2/5/19
Kaushal Mangipudi, Gerard Gaimari, Robert Kolmos, Kamden Chew

GitUp: A Simple, Portable Backup Manager with Practical Version Control

Motivation:

Backing up any project is essential to its reliable completion, and software projects are no exception. As more and more inexperienced non-developers work on software projects, they run into the problem of how to backup their work. Outside of software development, most people use products such as Google Drive, Microsoft OneDrive, Adobe Creative Cloud and Dropbox, which offer automated backup services to protect projects from being lost. Cloud-backup products are more reliable, secure, and easy to use than traditional external drive backups, which explains a lot of their widespread appeal. Some also offer version tracking that allows users to restore intermediary changes to files. However, these more traditional backup systems are far from ideal for backing up software projects. Cloud backup products are inadequate at keeping track of and comparing previous versions of a project. Previous versions only exist for a given time frame and changes are not grouped meaningfully. In software projects, even minute differences in program files can lead to wildly different program behavior, so it's important that coders be able to precisely identify changes in their source code over time.

To that end, many developers backup their work using distributed version control systems like git. Git provides a robust system for tracking changes in a project, and comes with a plethora of features that are invaluable for teams working on projects like pushing/pulling, tagging, branching, checking out past versions, etc. Its power has allowed it to become one of the dominant version control systems out there - a StackOverflow survey found that in 2018, almost 90% of developers used git to backup their work.[1] Git's popularity does not mean that it is considered perfect. We have had our fair share of mishaps with git, and have struggled with things like causing merge conflicts by failing to pull changes and reverting past versions of files. As single developers working on small scale projects, a lot of the git functionality was unnecessary, and was in fact detrimental to success. We were often overwhelmed by commands we almost never needed to use, and sometimes ended up using the wrong command as a result of the excessive complexity. Git, while usable for single users, was far from optimal. It felt bloated at times, and we were often bogged down and confused by functionality that we would almost certainly never use. We're not alone in our git woes - studies have shown that over seventy percent of users believe that git's UI and documentation need improvement.[2] Additionally, many users believe that git is too complex and has too many features, which can overwhelm even experienced developers, and make it very difficult for inexperienced users to use.[2]

There are many wrappers for git like GitKraken that aim to make git more user-friendly. They simplify many commands down to a couple button clicks, while also making it easier for teams to coordinate.[3] However, these wrappers still include the overwhelming and excessive amount of functionality that can confuse users. A lot of this functionality is necessary for large teams working on projects, but for single users, it only bogs them down. For example, merge conflicts are often inadvertent in large scale projects, but for a single user, they would mostly occur as the result of user error, and can and should be handled in a way to prevent the problems from occurring in the first place. Other git alternatives also run into similar problems. Gitless is a alternative version control system that arose directly in response to git's inadequacies.[2] It aimed to greatly simplify aspects of git that were unintuitive and difficult to grasp, but similarly to GitKraken, it also included a lot of functionality that is unneeded and confusing for new users. Even worse, it relies on a command line interface, which makes it even harder for non-developers to use.

In conclusion, git's design clearly caters to large-scale projects worked on by teams of more experienced developers that may require a more feature-rich environment. It begs for an alternative for smaller, single-user projects with very inexperienced coders who don't work in software. Git wrappers simplify many aspects of git, but still contain unnecessary functionality that single users don't need, while other alternatives like Gitless also retain excessive functionality and rely on unintuitive command line interfaces.

Proposed Solution:

GitUp is a product with the goal of preserving the reliability, automatic nature, and security that existing cloud-based backup systems provide while also providing users with the robust version tracking of files they need for software projects in an intuitive way. GitUp will be designed for users working on single-developer projects, with a focus on inexperienced users who need to code but work outside of software. To use GitUp, a user will open the application and login to their GitHub account. Once they are logged in, they and select the project they want to work on. GitUp will automatically track project file versions and update the backup whenever it detects a change. It will provide simple version restoration with granularity ranging from the entire project to individual file changes.

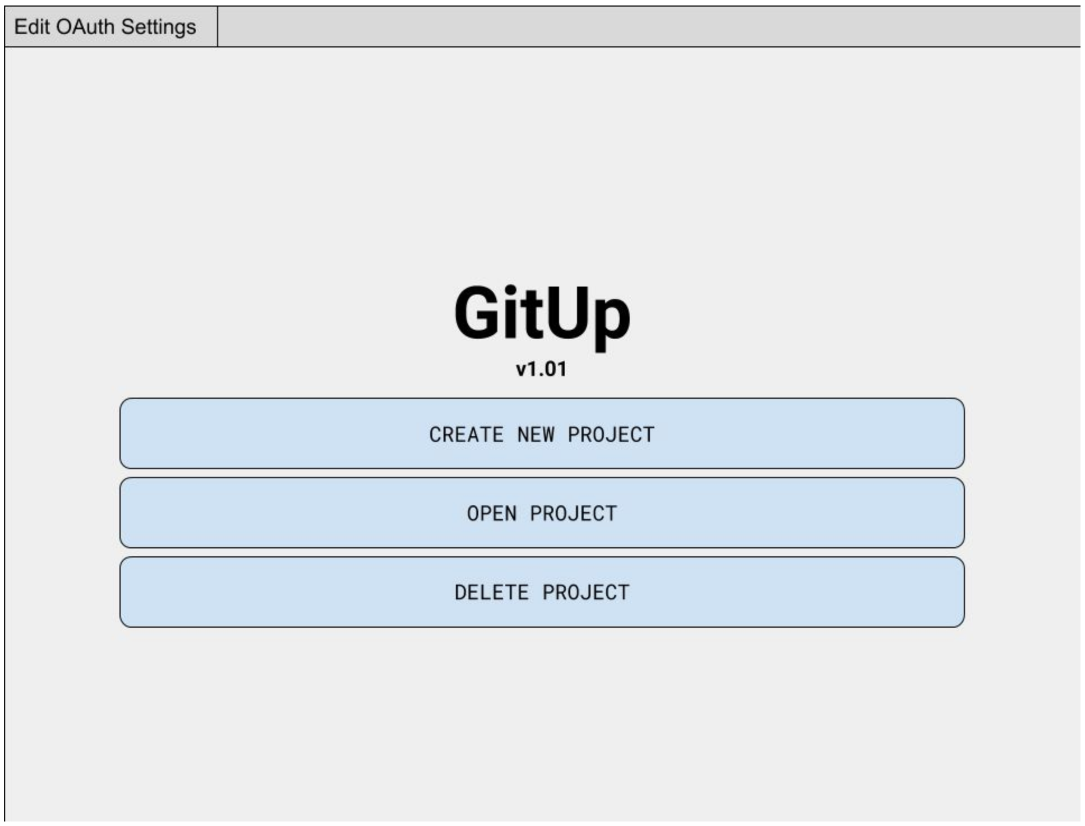
No other system has been created with the idea of single user projects in mind. Existing products are sufficient, but not ideal, for single users who are familiar with git, because they can get overwhelmed by the excessive functionality. However, users who work outside of software and have never used version control systems won't be able to use said products without investing considerable amounts of time to learning them. GitUp will be created to allow these inexperienced users to enjoy robust version tracking without having to learn about the underlying architecture of git or some other version control system.

GitUp's Target Functionality:

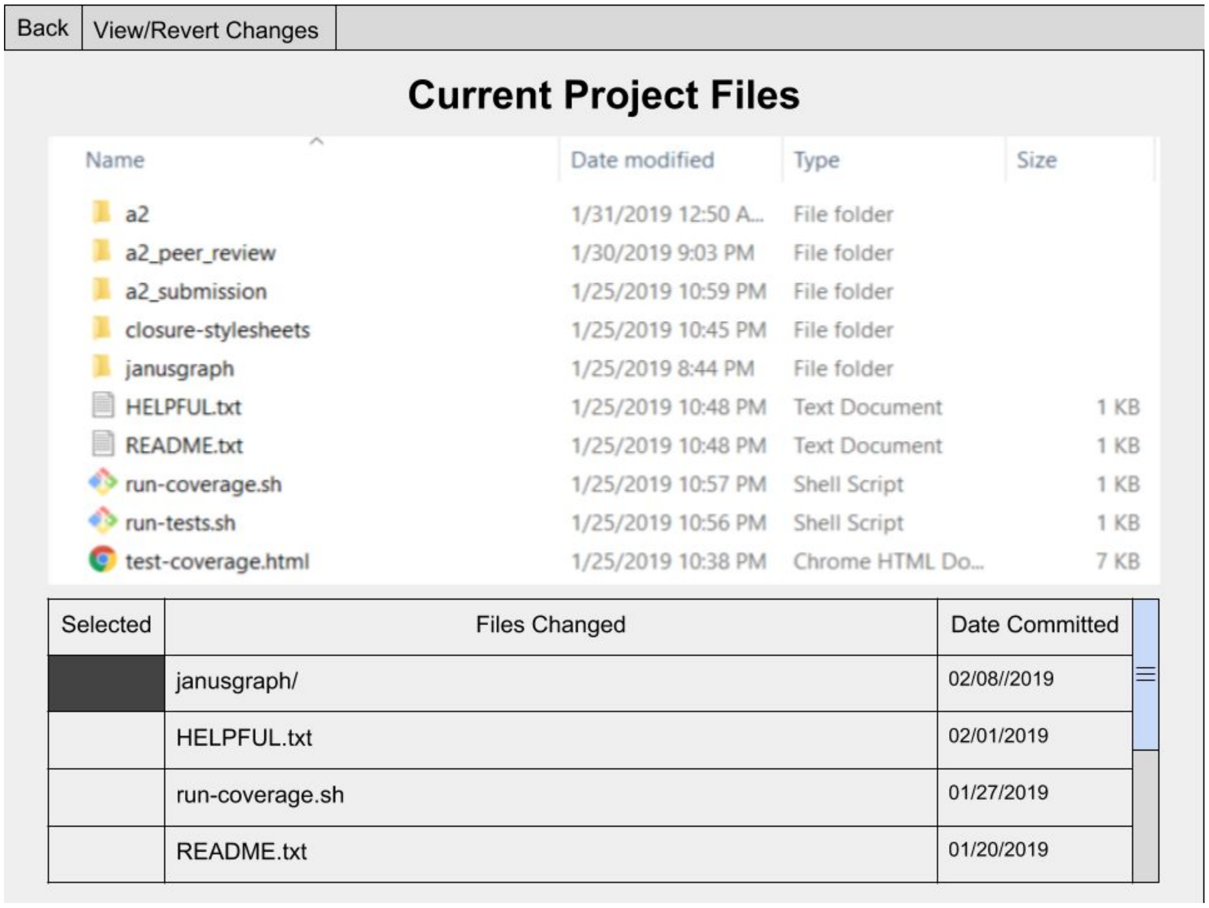
1. The user should be able to create a new GitHub project or open an existing one
2. Any locally saved changes should be automatically backed up on the remote repository
3. Whenever a project is opened up, the local version of it should be automatically synced with the remote version
4. Automatically backed up past versions will be grouped by files modified and/or time modified before being visible to users
5. The user should be able to view past versions of files and revert a file to a past version if they desire to
6. The risk of merge conflicts will be almost completely eliminated, and the user will be warned in advance about any potential merge conflicts that could occur
7. A very inexperienced user with minimal programming experience and no experience with version control systems should be able to do all of the above with the help of the user manual

User Interface:

When the user opens GitUp, they are greeted by the menu below. The user has the choice to either create a new project, open an existing project, or delete an existing project. From this screen, the user can also edit their OAuth settings by clicking in the top left corner



If the user selects a project, they are greeted by the menu below. The user is shown the current state of all the files in a file chooser interface, as well as a history of the groups of changes that have been made (found below the current files). Users can then select the commit they wish to view or revert. Selecting a commit will also mark the commit by filling in the "Selected" box adjacent to it. The user can then choose to View/Revert changes by clicking the "View/Revert Changes" button on the top left of the window. There is also a "Back" button at the top left of the window to allow the user to return to the main screen and select a different project.

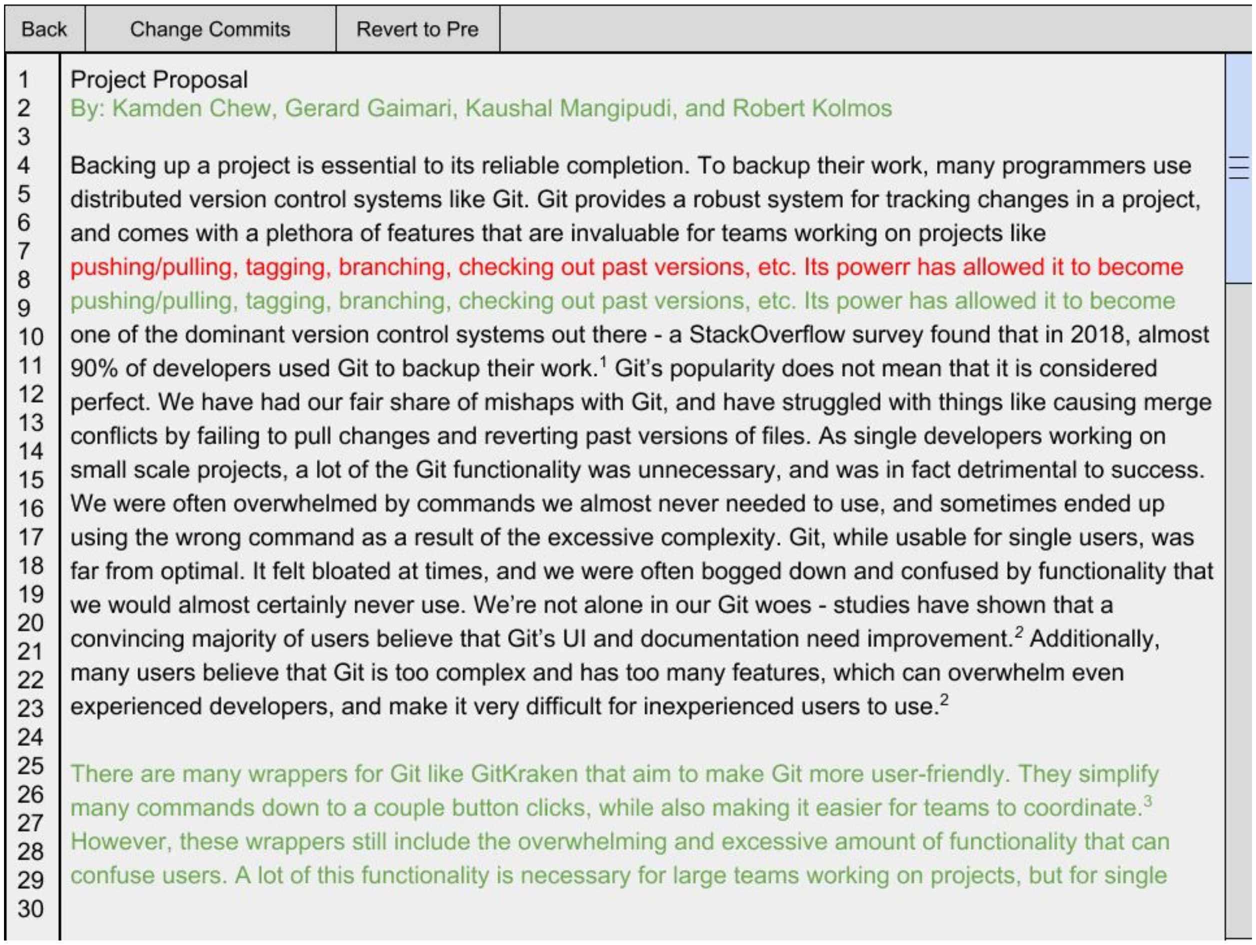


Current Project Files

Name	Date modified	Type	Size
a2	1/31/2019 12:50 A...	File folder	
a2_peer_review	1/30/2019 9:03 PM	File folder	
a2_submission	1/25/2019 10:59 PM	File folder	
closure-stylesheets	1/25/2019 10:45 PM	File folder	
janusgraph	1/25/2019 8:44 PM	File folder	
HELPFUL.txt	1/25/2019 10:48 PM	Text Document	1 KB
README.txt	1/25/2019 10:48 PM	Text Document	1 KB
run-coverage.sh	1/25/2019 10:57 PM	Shell Script	1 KB
run-tests.sh	1/25/2019 10:56 PM	Shell Script	1 KB
test-coverage.html	1/25/2019 10:38 PM	Chrome HTML Do...	7 KB

Selected	Files Changed	Date Committed
■	janusgraph/	02/08//2019
	HELPFUL.txt	02/01/2019
	run-coverage.sh	01/27/2019
	README.txt	01/20/2019

After clicking the “View/Revert Changes” button on the project screen, the user is greeted with the interface below. The user will be able to use the diff’s of the current file and the commit they selected before hitting “View/Revert Changes”. Insertions are marked in green, while deletions are marked in red.

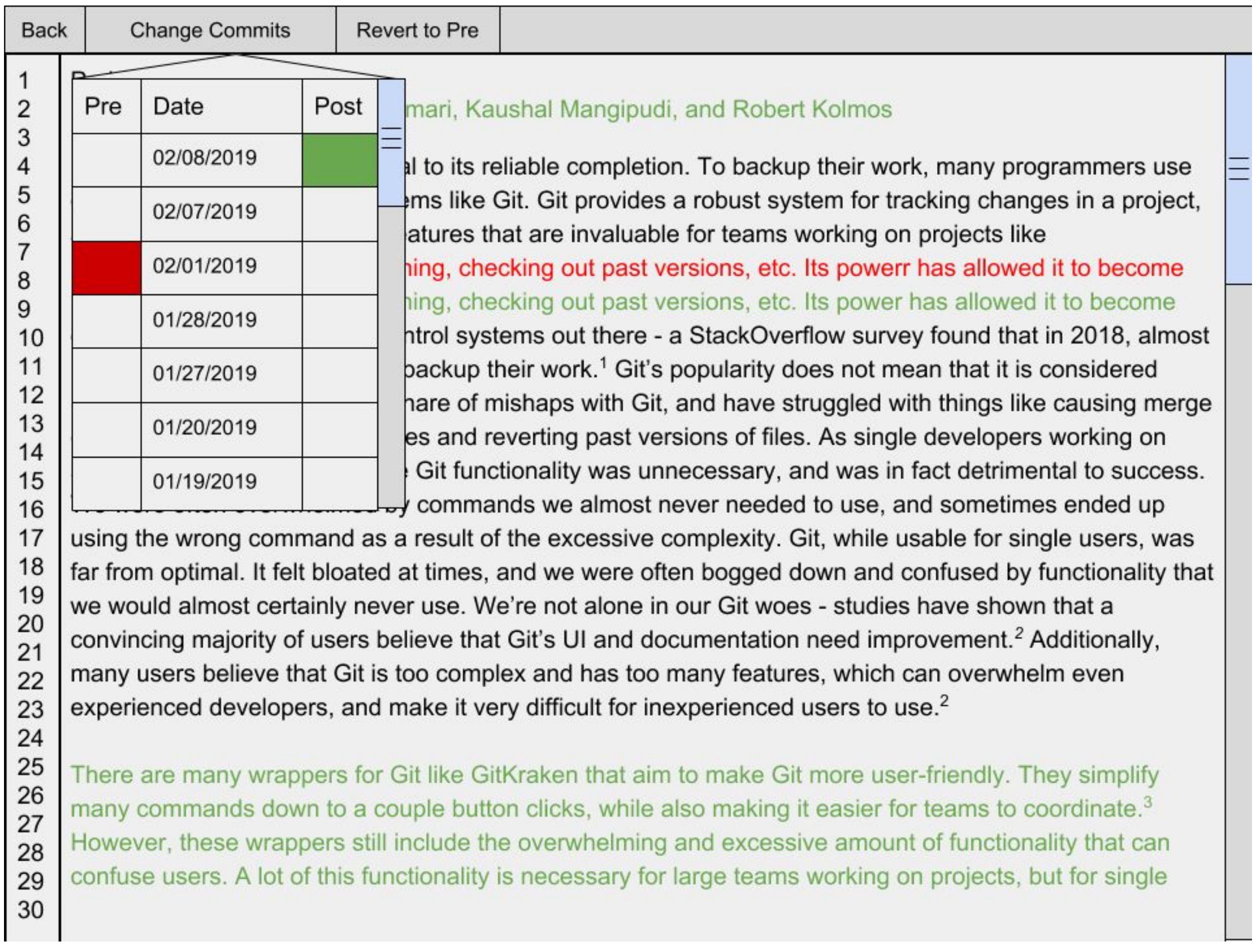


Project Proposal
By: Kamden Chew, Gerard Gaimari, Kaushal Mangipudi, and Robert Kolmos

Backing up a project is essential to its reliable completion. To backup their work, many programmers use distributed version control systems like Git. Git provides a robust system for tracking changes in a project, and comes with a plethora of features that are invaluable for teams working on projects like
pushing/pulling, tagging, branching, checking out past versions, etc. Its powerr has allowed it to become
pushing/pulling, tagging, branching, checking out past versions, etc. Its power has allowed it to become
one of the dominant version control systems out there - a StackOverflow survey found that in 2018, almost 90% of developers used Git to backup their work.[1] Git’s popularity does not mean that it is considered perfect. We have had our fair share of mishaps with Git, and have struggled with things like causing merge conflicts by failing to pull changes and reverting past versions of files. As single developers working on small scale projects, a lot of the Git functionality was unnecessary, and was in fact detrimental to success. We were often overwhelmed by commands we almost never needed to use, and sometimes ended up using the wrong command as a result of the excessive complexity. Git, while usable for single users, was far from optimal. It felt bloated at times, and we were often bogged down and confused by functionality that we would almost certainly never use. We’re not alone in our Git woes - studies have shown that a convincing majority of users believe that Git’s UI and documentation need improvement.[2] Additionally, many users believe that Git is too complex and has too many features, which can overwhelm even experienced developers, and make it very difficult for inexperienced users to use.[2]

There are many wrappers for Git like GitKraken that aim to make Git more user-friendly. They simplify many commands down to a couple button clicks, while also making it easier for teams to coordinate.[3] However, these wrappers still include the overwhelming and excessive amount of functionality that can confuse users. A lot of this functionality is necessary for large teams working on projects, but for single

If the user clicks the “Change Commits” button in the top left part of the window, then a table is brought up, allowing them to select a which two commits they are comparing in terms of the diff’s. The “Revert to Pre” button will allow the user to set the current file back to whatever previous commit the user currently has marked in the “Pre” section of the “Change Commits” table



Pre	Date	Post
	02/08/2019	🟩
	02/07/2019	
🟥	02/01/2019	
	01/28/2019	
	01/27/2019	
	01/20/2019	
	01/19/2019	

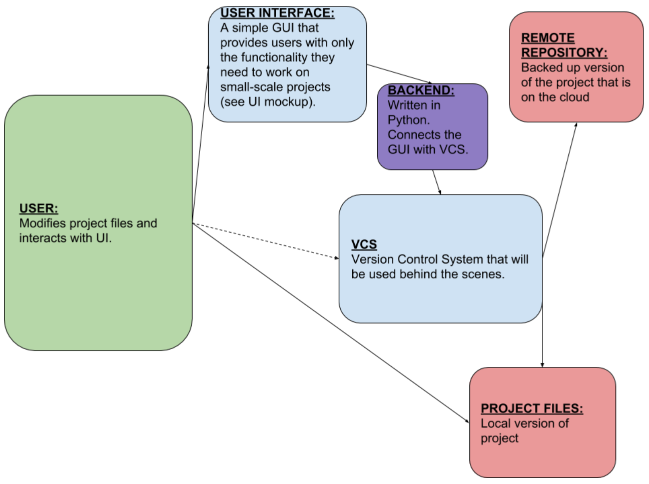
mari, Kaushal Mangipudi, and Robert Kolmos

al to its reliable completion. To backup their work, many programmers use
ems like Git. Git provides a robust system for tracking changes in a project,
atures that are invaluable for teams working on projects like
hing, checking out past versions, etc. Its powerr has allowed it to become
hing, checking out past versions, etc. Its power has allowed it to become
ntrol systems out there - a StackOverflow survey found that in 2018, almost
backup their work.[1] Git’s popularity does not mean that it is considered
hare of mishaps with Git, and have struggled with things like causing merge
es and reverting past versions of files. As single developers working on
Git functionality was unnecessary, and was in fact detrimental to success.
commands we almost never needed to use, and sometimes ended up
using the wrong command as a result of the excessive complexity. Git, while usable for single users, was far from optimal. It felt bloated at times, and we were often bogged down and confused by functionality that we would almost certainly never use. We’re not alone in our Git woes - studies have shown that a convincing majority of users believe that Git’s UI and documentation need improvement.[2] Additionally, many users believe that Git is too complex and has too many features, which can overwhelm even experienced developers, and make it very difficult for inexperienced users to use.[2]

There are many wrappers for Git like GitKraken that aim to make Git more user-friendly. They simplify many commands down to a couple button clicks, while also making it easier for teams to coordinate.[3] However, these wrappers still include the overwhelming and excessive amount of functionality that can confuse users. A lot of this functionality is necessary for large teams working on projects, but for single

Architecture:

GitUp will be designed to work on Linux machines, though we may expand it to other Operating Systems if time permits us to. It will use a version control system behind the scenes. This could have been any version control system, but it was decided we’d use git since it’s the version control system we are most familiar with and because we plan on using OAuth to link GitUp with the user’s GitHub account. The user can work on a project using multiple devices, but each project


USER INTERFACE:
A simple GUI that provides users with only the functionality they need to work on small-scale projects (see UI mockup).
REMOTE REPOSITORY:
Backed up version of the project that is on the cloud
BACKEND:
Written in Python. Connects the GUI with VCS.
USER:
Modifies project files and interacts with UI.
VCS
Version Control System that will be used behind the scenes.
PROJECT FILES:
Local version of project

can only be worked on by a single user, and the user should only use one device at a time. GitUp's architecture will be based around a simple GUI that aims to emulate the interface of cloud-base backup systems like Google Drive and Dropbox, with additional functionality involving viewing and reverting past versions of files. When the user interacts with GitUp's GUI, GitUp will use git commands in the background to provide the necessary functionality. The user can still technically interact with their project using git in the command line, but the user will never need to, as GitUp will handle all interactions between the local and remote repositories, and will in fact be encouraged not to in the user manual. GitUp will use git to help sync the local and remote versions of the project. Whenever a user opens a project in GitUp, it will try to to use git to automatically pull the project. If it fails, it will issue a warning to the user that any changes they make might to certain files might lead to merge conflicts down the road. Since there is a single user, they can know to avoid modifying those files until they have been properly synced. If an automatic push fails, GitUp will periodically reattempt to push the file to the remote repository. If the user closes GitUp before a push succeeds, GitUp will issue a warning that the file is out of sync.

Implementation:

The UI will be implemented with Tkinter[4] for Python since this library is platform independent allowing a single build to produce a GUI for OS X, Linux and Windows 10 if necessary. Tkinter provides a basic UI kit that should be sufficient to create a UI resembling the above mockups. Python was chosen for GUI implementation so we can easily interface with the GitPython[5] API that GitUp will use to utilize git in order to interact with the local project files and the remote repository. GitUp will automatically push by becoming aware of file opens and closes through Linux's inotify API.[6] When a user sets up a project, GitUp begins monitoring file opens, and closes in the directory associated with this project and it's subdirectories. When GitUp sees that a file has been opened it stores the current state of the file, and begins checking intermittently to monitor the magnitude of changes the user has made. When the user closes the file, or GitUp determines that a significant amount of changes have been made, GitUp automatically commits the changes to this file. If the file is still open when a commit is made, GitUp updates the stored current state of the file, and proceeds monitoring changes. When a file is closed GitUp stops intermittently monitoring it for changes. We will display a warning to a user about potential merge conflicts whenever an automatic push or pull fails, and to do so, we will use a popup implemented with Tkinter to do so. We will not put any hard blocks on modifications that could lead to merge conflicts, because the user is guaranteed to have seen a warning about merge conflicts since they are the only user working on the project. If the user still causes a merge conflict despite the warning, we will automatically resolve the merge conflict, though we haven't determined how exactly we will do so. With regards to grouping commits together to give the user a big picture of changes made to their project over time, we plan on grouping commits together by the files modified and/or the times they were modified. We will save our grouping data in a file that will be pushed to the repository in GitHub but will remain invisible to the user when using GitUp's interface. The exact algorithm to group commits will be finalized after testing.

Challenges and Risks:

Design Challenges:

One fundamental design risk that will need to be constantly addressed during the development of GitUp is that its design does not satisfy the problem it aims to solve, i.e. GitUp does not provide single users with almost all the functionality they need and/or it is bloated with too much functionality and difficult/unintuitive to use. To mitigate this risk, the GitUp team will conduct comprehensive user research at all stages of the development process using a group of developers familiar with git and a group of inexperienced developers with little to no version control experience who have kindly agreed to help us. When collecting information, we will focus on determining the most useful version control features needed in a backup system, user desires with respect to backup frequency, size and speed, and how we can ensure that user interaction with the system is simple and automated.

One technical challenge is how we plan on handling merge conflicts when the user causes one despite numerous warnings to be careful. We see three potential solutions to this. The first is to automatically replace the file with the version that is being pushed. A second possibility would be to save a renamed version of the file to the repository that the user could look at. The third would be to utilize a GUI based merge conflict resolution tool to help the user manually resolve the conflict. Given the complexity of difficulty of the third option, we are leaning towards either the first option or the second. Another technical challenge is grouping automatic commits together in meaningful ways that users will able to use to get a big picture idea of how their project has changed over time. To that end, we will group commits using a method that is time-based and/or file-based. If we decide to group commits by time, then we might choose to group commits that have happened since the last pre-defined time interval without changes. If we instead decided to group commits by file, then we might group all commits that edit a particular file. Either way, we will need to fine tune the intervals which we will use to divide all changes into groups of commits. It will be difficult to choose numbers to use for this, so we'll need to test various frequencies to determine the best one to use.

Implementation Schedule:

Date	Goal(s)	Test(s)	Implementers	Evaluators
02/04	Finish preliminary user usage data and finish designing a preliminary UX/UI	Be able to run example use cases through UX flowchart. Results of survey collected.	Rob, Kamden	Gerard, Kaushal
02/11	Automated pushing/pulling and OAuth	Be able to complete the basic project use case [1]	Kaushal, Gerard	Rob, Kamden
02/18	Edge case avoidance/File reverting	Be able to complete the syncing warnings use case [2] and reverting files use case [3]	Gerard, Kaushal	Rob, Kamden
02/25	Begin work on smarter grouping of changes	Collect customer data on what users think a 'smart grouping' of changes are, and write a smart grouping use case/test based off the customer data	Kaushal, Kamden	Rob, Gerard

03/04	Smarter grouping of changes	Be able to complete the smart grouping [4] and final use cases [5]	Kaushal, Kamden	Rob, Gerard
03/11	Collect and respond to feedback on finished GitUp	Feedback fully collected.	Rob, Gerard	Kaushal, Kamden
03/18	Feedback Implementation	Dependent on feedback received. Fixes implemented address most important user concerns with GitUp	Rob, Gerard, Kaushal	Kamden

Testing & Evaluation:

These use cases are intended as examples of specific behavior that should work in order for the relevant phase of the project to be considered finished. These are not the exact tests that will be written as part of the test suite, but are meant as high level behavioral requirements. It is assumed that test suites for each phase will be written in order for the phase to be considered completed.

[1] Basic project use case: This test is intended to ensure that the basic functionality of automatic pushing/pulling works. In order for this part of the project to be considered successful a user should be able to:

- Open GitUp on machine 1
- Login to GitUp using a GitHub account
- Create a project "hello world" on machine 1.
- Add file helloworld.txt to project, and make changes on machine 1, and save the changes
- Verify that the changes were automatically pushed.
- Close GitUp on machine 1
- Open GitUp on machine 2
- Open the "hello world" project on machine 2.
- Verify that the version of "hello world" on machine 2 is up to date
- Make changes to helloworld.txt on machine 2, and save
- Close GitUp on machine 2
- Open GitUp on machine 1
- Verify that helloworld.txt is up to date

[2] Merge conflict use case:

- Create a project "parallel test" on machine 1.
- Add file "paralleledit.txt" to project, and make changes on machine 1
- Turn off internet on machine 1
- Save the changes on machine 1
- Verify that the user is warned that the automatic push failed
- Close GitUp, verifying that the user is warned once again that their changes haven't been backed up

- Turn on the internet in machine 1
- Verify that the changes are pushed to the remote repository
- Close GitUp
- Verify that there are no warnings about merge conflicts
- Turn off the internet on machine 2
- Open GitUp on machine 2, and open parallel test
- Verify that a warning appears about not being able to sync the local version of parallel test with the remote version

[3] Reverting files use case:
- Create a project “Test revert” on machine 1
    - Create a file “version-test.txt” in the project with the line “version 1”.
    - Commit/push all changes on machine 1.
- Open “Test revert” on machine 2
    - Add the line “version 2” to version-test.txt
    - Create the file “extra-file.txt” with the line “Extra unrelated file”
    - Commit/push all changes on machine 2.
- Open “Test revert” on machine 1
    - Edit the file “version-text.txt” by adding the line “version 3” to it
    - Commit/push all changes on machine 1.
    - View the past version of version-text.txt with line “version 1”
    - View the past version of version-text.txt with lines “version 1” and “version 2”
    - Revert version-test.txt to the past version with lines “version 1” and “version 2”
- Open “Test revert” on machine 2
    - Validate that version-test.txt has been successfully reverted back to the version with lines “version 1” and “version 2”

[4] Smart grouping case:
We are choosing to delay defining what smart change grouping means exactly until we receive feedback from users as to how they value various styles of grouping commits (grouping based on files, grouping based on time changes have been made, etc.).

[5] Final use case:
- Run all permutations of orderings of previous phase tests, and verify that all tests still pass when called in any order.

Evaluating GitUp’s success will require that we not only test that the implementation is working as expected, but that it achieves the goal of making project management for new/novice developers easy and intuitive. To make sure that our product hits this mark, we have a pool of new/novice developers with little to no version control experience who are willing to try out GitUp after it’s implementation is complete. We will be able to evaluate if our product is useful to its target audience by seeing if our pool of novice testers are able to navigate the system easily, or at least more easily than competitors. To that end, we have standardized testing instructions and surveys for each phase of development. The instructions give general goals that we want the testers to be able to achieve, and they will attempt to satisfy those goals in a time limit by only using the user manual for assistance. After they complete the test (or run out of time), we have them fill out a survey about their experience. If we find that GitUp is an easier version control system for inexperienced users to use, then we will have filled the glaring gap that git leaves behind in regards to simple and easy version control for novice developers who are focused on small-scale single-user projects. Additionally, we'll ask a group of developers who are familiar with git to try making their own project using GitUp and fill out a survey focused on whether they were able to successfully do so using GitUp and multiple machines.

Work Used:

[1]**https://insights.stackoverflow.com/survey/2018 Accessed 2/12/19**

[2] **What's Wrong with Git? A Conceptual Design Analysis. Santiago Perez De Rosso, Daniel Jackson. Computer Science and Artificial Intelligence Lab at MIT.**
Accessed 2/12/19 at https://spderosso.github.io/onward13.pdf

[3]**https://www.gitkraken.com/git-client Accessed 2/12/19**

[4]**https://docs.python.org/3/library/tk.html#tkinter Accessed 2/12/19**

[5]**https://gitpython.readthedocs.io/en/stable/ Accessed 2/12/19**

Hours worked on report (team total): 72