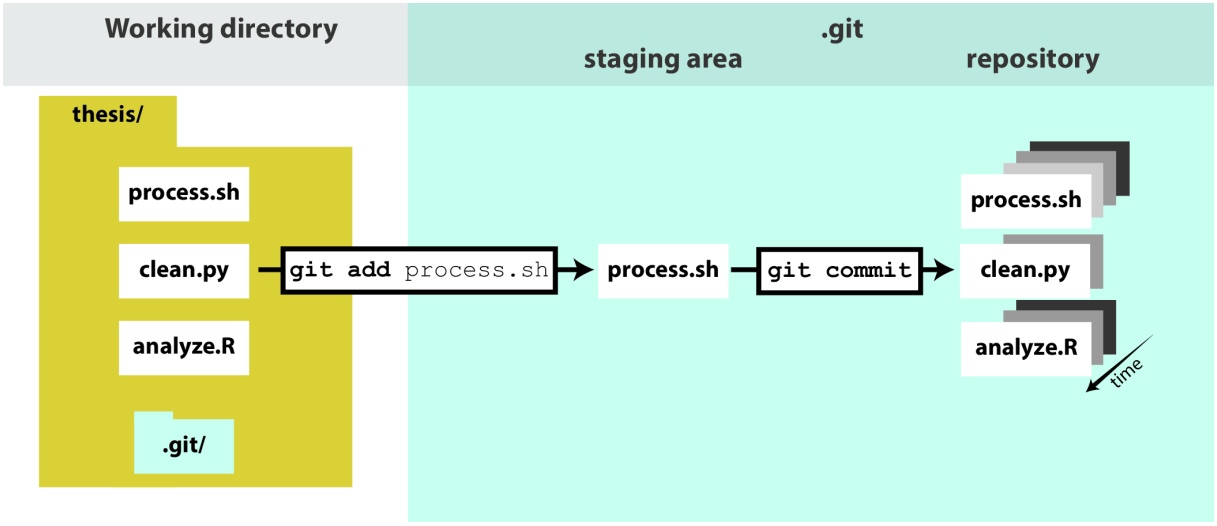

Figure 1.

References

[1] Ram K (2013) Git can facilitate greater reproducibility and increased transparency in science. Source Code Biol Med 8: 7.

[2] Wilson G, Aruliah D, Brown C, Chue HN, Davis M, et al. (2014) Best practices for scientific computing. PLoS Biol 12: e1001745.

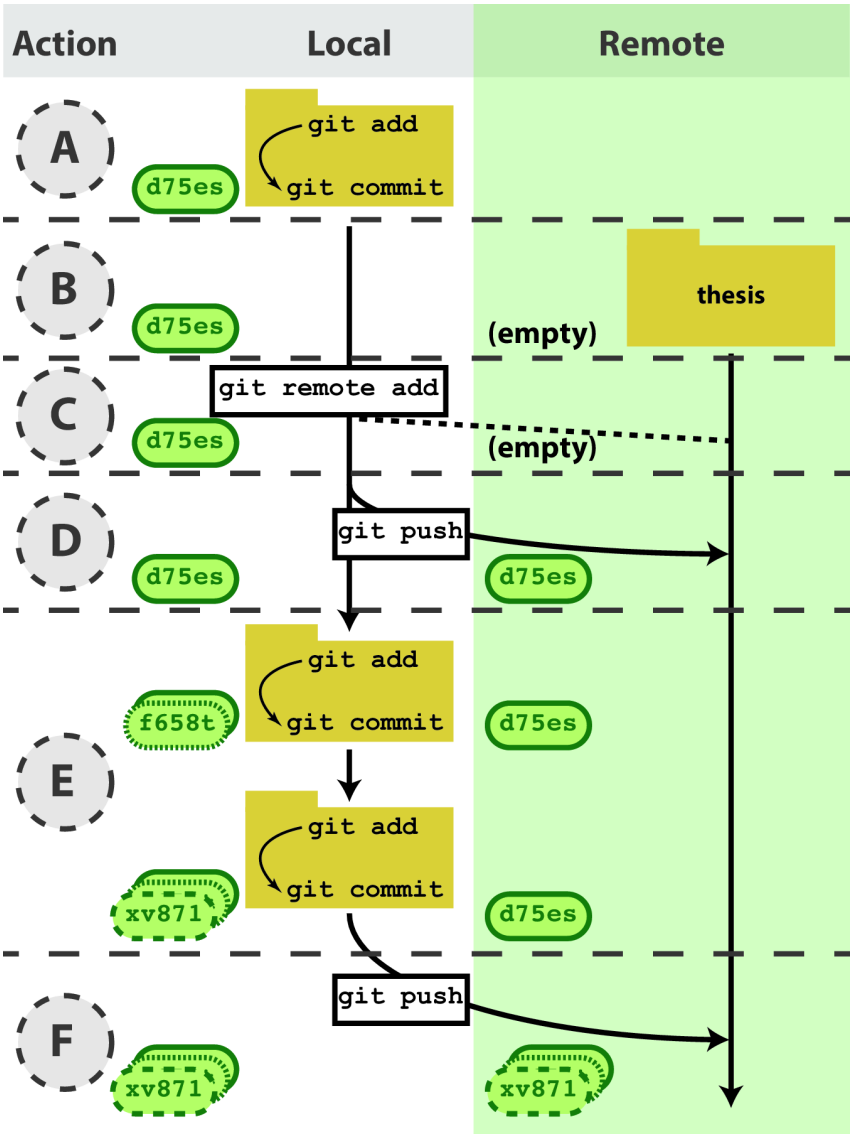
Action
Local
Remote
A
d75es
git add
git commit
B
d75es
thesis
(empty)
C
git remote add
d75es
(empty)
D
git push
d75es
d75es
E
f658t
git add
git commit
d75es
xv871
git add
git commit
d75es
F
git push
xv871
xv871

Figure 2.

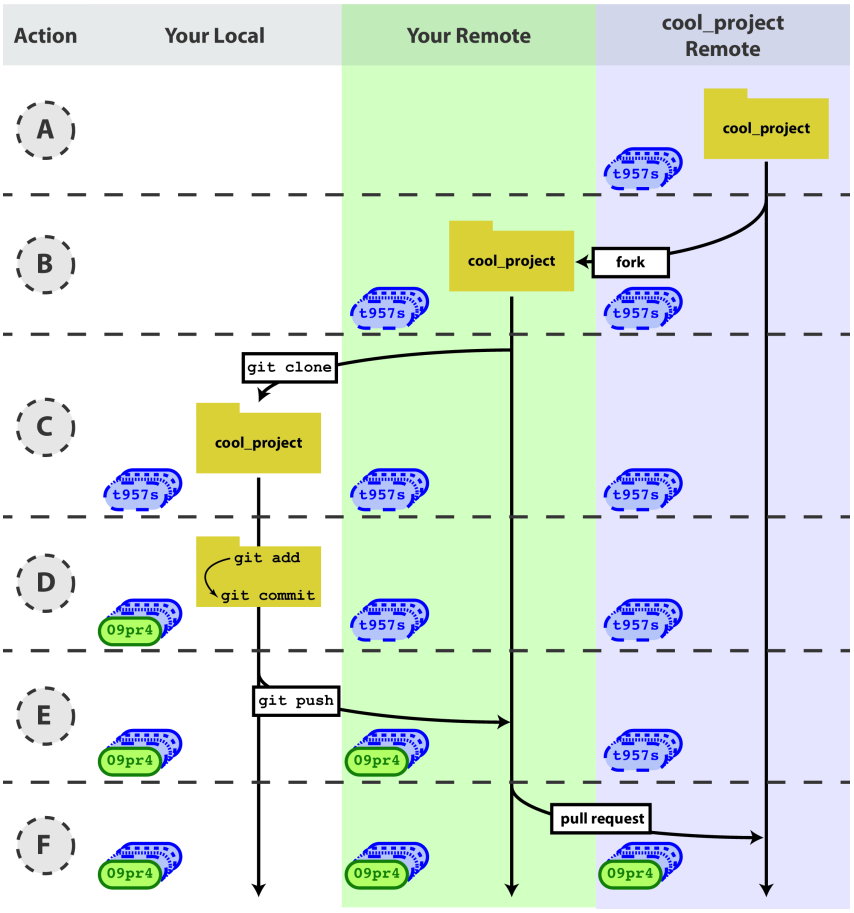
Action
Your Local
Your Remote
cool_project Remote
A
B
C
D
E
F
cool_project
t957s
fork
git clone
git add
git commit
git push
pull request
09pr4

Figure 3.

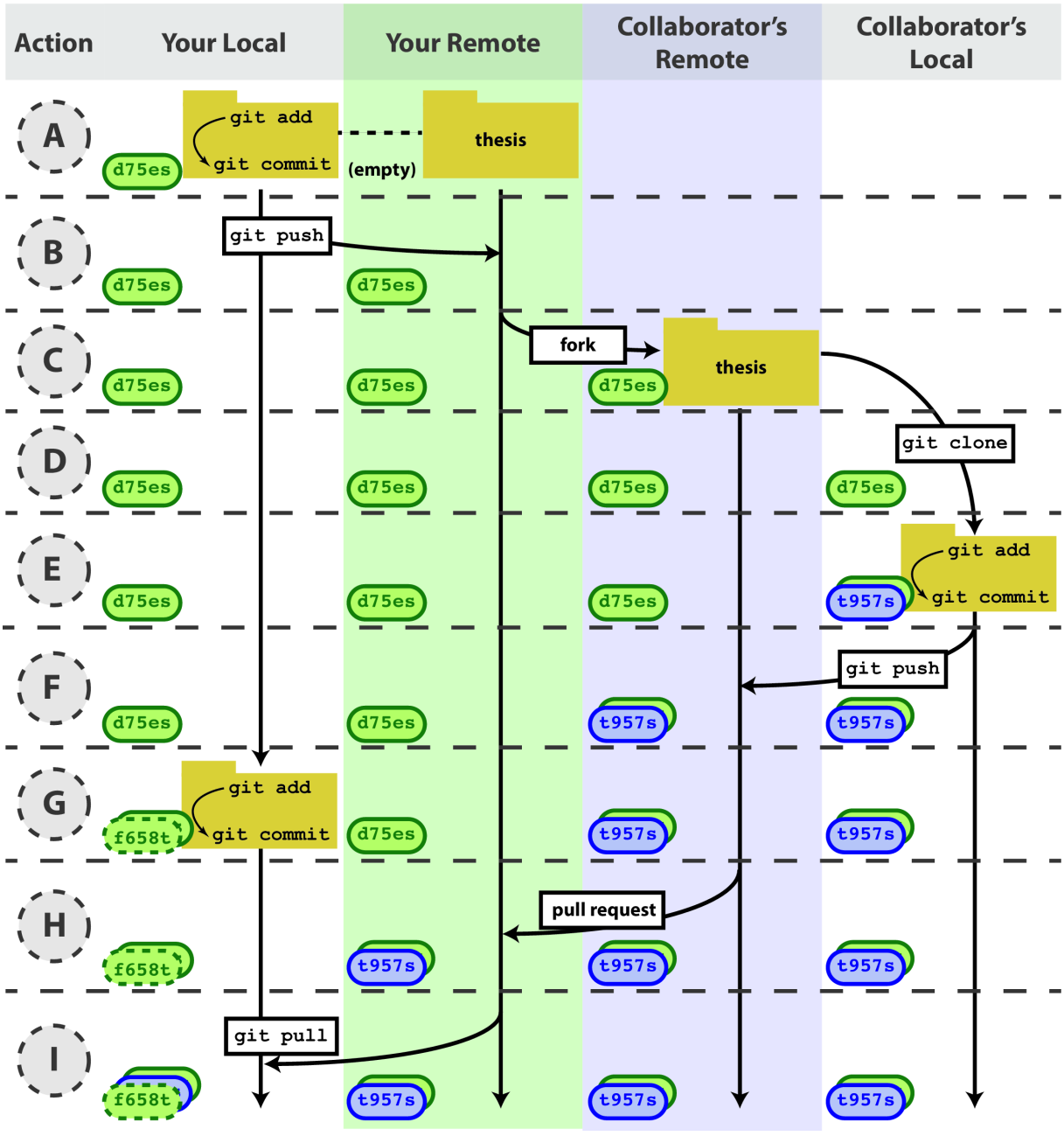
Action
Your Local
Your Remote
Collaborator's Remote
Collaborator's Local
A
B
C
D
E
F
G
H
I
d75es
git add
git commit
thesis
(empty)
git push
fork
thesis
git clone
git add
git commit
t957s
git push
git add
git commit
f658t
pull request
git pull

Figure 4.