**A White Paper by**

**Texas A&M University / Research Division **

**Research Information Systems / Maestro**

**3410 TAMU | College Station, TX 77843-3410**

[**http://maestro.tamus.edu**](http://maestro.tamus.edu)

**Git Technology and Process Migration**

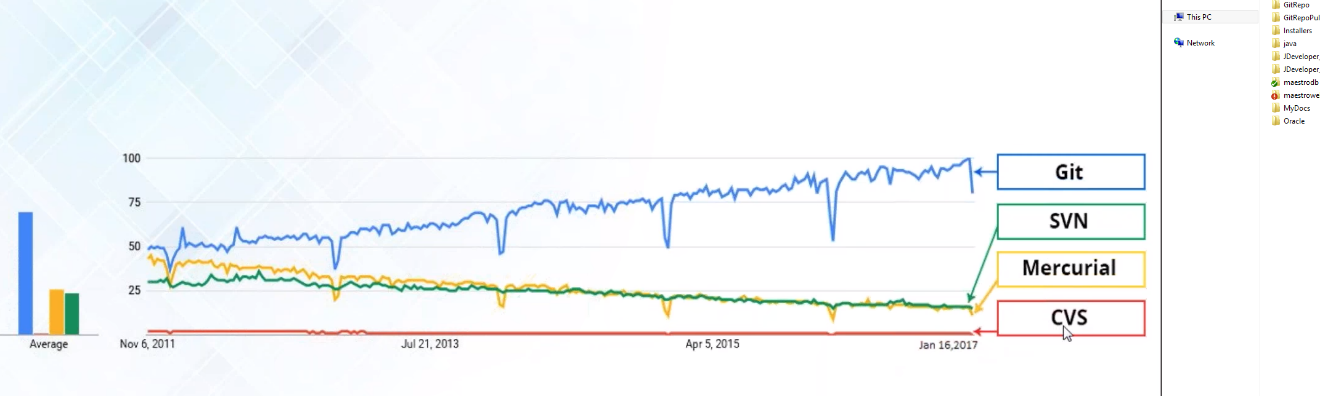
***Mohammad abu Shaira***

***Senior Lead Software Developer.***

***Software Development / Maestro***

# Why git?

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.



Minute 15 of https://www.youtube.com/watch?v=xuB1Id2Wxak

## Branching and Merging

The Git feature that really makes it stand apart from nearly every other SCM out there is its branching model.

Git allows and encourages you to have multiple local branches that can be entirely independent of each other. The creation, merging, and deletion of those lines of development takes seconds.

This means that you can do things like:

* **Frictionless Context Switching**. Create a branch to try out an idea, commit a few times, switch back to where you branched from, apply a patch, switch back to where you are experimenting, and merge it in.
* **Role-Based Codelines**. Have a branch that always contains only what goes to production, another that you merge work into for testing, and several smaller ones for day to day work.
* **Feature Based Workflow**. Create new branches for each new feature you're working on so you can seamlessly switch back and forth between them, then delete each branch when that feature gets merged into your main line.
* **Disposable Experimentation**. Create a branch to experiment in, realize it's not going to work, and just delete it - abandoning the work—with nobody else ever seeing it (even if you've pushed other branches in the meantime).



Notably, when you push to a remote repository, you do not have to push all of your branches. You can choose to share just one of your branches, a few of them, or all of them. This tends to free people to try new ideas without worrying about having to plan how and when they are going to merge it in or share it with others.

There are ways to accomplish some of this with other systems, but the work involved is much more difficult and error-prone. Git makes this process incredibly easy and it changes the way most developers work when they learn it.

## Small and Fast

**Git is fast**. With Git, nearly all operations are performed locally, giving it a huge speed advantage on centralized systems that constantly have to communicate with a server somewhere.

Git was built to work on the Linux kernel, meaning that it has had to effectively handle large repositories from day one. Git is written in C, reducing the overhead of runtimes associated with higher-level languages. Speed and performance has been a primary design goal of the Git from the start.

### Benchmarks

Let's see how common operations stack up against Subversion, a common centralized version control system that is similar to CVS or Perforce. Smaller is faster.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| init benchmarks | init benchmarks | init benchmarks | init benchmarks | init benchmarks | init benchmarks |
| init benchmarks | init benchmarks | init benchmarks | init benchmarks | init benchmarks | init benchmarks |

For testing, large AWS instances were set up in the same availability zone. Git and SVN were installed on both machines, the Ruby repository was copied to both Git and SVN servers, and common operations were performed on both.

In some cases the commands don't match up exactly. Here, matching on the lowest common denominator was attempted. For example, the 'commit' tests also include the time to push for Git, though most of the time you would not actually be pushing to the server immediately after a commit where the two commands cannot be separated in SVN.

All of these times are in seconds.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** |  | **Git** | **SVN** |
| Commit Files (A) | Add, commit and push 113 modified files (2164+, 2259-) | 0.64 | 2.60 | 4x |
| Commit Images (B) | Add, commit and push 1000 1k images | 1.53 | 24.70 | 16x |
| Diff Current | Diff 187 changed files (1664+, 4859-) against last commit | 0.25 | 1.09 | 4x |
| Diff Recent | Diff against 4 commits back (269 changed/3609+,6898-) | 0.25 | 3.99 | 16x |
| Diff Tags | Diff two tags against each other (v1.9.1.0/v1.9.3.0 ) | 1.17 | 83.57 | 71x |
| Log (50) | Log of the last 50 commits (19k of output) | 0.01 | 0.38 | 31x |
| Log (All) | Log of all commits (26,056 commits - 9.4M of output) | 0.52 | 169.20 | 325x |
| Log (File) | Log of the history of a single file (array.c - 483 revs) | 0.60 | 82.84 | 138x |
| Update | Pull of Commit A scenario (113 files changed, 2164+, 2259-) | 0.90 | 2.82 | 3x |
| Blame | Line annotation of a single file (array.c) | 1.91 | 3.04 | 1x |

Note that this is the best case scenario for SVN - a server with no load with an 80MB/s bandwidth connection to the client machine. Nearly all of these times would be even worse for SVN if that connection was slower, while many of the Git times would not be affected.

Clearly, in many of these common version control operations, **Git is one or two orders of magnitude faster than SVN**, even under ideal conditions for SVN.

One place where Git is slower is in the initial clone operation. Here, Git is downloading the entire history rather than only the latest version. As seen in the above charts, it's not considerably slower for an operation that is only performed once.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operation** |  | **Git\*** | **Git** | **SVN** |
| Clone | Clone and shallow clone(\*) in Git vs checkout in SVN | 21.0 | 107.5 | 14.0 |
| Size (M) | Size of total client side data and files after clone/checkout (in M) |  | 181.0 | 132.0 |

It's also interesting to note that the size of the data on the client side is very similar even though Git also has every version of every file for the entire history of the project. This illustrates how efficient it is at compressing and storing data on the client side.

## Distributed

One of the nicest features of any Distributed SCM, Git included, is that it's distributed. This means that instead of doing a "checkout" of the current tip of the source code, you do a "**clone**" of the entire repository.

### Multiple Backups

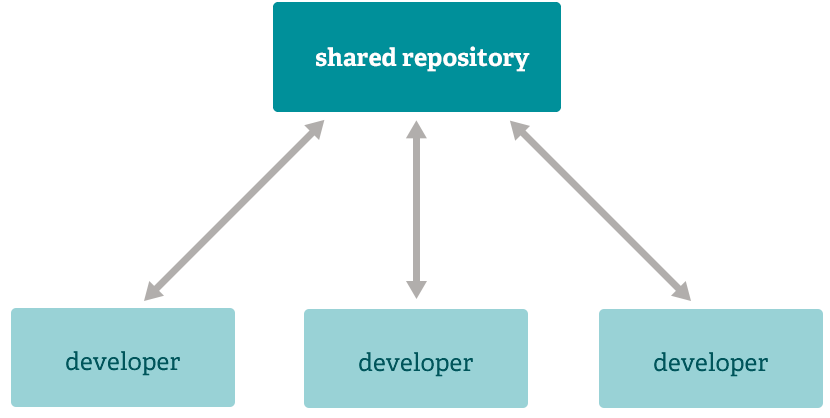
This means that even if you're using a centralized workflow, every user essentially has a full backup of the main server. Each of these copies could be pushed up to replace the main server in the event of a crash or corruption. In effect, there is no single point of failure with Git unless there is only a single copy of the repository.

### Any Workflow

Because of Git's distributed nature and superb branching system, an almost endless number of workflows can be implemented with relative ease.

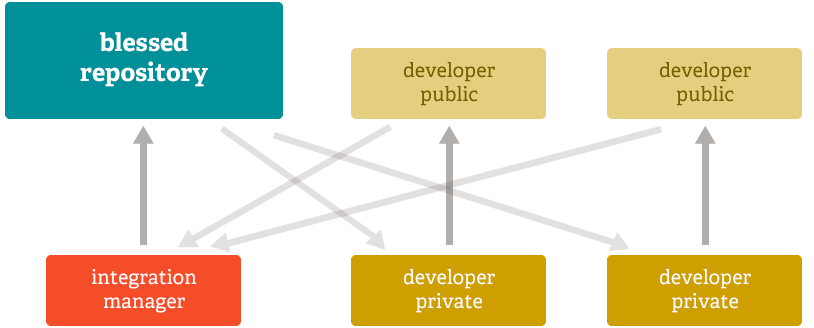
#### Subversion-Style Workflow

A centralized workflow is very common, especially from people transitioning from a centralized system. Git will not allow you to push if someone has pushed since the last time you fetched, so a centralized model where all developers push to the same server works just fine.



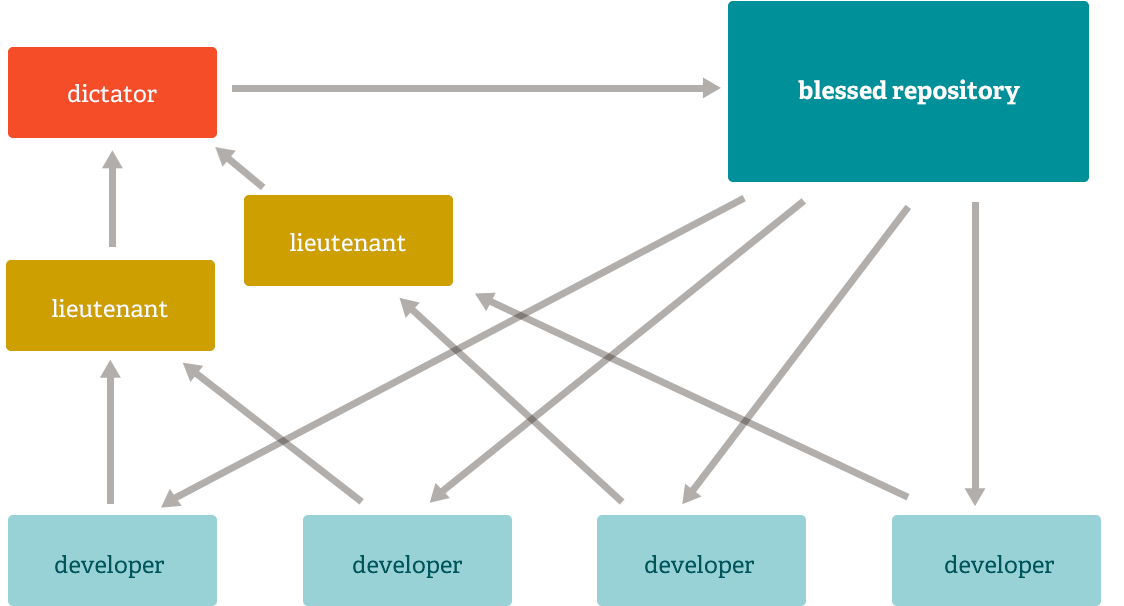
#### Integration Manager Workflow

Another common Git workflow involves an integration manager — a single person who commits to the 'blessed' repository. A number of developers then clone from that repository, push to their own independent repositories, and ask the integrator to pull in their changes. This is the type of development model often seen with open source or GitHub repositories.



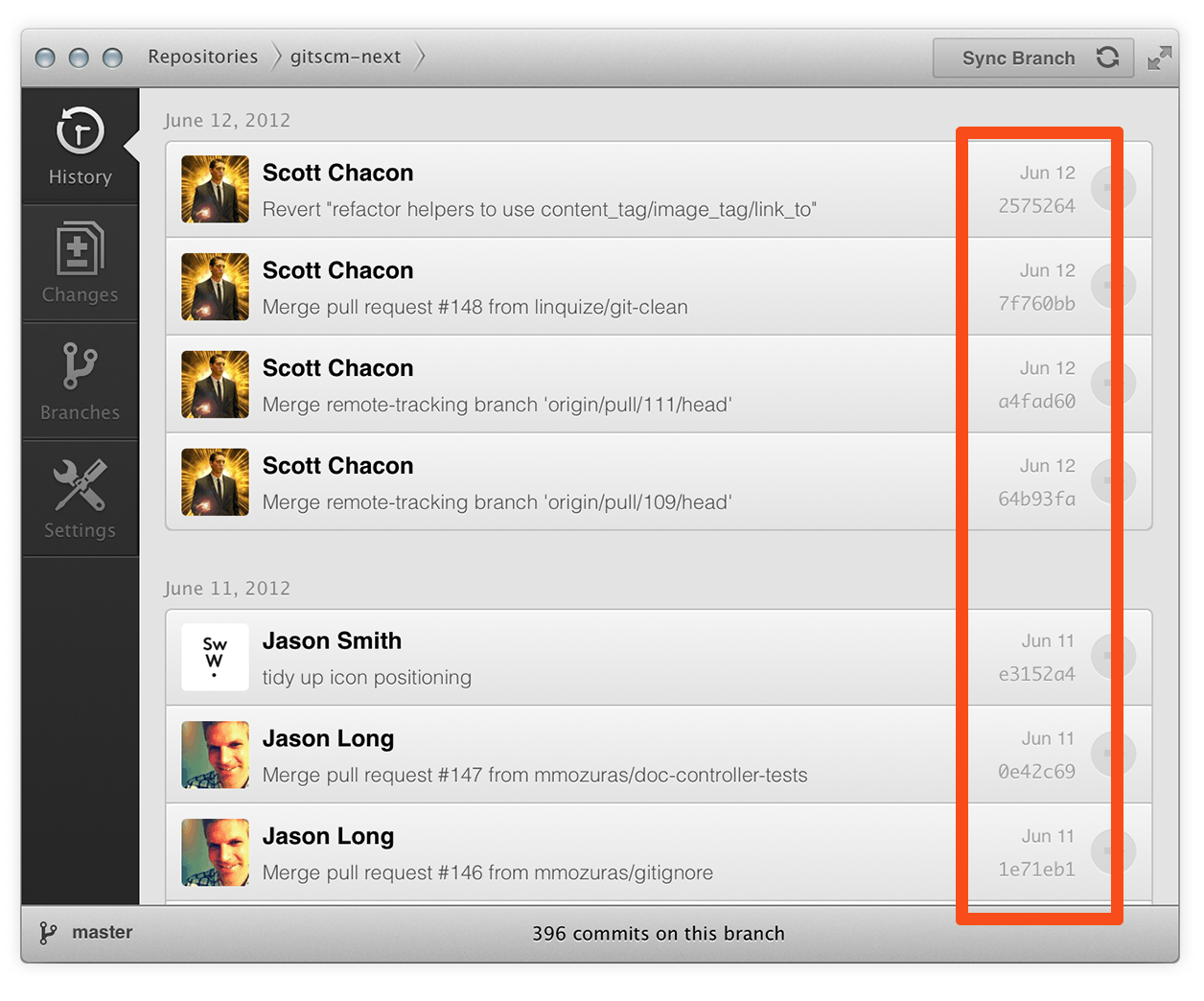
#### Dictator and Lieutenants Workflow

For more massive projects, a development workflow like that of the Linux kernel is often effective. In this model, some people ('lieutenants') are in charge of a specific subsystem of the project and they merge in all changes related to that subsystem. Another integrator (the 'dictator') can pull changes from only his/her lieutenants and then push to the 'blessed' repository that everyone then clones from again.



## Data Assurance

The data model that Git uses ensures the cryptographic integrity of every bit of your project. Every file and commit is checksummed and retrieved by its checksum when checked back out. It's impossible to get anything out of Git other than the **exact bits you put in**.



It is also impossible to change any file, date, commit message, or any other data in a Git repository without changing the IDs of everything after it. This means that if you have a commit ID, you can be assured not only that your project is exactly the same as when it was committed, but that nothing in its history was changed.

Most centralized version control systems provide no such integrity by default.

## Staging Area

Unlike the other systems, Git has something called the "staging area" or "index". This is an intermediate area where commits can be formatted and reviewed before completing the commit.

One thing that sets Git apart from other tools is that it's possible to quickly stage some of your files and commit them without committing all of the other modified files in your working directory or having to list them on the command line during the commit.



This allows you to stage only portions of a modified file. Gone are the days of making two logically unrelated modifications to a file before you realized that you forgot to commit one of them. Now you can just stage the change you need for the current commit and stage the other change for the next commit. This feature scales up to as many different changes to your file as needed.

Of course, Git also makes it easy to ignore this feature if you don't want that kind of control — just add a '-a' to your commit command in order to add all changes to all files to the staging area.



### Free and Open Source

Git is released under the [GNU General Public License version 2.0](http://opensource.org/licenses/GPL-2.0), which is an [open source license](http://www.opensource.org/docs/osd). The Git project chose to use GPLv2 to guarantee your freedom to share and change free software---to make sure the software is free for all its users.

However, we do restrict the use of the term "Git" and the [logos](https://git-scm.com/downloads/logos) to avoid confusion. Please see our [trademark](https://git-scm.com/trademark) policy for details.

# Process - Current VS Proposed

## **Used Technologies**

### Current

The used VCS System is **SVN**.

**Jira** is used for issues tracking.

**Jenkins** is used for automatic build and dev deploys.

### Proposed

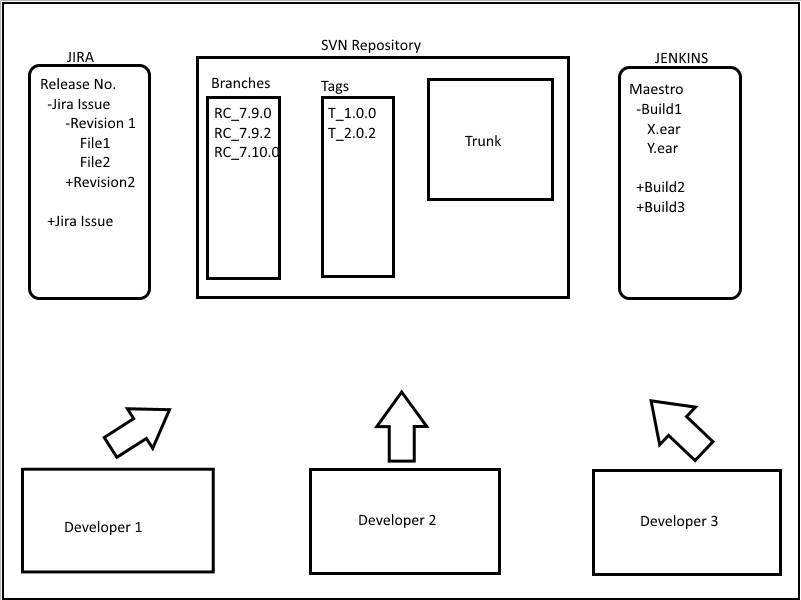
The proposed VCS system is a DVCS (Distributed Version Control System) which is **git**.

The proposed issue tracking system is **Jira**

The proposed tool system for automatic builds and dev deploys is **Jenkins**

## **Workflows**

### Current



1. Developers Checkout the Trunk from the SVN Repository.
2. Developers get assigned some Jira Issue, and start working on.
3. Developer 1 commits his work on Trunk for a specific JIRA issue – Uses Issue Key on Commit.
4. If Developer 1 is done with the JIRA Issue he will change its status from (In Progress -> Peer Review).
5. The peer review is done through Jira, and the reviewer can view the changes through Jira which opens files from SVN (URLs).
6. Jenkins after any commit will try to build Maestro, and checks if there is any missed file.
7. Jira after any commit will logs the amended files from the commit. Under this release you will see all JIRA issues, and under each JIRA issue you will see the revisions, and under each commit revision you will see the amended project files.
8. Now the Code Review (Peer Review) is done, and the reviewer changed the status to (Functional Review) and re-assigned the issue to a Functional Resource.
9. Now, the Release should be deployed to Dev, if Jinkens Tonight script did not deploy it, the developer will take this action
10. The developer will build Maestro using Jenkins (Dev).
11. The developer will deploy Maesto using Jenkins (Dev).
12. If the functional Review is Ok, the Build and Deploy should take place into (Test) environment.
13. Build and Deploy to (TEST):
    1. Start with Jira for the Specific Release you need to deploy to know which Jira Issues, Revisions, and Files that needs to be merged, in addition to know which DB script files needs to be moved/migrated to (Test).
    2. Send an email 2 days before the cutoff to notify the developers to finish their Jira issues in this release.
    3. For making it safe and for ordering purpose, it is a must to create a new excel file under the path Z:\RIS\M\Maestro\_Workarea\Developer\_Build\RC\_7.10.0, you should fill this file with the number of revisions that you will merge, and order these revisions ascending to merge them as the original order when they built for the first time.
    4. Create the new branch 7.10 on SVN, by clicking on SVN Checkout, and expanding the URL of the repository, coping the latest branch.
    5. Create a temp folder for the merge purpose under the C://RIS/Merge7.10 and checkout the new branch 7.10 in it.
    6. Merge, you can merge by right click on merge, and enter the specific revision.
    7. Commit, you need to put the issue key (which is the Jira No).
    8. You will see the merged files under Jira > Subversion tab.
    9. Modify jenkins build by Login to Jenkins, Click Production Tab, Click new Item button on the right, Give the build a name eg: RC\_5.5.0, Make sure you copy from the previous release. Press ok.

Make sure the new location is entered in the following build properties:build.full.version

Make sure that repository url is correct ie. <http://mpadm.tamu.edu/maestroweb/branches/RC_7.10.0>

1. Hand over to Build and Deploy to (QA)
2. Build and Deploy to (PROD)

### Proposed

There is no universal solutions for every team’s needs, but a good workflow positions all of the tools, processes, and people for optimum happiness and productivity.

Rather than presenting a prescriptive set of features or a pre-determined flow, git hub gives you the flexibility to define the best workflow for you.

One of the most straightforward branching workflow is the **GitHub workflow**.

https://guides.github.com/

TODO: https://www.youtube.com/watch?v=aJnFGMclhU8

https://www.youtube.com/watch?v=rFQbYSvz7ms&t=51s