PhUSE US Connect 2018

Paper TT10

# Building efficient programming teams using RStudio with Git in pharmaceutical industry

James J. Kim, Pfizer Inc., Cary, U.S.A.

## ABSTRACT

The proper use of a version control system (VCS) ensures the integrity and quality of the programs written by globally dispersed programmers. Git is a distributed VCS (DVCS) that is powerful and slowly replacing traditional VCS (e.g. subversion). But Git alone is often inefficient without implementing logical processes and rules around the people who use it. Programming in pharmaceutical industry provides unique challenges that require special considerations including: data vs. application-driven approaches, longevity of programs and libraries, validation methodology and etc. This paper introduces Git and DVCS concept in general, including popular online services such as GitHub and Bitbucket; its use in RStudio which is a popular integrated development environment (IDE) for R programming language with built-in Git and subversion client. And finally, it suggests some tips for processes and rules when more than few programmers are working together.

## INTRODUCTION

This paper is not a tutorial for learning version control system, programming languages nor any specific development interfaces. Therefore I will not cover the specifics of Git, R/SAS or RStudio. There is enormous information available on the Internet about each of these topics and I will provide some useful ones in the reference section. This paper will concentrate on special needs and limitations of our industry, pragmatic solutions for process improvements and leave some questions for discussions. I assume that readers are familiar with pharmaceutical/CRO industry and comfortable with technical topics including programming, version control and tools. The Microsoft Windows 10 64bit version was used for the mentioned programs and tools but Mac/Linux users should be able to replicate the same experience with minimal differences.

## WHY VERSION CONTROL?

If you have more than one person (not necessarily a programmer) working together, you understand the pain. This is one of the main reasons why adding two programmers to a task doesn't produce the outcome of 200%. Even doubling CPUs in a computer system doesn't result in doubling of performance because there is always the price of overhead for coordinating and scheduling between more than one entity. A common problem in this situation is handling *changes*. For example, adding, modifying and deleting programs that you may want to undo (or redo in some cases) take up bulk of your time. *Study 1 V1 Final.doc* or *Study 2 CRF Spec Draft V0.2.xsl* should sound sadly familiar. Furthermore, you may want to share your program with others for their reviews, want to know who made a change for what reason, when they made it and even want to store multiple versions in parallel. This is increasingly becoming important as many programming tasks are now distributed to many groups (internal programmers, external contractors and vendors), outsourced to different geographical regions/countries and often in different time zones.

A VCS can do all these tasks and make more advanced operations possible. In VCS, programmers *check out* programs into their own working space where they can work on them independently from the *repository*. When they are ready to save the changes, they *commit* their works. If there are others who work on the same programs, they get to review and *merge* those changes. All changes are saved and tracked over time – they can *revert* changes, create a new development *branch* for new features/tests without affecting the *master* and *tag* important milestone changes. So the importance of using a good VCS can't be overstated. If you are not using a VCS at your current workplace where any type of programming is involved, you have a serious issue.

### VCS HISTORY AND GIT

The following table describes a brief history of VCS in three generations:

| Generation | Network Repository | File operations | Examples |
|---|---|---|---|
| First | None (Local only) | One file at a time | RCS, SCCS |
| Second | Centralized (CVCS) | Multiple files | CVS, Subversion |
| Third | Distributed (DVCS) | Changesets | Git, Mercurial |

**Table 1: VCS Generations**

The most significant difference of DVCS compared to the previous generations is the existence of private copy of the entire repository to each programmer. Everyone can pretend he/she is the only programmer on the team and defer the overhead of coordination with others until the work is ready to be *pushed* to the team. The programmer can commit as often as needed without affecting others. (The term "team" is loosely being used here since there is no "central" repository in DVCS – the central repository is selected by the team's decision in DVCS, not by the existence of a central location of the repository. You can even have multiple repositories synchronizing together.)

There are other advantages of Git –

1. Speed – Working locally on your computer is always faster than working via network. The speed of DVCS will prevail until such time when everyone gets a Petabyte scale network.
2. Offline – You don't have to be connected to your company VPN or the Internet to work since the repository stays with your computer. Although this is becoming less of an issue, there are times that you have to work offline without Wi-Fi. You can still commit your changes until you are ready to push them to the team.
3. Non-linear development – Git supports rapid branching and merging. This offers flexibility in workflows among teams; for example, you may want to create a separate branch for a specific purpose.
4. Scale out – there is no powerful centralized server needed since the work is distributed among programmers. All the heavy lifting happens on the client side; similar to BitTorrent protocol.

There are disadvantages of Git too. It has a steep learning curve and the concept may sound too complicated to understand. Luckily, there is no shortage of online tutorials. And as a programmer working in pharmaceutical industry, you only need to be able to use some basic commands:

1. How to clone or initialize a new repository locally
2. How to pull a remote repository
3. How to add and commit changes
4. How to push changes
5. How to create and setup a private key for authentication
6. Understand some terms such as "remote" and "master"

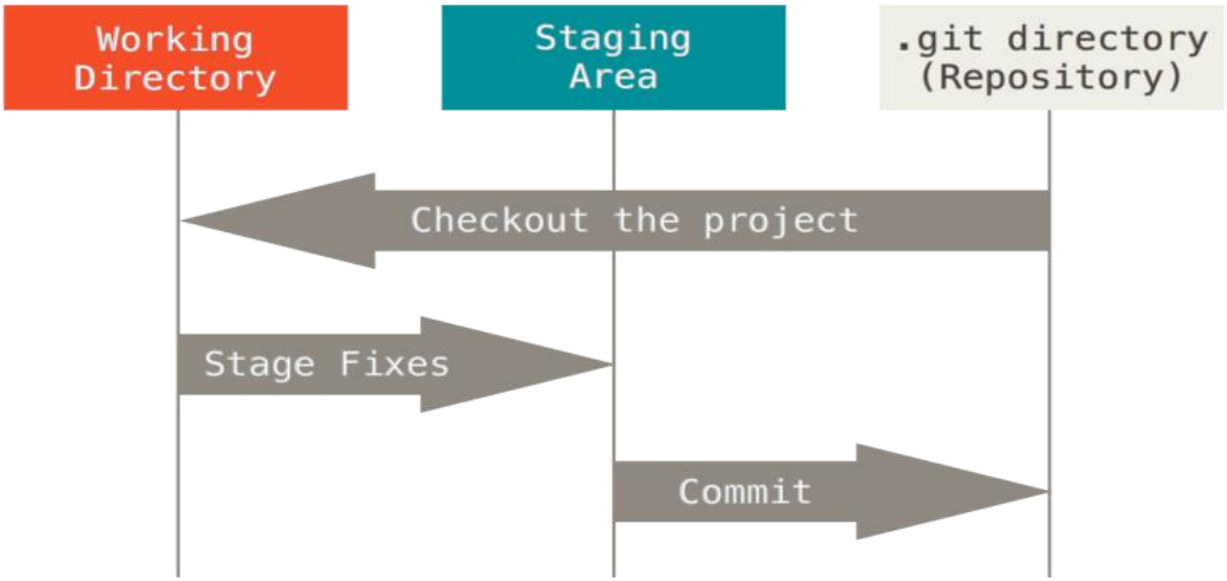


**Figure 1: Git Process**

**GIT HOSTING SERVICES**

I've mentioned that the strength of DVCS is the distributed nature of the repository. However for any collaborative work in Git, you will need to have a remote Git repository (Git Server). Technically speaking, you can push changes to and pull changes from anyone's repositories but doing so is discouraged because you can fairly easily confuse what others are working on if you are not careful. Furthermore, you want your colleagues to be able to access the repository even if your computer is offline – having a more reliable common repository is useful. Therefore, the preferred method for collaborating with someone is to set up an intermediate repository that you both have access to, and push to and pull from that. While you can setup your own Git server, it is more common to use Git hosting services provided by your organization or external services. At Pfizer, we use TeamForge with Git integration internally but we also have GitHub account (https://github.com/PfizerRD) for external collaboration. GItLab and Bitbucket also provide similar services for both private and public repositories.

## HOW R/RSTUDIO SUPPORTS GIT?

### R PACKAGE MANAGER

R has two packages to install R packages hosted on GitHub directly instead of CRAN.

1. *devtools* package offers "*install_github*("DeveloperNmae/PackageName")" function.

```
library(devtools)
install_github("hadley/dplyr")
```

2. *githubinstall* package offers "*githubinstall*("PackageName")" function and other useful functions.

```
library(githubinstall)
githubinstall("AnomalyDetection")
gh_suggest()
gh_suggest_username()
gh_list_packages()
gh_search_packages()
gh_show_source()
gh_update_package_list()
```

### RSTUDIO GUI

RStudio is a free and open-source integrated development environment (IDE) for R. It is installed separately from the base R package. It is probably the only IDE available for R; therefore popular among many R programmers. RStudio supports Subversion (SVN) and Git integration with R but the client programs have to be downloaded and installed separately as well.

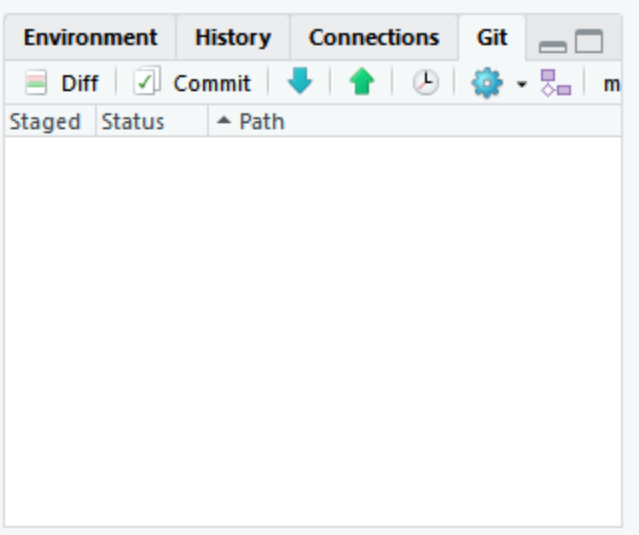


**Figure 2: RStudio Git GUI**

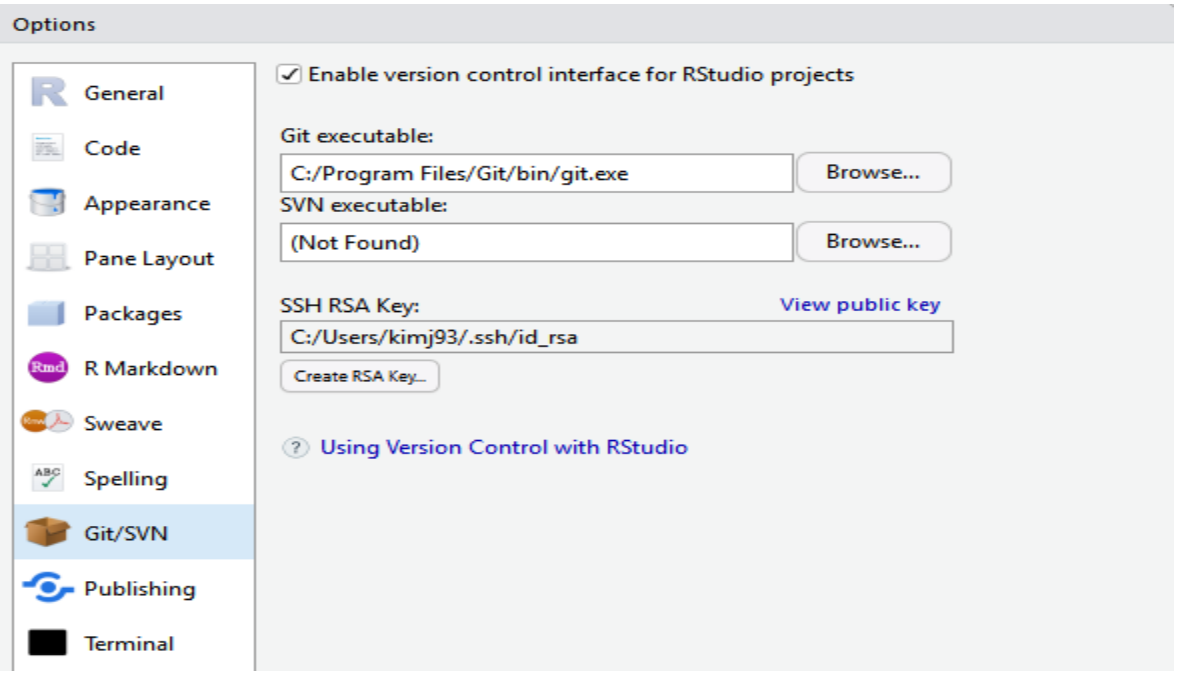


**Figure 3: RStudio Git/SVN options**

Although RStudio supports the Git integration via GUI, the functionality is limited to basic commands. The best and raw power of Git can be only used via the command-line; fortunately, many programmers should be able to use the basic commands to complete their daily tasks.

**NOTE TO SAS PROGAMMERS**

There is nothing preventing SAS programmers from using Git. As a matter of fact, SAS Enterprise Guide (EG) uses a compressed Git repository as its project file (*.egp) to track files and history. The function of Git within SAS EG is very limited and unlike RStudio, it doesn't provide any external means to access the repository; therefore unless you have a special need for SAS EG project, I discourage you from using it.

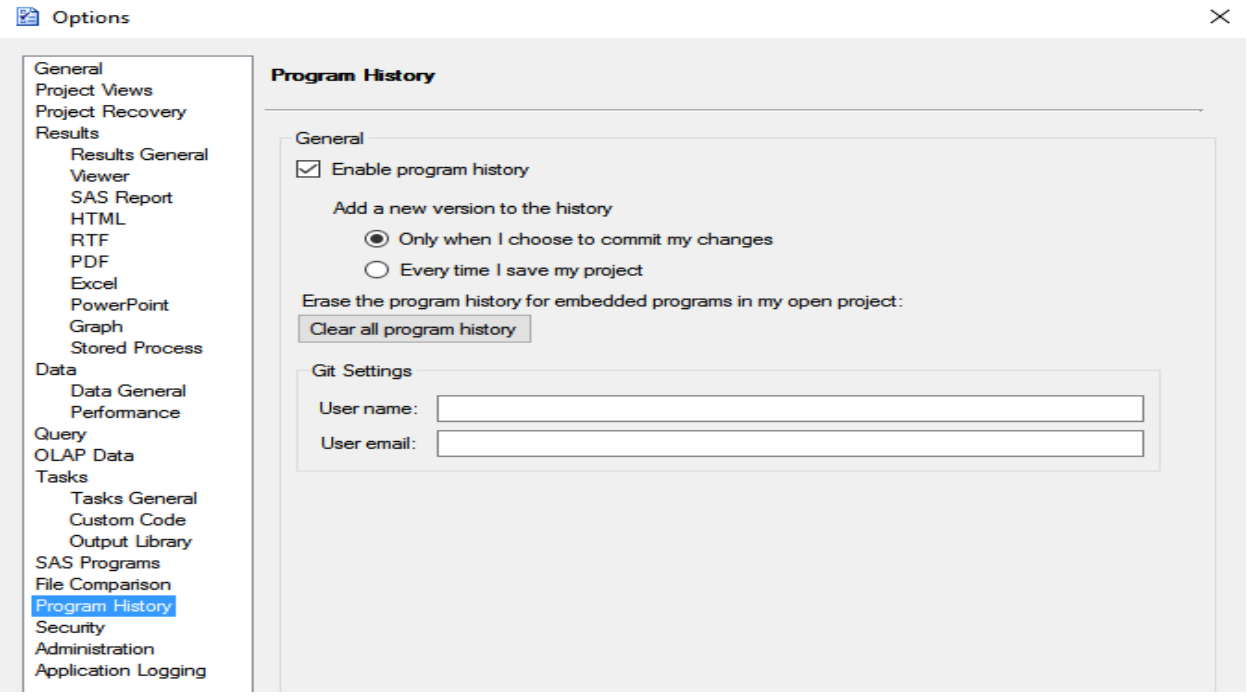


**Figure 4: SAS EG Program History Options**

## BEST PRACTICES

Working as a programmer in pharmaceutical industry presents unique challenges, including:

1. We work in a regulated environment. We are expected to follow a set of written procedures within validated systems while maintaining confidentiality of our data.
2. We normally don't worry about memory allocation, garbage collection or sorting algorithms. Both R and SAS are high-level languages that take care most of these complex issues at lower level. With abundance of RAM and disk drive spaces, we care less about the available resources but more about delivering on time.
3. We don't ship any final software product or send out bug patches – our final product is a set of data – raw, derived, cleaned and analyzed to support conclusive evidence whether a drug is effective or not. This is a big difference from other programming environments and we need to consider it carefully.

#### COPYING/STEALING DATA CAN BE EVIL

Our intellectual property lies with our data collected through clinical trials. Our programs are written to manipulate the data to be used for submission of new drugs.

The following illustrates the rough process cycle:

1. Input data is received (Raw data)
2. Output data is mapped and generated (e.g. SDTM/ADaM)
3. Table, Listing and Figures (TLFs) are programmed for CSR inclusion
4. Repeat as needed

The rest of the world has a different cycle:

1. Design a program to do certain tasks
2. Program/Debug/Test
3. Release
4. Work on the next patch/minor/major release

#### COPYING/STEALING PROGRAMS CANNOT BE EVIL

With the adaption of industry standards such as CDISC, many programs can be recycled and reused. Building of common packages and libraries within organizations is a common practice to save time and effort from starting programs from scratch. However copying someone's program and reusing it in your own program is still being frowned upon – why is this? Copying a validated program (or portion of it) guarantees that it will be more reliable than writing it from scratch and should be encouraged. This is a misconception that somehow this is unethical – copying and possibly improving program is perfectly fine and it is the principle of open source programming. In the previous section, I've made it clear where our important asset resides and why it is okay to share programs as long as proprietary data is not shared. This will break down silos, encourage internal collaboration and accelerate new hires on-boarding.

For example, the following SAS program is a popular snippet on the Internet for calculating a person's integer age from birthdate (dob):

```
FLOOR((INTCK('month',dob,eventdate)-(day(eventdate)<day(dob)))/12)
```

I've seen numerous cases where I saw the above calculation copied straight from somewhere and landed on both primary and QC programs. What is the point of doing DP in this case? Is one or both of these programmers guilty in plagiarism, copyright violation or company policy?

By the way, if you have SAS 9.3 or later you can use YRDIF function:

```
YRDIF(dob,eventdt,'AGE')
```

#### DOUBLE-PROGRAMMING (DP)

DP relies on the notion that two programmers work independently from one another (i.e. "Blinded") and the generation of two identical outputs validates the accuracy. There are few issues with this approach:

1. It is time-consuming and resource intensive. This becomes more serious when the programmers are separated geographically as their work schedules may not overlap enough to work together.
2. It is inflexible. The roles are locked (primary and QC) and can't be reversed. The problem becomes more evident when one or both programmers become unavailable.
3. Most importantly since the programs are not reviewed by each other, no one really knows how the programs work other than the original author.

I don't know the source of DP or where the practice originated from. It is possible that it may have come from double-independent entry practice in data management; first entry is validated by second/third entry and adjudication. But programming data is different from entering data – this practice may not be as accurate as you think and it is not a good model for fast-paced environments with shifting resources such as CROs.

For example, the following two programs produce the exact same results:

```
Program a:
   a <- strsplit("Fun in R",split="")
   for (i in length(a)) {
     print(a[i])
   }
Program b:
   b <-c("F","u","n"," ","i","n"," ","R")
   print(b)
```

They both produce:

```
   [1] "F" "u" "n" " " "i" "n" " " "R"
```

The second program is a textbook version of "hard-coding" but there is no way you will find that out unless you review the program. Also some sharp-eyed readers will note that the first program returns a list while the second program will return a character vector.

The key point is that the biggest weakness in DP cannot be simply ignored – if your program is not reviewed by another person ("blinded"), your result can not be validated 100%. Some organizations try to compensate this by performing "Triple QC Programming" – one more person who reviews both primary and QC programs for validation but it is extremely inefficient and resource-intensive.

**OPEN SOURCE**

Lack of transparency is an obvious setback for closed source programs. Both R and Git were developed under open source. You are free to improve, distribute and share as long as you acknowledge their use and share your contribution freely. In the previous sections, the benefits of open sourcing your programs were presented. I encourage more people and organizations to share your programs and ideas.

There is no need for anyone to reinvent the wheel. For example, it is perfectly possible for you to write a program to perform regression analysis in R. But this is not a good idea since there are many R packages available to do the same job, often optimized and tested rigorously at a level you can't replicate.

**TIPS TO KEEP IT CLEAN**

Here are some tips to work with other programmers – targeted for Git but applicable to all VCS.

1. Pull before starting your day – your repository is most likely out of date especially if you have colleagues working in India or China. Pull to get the latest update.
2. Run *diff* before commit – it is always good idea to review what you are committing before commit. There are many diff tools available for doing so.
3. Read diffs from other programmers too – it is also good ideas to review what others have done. When you do this, two good things might happen: the program might get better when you notice something needs to be fixed and you might learn something you never knew from your colleagues.
4. Keep your directories logical – doing it right in the first place will save you a lot of efforts down the road. This is an example of how one may structure a project.

```
/my_project
   |--/.git
   |--.gitignore
   |--README.md
   |--/data
      |--data1.txt
      |--data2.txt
      |--etc...
   |--/R_scripts
      |--Rfile1.R
      |--Rfile2.R
      |--etc...
   |--/binaries
      |--R1.RData
      |--SAS1.sas7bdat
      |--etc...
```

By putting all data files in a single directory and adding "/data" to .gitignore file, the data files will only reside in my local repository but won't be tracked; therefore I get the benefits of VCS without risking my data.

5. Keep your repositories small and clean – since every programmer gets a copy of the repository, it is better to keep it small. For large organization, it is not recommended to have one repository for all projects.
6. Don't commit binaries or generated files – A good rule of thumb is to include ones that were manually generated by programmers. The HTML files generated by R markdown language are good example that should not be added into the repository. Any intermediate data files should not be included either. Binary files such as R/SAS datasets take up much space without the benefit of diff so you should carefully consider committing them into the repository.
7. Explain your commits well – programmers should be able to communicate with inline comments and log messages. A well-written program doesn't even need many comments because it is self-explanatory. You should enter enough messages to explain what is going on. Don't just enter "fixed an error message". Tell us what the error message was.
8. Don't break the repository – if you did, fix it. Don't commit your programs that contain bugs or incomplete steps. Remember that you are working with others. Leaving your work in this state for others to find and fix is considered to be rude and unprofessional.

## EXTERNAL TOOLS

I stated above that the raw power of Git can only be unleashed via the command-line. The external tools listed below are some of the more common GUI tools you can use if you are not comfortable with learning the command-line syntax of Git. They take away much of the complexity and learning curve of Git yet provide enough power to manage day-to-day tasks.

### GITHUB DESKTOP

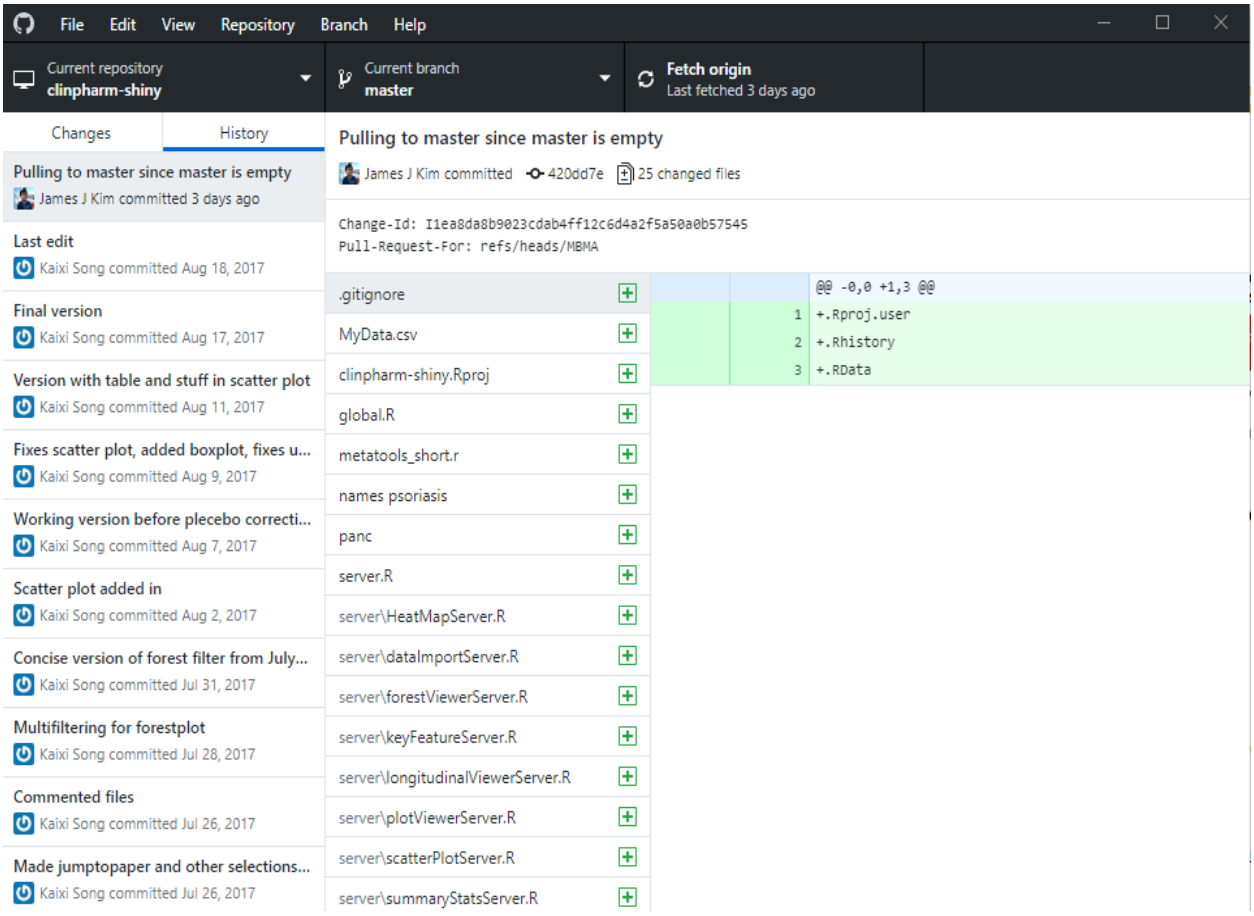


**Figure 5: GitHub Desktop**

GitHub is the most popular Git repository service especially with open source projects. And its desktop tool is a fantastic GUI that makes it easy to manage local and remote Git repositories. I don't recommend editing programs directly within GitHub Desktop since there are better IDEs (e.g. RStudio, SAS EG, etc.) or text editors (e.g. Sublime, Microsoft VS Code, etc.)

**TORTOISEGIT**

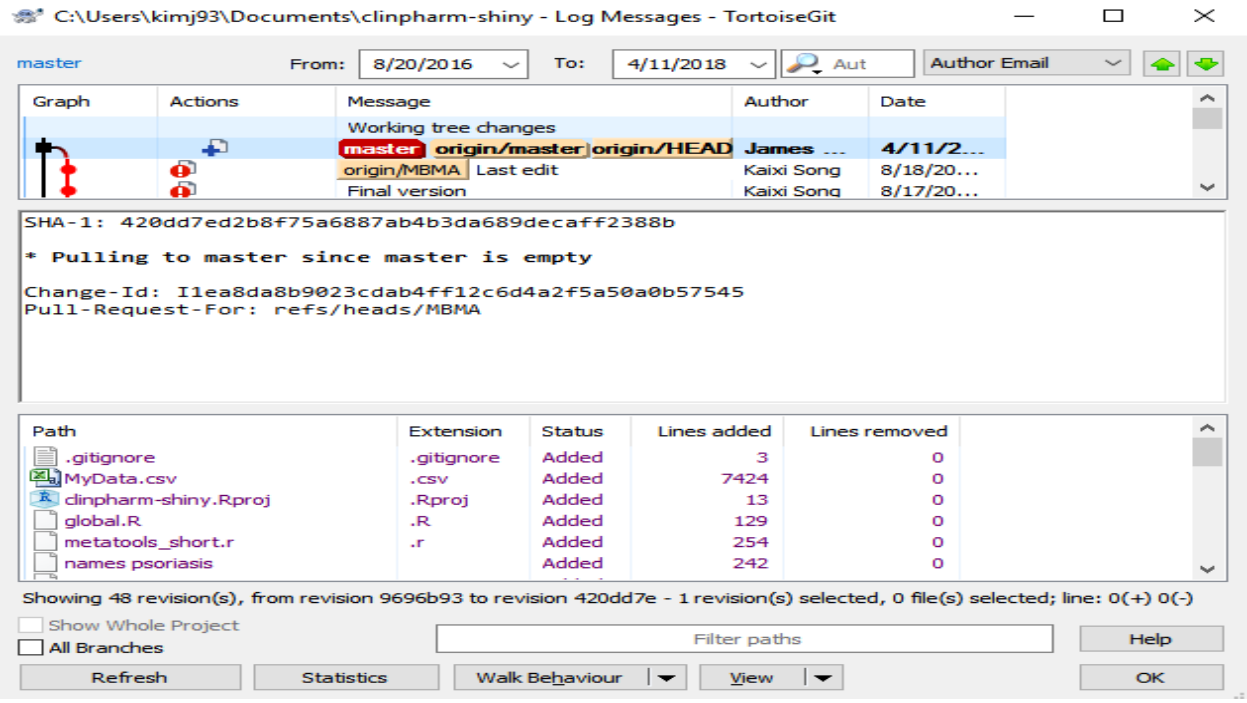


**Figure 6: TortoiseGit**

TortoiseGit is a Windows shell interface to Git and based on TortoiseSVN. Since it's not an integration for a specific IDE, you can use it freely outside of IDEs. Main interaction with TortoiseGit will be using the context menu of the Windows explorer. TortoiseGit supports regular tasks, such as committing, showing logs, difffing two versions, creating branches and tags, etc.

**WAFFLE.IO**

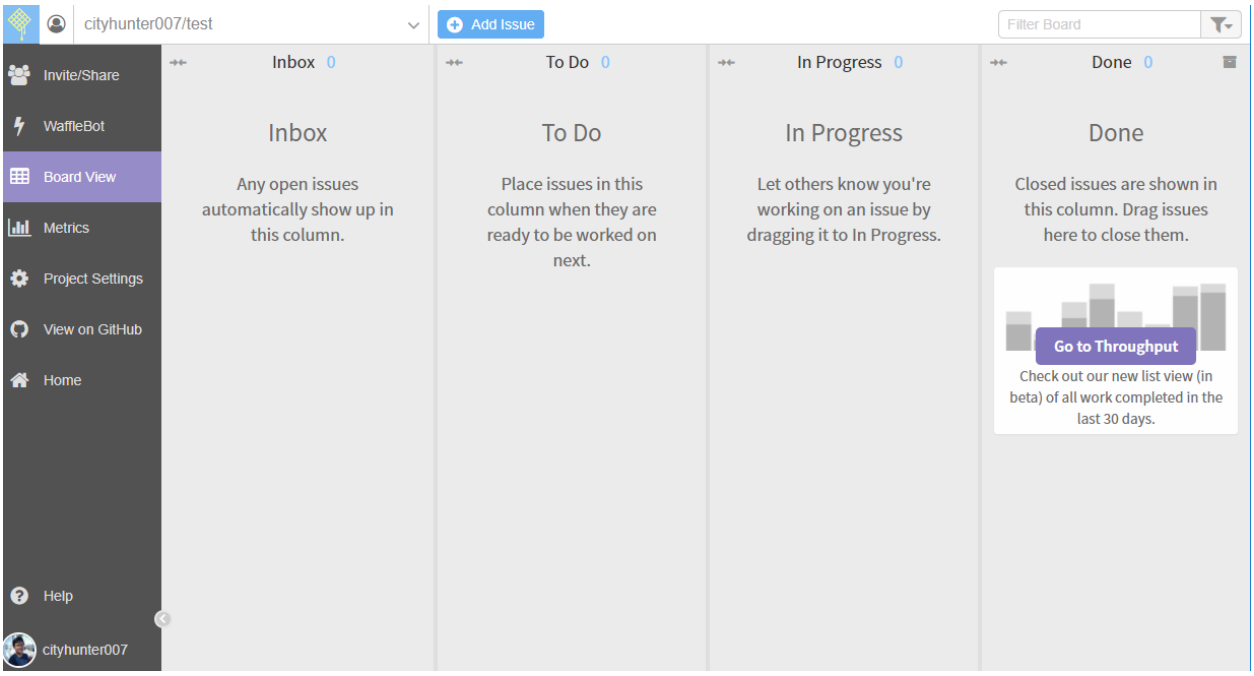


**Figure 7: Waffle.io**

Waffle is technically not a Git tool – it is a project management tool exclusively designed for GitHub. But it is worth mentioning since it has a tight integration with Git and some may find the tool useful. It helps to follow Kanban-style board for GitHub projects and issue tracking. There are other similar tools combinations such as Bitbucket/Jira integration, Slack and Keybase that offer repository hosting as well as project management tools.

## CONCLUSION

Many topics were presented for your information spanning from version control, programming methodology, tools and etc. Some topics such as double-programming are intentionally left open for debate and feedback. As we continue to move forward with adapting new techniques for building efficient programming teams across the globe in pharmaceutical industry, we must be transparent of our ethics and procedures. I believe supporting open attitude toward sharing and collaboration of codes is the best way to accomplish this.

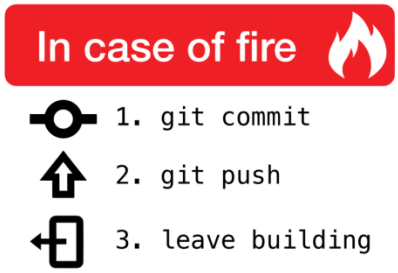


**Figure 8: In case of fire**

## REFERENCES

R Project – https://www.r-project.org/
The Comprehensive R Archive Network (CRAN) – https://cran.r-project.org/
RStudio – https://www.rstudio.com/
Git - https://git-scm.com/
GitHub – https://github.com/
GitHub Git cheatsheet – https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf
Bitbucket – https://bitbucket.org/
GitLab – https://gitlab.com/
TortoiseGit – https://tortoisegit.org/
Waffle.io – https://waffle.io/
Slack – https://slack.com/
Keybase – https://keybase.io/

## ACKNOWLEDGMENTS

The author likes to thank everyone who contributes to open source software for their hard works, including those who constantly contribute their R/SAS programs for sharing.

## RECOMMENDED READING

Git Tutorial – https://try.github.io/
RStudio Online learning – https://www.rstudio.com/online-learning/
R and Python tutorials at Datacamp – https://www.datacamp.com/

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

James Kim
Pfizer Inc.
james.j.kim@pfizer.com
GitHub – cityhunter007