Atlassian Bitbucket

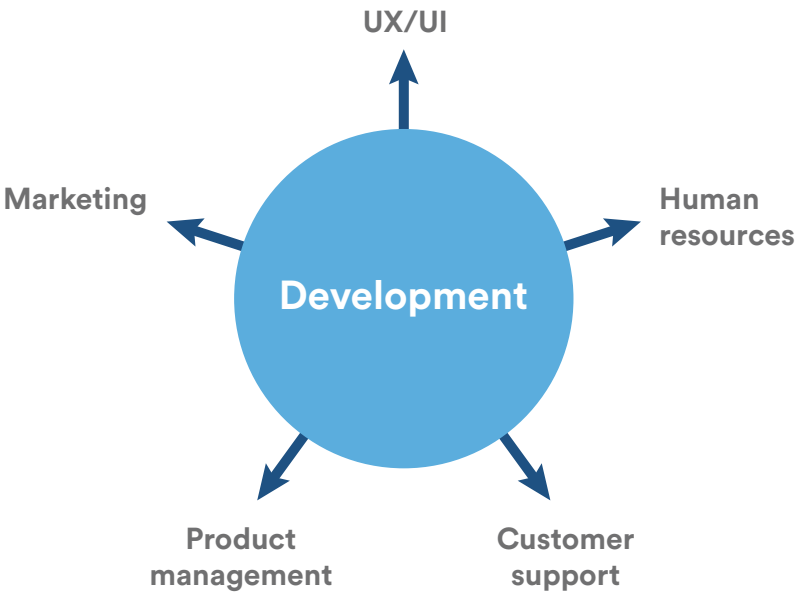
# SVN to Git

## Why teams are migrating and how to prepare for success

# Contents

# For the whole organization

Switching from a centralized version control system to Git changes the way your development team creates software. And, if you're a company that relies on its software for mission-critical applications, altering your development workflow impacts your entire business.

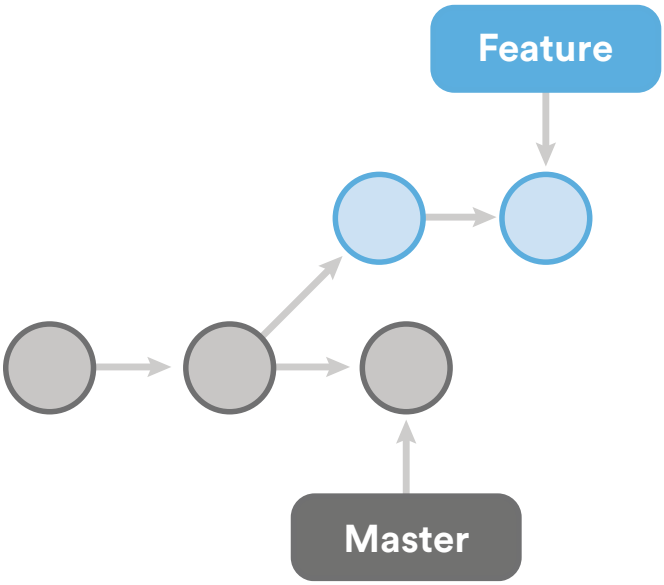



In this article, we'll discuss how Git benefits each aspect of your organization, from your development team to your marketing team, and everything in between. By the end of this article, it should be clear that Git isn't just for agile software development—it's for agile business.

## Git For Developers

### Feature Branch Workflow

One of the biggest advantages of Git is its branching capabilities. Unlike centralized version control systems, Git branches are cheap and easy to merge. This facilitates the feature branch workflow popular with many Git users.

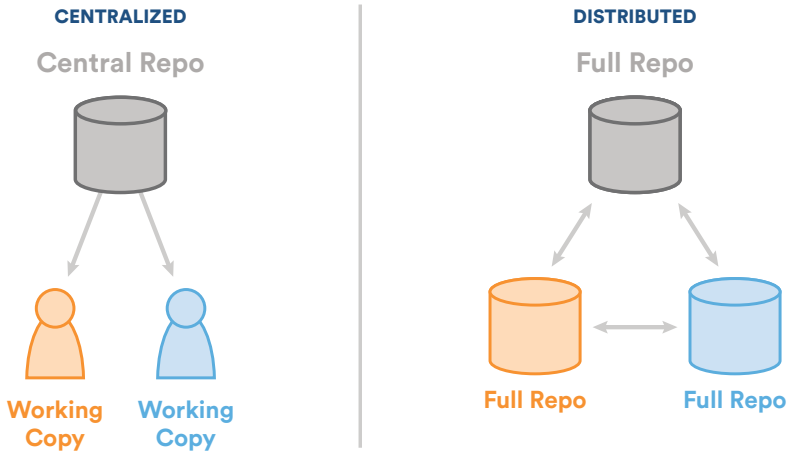



Feature branches provide an isolated environment for every change to your codebase. When a developer wants to start working on something—no matter how big or small—they create a new branch. This ensures that the master branch always contains production-quality code.

Using feature branches is not only more reliable than directly editing production code, but it also provides organizational benefits. They let you represent development work at the same granularity as the your agile backlog. For example, you might implement a policy where each JIRA ticket is addressed in its own feature branch.

### Distributed Development

In SVN, each developer gets a working copy that points back to a single central repository. Git, however, is a distributed version control system. Instead of a working copy, each developer gets their own local repository, complete with a full history of commits.



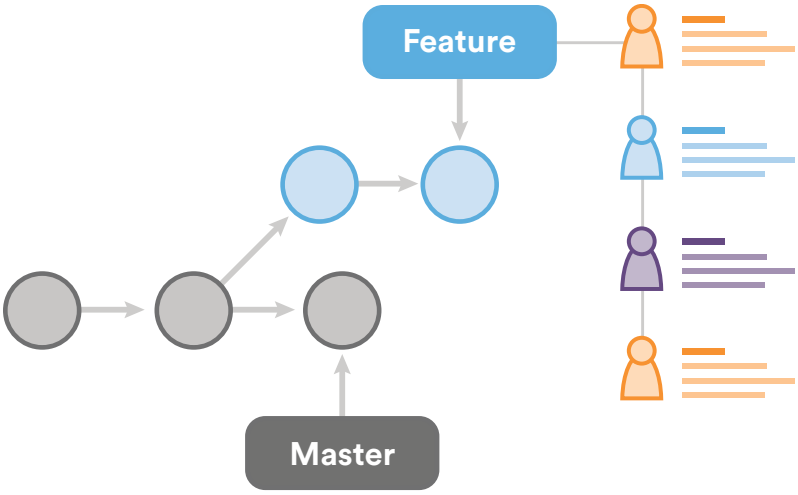

Having a full local history makes Git fast, since it means you don't need a network connection to create commits, inspect previous versions of a file, or perform diffs between commits.

Distributed development also makes it easier to scale your engineering team. If someone breaks the production branch in SVN, other developers can't check in their changes until it's fixed. With Git, this kind of blocking doesn't exist. Everybody can continue going about their business in their own local repositories.

And, similar to feature branches, distributed development creates a more reliable environment. Even if a developer obliterates their own repository, they can simply clone someone else's and start anew.

## Pull Requests

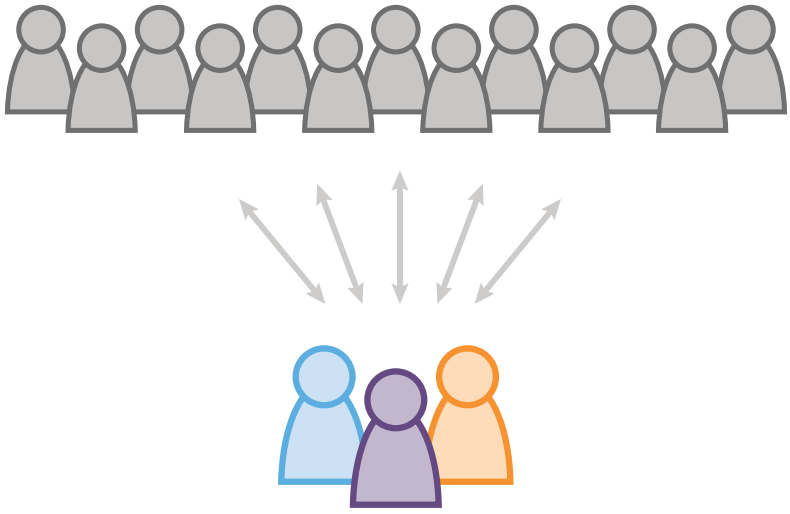
Many source code management tools such as Bitbucket or Stash enhance core Git functionality with pull requests. A pull request is a way to ask another developer to merge one of your branches into their repository. This not only makes it easier for project leads to keep track of changes, but also lets developers initiate discussions around their work before integrating it with the rest of the codebase.




Since they're essentially a comment thread attached to a feature branch, pull requests are extremely versatile. When a developer gets stuck with a hard problem, they can open a pull request to ask for help from the rest of the team. Alternatively, junior developers can be confident that they aren't destroying the entire project by treating pull requests as a formal code review.

## Community

In many circles, Git has come to be the expected version control system for new projects. If your team is using Git, odds are you won't have to train new hires on your workflow, because they'll already be familiar with distributed development.



In addition, Git is very popular among open source projects. This means it's easy to leverage 3rd-party libraries and encourage others to fork your own open source code.

## Faster Release Cycle

The ultimate result of feature branches, distributed development, pull requests, and a stable community is a faster release cycle. These capabilities facilitate an agile workflow where developers are encouraged to share smaller changes more frequently. In turn, changes can get pushed down the deployment pipeline faster than the monolithic releases common with centralized version control systems.


As you might expect, Git works very well with continuous integration and continuous delivery environments. Git hooks allow you to run scripts when certain events occur inside of a repository, which lets you automate deployment to your heart's content. You can even build or deploy code from specific branches to different servers.

For example, you might want to configure Git to deploy the most recent commit from the develop branch to a test server whenever anyone merges a pull request

into it. Combining this kind of build automation with peer review means you have the highest possible confidence in your code as it moves from development to staging to production.

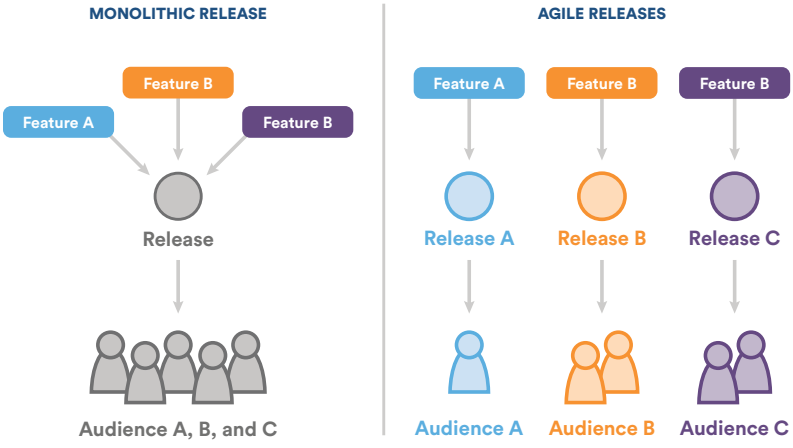
## Git For Marketing

To understand how switching to Git affects your company's marketing activities, imagine your development team has three distinct changes scheduled for completion in the next few weeks:

- The entire team is finishing up a game-changing feature that they've been working on for the last 6 months.
- Mary is implementing a smaller, unrelated feature that only impacts existing customers.
- Rick is making some much-needed updates to the user interface.

If you're using a traditional development workflow that relies on a centralized VCS, all of these changes would probably be rolled up into a single release. Marketing can only make one announcement that focuses primarily on the game-changing feature, and the marketing potential of the other two updates is effectively ignored.

The shorter development cycle facilitated by Git makes it much easier to divide these into individual releases. This gives marketers more to talk about, more often. In the above scenario, marketing can build out three campaigns that revolve around each feature, and thus target very specific market segments.

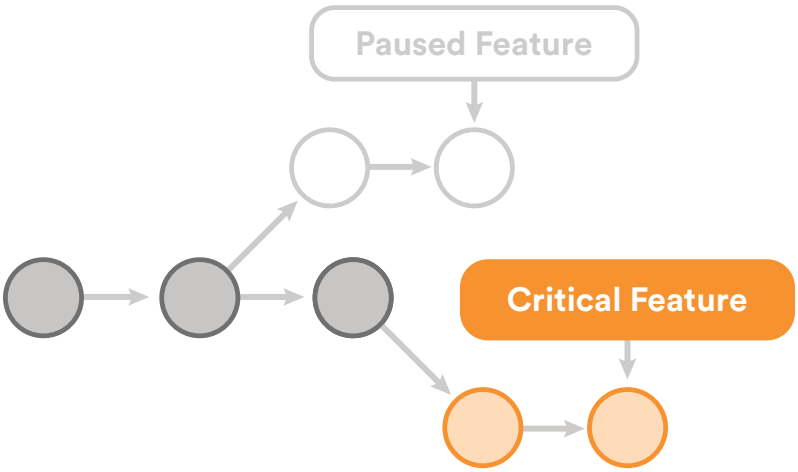



For instance, they might prepare a big PR push for the game changing feature, a corporate blog post and newsletter blurb for Mary's feature, and some guest posts

about Rick's underlying UX theory for sending to external design blogs. All of these activities can be synchronized with a separate release.

## Git For Product Management

The benefits of Git for product management is much the same as for marketing. More frequent releases means more frequent customer feedback and faster updates in reaction to that feedback. Instead of waiting for the next release 8 weeks from now, you can push a solution out to customers as quickly as your developers can write the code.

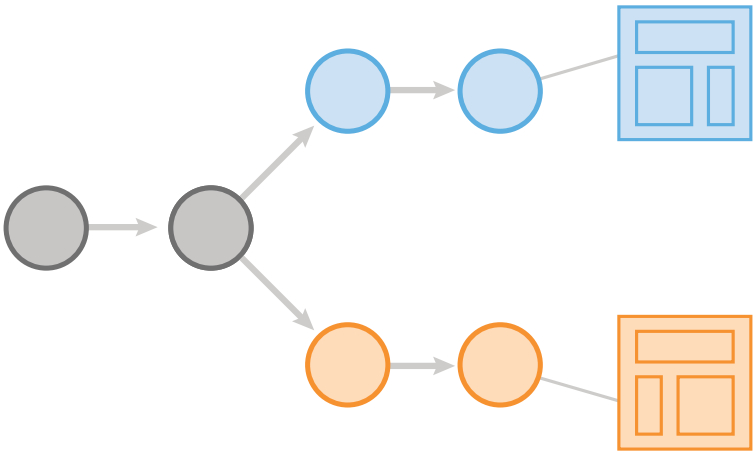



The feature branch workflow also provides flexibility when priorities change. For instance, if you're halfway through a release cycle and you want to postpone one feature in lieu of another time-critical one, it's no problem. That initial feature can sit around in its own branch until engineering has time to come back to it.

This same functionality makes it easy to manage innovation projects, beta tests, and rapid prototypes as independent codebases.

## Git For Designers

Feature branches lend themselves to rapid prototyping. Whether your UX/UI designers want to implement an entirely new user flow or simply replace some icons, checking out a new branch gives them a sandboxed environment to play with. This lets designers see how their changes will look in a real working copy of the product without the threat of breaking existing functionality.



Encapsulating user interface changes like this makes it easy to present updates to other stakeholders. For example, if the director of engineering wants to see what the design team has been working on, all they have to do is tell the director to check out the corresponding branch.

Pull requests take this one step further and provide a formal place for interested parties to discuss the new interface. Designers can make any necessary changes, and the resulting commits will show up in the pull request. This invites everybody to participate in the iteration process.

Perhaps the best part of prototyping with branches is that it's just as easy to merge the changes into production as it is to throw them away. There's no pressure to do either one. This encourages designers and UI developers to experiment while ensuring that only the best ideas make it through to the customer.

## Git For Customer Support

Customer support and customer success often have a different take on updates than product managers. When a customer calls them up, they're usually experiencing some kind of problem. If that problem is caused by your company's software, a bug fix needs to be pushed out as soon as possible.

Git's streamlined development cycle avoids postponing bug fixes until the next monolithic release. A developer can patch the problem and push it directly to production. Faster fixes means happier customers and fewer repeat support tickets. Instead of being stuck with, "Sorry, we'll get right on that" your customer support team can start responding with "We've already fixed it!

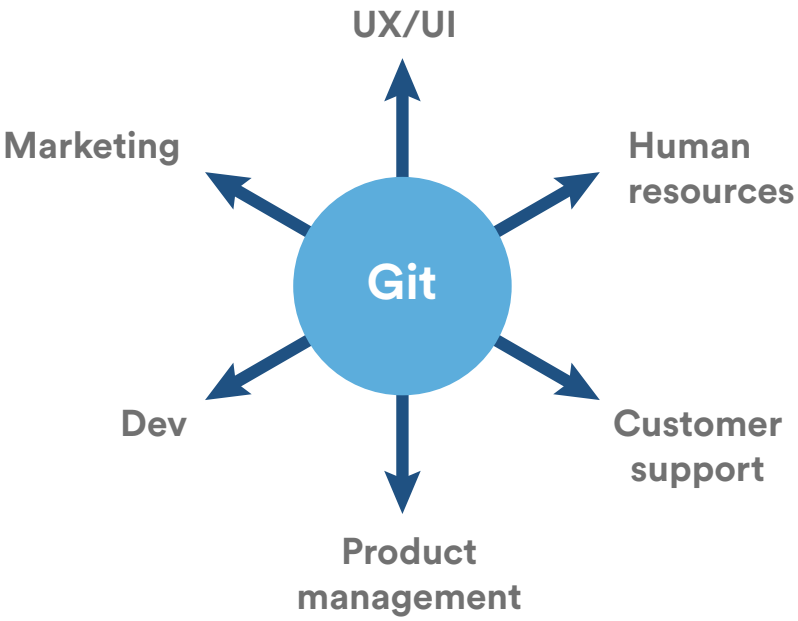
## Git For Human Resources

To a certain extent, your software development workflow determines who you hire. It always helps to hire engineers that are familiar with your technologies and workflows, but using Git also provides other advantages.

Employees are drawn to companies that provide career growth opportunities, and understanding how to leverage Git in both large and small organizations is a boon to any programmer. By choosing Git as your version control system, you're making the decision to attract forward-looking developers.

## Git For Anyone Managing a Budget

Git is all about efficiency. For developers, it eliminates everything from the time wasted passing commits over a network connection to the man hours required to integrate changes in a centralized version control system. It even makes better use of junior developers by giving them a safe environment to work in. All of this affects the bottom line of your engineering department.




But, don't forget that these efficiencies also extend outside your development team. They prevent marketing from pouring energy into collateral for features that aren't popular. They let designers test new interfaces on the actual product with little overhead. They let you react to customer complaints immediately.

Being agile is all about finding out what works as quickly as possible, magnifying efforts that are successful, and eliminating ones that aren't. Git serves as a multiplier for all your business activities by making sure every department is doing their job more efficiently.

# Prepping your team for the migration

In Part 1, we discussed the many ways that Git can help your team become more agile. Once you've decided to make the switch, your next step is to figure out how to migrate your existing development workflow to Git.

This article explains some of the biggest changes you'll encounter while transitioning your team from SVN to Git. The most important thing to remember during the migration process is that Git is not SVN. To realize the full potential of Git, try your best to open up to new ways of thinking about version control.

## For administrators

Adopting Git can take anywhere from a few days to several months depending on the size of your team. This section addresses some of the main concerns for engineering managers when it comes to training employees on Git and migrating repositories from SVN to Git.

### Learning Git & basic Git commands

Git once had a reputation for a steep learning curve. However the Git maintainers have been steadily releasing new improvements like sensible defaults and contextual help messages that have made the on-boarding process a lot more pleasant.

Atlassian offers a comprehensive series of self-paced Git tutorials, as well as webinars and live training sessions. Together, these should provide all the training options your team needs to get started with Git. To get you started, here are a list of some basic Git commands to get you going with Git:

| Git task | Notes | Git commands |
|---|---|---|
| Tell Git who you are | Configure the author name and email address to be used with your commits. Note that Git strips some characters (for example trailing periods) from user.name. | git config --global user.name \"Sam Smith\"git config --global user.email sam@example.com |
| Create a new local repository | | git init |
| Check out a repository | Create a working copy of a local repository: | git clone /path/to/repository |
| | For a remote server, use: | git clone username@host:/path/to/repository |
| Add files | Add one or more files to staging (index): | git add <filename><br><br>git add * |
| Commit | Commit changes to head (but not yet to the remote repository): | git commit -m \"Commit message\" |
| | Commit any files you've added with git add, and also commit any files you've changed since then: | git commit -a |
| Push | Send changes to the master branch of your remote repository: | git push origin master |
| Status | List the files you've changed and those you still need to add or commit: | git status |
| Connect to a remote repository | If you haven't connected your local repository to a remote server, add the server to be able to push to it: | git remote add origin <server> |
| | List all currently configured remote repositories: | git remote -v |
| Branches | Create a new branch and switch to it: | git checkout -b <branchname> |
| | Switch from one branch to another: | git checkout <branchname> |
| | List all the branches in your repo, and also tell you what branch you're currently in: | git branch |
| | Delete the feature branch: | git branch -d <branchname> |
| | Push the branch to your remote repository, so others can use it: | git push origin <branchname> |
| | Push all branches to your remote repository: | git push --all origin |
| | Delete a branch on your remote repository: | git push origin :<branchname> |

| Git task | Notes | Git commands |
|---|---|---|
| Update from the remote repository | Fetch and merge changes on the remote server to your working directory: | git pull |
| | To merge a different branch into your active branch: | git merge <branchname> |
| | View all the merge conflicts:View the conflicts against the base file:Preview changes, before merging: | git diffgit diff --base <filename>git diff <sourcebranch> <targetbranch> |
| | After you have manually resolved any conflicts, you mark the changed file: | git add <filename> |
| Tags | You can use tagging to mark a significant changeset, such as a release: | git tag 1.0.0 <commitID> |
| | CommitId is the leading characters of the changeset ID, up to 10, but must be unique. Get the ID using: | git log |
| | Push all tags to remote repository: | git push --tags origin |
| Undo local changes | If you mess up, you can replace the changes in your working tree with the last content in head:Changes already added to the index, as well as new files, will be kept. | git checkout -- <filename> |
| | Instead, to drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it, do this: | git fetch origin<br><br>git reset --hard origin/master |
| Search | Search the working directory for foo(): | git grep \"foo()\" |

## Migration tools

There's a number of tools available to help you migrate your existing projects from SVN to Git, but before you decide what tools to use, you need to figure out how you want to migrate your code. Your options are:

- Migrate your entire codebase to Git and stop using SVN altogether.
- Don't migrate any existing projects to Git, but use Git for all new projects.
- Migrate some of your projects to Git while continuing to use SVN for other projects.
- Use SVN and Git simultaneously on the same projects.

An complete transition to Git limits the complexity in your development workflow, so this is the preferred option. However, this isn't always possible in larger companies with dozens of development teams and potentially hundreds of projects. In these

situations, a hybrid approach is a safer option.

Your choice of migration tool(s) depends largely on which of the above strategies you choose. Some of the most common SVN-to-Git migration tools are introduced below.

### Atlassian's migration scripts

If you're interested in making an abrupt transition to Git, Atlassian's migration scripts are a good choice for you. These scripts provide all the tools you need to reliably convert your existing SVN repositories to Git repositories. The resulting native-Git history ensures you won't need to deal with any SVN-to-Git interoperability issues after the conversion process.

We've provided a complete technical walkthrough for using these scripts to convert your entire codebase to a collection of Git repositories. This walkthrough explains everything from extracting SVN author information to re-organizing non-standard SVN repository structures.

### SVN Mirror for Stash (now Bitbucket Server) plugin

SVN Mirror for Stash is a Bitbucket Server plugin that lets you easily maintain a hybrid codebase that works with both SVN and Git. Unlike Atlassian's migration scripts, SVN Mirror for Stash lets you use Git and SVN simultaneously on the same project for as long as you like.

This compromise solution is a great option for larger companies. It enables incremental Git adoption by letting different teams migrate workflows at their convenience.

### Git-SVN

The git svn tool that comes with Git serves as an interface between a local Git repository and a remote SVN repository. It lets developers write code and create commits locally with Git, then push them up to a central SVN repository with svn commit-style behavior.

git svn is a good option if you're not sure about making the switch to Git and want to let some of your developers explore Git commands without committing to a full-on migration. It's also perfect for the training phase—instead of an abrupt transition, your team can ease into it with local Git commands before worrying about

collaboration workflows.

Note that git svn should only be a temporary phase of your migration process. Since it still depends on SVN for the "backend," it can't leverage the more powerful Git features like branching or advanced collaboration workflows.

## Rollout Strategies

Migrating your codebase is only one aspect of adopting Git. You also need to consider how to introduce Git to the people behind that codebase. External consultants, internal Git champions, and pilots teams are the three main strategies for moving your development team over to Git.

## External Git Consultants

Git consultants can essentially handle the migration process for you for a nominal fee. This has the advantage of creating a Git workflow that's perfectly suited to your team without investing the time to figure it out on your own. It also makes expert training resources available to you while your team is learning Git. Atlassian Experts are pros when it comes to SVN to Git migration and are a good resource for sourcing a Git consultant.

On the other hand, designing and implementing a Git workflow on your own is a great way for your team to understand the inner workings of their new development process. This avoids the risk of your team being left in the dark when your consultant leaves.

## Internal Git Champions

A Git champion is a developer inside of your company who's excited to start using Git. Leveraging a Git champion is a good option for companies with a strong developer culture and eager programmers comfortable being early adopters. The idea is to enable one of your engineers to become a Git expert so they can design a Git workflow tailored to your company and serve as an internal consultant when it's time to transition the rest of the team to Git.

Compared to an external consultant, this has the advantage of keeping your Git expertise in-house. However, it requires a larger time investment to train that Git champion, and it runs the risk of choosing the wrong Git workflow or implementing it incorrectly.

## Pilot Teams

The third option for transitioning to Git is to test it out on a pilot team. This works best if you have a small team working on a relatively isolated project. This could work even better by combining external consultants with internal Git champions in the pilot team for a winning combo.

This has the advantage of requiring buy-in from your entire team, and also limits the risk of choosing the wrong workflow, since it gets input from the entire team while designing the new development process. In other words, it ensures any missing pieces are caught sooner than when a consultant or champion designs the new workflow on their own.

On the other hand, using a pilot team means more initial training and setup time: instead of one developer figuring out a new workflow, there's a whole team that could potentially be temporarily less productive while they're getting comfortable with their new workflow. However, this short term pain is absolutely worth the long term gain.

## Security and Permissions

Access control is an aspect of Git where you need to fundamentally re-think how you manage your codebase.

In SVN, you typically store your entire codebase in a single central repository, then limit access to different teams or individuals by folder. In Git, this is not possible: developers must retrieve the entire repository to work with it. You typically can not retrieve a subset of the repository, as you can with SVN. permissions can only be granted to entire Git repositories.

This means you have to split up your large, monolithic SVN repository into several small Git repositories. We actually experienced this first hand here at Atlassian when our JIRA development team migrated to Git. All of our JIRA plugins used to be stored in a single SVN repository, but after the migration, each plugin ended up in its own repository.

Keep in mind that Git was designed to securely integrate code contributions from thousands of independent Linux developers, so it definitely provides some way to set up whatever kind of access control your team needs. This may, however, require a fresh look at your build cycle.

If you're concerned about maintaining dependencies between your new collection of Git repositories, you may find a dependency management layer on top of Git helpful. A dependency management layer will help with build times because as a project grows, you need "caching" in order to speed up your build time. A list of recommended dependency management layer tools for every technology stack can be found in this helpful article: "Git and project dependencies."

## For Developers

### A Repository for Every Developer

As a developer, the biggest change you'll need to adjust to is the distributed nature of Git. Instead of a single central repository, every developer has their own copy of the entire repository. This dramatically changes the way you collaborate with your fellow programmers.

Instead of checking out an SVN repository with svn checkout and getting a working copy, you clone the entire Git repository to your local machine with git clone.

Collaboration occurs by moving branches between repositories with either git push, git fetch, or git pull. Sharing is commonly done on the branch level in Git but can be done on the commit level, similar to SVN. But in Git, a commit represents the entire state of the whole project instead rather than file modifications. Since you can use branches in both Git and SVN, the important distinction here is that you can commit locally with Git, without sharing your work. This enables you to experiment more freely, work more effectively offline and speeds up almost all version control related commands.

However, it's important to understand that a remote repository is not a direct link into somebody else's repository. It's simply a bookmark that prevents you from having to re-type the full URL each time you interact with a remote repository. Until you explicitly pull or push a branch to a remote repository, you're working in an isolated environment.

The other big adjustment for SVN users is the notion of "local" and "remote" repositories. Local repositories are on your local machine, and all other repositories are referred to as remote repositories. The main purpose of a remote repository is to make your code accessible to the rest of the team, and thus no active development takes place in them. Local repositories reside on your local machine, and it's where you do all of your software development.

## Don't Be Scared of Branching or Merging

In SVN, you commit code by editing files in your working copy, then running svn commit to send the code to the central repository. Everybody else can then pull those changes into their own working copies with svn update. SVN branches are usually reserved for large, long-running aspects of a project because merging is a dangerous procedure that has the potential to break the project.

Git's basic development workflow is much different. Instead of being bound to a single line of development (e.g., trunk/), life revolves around branching and merging.

When you want to start working on anything in Git, you create and check out a new branch with git checkout -b <branch-name>. This gives you a dedicated line of development where you can write code without worrying about affecting anyone else on your team. If you break something beyond repair, you simply throw the branch away with git branch -d <branch-name>. If you build something useful, you file a pull request asking to merge it into the master branch.

## Potential Git Workflows

When choosing a Git workflow it is important to consider your team's needs. A simple workflow can maximise development speed and flexibility, while a more complex workflow can ensure greater consistency and control of work in progress. You can adapt and combine the general approaches listed below to suit your needs and the different roles on your team. A core developer might use feature branches while a contractor works from a fork, for example.

- A centralized workflow provides the closest match to common SVN processes, so it's a good option to get started.
- Building on that idea, using a feature branch workflow lets developers keep their work in progress isolated and important shared branches protected. Feature branches also form the basis for managing changes via pull requests.
- A Gitflow workflow is a more formal, structured extension to feature branching, making it a great option for larger teams with well-defined release cycles.
- Finally, consider a forking workflow if you need maximum isolation and control over changes, or have many developers contributing to one repository.

But, if you really want to get the most out of Git as a professional team, you should

consider the feature branch workflow. This is a truly distributed workflow that is highly secure, incredibly scalable, and quintessentially agile.

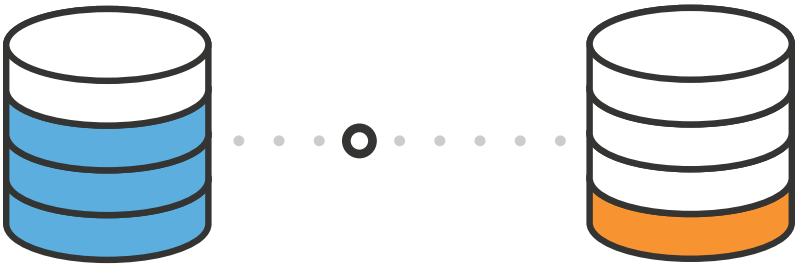
## Conclusion

Transitioning your team to Git can be a daunting task, but it doesn't have to be. This article introduced some of the common options for migrating your existing codebase, rolling out Git to your development teams, and dealing with security and permissions. We also introduced the biggest challenges that your developers should be prepared for during the migration process.

Hopefully, you now have a solid foundation for introducing distributed development to your company, regardless of its size or current development practices.

# Migrate to Git from SVN

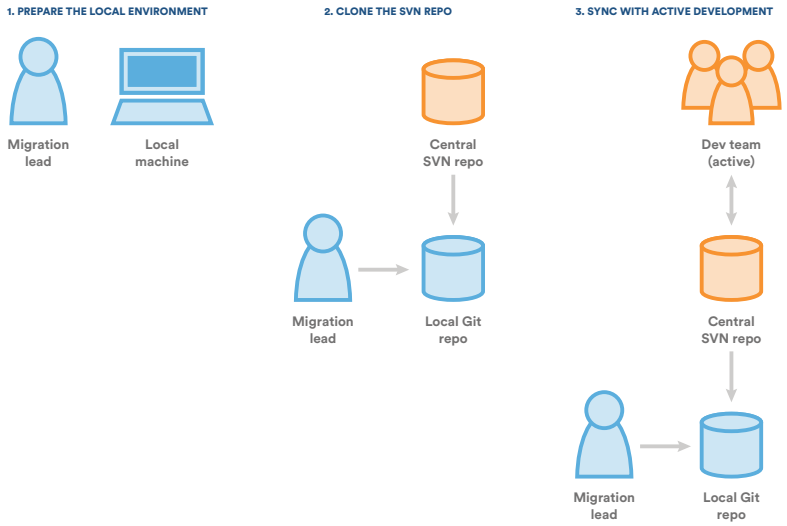
## Overview



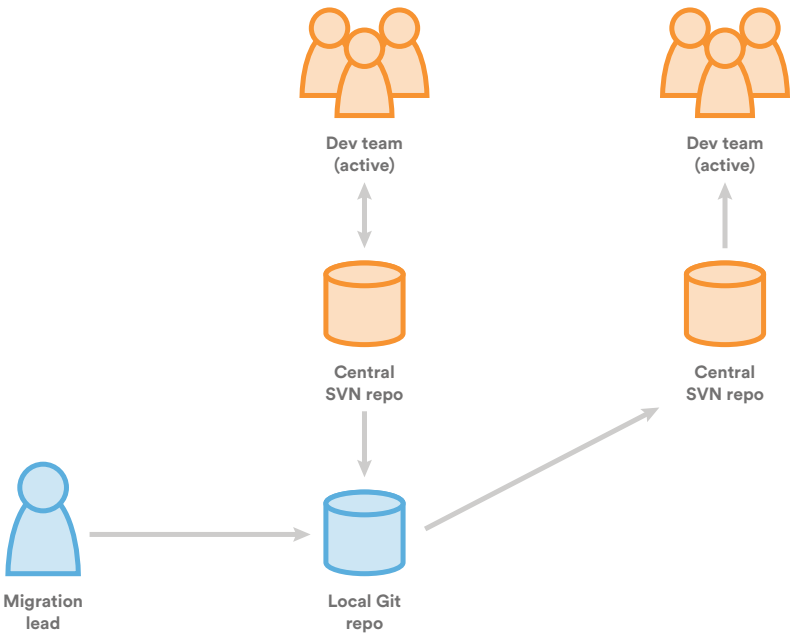
We've broken down the SVN-to-Git migration process into 5 simple steps:

1. Prepare your environment for the migration.
2. Convert the SVN repository to a local Git repository.
3. Synchronize the local Git repository when the SVN repository changes.
4. Share the Git repository with your developers via Bitbucket.
5. Migrate your development efforts from SVN to Git.

The prepare, convert, and synchronize steps take a SVN commit history and turn it into a Git repository. The best way to manage these first 3 steps is to designate one of your team members as the migration lead (if you're reading this guide, that person is probably you). All 3 of these steps should be performed on the migration lead's local computer.



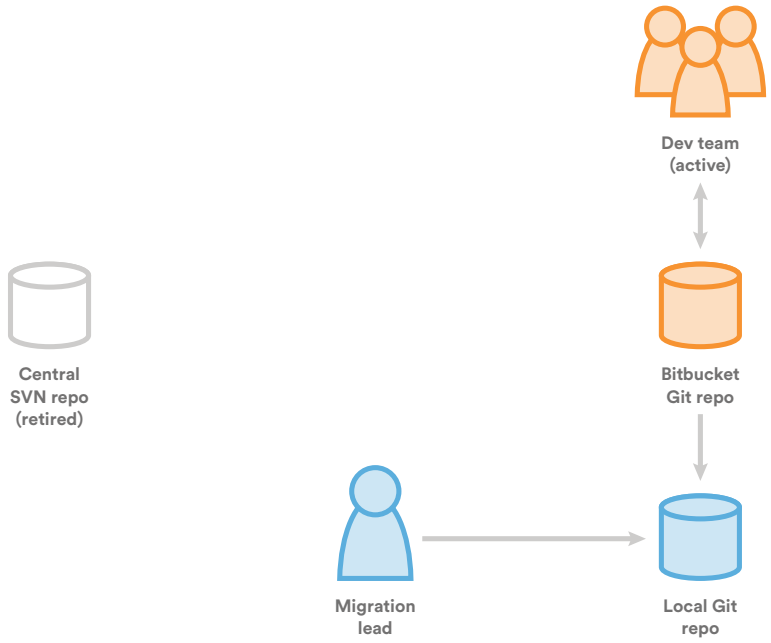

After the synchronize phase, the migration lead should have no trouble keeping a local Git repository up-to-date with an SVN counterpart. To share the Git repository, the migration lead can share his local Git repository with other developers by pushing it to Bitbucket, a Git hosting service.




Once it's on Bitbucket, other developers can clone the converted Git repository to their local machines, explore its history with Git commands, and begin integrating it into their build processes. However, we advocate a one-way synchronization from

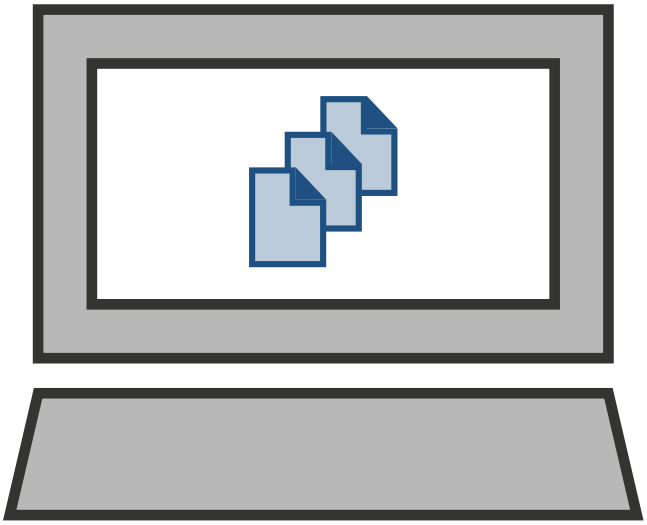
SVN to Git until your team is ready to switch to a pure Git workflow. This means that everybody should treat their Git repository as read-only and continue committing to the original SVN repository. The only changes to the Git repository should happen when the migration lead synchronizes it and pushes the updates to Bitbucket.

This provides a clear-cut transition period where your team can get comfortable with Git without interrupting your existing SVN-based workflow. Once you're confident that your developers are ready to make the switch, the final step in the migration process is to freeze your SVN repository and begin committing with Git instead.




This switch should be a very natural process, as the entire Git workflow is already in place and your developers have had all the time they need to get comfortable with it. By this point, you have successfully migrated your project from SVN to Git.

## Prepare



The first step to migrating a project from SVN to Git-based version control is to prepare the migration lead's local machine. In this phase, you'll download a convenient utility script, mount a case-sensitive filesystem (if necessary), and map author information from SVN to Git.

All of the the following steps should be performed on the migration lead's local machine.

### Download the migration script

Git comes with most of the necessary tools for importing an SVN repository; however, there are a few missing bits of functionality that Atlassian has rolled into a handy JAR file. This file will be integral to the migration, so be sure to download `svn-migration-scripts.jar` from Atlassian's Bitbucket account. This guide assumes that you've saved it in your home directory.

Once you've downloaded it, it's a good idea to verify the scripts to make sure you have the Java Runtime Environment, Git, Subversion, and the git-svn utility installed. Open a command prompt and run the following:

```
java -jar ~/svn-migration-scripts.jar verify
```

This will display an error message in the console if you don't have the necessary programs for the migration process. Make sure that any missing software is installed

before moving on.

If you get a warning about being unable to determine a version, run export LANG=C (*nix) or SET LANG=C (Windows) and try again.

If you're performing the migration on a computer running OS X, you'll also see the following warning:

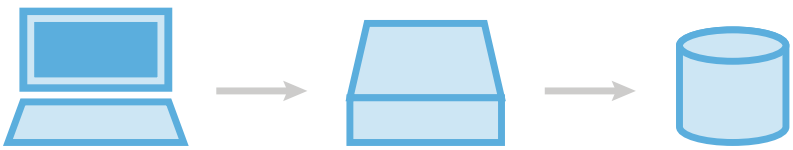
```
You appear to be running on a case-insensitive file-system. This is
unsupported, and can result in data loss.
```

We'll address this in the next section.

## Mount a case-sensitive disk image

Migrating to Git should be done on a case-sensitive file system to avoid corrupting the repository. This is a problem if you're performing the migration on an OS X computer, as the OS X filesystem isn't case-sensitive.

If you're not running OS X, all you need to do is create a directory on your local machine called `~/GitMigration`. This is where you will perform the conversion. After that, you can skip to the next section.




If you are running OS X, you need to mount a case-sensitive disk image with the `create-disk-image` script included in `svn-migration-scripts.jar`. It takes two parameters:

1. The size of the disk image to create in gigabytes. You can use any size you like, as long as it's bigger than the SVN repository that you're trying to migrate.
2. The name of the disk image. This guide uses `GitMigration` for this value.

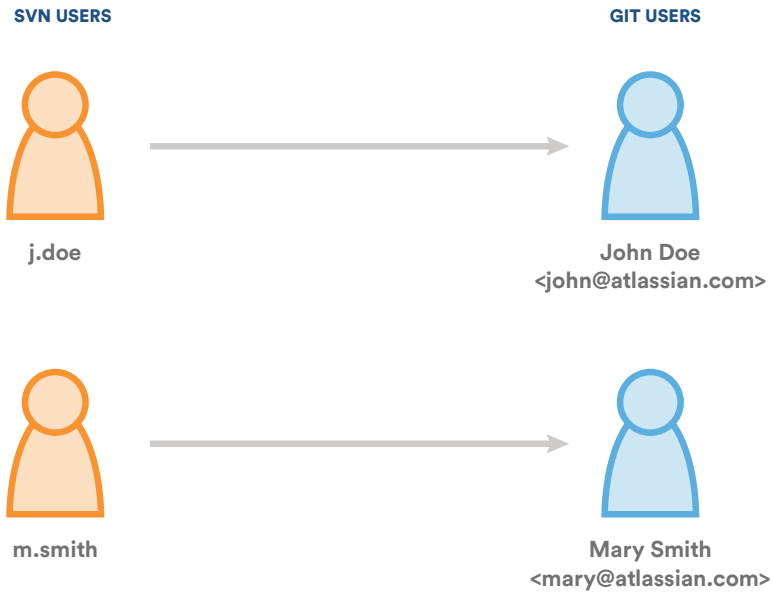
For example, the following command creates a 5GB disk image called `GitMigration`:

```
java -jar ~/svn-migration-scripts.jar create-disk-image 5 GitMigration
```

The disk image is mounted in your home directory, so you should now see a directory called `~/GitMigration` on your local machine. This serves as a virtual case-sensitive filesystem, and it's where you'll store the converted Git repository.

## Extract the author information

SVN only records the username of the author for each revision. Git, however, stores the full name and email address of the author. This means that you need to create a text file that maps SVN usernames to their Git counterparts.




Run the following commands to automatically generate this text file:

```
cd ~/GitMigration
java -jar ~/svn-migration-scripts.jar authors <svn-repo> > authors.txt
```

Be sure to replace `<svn-repo>` with the URI of the SVN repository that you want to migrate. For example, if your repository resided at `https://svn.example.com`, you would run the following:

```
java -jar ~/svn-migration-scripts.jar authors https://svn.example.com >
authors.txt
```

This creates a text file called `authors.txt` that contains the username of every

author in the SVN repository along with a generated name and email address. It should look something like this:

```
j.doe = j.doe <j.doe@mycompany.com>
 m.smith = m.smith <m.smith@mycompany.com>
```

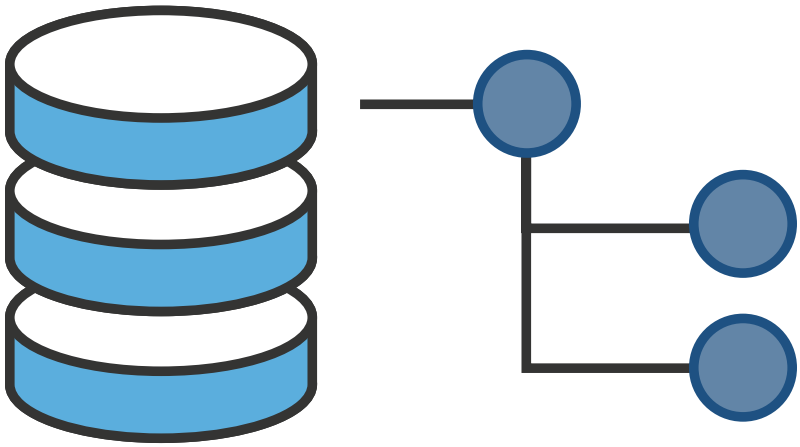
Change the portion to the right of the equal sign to the full name and email address of the corresponding user. For example, you might change the above authors to:

```
j.doe = John Doe <john.doe@atlassian.com>
 m.smith = Mary Smith <mary.smith@atlassian.com>
```

### Summary

Now that you have your migration scripts, disk image (OS X only), and author information, you're ready to import your SVN history into a new Git repository. The next phase explains how this conversion works.

## Convert



The next step in the migration from SVN to Git is to import the contents of the SVN repository into a new Git repository. We'll do this with the `git svn` utility that is included with most Git distributions, then we'll clean up the results with `svn-migration-scripts.jar`.

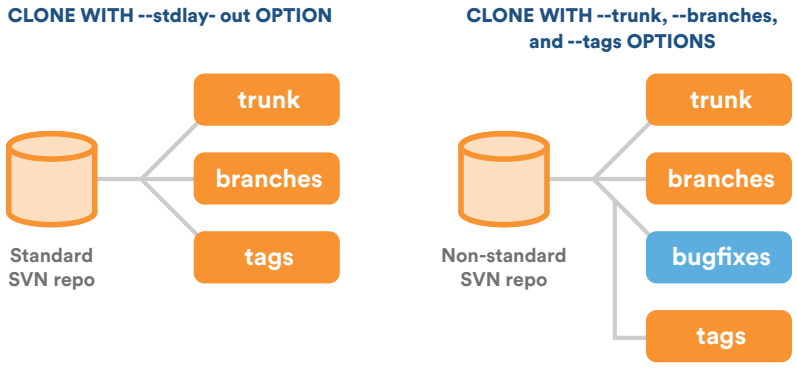
Beware that the conversion process can take a significant amount of time for larger repositories, even when cloning from a local SVN repository. As a benchmark,

converting a 400MB repository with 33,000 commits on master took around 12 hours to complete.

For reasonably sized repositories, the following steps should be run on the migration lead's local computer. However, if you have a very large SVN repository and want to cut down on the conversion time, you can run `git svn clone` on the SVN server instead of on the migration lead's local machine. This will avoid the overhead of cloning via a network connection.

## Clone the SVN repository

The git svn clone command transforms the trunk, branches, and tags in your SVN repository into a new Git repository. Depending on the structure of your SVN repo, the command needs to be configured differently.




## Standard SVN layouts

If your SVN project uses the standard `/trunk`, `/branches`, and `/tags` directory layout, you can use the `--stdlayout` option instead of manually specifying the repository's structure. Run the following command in the `~/GitMigration` directory:

```
git svn clone --stdlayout --authors-file=authors.txt
 <svn-repo>/<project> <git-repo-name>
```

Where `<svn-repo>` is the URI of the SVN repository that you want to migrate and, `<project>` is the name of the project that you want to import, and `<git-repo-name>` is the directory name of the new Git repository.

For example, if you were migrating a project called `Confluence`, hosted on

`https://svn.atlassian.com`, you might run the following:

```
git svn clone --stdlayout --authors-file=authors.txt
 https://svn.atlassian.com/Confluence ConfluenceAsGit
```

## Non-standard SVN layouts

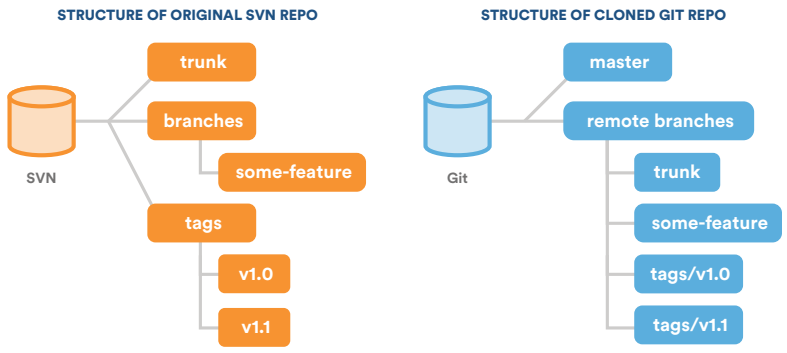
If your SVN repository doesn't have a standard layout, you need to provide the locations of your trunk, branches, and tags using the `--trunk`, `--branches`, and `--tags` command line options. For example, if you have branches stored in both the `/branches` directory and the `/bugfixes` directories, you would use the following command:

```
git svn clone --trunk=/trunk --branches=/branches
 --branches=/bugfixes --tags=/tags --authors-file=authors.txt
 <svn-repo>/<project> <git-repo-name>
```

## Inspect the new Git repository

After `git svn clone` has finished (this might take a while), you'll find a new directory called `<git-repo-name>` in `~/GitMigration`. This is the converted Git repository. You should be able to switch into `<git-repo-name>` and run any of the standard Git commands to explore your project.

Branches and tags are not imported into the new Git repository as you might expect. You won't find any of your SVN branches in the `git branch` output, nor will you find any of your SVN tags in the `git tag` output. But, if you run `git branch -r`, you'll find all of the branches and tags from your SVN repository. The `git svn clone` command imports your SVN branches as remote branches and imports your SVN tags as remote branches prefixed with `tags/`.



This behavior makes certain two-way synchronization procedures easier, but it can be very confusing when trying to make a one-way migration Git. That's why our next step will be to convert these remote branches to local branches and actual Git tags.

## Clean the new Git repository

The `clean-git` script included in `svn-migration-scripts.jar` turns the SVN branches into local Git branches and the SVN tags into full-fledged Git tags. Note that this is a **destructive** operation, and you will not be able to move commits from the Git repository back into the SVN repository.

If you're following this migration guide, this isn't a problem, as it advocates a one-way sync from SVN to Git (the Git repository is considered read-only until after the Migrate step). However, if you're planning on committing to the Git repository *and* the SVN repository during the migration process, you should not perform the following commands. This is an advanced task, as is not recommended for the typical project.

To see what can be cleaned up, run the following command in `~/GitMigration/<git-repo-name>`:

```
java -Dfile.encoding=utf-8 -jar ~/svn-migration-scripts.jar clean-git
```

This will output all of the changes the script wants to make, but it won't actually make any of them. To execute these changes, you need to use the `--force` option, like so:

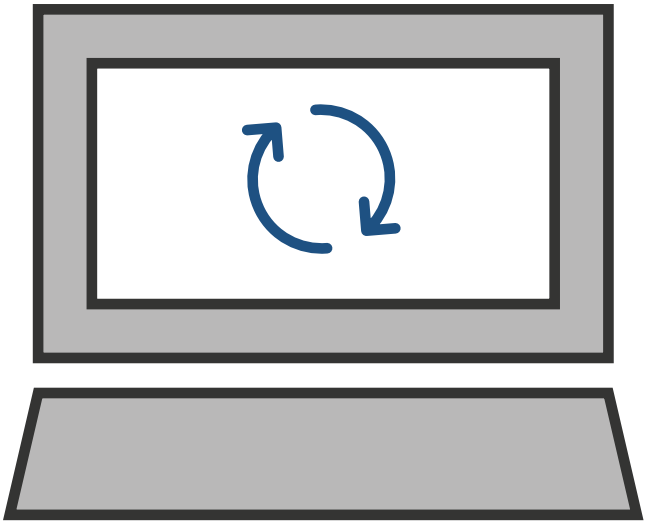
```
java -Dfile.encoding=utf-8 -jar ~/svn-migration-scripts.jar clean-git
--force
```

You should now see all of your SVN branches in the git branch output, along with your SVN tags in the `git tag` output. This means that you've successfully converted your SVN project to a Git repository.

## Summary

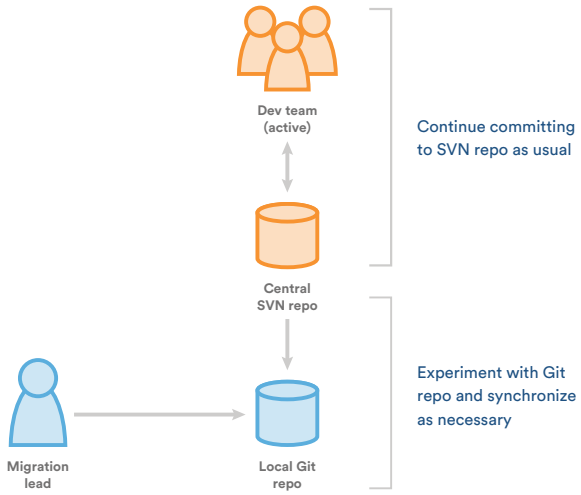
In this step, you turned an SVN repository into a new Git repository with the `git svn clone` command, then cleaned up the structure of the resulting repository with `svn-migration-scripts.jar`. In the next step, you'll learn how to keep this new Git repo in sync with any new commits to the SVN repository. This will be a similar process to the conversion, but there are some important workflow considerations during this transition period.

## Synchronize



It's very easy to synchronize your Git repository with new commits in the original SVN repository. This makes for a comfortable transition period in the migration process where you can continue to use your existing SVN workflow, but begin to experiment with Git.

It's possible to synchronize in both directions. However, we recommend a one-way sync from SVN to Git. During your transition period, you should only commit to your SVN repository, not your Git repo. Once you're confident that your team is ready to make the switch, you can complete the migration process and begin to commit changes with Git instead of SVN.



In the meantime, you should continue to commit to your SVN repository and synchronize your Git repository whenever necessary. This process is similar to the Convert phase, but since you're only dealing with incremental changes, it should be much more efficient.

### Update the authors file

The `authors.txt` file that we used to map SVN usernames to full names and email addresses is essential to the synchronization process. If it has been moved from the `~/GitMigration/authors.txt` location that we've been using thus far, you need to update its location with:

```
git config svn.authorsfile <path-to-authors-file>
```

If new developers have committed to the SVN repository since the last sync (or the initial clone), the authors file needs to be updated accordingly. You can do this by manually appending new users to `authors.txt`, or you can use the `--authors-prog` option, as discussed in the next section.

For one-off synchronizations it's often easier to directly edit the authors file; however, the `--authors-prog` option is preferred if you're performing unsupervised syncs (i.e. in a scheduled task).

### Automatically generating Git authors

If your authors file doesn't need to be updated, you can skip to the next section.

The git svn command includes an option called `--authors-prog`, which points to a script that automatically transforms SVN usernames into Git authors. You'll need to configure this script to accept the SVN username as its only argument and return a single line in the form of Name `<email>` (just like the right hand side of the existing authors file). This option can be very useful if you need to periodically add new developers to your project.

If you want to use the `--authors-prog` option, create a file called authors.sh option in `~/GitMigration`. Add the following line to authors.sh to return a dummy Git name and email for any authors that aren't found in `authors.txt`:

```
echo "$1 <$1@example.com>"
```

Again, this will only generate a dummy name and email based on the SVN

username, so feel free to alter it if you can provide a more meaningful mapping.

### Fetch the new SVN commits

Unlike SVN, Git makes a distinction between *downloading* upstream commits and *integrating* them into the project. The former is called "fetching", while the latter can be done via merging or rebasing. In the `~/GitMigration` directory, run the following command to fetch any new commits from the original SVN repository.

```
git svn fetch
```

This is similar to the `git svn clone` command from the previous phase in that it only updates the Git repository's remote branches—the local branches will not reflect any of the updates yet. Your remote branches, on the other hand, should exactly match your SVN repo's history.

If you're using the `--authors-prog` option, you need include it in the above command, like so:

```
git svn fetch --authors-prog=authors.sh
```

### Synchronize with the fetched commits

To apply the downloaded commits to the repository, run the following command:

```
java -Dfile.encoding=utf-8 -jar ~/svn-migration-scripts.jar sync-rebase
```

This will rebase the fetched commits onto your local branches so that they match their remote counterparts. You should now be able to see the new commits in your `git log` output.

### Clean up the Git repo (again)

It's also a good idea to run the `git-clean` script again to remove any obsolete tags or branches that were deleted from the original SVN repository since the last sync:

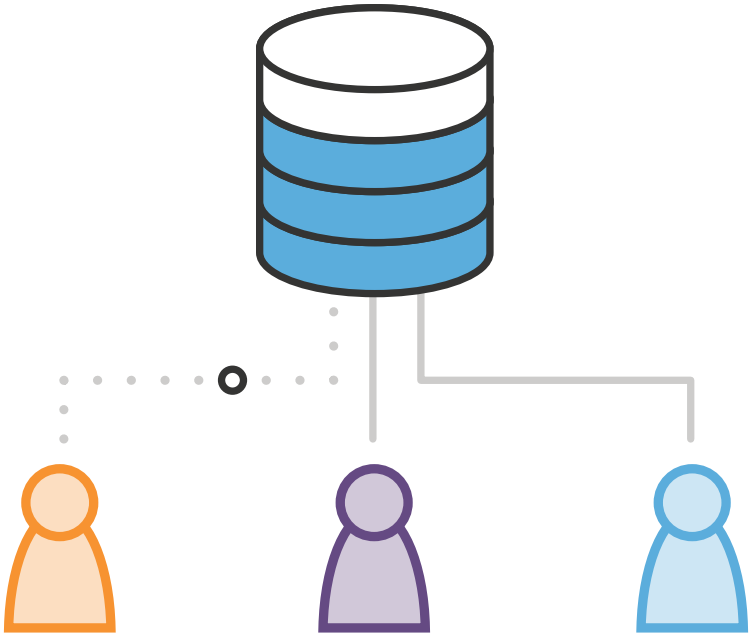
```
java -Dfile.encoding=utf-8 -jar ~/svn-migration-scripts.jar clean-git
 --force
```

Your local Git repository should now be synchronized with your SVN repository.

### Summary

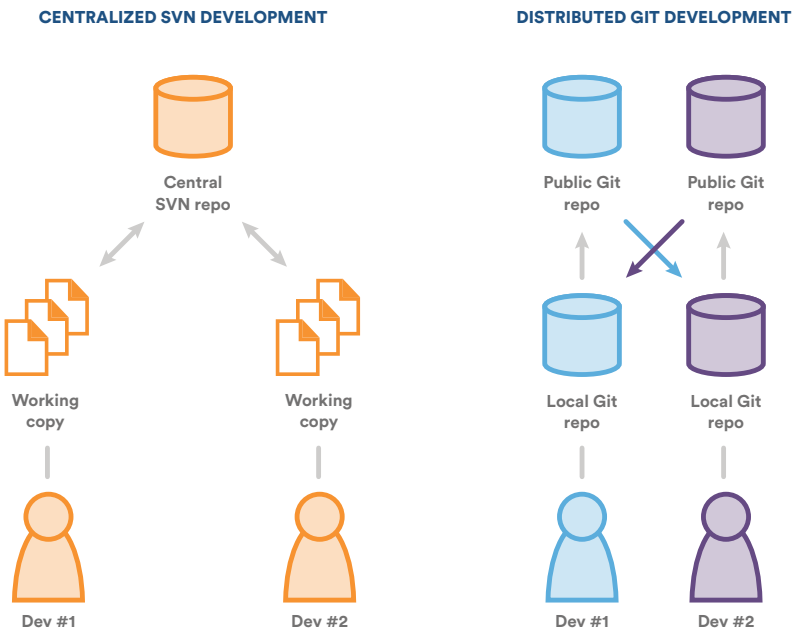
During this transition period, it's very important that your developers only commit to the original SVN repository. The only time the Git repository should be updated is via the synchronization process discussed above. This is much easier than managing a two-way synchronization workflow, but it still allows you to start integrating Git into your build process.

## Share



In SVN, developers share contributions by committing changes from a working copy on their local computer to a central repository. Then, other developers pull these updates from the central repo into their own local working copies.

Git's collaboration workflow is much different. Instead of differentiating between working copies and the central repository, Git gives each developer their own local copy of the *entire* repository. Changes are committed to this local repository instead of a central one. To share updates with other developers, you need to push these local changes to a public Git repository on a server. Then, the other developers can pull your new commits from the public repo into their own local repositories.




Giving each developer their own complete repository is the heart of distributed version control, and it opens up a wide array of potential workflows. You can read more about these workflows from our Git Workflows section.

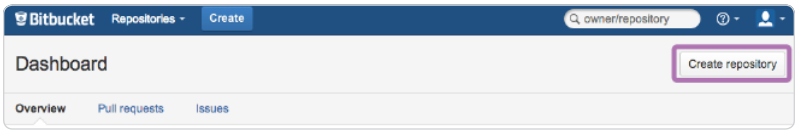
So far, you've only been working with a local Git repository. This page explains how to push this local repo to a public repository hosted on Bitbucket. Sharing the Git repository during the migration allows your team to experiment with Git commands without affecting their active SVN development. Until you're ready to make the switch, it's very important to **treat the shared Git repositories as read-only**. All development should continue to be committed to the original SVN repository.

## Create a Bitbucket account

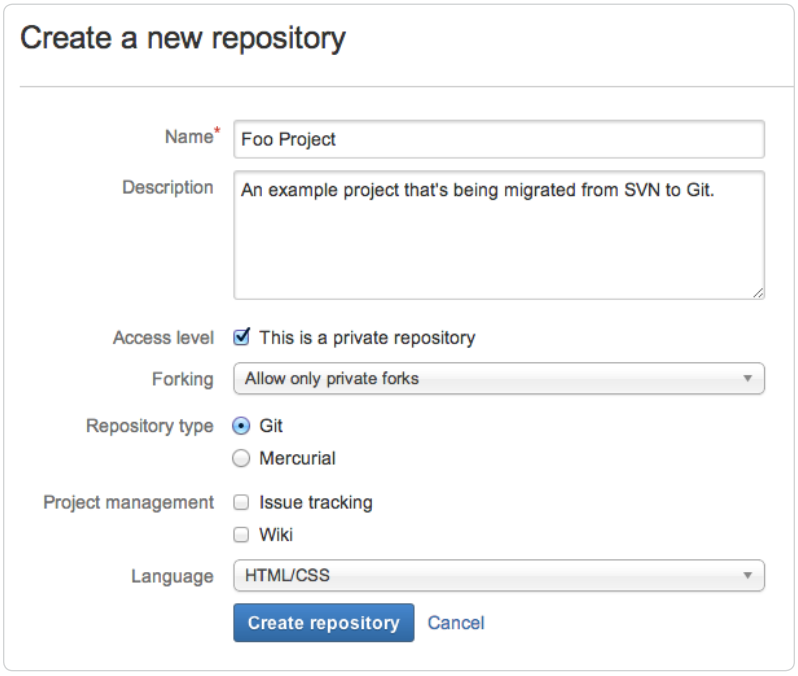
If you don't already have a Bitbucket account, you'll need to create one. Hosting is free for up to 5 users, so you can start experimenting with new Git workflows right away.

## Create a Bitbucket repository

Next, you'll need to create a Bitbucket repository. Bitbucket makes it very easy to administer your hosted repositories via a web interface. All you have to do is click the Create repository button after you've logged in.




In the resulting form, add a name and description for your repository. If your project is private, keep the *Access level* option checked so that only designated developers are allowed to clone it. For the *Forking* field, use *Allow only private forks*. Use *Git* for the *Repository type*, select any project management tools you want to use, and select the primary programming language of your project in the *Language* field.




To create the hosted repository, submit the form by clicking the *Create repository* button. After your repository is set up, you'll see a *Next steps* page that describes some useful commands for importing an existing project. The rest of this page will walk you through those instructions step-by-step.

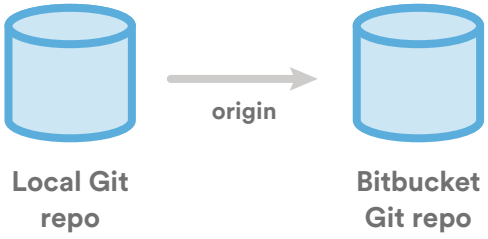
## Add an origin remote

To make it easier to push commits from your local Git repository to the Bitbucket repository you just created, you should record the Bitbucket repo's URL in a remote. A remote is just a convenient shortcut for a URL. Technically, you can use anything

you like for the shortcut, but if the remote repository serves as the official codebase for the project, it's conventionally referred to as `origin`. Run the following in your local Git repository to add your new Bitbucket repository as the `origin` remote.

```
git remote add origin https://<user>@bitbucket.org/<user>/<repo>.git
```

Be sure to change `<user>` to your Bitbucket username and `<repo>` to the name of the Bitbucket repository. You should also be able to copy and paste the complete URL from the Bitbucket web interface.




After running the above command, you can use `origin` in other Git commands to refer to your Bitbucket repository.

## Push the local repository to Bitbucket

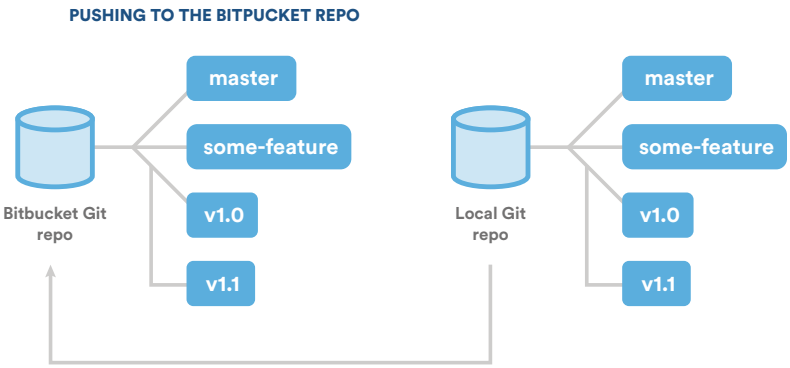
Next, you need to populate your Bitbucket repository with the contents of your local Git repository. This is called "pushing," and can be accomplished with the following command:

```
git push -u origin --all
```

The `-u` option tells Git to track the upstream branches. This enables Git to tell you if the remote repo's commit history is ahead or behind your local ones. The `--all` option pushes all of the local branches to the remote repository.

You also need to push your local tags to the Bitbucket repository with the `--tags` option:

```
git push --tags
```

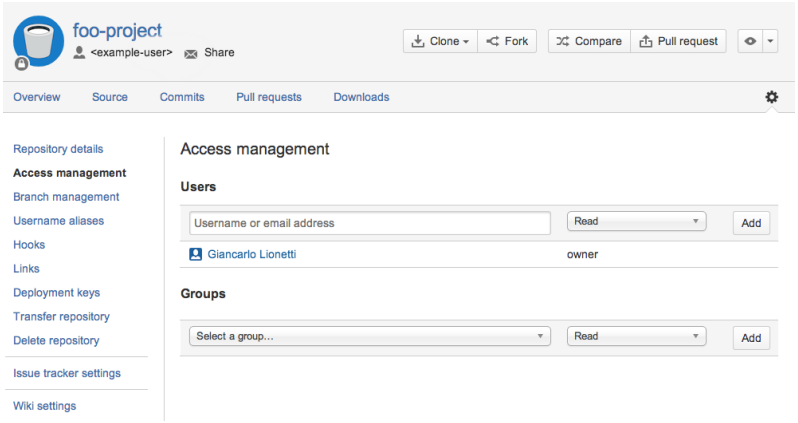



Your Bitbucket repository is now essentially a clone of your local repository. In the Bitbucket web interface, you should be able to explore the entire commit history of all of your branches.

## Share the repository with your team

All you have to do now is share the URL of your Bitbucket repository with any other developers that need access to the repository. The URL for any Git repository can be copy-and-pasted from the repository home page on Bitbucket:


If your repository is private, you'll also need to grant access to your team members in the *Administration* tab of the Bitbucket web interface. Users and groups can be managed by clicking the *Access management* link the left sidebar.



As an alternative, you can use Bitbucket's built-in invitation feature to invite other developers to fork the repository. The invited users will automatically be given access to the repository, so you don't need to worry about granting permissions.

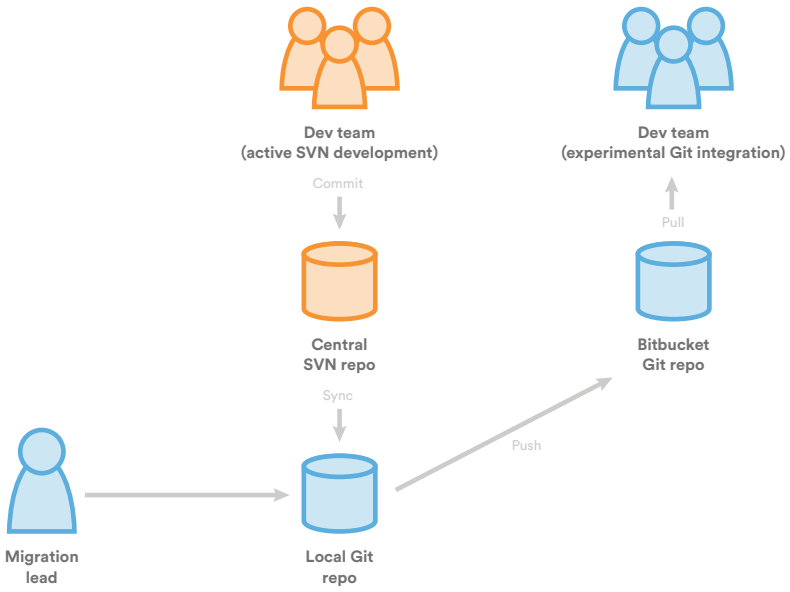
Once they have the URL of your repository, another developer can copy the repository to their local machine with `git clone` and begin working with the project. For example, after running the following command on their local machine, another developer would find a new Git repository containing the project in the `<destination>` directory.

```
git clone https://<user>@bitbucket.org/<user>/<project>.git <destination>
```

## Continue committing with SVN, not Git

You should now be able to push your local project to a remote repository, and your team should be able to use that remote repository to clone the project onto their local machines. These are all the tools you need to start collaborating with Git. However, you and your team should continue to commit changes using SVN until everybody is ready to make the switch.

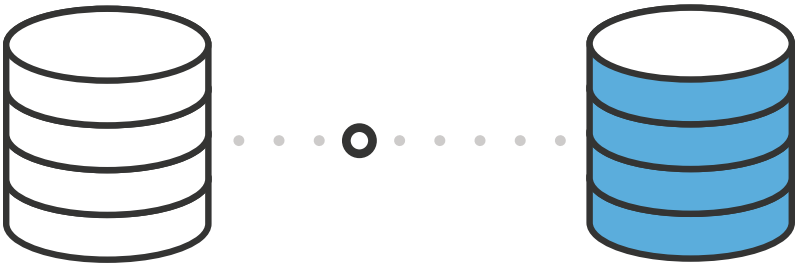
The only changes to the Git repository should come from the original SVN repository using the synchronization process discussed on the previous page. For all intents and purposes, this means that all of your Git repositories (both local and remote) are read-only. Your developers can experiment with them, and you can begin to integrate them into your build process, but you should avoid committing any permanent changes using Git.



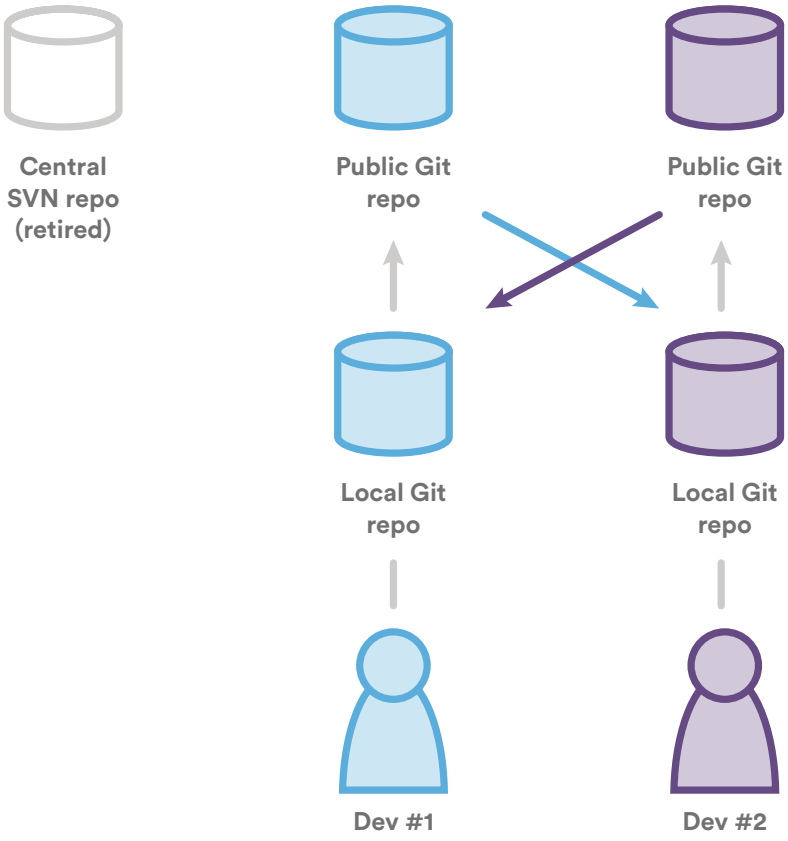

## Summary

In this step, you set up a Bitbucket repository to share your converted Git repository with other developers. You should now have all the tools you need to implement any of the git workflows described in Git Workflows. You can continue synchronizing with the SVN repository and sharing the resulting Git commits via Bitbucket for as long as it takes to get your development team comfortable with Git. Then, you can complete the migration process by retiring your SVN repository.

## Migrate



This migration guide advocates a one-way synchronization from SVN to Git during the transition period. This means that while your team is getting comfortable with Git, they should still only be committing to the original SVN repository. When you're ready to make the switch, the SVN repository should freeze at whatever state it's in. Then, developers should begin committing to their local Git repositories and sharing them via Bitbucket.



The discrete switch from SVN to Git makes for a very intuitive migration. All of your developers should already understand the new Git workflows that they'll be using, and they should have had plenty of time to practice using Git commands on the local repositories they cloned from Bitbucket.

This page guides you through the final step of the migration.

## Synchronize the Git repository

Before finalizing your migration to Git, you should make sure that your Git repository contains any new changes that have been committed to your SVN repository. You can do this with the same process described in the Synchronize phase.

```
git svn fetch
java -Dfile.encoding=utf-8 -jar ~/svn-migration-scripts.jar sync-rebase
java -Dfile.encoding=utf-8 -jar ~/svn-migration-scripts.jar clean-git --force
```

## Back up the SVN repository

While you can still see your pre-Git project history in the migrated repository, it's a good idea to backup the SVN repository just in case you ever need to explore the raw SVN data. An easy way to backup an SVN repo is to run the following on the machine that hosts the central SVN repository. If your SVN repo is hosted on a Linux machine, you can use the following:

```
svnadmin dump <svn-repo> | gzip -9 > <backup-file>
```

Replace `<svn-repo>` with the file path of the SVN repository that you're backing up, and replace `<backup-file>` with the file path of the compressed file containing the backup.

## Make the SVN repository read-only

All of your developers should now be committing with Git. To enforce this convention, you can make your SVN repository read-only. This process can vary depending on your server setup, but if you're using the `svnserve` daemon, you can accomplish this by editing your SVN repo's `conf/svnserve.conf` file. It's `[general]` section should contain the following lines:

```
anon-access = read
auth-access = read
```

This tells svnserve that both anonymous and authenticated users only have read permissions.

## Summary

And that's all there is to migrating a project to Git. Your team should now be developing with a pure Git workflow and enjoying all of the benefits of distributed development. Good job!

# Bitbucket Data Center

# Git that grows with you

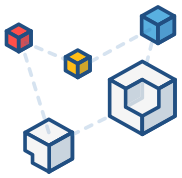
The Data Center deployment option is designed for high availability and performance at scale when hosting our applications in your own data center.

### Next generation Git

Bitbucket Data Center offers true active-active clustering for high availability, and Git mirroring for performance across distributed teams. Plus, strong security with LDAP support, permissions at the branch level, and workflow control.

### Code collaboration at scale

With innovative features like pull requests, smart commits, code search, Git mirroring, and Git LFS (Large File Storage), teams can stretch Git to support a growing business at the speed of LAN.



### Integrations that matter

Teams can customize their development platform using integrations and add-ons. Integrations with other Atlassian tools come out-of-the-box while the Atlassian Marketplace offers hundreds of useful add-ons.

### Disaster recovery

Deploy an offsite disaster recovery system in the event of a complete system outage with index replication and database synchronization.

### Performance at scale

Intelligently share load between nodes to bolster concurrent user and build capacity and avoid performance degradation.

### Instant scalability

Add nodes to your cluster without downtime or additional license cost to maximize resilience and uptime as you grow.

### Code smarter, faster

Use inline comments to collaborate on code right where it lives. Control code reviews and pull requests with ease.

### Continuous integration

Monitor your builds and detect issues before they become a problem by linking to CI and issue tracking tools without impacting performance.

### Enterprise support and services

Premier Support for 24/7 help and Technical Account Management for proactive analysis and advice.

> **With Git and Bitbucket, we're able to complete 3x as many code reviews, which ultimately results in fewer bugs, fewer support tickets, and better software.**
>
> —Kurt Chase, Director of Release Engineering, Splunk

Learn more