4
Everything is Code

Everything is code, and you need somewhere to store it. The organization's source code management system is that place.

Developers and operations personnel share the same central storage for their different types of code.

There are many ways of hosting the central code repository:

- You can use a software as a service solution, such as GitHub, Bitbucket, or GitLab. This can be cost-effective and provide good availability.
- You can use a cloud provider, such as AWS or Rackspace, to host your repositories.

Some types of organization simply can't let their code leave the building. For them, a private in-house setup is the best choice.

In this chapter, we will explore different options, such as Git, and web-based frontends to Git, such as Gerrit and GitLab.

This exploration will serve to help you find a Git hosting solution that covers your organization's needs.

In this chapter, we will start to experience one of the challenges in the field of DevOps—there are so many solutions to choose from and explore! This is particularly true for the world of source code management, which is central to the DevOps world.

Therefore, we will also introduce the software virtualization tool Docker from a user's perspective so that we can use the tool in our exploration.

The need for source code control

Terence McKenna, an American author, once said that everything is code.

While one might not agree with McKenna about whether everything in the universe can be represented as code, for DevOps purposes, indeed nearly everything can be expressed in codified form, including the following:

- The applications that we build
- The infrastructure that hosts our applications
- The documentation that documents our products

Even the hardware that runs our applications can be expressed in the form of software.

Given the importance of code, it is only natural that the location that we place code, the source code repository, is central to our organization. Nearly everything we produce travels through the code repository at some point in its life cycle.

The history of source code management

In order to understand the central need for source code control, it can be illuminating to have a brief look at the development history of source code management. This gives us an insight into the features that we ourselves might need. Some examples are as follows:

- Storing historical versions of source in separate archives.

  This is the simplest form, and it still lives on to some degree, with many free software projects offering tar archives of older releases to download.

- Centralized source code management with check in and check out.

  In some systems, a developer can lock files for exclusive use. Every file is managed separately. Tools like this include RCS and SCCS.

  [Normally, you don't encounter this class of tool anymore, except the occasional file header indicating that a file was once managed by RCS.]

- A centralized store where you merge before you commit. Examples include **Concurrent Versions System** (**CVS**) and **Subversion**.

Subversion in particular is still in heavy use. Many organizations have centralized workflows, and Subversion implements such workflows well enough for them.

- A decentralized store.

  On each step of the evolutionary ladder, we are offered more flexibility and concurrency as well as faster and more efficient workflows. We are also offered more advanced and powerful guns to shoot ourselves in the foot with, which we need to keep in mind!

Currently, Git is the most popular tool in this class, but there are many other similar tools in use, such as Bazaar and Mercurial.

Time will tell whether Git and its underlying data model will fend off the contenders to the throne of source code management, who will undoubtedly manifest themselves in the coming years.

Roles and code

From a DevOps point of view, it is important to leverage the natural meeting point that a source code management tool is. Many different roles have a use for source code management in its wider meaning. It is easier to do so for technically-minded roles but harder for other roles, such as project management.

Developers live and breathe source code management. It's their bread and butter.

Operations personnel also favor managing the descriptions of infrastructure in the form of code, scripts, and other artifacts, as we will see in the coming chapters. Such infrastructural descriptors include network topology, versions of software that should be installed on particular servers, and so on.

Quality assurance personnel can store their automated tests in codified form in the source code repository. This is true for software testing frameworks such as Selenium and Junit, among others.

There is a problem with the documentation of the manual steps needed to perform various tasks, though. This is more of a psychological or cultural problem than a technical one.

While many organizations employ a wiki solution such as the wiki engine powering Wikipedia, there is still a lot of documentation floating around in the Word format on shared drives and in e-mails.

This makes documentation hard to find and use for some roles and easy for others. From a DevOps viewpoint, this is regrettable, and an effort should be made so that all roles can have good and useful access to the documentation in the organization.

It is possible to store all documentation in the wiki format in the central source code repository, depending on the wiki engine used.

Which source code management system?

There are many source code management (SCM) systems out there, and since SCM is such an important part of development, the development of these systems will continue to happen.

Currently, there is a dominant system, however, and that system is Git.

Git has an interesting story: it was initiated by Linus Torvalds to move Linux kernel development from BitKeeper, which was a proprietary system used at the time. The license of BitKeeper changed, so it wasn't practical to use it for the kernel anymore.

Git therefore supports the fairly complicated workflow of Linux kernel development and is, at the base technical level, good enough for most organizations.

The primary benefit of Git versus older systems is that it is a distributed version control system (DVCS). There are many other distributed version control systems, but Git is the most pervasive one.

Distributed version control systems have several advantages, including, but not limited to, the following:

- It is possible to use a DVCS efficiently even when not connected to a network. You can take your work with you when traveling on a train or an intercontinental flight.
- Since you don't need to connect to a server for every operation, a DVCS can be faster than the alternatives in most scenarios.
- You can work privately until you feel your work is ready enough to be shared.
- It is possible to work with several remote logins simultaneously, which avoids a single point of failure.

Other distributed version control systems apart from Git include the following:

- **Bazaar**: This is abbreviated as bzr. Bazaar is endorsed and supported by Canonical, which is the company behind Ubuntu. Launchpad, which is Canonical's code hosting service, supports Bazaar.
- **Mercurial**: Notable open source projects such as Firefox and OpenJDK use Mercurial. It originated around the same time as Git.

Git can be complex, but it makes up for this by being fast and efficient. It can be hard to understand, but that can be made easier by using frontends that support different tasks.

A word about source code management system migrations

I have worked with many source code management systems and experienced many transitions from one type of system to another.

Sometimes, much time is spent on keeping all the history intact while performing a migration. For some systems, this effort is well spent, such as for venerable free or open source projects.

For many organizations, keeping the history is not worth the significant expenditure in time and effort. If an older version is needed at some point, the old source code management system can be kept online and referenced. This includes migrations from Visual SourceSafe and ClearCase.

Some migrations are trivial though, such as moving from Subversion to Git. In these cases, historic accuracy need not be sacrificed.

Choosing a branching strategy

When working with code that deploys to servers, it is important to agree on a branching strategy across the organization.

A branching strategy is a convention, or a set of rules, that describes when branches are created, how they are to be named, what use branches should have, and so on.

Branching strategies are important when working together with other people and are, to a degree, less important when you are working on your own, but they still have a purpose.

Shared authentication

In most organizations, there is some form of a central server for handling authentication. An LDAP server is a fairly common choice. While it is assumed that your organization has already dealt with this core issue one way or the other and is already running an LDAP server of some sort, it is comparatively easy to set up an LDAP server for testing purposes.

One possibility is using 389 Server, named after the port commonly used for the LDAP server, together with the phpLDAPadmin web application for administration of the LDAP server.

Having this test LDAP server setup is useful for the purposes of this book, since we can then use the same LDAP server for all the different servers we are going to investigate.

Hosted Git servers

Many organizations can't use services hosted within another organization's walls at all.

These might be government organizations or organizations dealing with money, such as bank, insurance, and gaming organizations.

The causes might be legal or, simply nervousness about letting critical code leave the organization's doors, so to speak.

If you have no such qualms, it is quite reasonable to use a hosted service, such as GitHub or GitLab, that offers private accounts.

Using GitHub or GitLab is, at any rate, a convenient way to get to learn to use Git and explore its possibilities.

Both vendors are easy to evaluate, given that they offer free accounts where you can get to know the services and what they offer. See if you really need all the services or if you can make do with something simpler.

Some of the features offered by both GitLab and GitHub over plain Git are as follows:

- Web interfaces
- A documentation facility with an inbuilt wiki
- Issue trackers
- Commit visualization

- Branch visualization
- The pull request workflow

While these are all useful features, it's not always the case that you can use the facilities provided. For instance, you might already have a wiki, a documentation system, an issue tracker, and so on that you need to integrate with.

The most important features we are looking for, then, are those most closely related to managing and visualizing code.

Large binary files

GitHub and GitLab are pretty similar but do have some differences. One of them springs from the fact that source code systems like Git traditionally didn't cater much for the storage of large binary files. There have always been other ways, such as storing file paths to a file server in plain text files.

But what if you actually have files that are, in a sense, equivalent to source files except that they are binary, and you still want to version them? Such file types might include image files, video files, audio files, and so on. Modern websites make increasing use of media files, and this area has typically been the domain of content management systems (CMSes). CMSes, however nice they might be, have disadvantages compared to DevOps flows, so the allure of storing media files in your ordinary source handling system is strong. Disadvantages of CMSes include the fact that they often have quirky or nonexistent scripting capabilities. Automation, another word that should have really been included in the "DevOps" portmanteau, is therefore often difficult with a CMS.

You can, of course, just check in your binary to Git, and it will be handled like any other file. What happens then is that Git operations involving server operations suddenly become sluggish. And then, some of the main advantages of Git—efficiency and speed—vanish out the window.

Solutions to this problem have evolved over time, but there are no clear winners yet. The contenders are as follows:

- **Git LFS**, supported by GitHub
- **Git Annex**, supported by GitLab but only in the enterprise edition

Git Annex is the more mature of these. Both solutions are open source and are implemented as extensions to Git via its plugin mechanism.

There are several other systems, which indicates that this is an unresolved pain point in the current state of Git. The Git Annex has a comparison between the different breeds at `http://git-annex.branchable.com/not/`.

If you need to perform version control of your media files, you should start by exploring Git Annex. It is written in Haskell and is available through the package system of many distributions.

It should also be noted that the primary benefit of this type of solution is the ability to version media files together with the corresponding code. When working with code, you can examine the differences between versions of code conveniently. Examining differences between media files is harder and less useful.

In a nutshell, Git Annex uses a tried and tested solution to data logical problems: adding a layer of indirection. It does this by storing symlinks to files in the repository. The binary files are then stored in the filesystem and fetched by local workspaces using other means, such as rsync. This involves more work to set up the solution, of course.

Trying out different Git server implementations

The distributed nature of Git makes it possible to try out different Git implementations for various purposes. The client-side setup will be similar regardless of how the server is set up.

You can also have several solutions running in parallel. The client side is not unduly complicated by this, since Git is designed to handle several remotes if need be.

Docker intermission

In *Chapter 7*, *Deploying the Code*, we will have a look at a new and exciting way to package our applications, called **Docker**.

In this chapter, we have a similar challenge to solve. We need to be able to try out a couple of different Git server implementations to see which one suits our organization best.

We can use Docker for this, so we will take this opportunity to peek at the possibilities of simplified deployments that Docker offers us.

Since we are going to delve deeper into Docker further along, we will cheat a bit in this chapter and claim that Docker is used to download and run software. While this description isn't entirely untrue, Docker is much more than that. To get started with Docker, follow these steps:

1. To begin with, install Docker according to the particular instructions for your operating system. For Red Hat derivatives, it's a simple `dnf install docker-io` command.

   > The `io` suffix might seem a bit mysterious, but there was already a Docker package that implemented desktop functionality, so `docker-io` was chosen instead.

2. Then, the `docker` service needs to be running:

   ```
   systemctl enable docker
   systemctl start docker
   ```

3. We need another tool, Docker Compose, which, at the time of writing, isn't packaged for Fedora. If you don't have it available in your package repositories, follow the instructions on this page `https://docs.docker.com/compose/install/`

> Docker Compose is used to automatically start several Docker-packaged applications, such as a database server and a web server, together, which we need for the GitLab example.

Gerrit

A basic Git server is good enough for many purposes.

Sometimes, you need precise control over the workflow, though.

One concrete example is merging changes into configuration code for critical parts of the infrastructure. While my opinion is that it's core to DevOps to not place unnecessary red tape around infrastructure code, there is no denying that it's sometimes necessary. If nothing else, developers might feel nervous committing changes to the infrastructure and would like for someone more experienced to review the code changes.

Gerrit is a Git-based code review tool that can offer a solution in these situations. In brief, Gerrit allows you to set up rules to allow developers to review and approve changes made to a codebase by other developers. These might be senior developers reviewing changes made by inexperienced developers or the more common case, which is simply that more eyeballs on the code is good for quality in general.

Gerrit is Java-based and uses a Java-based Git implementation under the hood.

Gerrit can be downloaded as a Java WAR file and has an integrated setup method. It needs a relational database as a dependency, but you can opt to use an integrated Java-based H2 database that is good enough for evaluating Gerrit.

An even simpler method is using Docker to try out Gerrit. There are several Gerrit images on the Docker registry hub to choose from. The following one was selected for this evaluation exercise: `https://hub.docker.com/r/openfrontier/gerrit/`

To run a Gerrit instance with Docker, follow these steps:

1. Initialize and start Gerrit:

   ```
   docker run -d -p 8080:8080 -p 29418:29418 openfrontier/gerrit
   ```

2. Open your browser to `http://<docker host url>:8080`

   Now, we can try out the code review feature we would like to have.

Installing the git-review package

Install `git-review` on your local installation:

```
sudo dnf install git-review
```

This will install a helper application for Git to communicate with Gerrit. It adds a new command, `git-review`, that is used instead of `git push` to push changes to the Gerrit Git server.

The value of history revisionism

When we work with code together with other people in a team, the code's history becomes more important than when we work on our own. The history of changes to files becomes a way to communicate. This is especially important when working with code review and code review tools such as Gerrit .

The code changes also need to be easy to understand. Therefore, it is useful, although perhaps counterintuitive, to edit the history of the changes in order to make the resulting history clearer.

As an example, consider a case where you made a number of changes and later changed your mind and removed them. It is not useful information for someone else that you made a set of edits and then removed them. Another case is when you have a set of commits that are easier to understand if they are a single commit. Adding commits together in this way is called **squashing** in the Git documentation.

Another case that complicates history is when you merge from the upstream central repository several times, and merge commits are added to the history. In this case, we want to simplify the changes by first removing our local changes, then fetching and applying changes from the upstream repository, and then finally reapplying our local changes. This process is called **rebasing**.

Both squashing and rebasing apply to Gerrit.

The changes should be clean, preferably one commit. This isn't something that is particular to Gerrit; it's easier for a reviewer to understand your changes if they are packaged nicely. The review will be based on this commit.

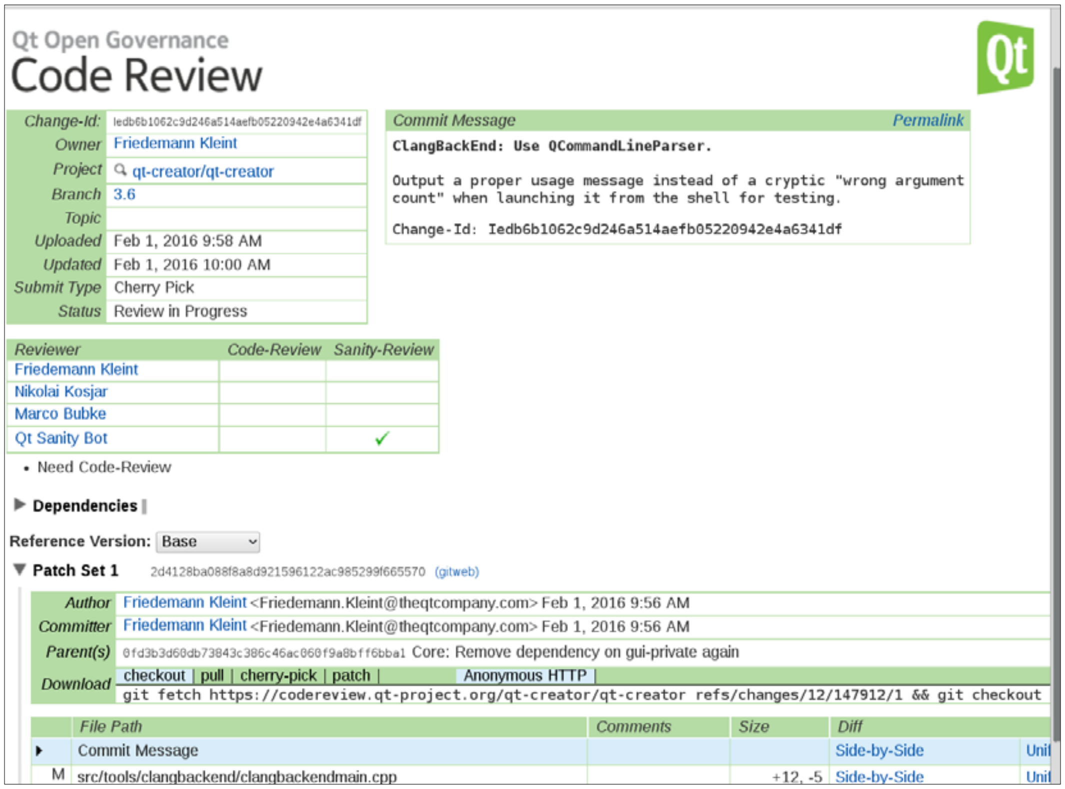
1. To begin with, we need to have the latest changes from the Git/Gerrit server side. We rebase our changes on top of the server-side changes:

   ```
   git pull --rebase origin master
   ```

2. Then, we polish our local commits by squashing them:

   ```
   git rebase -i origin/master
   ```

Now, let's have a look at the Gerrit web interface:



Qt Open Governance
Code Review

Change-Id:	Iedb6b1062c9d246a514aefb05220942e4a6341df
Owner	Friedemann Kleint
Project	qt-creator/qt-creator
Branch	3.6
Topic	
Uploaded	Feb 1, 2016 9:58 AM
Updated	Feb 1, 2016 10:00 AM
Submit Type	Cherry Pick
Status	Review in Progress

Commit Message — Permalink

```
ClangBackEnd: Use QCommandLineParser.

Output a proper usage message instead of a cryptic "wrong argument
count" when launching it from the shell for testing.

Change-Id: Iedb6b1062c9d246a514aefb05220942e4a6341df
```

Reviewer	Code-Review	Sanity-Review
Friedemann Kleint		
Nikolai Kosjar		
Marco Bubke		
Qt Sanity Bot		✓

- Need Code-Review

Dependencies

Reference Version: Base

Patch Set 1 2d4128ba088f8a8d921596122ac985299f665570 (gitweb)

Author	Friedemann Kleint <Friedemann.Kleint@theqtcompany.com> Feb 1, 2016 9:56 AM
Committer	Friedemann Kleint <Friedemann.Kleint@theqtcompany.com> Feb 1, 2016 9:56 AM
Parent(s)	0fd3b3d60db73843c386c46ac060f9a8bff6bba1 Core: Remove dependency on gui-private again
Download	checkout | pull | cherry-pick | patch | Anonymous HTTP |

```
git fetch https://codereview.qt-project.org/qt-creator/qt-creator refs/changes/12/147912/1 && git checkout
```

	File Path	Comments	Size	Diff	
▸	Commit Message			Side-by-Side	Unif
M	src/tools/clangbackend/clangbackendmain.cpp		+12, -5	Side-by-Side	Unif

We can now approve the change, and it is merged to the master branch.

There is much to explore in Gerrit, but these are the basic principles and should be enough to base an evaluation on.

Are the results worth the hassle, though? These are my observations:

- Gerrit allows fine-grained access to sensitive codebases. Changes can go in after being reviewed by authorized personnel.

  This is the primary benefit of Gerrit. If you just want to have mandatory code reviews for unclear reasons, don't. The benefit has to be clear for everyone involved. It's better to agree on other more informal methods of code review than an authoritative system.

- If the alternative to Gerrit is to not allow access to code bases at all, even read-only access, then implement Gerrit.

Some parts of an organization might be too nervous to allow access to things such as infrastructure configuration. This is usually for the wrong reasons. The problem you usually face isn't people taking an interest in your code; it's the opposite.

Sometimes, sensitive passwords are checked in to code, and this is taken as a reason to disallow access to the source. Well, if it hurts, don't do it. Solve the problem that leads to there being passwords in the repositories instead.

The pull request model

There is another solution to the problem of creating workflows around code reviews: the pull request model, which has been made popular by GitHub.

In this model, pushing to repositories can be disallowed except for the repository owners. Other developers are allowed to fork the repository, though, and make changes in their fork. When they are done making changes, they can submit a pull request. The repository owners can then review the request and opt to pull the changes into the master repository.

This model has the advantage of being easy to understand, and many developers have experience in it from the many open source projects on GitHub.

Setting up a system capable of handling a pull request model locally will require something like GitHub or GitLab, which we will look at next.

GitLab

GitLab supports many convenient features on top of Git. It's a large and complex software system based on Ruby. As such, it can be difficult to install, what with getting all the dependencies right and so on.

There is a nice Docker Compose file for GitLab available at `https://registry.hub.docker.com/u/sameersbn/gitlab/`. If you followed the instructions for Docker shown previously, including the installation of `docker-compose`, it's now pretty simple to start a local GitLab instance:

```
mkdir gitlab
cd gitlab
wget https://raw.githubusercontent.com/sameersbn/docker-gitlab/master/
docker-compose.yml
docker-compose up
```

The `docker-compose` command will read the `.yml` file and start all the required services in a default demonstration configuration.

If you read the startup log in the console window, you will notice that three separate application containers have been started: `gitlab postgresql1`, `gitlab redis1`, and `gitlab gitlab1`.

The GitLab container includes the Ruby base web application and Git backend functionality. Redis is distributed key-value store, and PostgreSQL is a relational database.

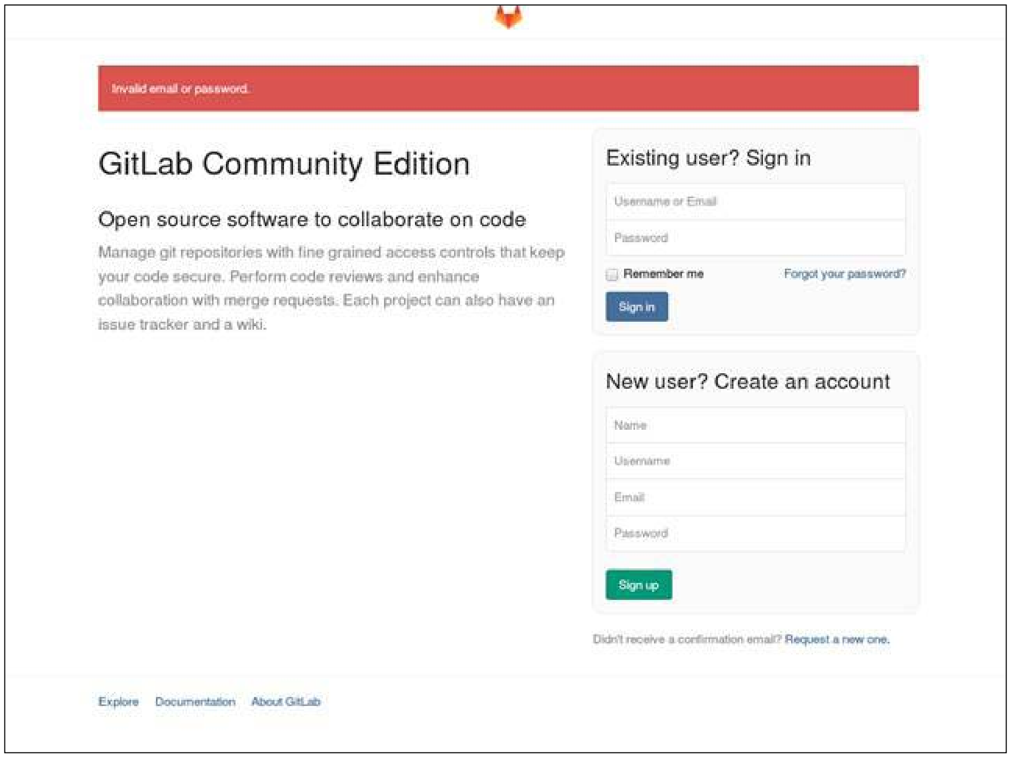
If you are used to setting up complicated server functionality, you will appreciate that we have saved a great deal of time with `docker-compose`.

The `docker-compose.yml` file sets up data volumes at `/srv/docker/gitlab`.

To log in to the web user interface, use the administrator password given with the installation instructions for the GitLab Docker image. They have been replicated here, but beware that they might change as the Docker image author sees fit:

- Username: root
- Password: 5iveL!fe

Here is a screenshot of the GitLab web user interface login screen:



Invalid email or password.

GitLab Community Edition

Open source software to collaborate on code

Manage git repositories with fine grained access controls that keep your code secure. Perform code reviews and enhance collaboration with merge requests. Each project can also have an issue tracker and a wiki.

Existing user? Sign in

Username or Email

Password

Remember me

Forgot your password?

Sign in

New user? Create an account

Name

Username

Email

Password

Sign up

Didn't receive a confirmation email? Request a new one.

Explore Documentation About GitLab

Try importing a project to your GitLab server from, for instance, GitHub or a local private project.

Have a look at how GitLab visualizes the commit history and branches.

While investigating GitLab, you will perhaps come to agree that it offers a great deal of interesting functionality.

When evaluating features, it's important to keep in mind whether it's likely that they will be used after all. What core problem would GitLab, or similar software, solve for you?

It turns out that the primary value added by GitLab, as exemplified by the following two features, is the elimination of bottlenecks in DevOps workflows:

- The management of user ssh keys
- The creation of new repositories

These features are usually deemed to be the most useful.

Visualization features are also useful, but the client-side visualization available with Git clients is more useful to developers.

Summary

In this chapter, we explored some of the options available to us for managing our organization's source code. We also investigated areas where decisions need to be made within DevOps, version numbering, and branching strategies.

After having dealt with source code in its pure form, we will next work on building the code into useful binary artifacts.