Deploying GitKraken with GitHub at Scale

Adopting a graphical user interface to overcome the challenges of implementing Git

Table of Contents

Introduction

"Git is the dominant choice for version control for developers today, with almost 90% of developers checking in their code via Git."
-Stack Overflow, **Developer Survey 2018**

The Problem

Without-a-doubt Git is the industry standard for version control today. Unfortunately, adopting Git often comes with a lot of additional overhead. In time, Git boosts an organization's overall efficiency; however, developers initially face a steep learning curve. Failure to address this problem increases the time it takes to ship code and decreases code stability.

What to Expect

This white paper will walk you through the benefits of adopting Git as your version control system (VCS), and how to tackle the accompanying challenges head on. We will cover common implementation practices associated with the popular Git hosting services GitHub.com and GitHub Enterprise, and how to enhance their functionality with GitKraken. You'll find out how to deploy Git, GitHub and GitKraken at scale, and how this process will save you time with a graphical user interface (GUI) approach.

After exploring solutions to this problem, we will address how to maintain security standards that are common to industries such as defense, medicine, finance, government, and others with offline/firewalled environments.

After reading this whitepaper you will be equipped with the knowledge needed to: easily transition your organization to Git, roll out a unified branching strategy, and improve your users' understanding of actions between their local and remote repositories.

Why Git?

One look at Google Trends will show Git has without-a-doubt become the dominant version control method when compared to competing systems used in the past:
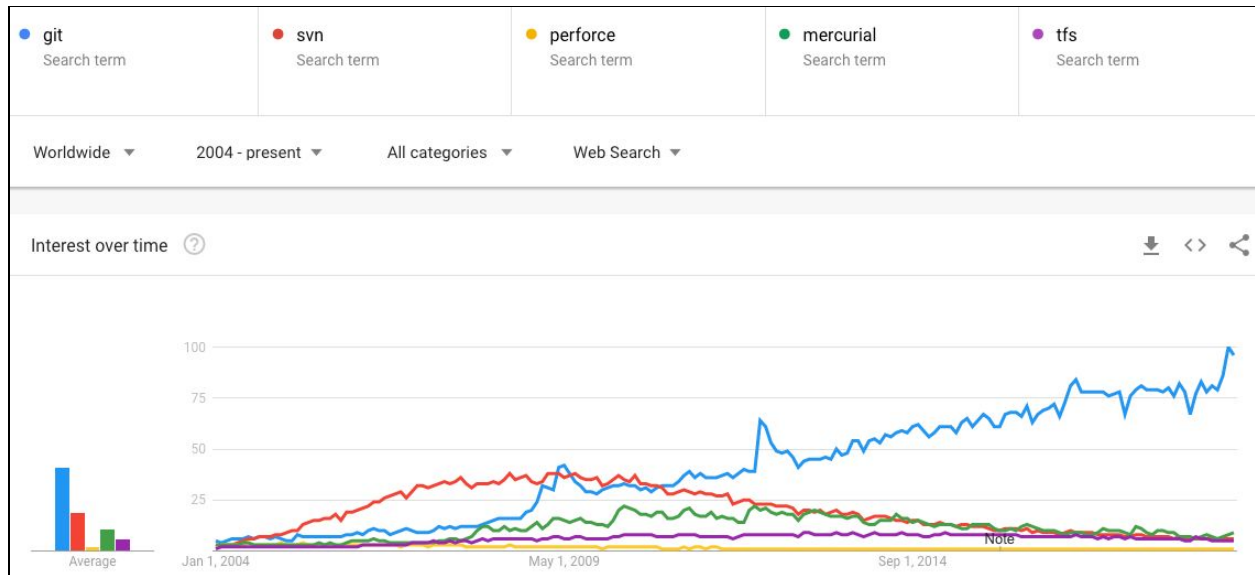
Git empowers teams to save time through its history, collaboration, and branching feature set. We will explore these key advantages in the following sections.

History

History is critical for maintaining source code because it allows developers to locate the exact commit with a given changeset. Once a commit is identified, it is then possible to identify the commit author through 'file blame' or 'file history' and request fixes or additional changes.

Distributed Version Control

Git's distributed nature provides users with better collaboration capabilities when compared to older version control systems. Any initialized repository can be cloned using the `git clone` command. This affords collaborators the ability to copy the entire project and work in parallel with their team members. No need to wait your turn to check out files or push changes.

Because any clone of a project can be arbitrarily designated as the main project, it is a common practice to designate the main project on a Git hosting service such as GitHub.

Collaborators may then 'fork' the repository, which is the `git clone` operation but performed by the Git hosting service on the collaborator's behalf.

Hosting repositories on a Git hosting service such as GitHub makes the project more accessible to any collaborator with the appropriate permissions. If the project is open source, then it is accessible to the public. A remote repository, often called a remote, is a Git repository hosted online—such as on a Git hosting service—or some other network.

Cloning a remote or a fork creates a local version of that entire repository on a user's machine, giving the developer a sandbox to experiment without affecting the original codebase. Cloning also establishes a connection between the local repository on the developer's machine and the remote repository, allowing 'push' and 'pull' actions with the remote project. However it is not necessary to stay connected to a remote repository until a user wishes to push up their changes or pull in changes from their team members.

Collaboration is mainly possible because of Git's branching feature. Creating a branch using the `git branch` command allows users to create a sandbox for features and fixes. Each developer is able to not only work in parallel with other users, but they can also create any number of branches to work in parallel with themselves.

Additionally, switching branches is an inexpensive operation in Git, therefore any user can checkout a branch and immediately get to work.

With this boost in efficiency, it is clear to see why software development teams have gravitated towards Git for source control.

Everyday challenges of using Git

While Git helps increase an organization's efficiency, there is a steep learning curve to master Git.

Git commands are primarily typed into the command line interface (or CLI for short). This minimalist interface demands the user understand the meaning behind their commands because there is little-to-no immediate visual feedback affirming an action has been performed as expected.

```
Last login: Mon Aug 12 08:56:47 on ttys001
  jonathans@jonathans-mbp   ~/Sites/support.gitkraken.com   ⑂ dev   git branch fe
ature2
  jonathans@jonathans-mbp   ~/Sites/support.gitkraken.com   ⑂ dev
```

To provide context, the **Git-scm reference** page has links to over 50 commands. Each page then has additional flags and rules to consider, which makes it difficult to remember all the possible combinations and proper syntaxes.

For example, say a user wishes to checkout a different branch using the command line. First they should confirm what branches are available for checkout by entering the command: `git checkout`

Then, the user will need to identify the target branch and enter the following command to perform the checkout:

`git checkout -b <new-branch> <existing-branch>`

While this may seem elementary to a seasoned user, they are still required to remember or reference the command, flags, and syntax for the operation. If the user is unable to recall the exact format, they must then either consult an online resource or ask a fellow teammate.

```
[mannyg-macos:~ mannyg$ git checkout -h
usage: git checkout [<options>] <branch>
   or: git checkout [<options>] [<branch>] -- <file>...

    -q, --quiet              suppress progress reporting
    -b <branch>              create and checkout a new branch
    -B <branch>              create/reset and checkout a branch
    -l                       create reflog for new branch
    --detach                 detach HEAD at named commit
    -t, --track              set upstream info for new branch
    --orphan <new-branch>
                             new unparented branch
    -2, --ours               checkout our version for unmerged files
    -3, --theirs             checkout their version for unmerged files
    -f, --force              force checkout (throw away local modifications)
    -m, --merge              perform a 3-way merge with the new branch
    --overwrite-ignore       update ignored files (default)
    --conflict <style>       conflict style (merge or diff3)
    -p, --patch              select hunks interactively
    --ignore-skip-worktree-bits
                             do not limit pathspecs to sparse entries only
    --ignore-other-worktrees
                             do not check if another worktree is holding the given ref
    --recurse-submodules[=<checkout>]
                             control recursive updating of submodules
    --progress               force progress reporting

mannyg-macos:~ mannyg$ ▊
```

This paper does not doubt or question a developer's ability to learn the necessary commands to properly interact with their repositories. However, *a developer's time is better spent programming than learning to manage the overhead associated with Git commands.*

Additionally, migrating to a Git hosting service such as GitHub does not completely address this learning curve. While it is possible to review the files directly on a GitHub repo, it is not

designed to detect changes made locally to your repository. Any changes made directly on the GitHub repo will be made immediately available to any other collaborators with access to the remote repository.

Teams commonly implement a solution like GitHub or GitHub Enterprise, but then expect users to depend on the command line interface (CLI) for their daily Git operations. However, if teams are able to reduce the time it takes their users to perform Git commands, these teams will be rewarded with faster release cycles, more stable code, and more confident developers. This is where a GUI approach comes into play.

A GUI Approach

Graphical user interfaces help users visualize information and actions more clearly. In much the same way that operating systems have become easier to use by offering a graphical user interface for managing applications, dragging-and-dropping files, and clicking on icons to perform actions, developers can take advantage of GitKraken to dramatically improve their productivity, reduce error rates, and learn Git in a more intuitive way. In the following sections, we'll discuss how GitKraken's intuitive visual interface makes developers more productive.

Visualizing your repository's history

Instead of manually verifying whether Git commands performed the intended action in the CLI, users need only open their repository in GitKraken to view their commit history and more.
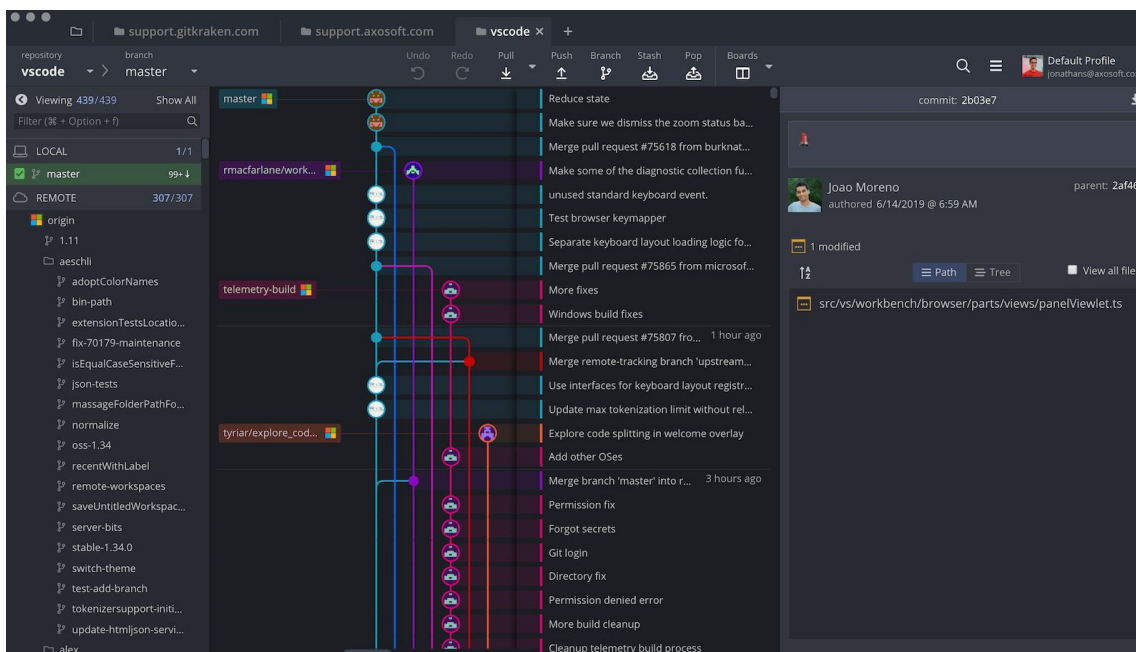
In the CLI, a user is required to enter a command to either checkout a new branch or view the contents of a commit. Prior to that, the user needs to confirm the branch name or have the commit SHA on hand.

Contrastingly, in the GitKraken GUI, the commits are drawn from one child commit to its parent commit(s). Users need only click on a given commit to review the list of files that were modified, renamed, added or deleted. Clicking on the file name then leads to the file diff, which tells users which lines and sections of code were modified.
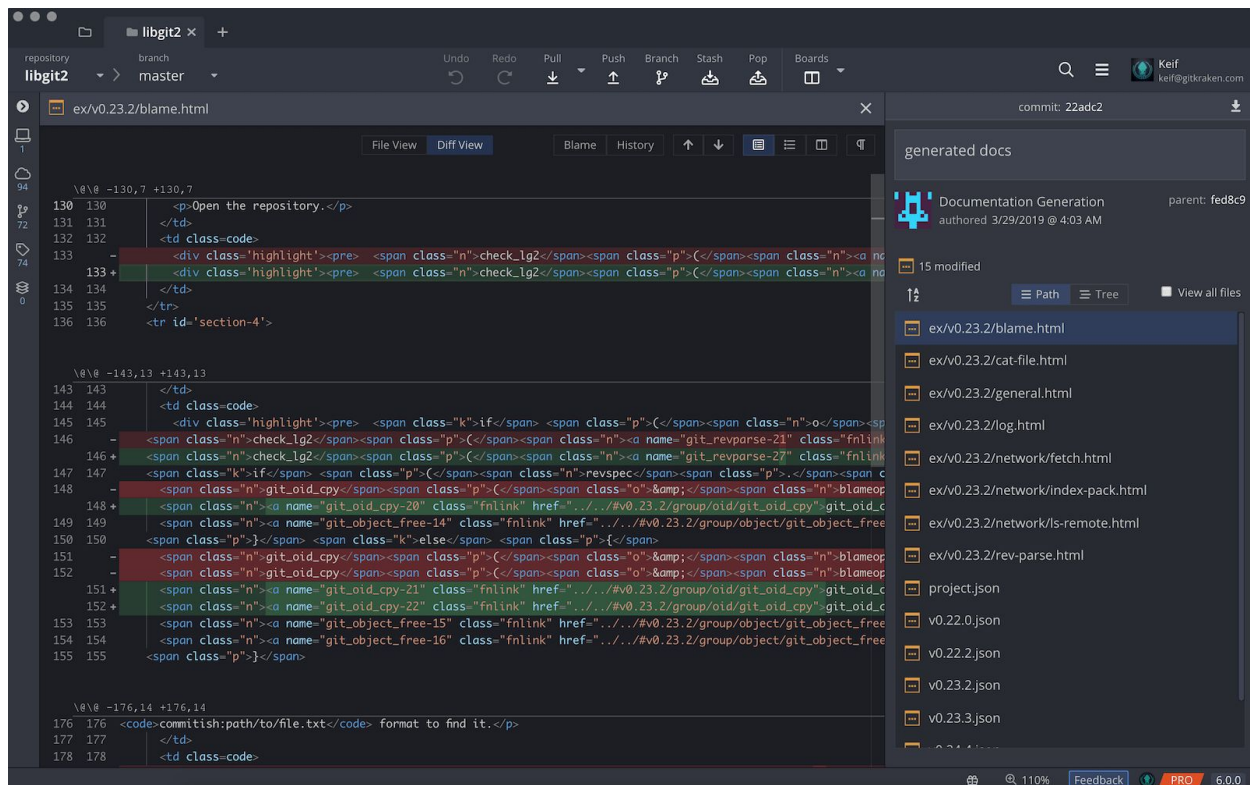

Training Resource: The **GitKraken Cheat Sheet** is a great way to quickly acclimate users with the GitKraken user interface.

Drag-and-drop to simplify power tasks

If a user has 2 branches, what actions are available between those branches? Is it possible to rebase or can they only merge? Even Git power-users may need to take a few moments to evaluate the state of their repo through the CLI before uncovering the answers to these questions.

However, users are able to completely bypass that uncertainty by performing a drag-and-drop action in GitKraken. The action will reveal what operations are possible for the 2 branches, without locking the user to any of those actions.

Users are empowered to freely explore their options between branches, within the context of their repo history.
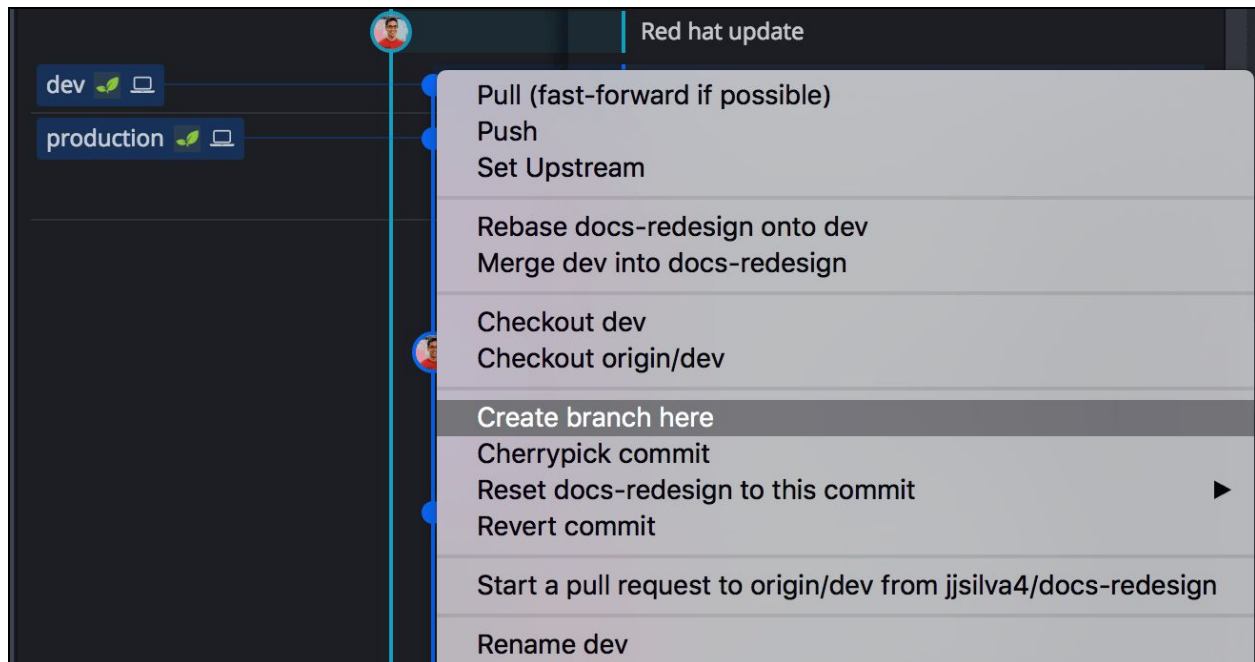

Training Resource: In under 4 minutes, this **Git tutorial video** will show you how to rebase in the CLI vs the GitKraken Git Client. You'll see what happens when a merge conflict occurs, and how to resolve it.

Collaboration through remotes

Remote repositories are the key to collaboration within a team. By adding remotes, users get visibility into their teammates branches. This visibility then makes it possible for users to checkout these remote branches to perform a code review for their pull requests.

What would normally take a `git remote add` command, followed by the name of a remote that a user has to lookup in the CLI, is performed with just a click in GitKraken. If a user has the GitHub integration set up in GitKraken, they will see the list of GitHub remote repositories they have permission to access.
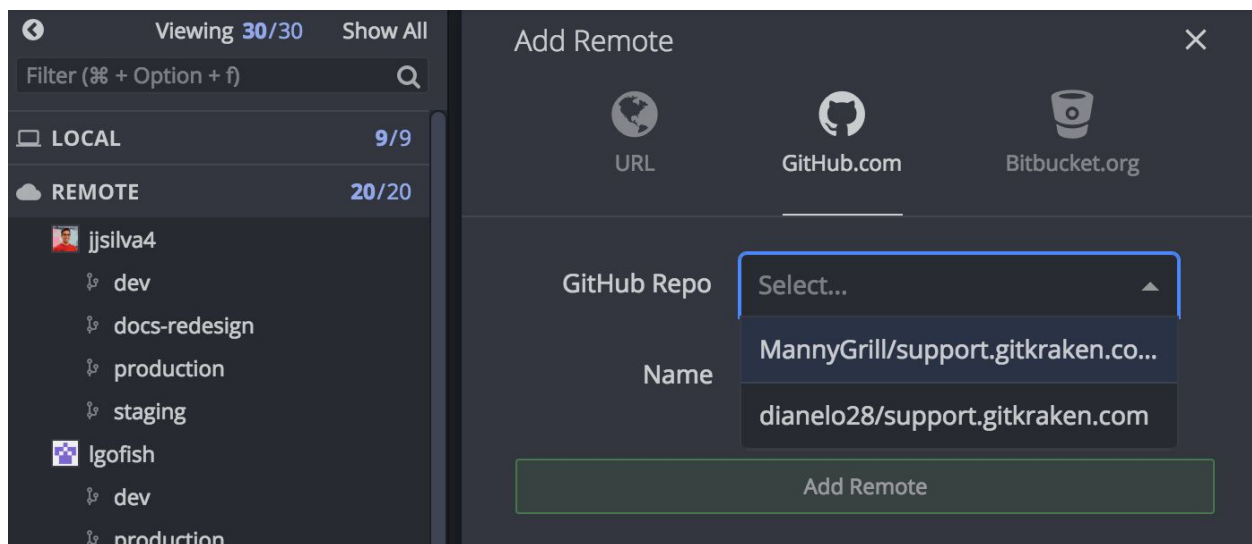

Remote branches are automatically added to the graph, making it quicker to identify how far along your teammates are with their work.
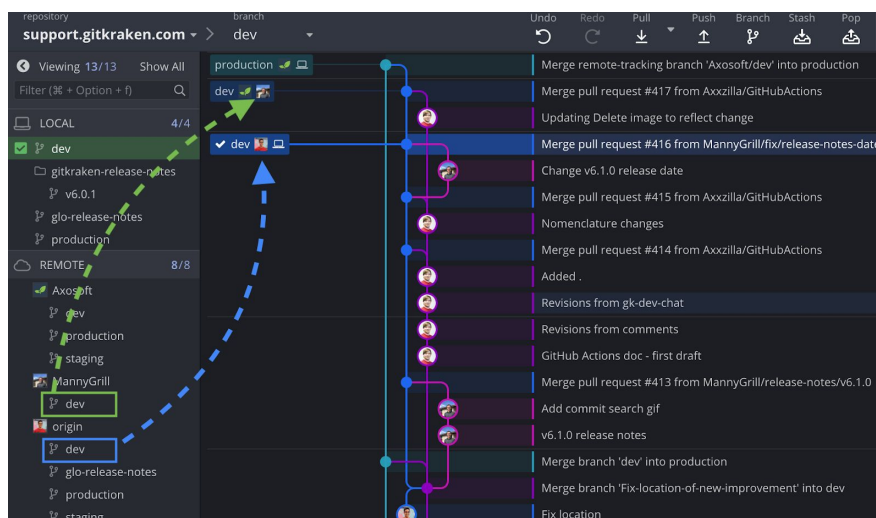
Note: GitKraken nevers hosts the source code for a repository. The app will either open a repository from a user's local machine or the app will authenticate with the Git hosting service over HTTPS or SSH. This remains true for all versions of the app.

## Collaboration through pull requests

Because remotes are better visualized in a GUI, it sets the stage for easier pull request creation. As pictured below, in GitKraken, users need only drag and drop one branch onto another to access the pull request option.
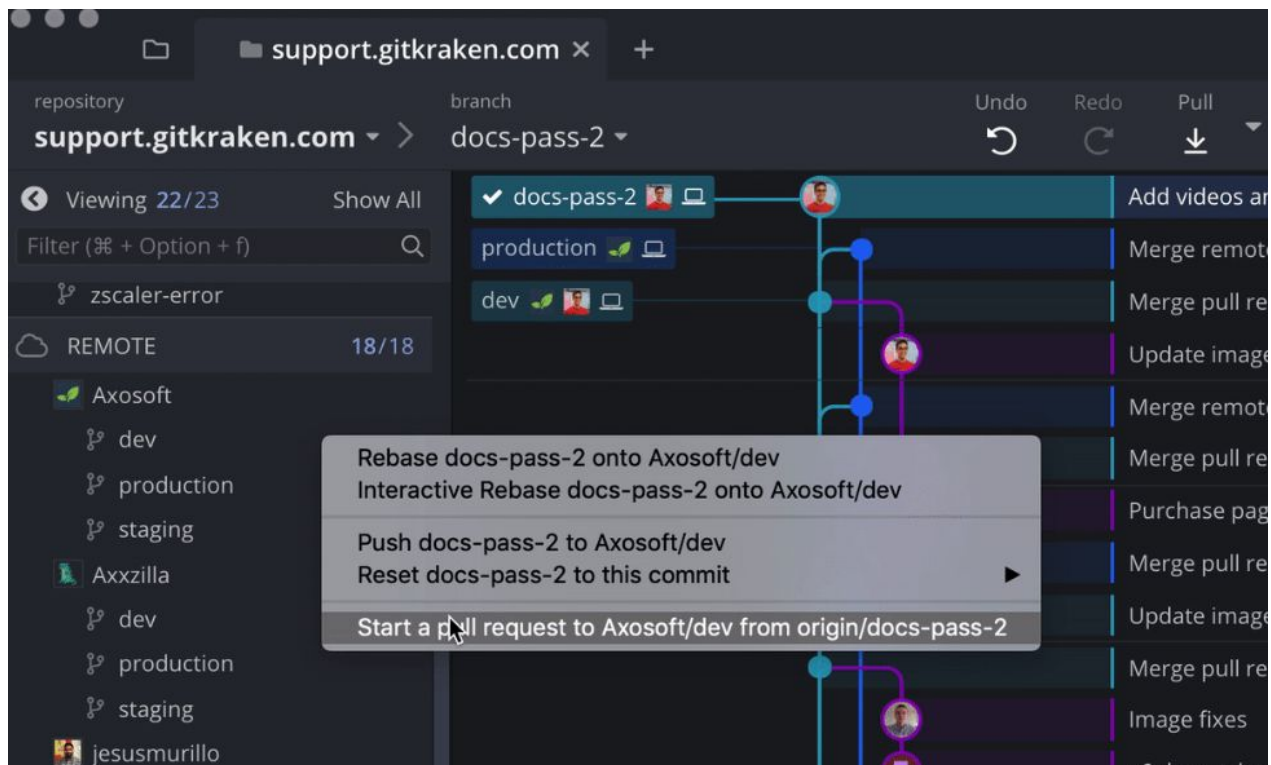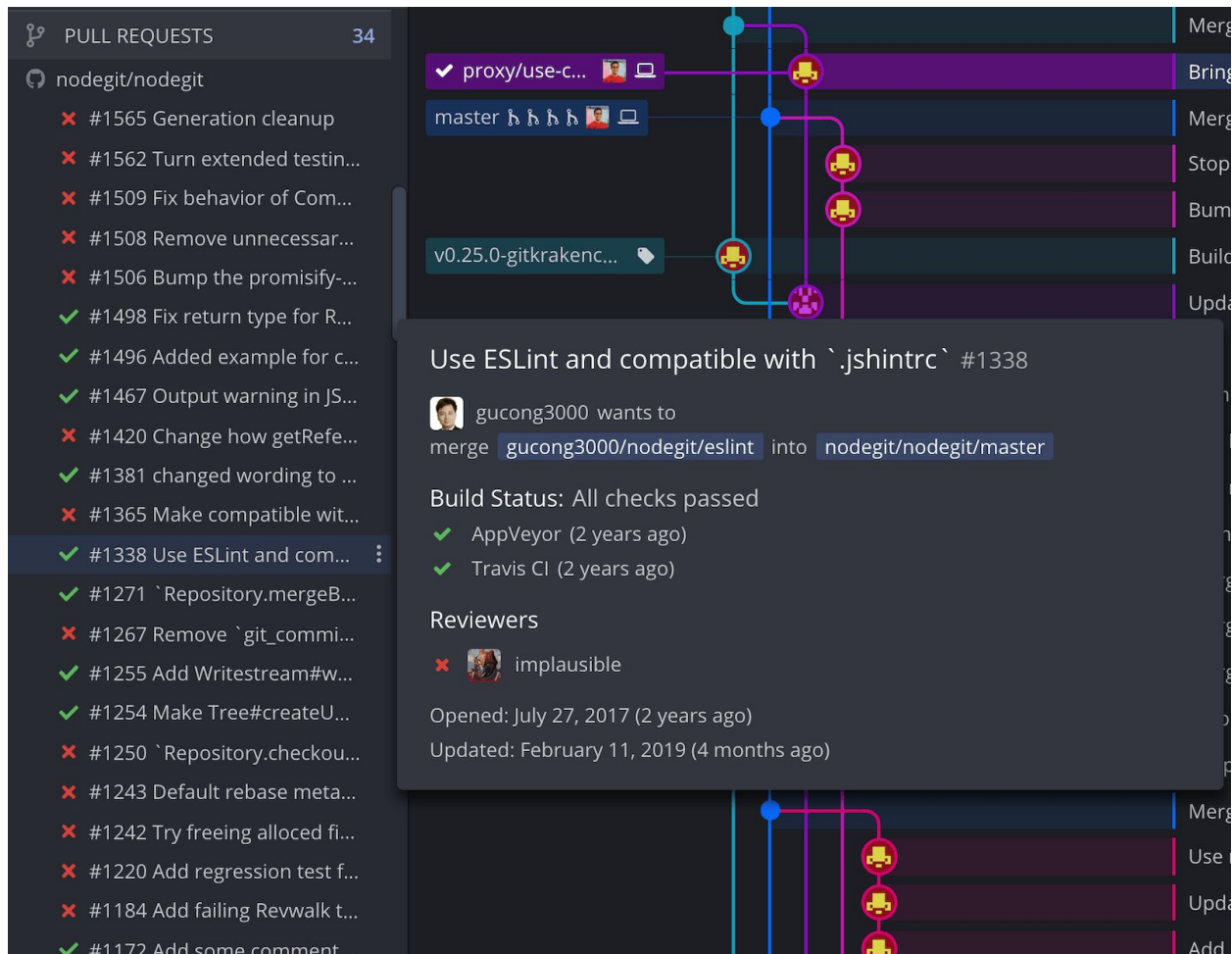
This drag and drop action prefills the repo, branch, title, and description information:
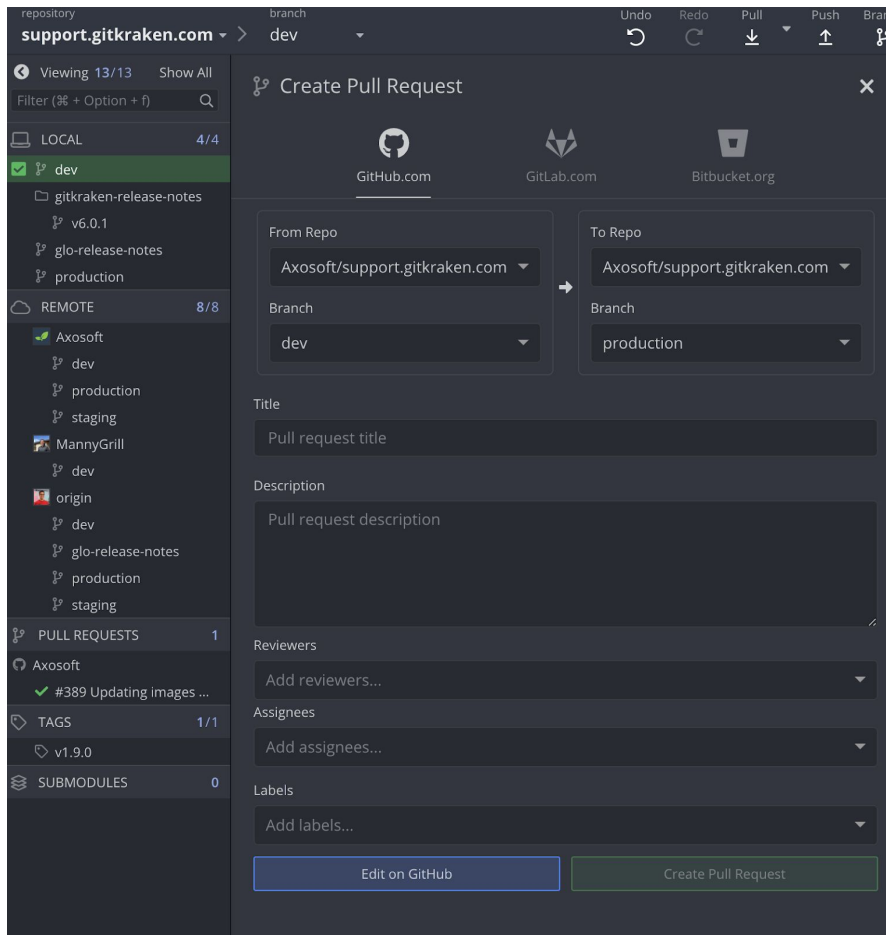
Furthermore, with the GitHub integration users can add reviewers, assignees, and labels to their pull request.
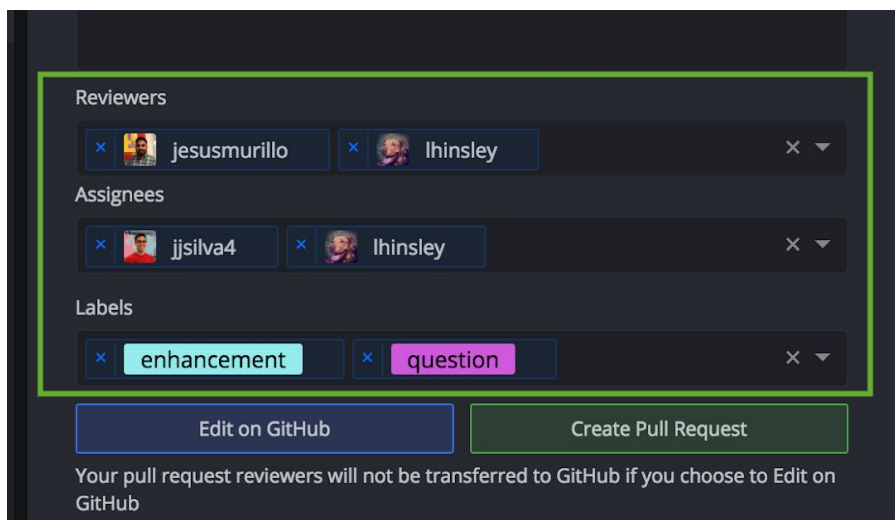
Pull requests are gathered in the left pane, and if a team has integration with a CI/CD tool, GitKraken will show the build status on hover.

All of this information is centralized in one location, saving the user the need to open up GitHub or any other tool to create a pull request, assign reviewers, or check on a build status.

Training Resource: In this comprehensive video guide, find out **How to use GitHub with GitKraken**, and how to set up integrations with GitHub.com and GitHub Enterprise.

Resolve merge conflicts faster

Merge conflicts force users to compare at least 2 different file versions and decide what changes to keep from each. While it is possible to have as little as 1 file in conflict between 2 branches, it is possible to have many more files in conflict—each with hunks of differences that must be reconciled.

Without a GUI, users may see something like this in the CLI:

```
jmarhee: ~/repos/test-rebase 🖋 git checkout feature_a                          [master]
Switched to branch 'feature_a'
jmarhee: ~/repos/test-rebase 🖋 vi test.txt                                    [feature_a]
jmarhee: ~/repos/test-rebase 🖋 cat test.txt                                   [feature_a*]
0804201901
jmarhee: ~/repos/test-rebase 🖋 git add test.txt                               [feature_a*]
jmarhee: ~/repos/test-rebase 🖋 git commit -m "done with feature a"            [feature_a*]
[feature_a 6cc182c] done with feature a
 1 file changed, 1 insertion(+), 1 deletion(-)
jmarhee: ~/repos/test-rebase 🖋 git push origin feature_a                      [feature_a]
Counting objects: 3, done.
Writing objects: 100% (3/3), 259 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:jmarhee/test-rebase.git
 * [new branch]      feature_a -> feature_a
jmarhee: ~/repos/test-rebase 🖋 git checkout master                           [feature_a]
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
jmarhee: ~/repos/test-rebase 🖋 git merge feature_a                            [master]
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.
```

From here, the user must open the conflicted file in a code editor and locate any sections that are partitioned by these **<<<** symbols.

```
diff --cc test.txt
index 20b652c,e932d49..0000000
--- a/test.txt
+++ b/test.txt
@@@ -1,1 -1,1 +1,5 @@@
++<<<<<<< HEAD
 +0804201702
++=======
+ 0804201901
++>>>>>>> feature_a
(END)
```

It is not clear in the text editor which branch's changeset the user is currently viewing, nor is it clear how the file's final output will look. While it is possible to resolve conflicts in the CLI and a code editor, GitKraken provides a much faster, more efficient method to address a merge conflict.


Training Resource: In under 3 minutes, this **Git tutorial video** will users what a merge conflict is, and how to resolve merge conflicts in the GitKraken Git Client.

Built-in forgiveness

Developers often break code before they can fix it, which is why it's important to build in fast ways to undo actions without needing to consult the web or colleagues.

With GitKraken, in one click users can undo a commit, merge, or checkout action so long as the change has not been pushed up to the remote.
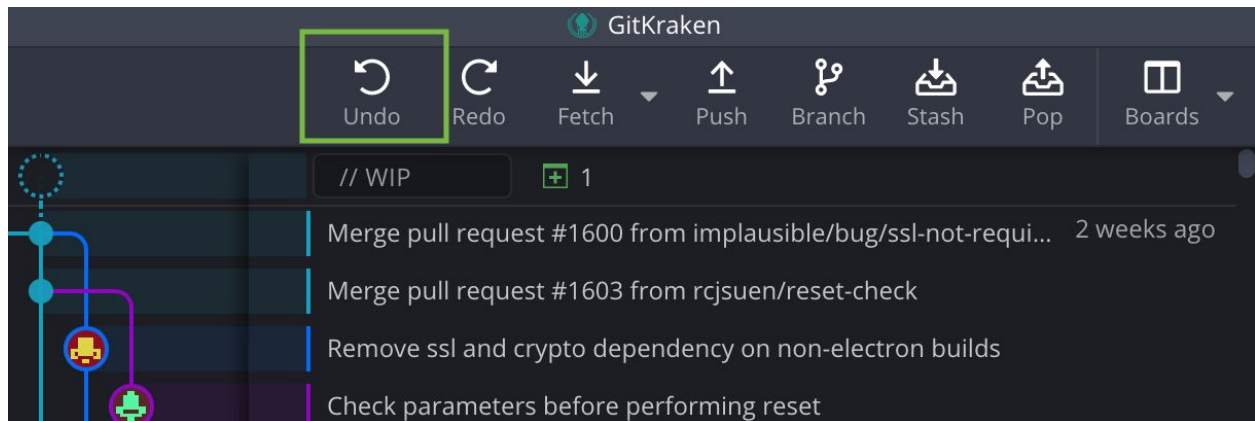
Additionally, because Git keeps the history of a project, the graph in GitKraken makes it easier for a user to identify a point in time they wish to explore. It is then possible to reset the current branch back to that point in time by simply right-clicking and selecting the preferred reset option.
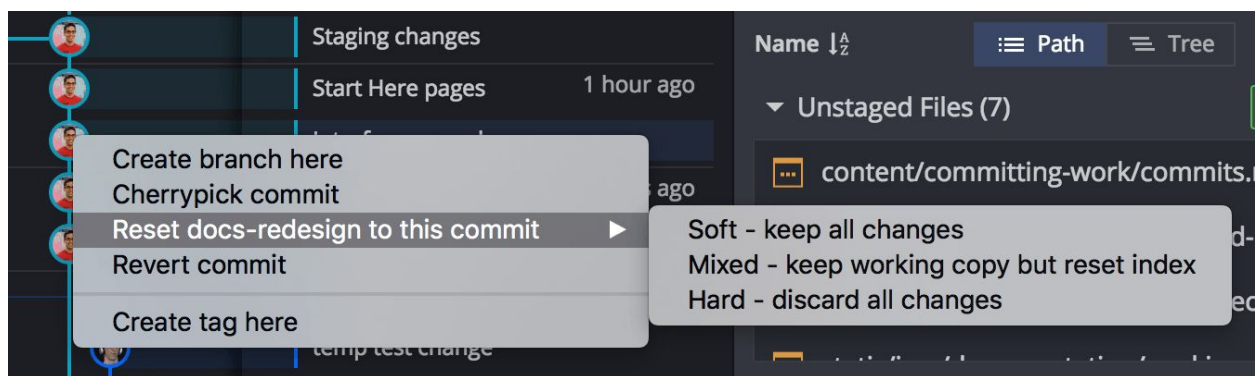
Without GitKraken, a developer would need to take the time to revert commits in the CLI until they got to a point where they had "undone" a commit. Git provides multiple paths to solving a version control problem, and GitKraken makes those options more accessible and discoverable to users.

Working in isolated networks behind a firewall

Organizations from industries such as defense contracting, medicine, finance, or government, often isolate their networks from the Internet for security purposes. However, software developers from these industries still demand the best productivity tools available on the market. These entities are thus drawn to self-hosted software solutions in order to meet their organizations' security standards while still delivering the tools their teams need.

GitKraken Enterprise addresses these security requirements by providing a self-hosted and a stand-alone option.

GitKraken Enterprise: Self-Hosted

GitKraken Enterprise Self-Hosted ships with its own account management system, release manager and the client downloads. This solution runs on a Linux server or virtual machine inside your network. Once installed and configured, your users' GitKraken clients will route user accounts, license validation and new release checks to your self-hosted GitKraken Enterprise server.

The GitKraken Enterprise self-hosted solution supports LDAP integration, which makes it easier to manage user licenses. In summary, the self-hosted solution provides the following:

- Self-hosted account management server for license management
- LDAP integration support
- Control over version updates

GitKraken Enterprise: Stand-Alone

GitKraken Enterprise Stand-Alone serves the same security needs as GitKraken Self-Hosted, but it saves teams from needing to maintain a server. Instead, only a license file and the GitKraken Stand-Alone clients are shipped.

The license file activates a user's GitKraken Stand-Alone client, and then they can get straight to work.

Deploying GitKraken Enterprise Stand-Alone

Because Stand-Alone ships without the account management system, organizations are required to manage the license allocation. Here are some best practices for dispersing the license file to users:

- Make the license file and client downloads available to users on a shared network folder.
- Email users the license file and a link to download the client.
- Share the license file and link to download the client via an internal communication tool.
- Track how many users use GitKraken to ensure you don't exceed the number of licenses purchased.

Training and educational resources

To make adopting Git and GitKraken easier, we have a series of educational videos, cheat sheets, and web resources that can help your developers come up to speed faster. You can share these resources with your teams:

- **Tutorial videos for learning Git**
- **GitKraken training resources for GitHub users**

Conclusion

A developer's time is better spent focused on programming, rather than dealing with the overhead that comes with Git. When developers—regardless of experience level—rely on the command line interface, it results in time wasted referencing Git commands, operations, and syntax online or through colleagues. Providing developers the GitKraken GUI will reduce the time it takes teams to collaborate and ship software on time and on budget.

About GitKraken

GitKraken is a product of Axosoft, where we develop software that is used by the world's most elite software engineers. Companies like Apple, Google, Microsoft, Amazon and thousands of other leading companies use GitKraken, Glo Boards and Axosoft.