Final Report

Git OSS-um

Gary Baker
Van Steinbrenner
Stephen White

May 8th, 2019

Project Sponsor: Dr Igor Steinmacher,
Assistant professor, SICCS, Northern Arizona University

Faculty Mentor: Ana Paula Chaves Steinmacher

Version 1.0

Table of Contents

Introduction

Open Source Software (OSS) is a type of computer software in which source code is released under a license, where the copyright holder grants users the rights to study, change, and distribute the software to anyone and for any purpose. OSS may be developed in a collaborative public manner using a version control system named Git and hosting the project itself on a site known as GitHub[1]. This version control system tracks changes made to a project and easily allows projects to have multiple contributors. It also allows for a project owner or manager to accept or decline contributions depending on their opinion of the quality of the change through what is called a pull request.

Unfortunately, a substantial amount of developers do not contribute to open source projects. We have all been there: we do not know where to start, what files are important, or what needs to be changed. The community may be friendly and quick to provide assistance to a newcomer, or the contributor may get berated for submitting a pull request that does not adhere to the project's standards. It is so daunting that the newcomer gives up and moves on. This experience shows that there is a need in the open source community for a tool that will allow a novice passing through to quickly and efficiently determine the newcomer "friendliness" of a repository.

Our client, Dr. Igor Steinmacher, is a researcher and assistant professor at Northern Arizona University. Dr. Steinmacher and his colleagues have been looking into the issues newcomers come across when trying to make their first contributions to Open

[1] GitHub: https://github.com/

Source Software. The main problems with contributing to Open Source Software as seen from the newcomers' perspective are as follows:

- Information overload
- Inactivity/no feedback
- Acceptance v.s discouragement

Due to these issues, we will be spearheading a solution to help the newcomer simplify the process in starting with the right repository for them. Information overload refers to situations where the newcomer navigates to a repositories landing page and there is an hours worth of reading material and a ton of different downloads that need to be done just to start making their first contribution. Inactivity/no feedback refers to the situation where a newcomer makes it through the process of making their contribution and submits it, but never receives any feedback. Their contribution is neither accepted nor declined, leaving the contributor in a state of uncertainty whether they did their contribution right or wrong. Lastly, acceptance versus discouragement is where a newcomer to Open Source attempts to contribute to a repository and after submitting their contribution, it is either accepted or declined. Some communities surrounding certain Open Source Software can be very harsh and quick to dismiss someone's contribution as useless, being very discouraging in the process. This is a negative experience that makes the newcomer feel bad, not want to attempt to contribute, and discouraging them from Open Source in general. If experiences like this continue and grow in number of occurrences, the community and industry of Open Source itself may have it's growth stunted and potentially begin the decline of the industry as a whole.

Our job is aimed at giving power back to the newcomer by creating an autonomous process where relevant data of a requested repository is presented to the user. This will allow them to decide if the specific OSS project will support their needs and help them feel at ease when picking their first open source project.

In this document, we will be going over the entire process and final product that was created as a result of our Capstone experience. In the following sections, the Process Overview, Requirements, Architecture and Implementation, Testing, Project Timeline, and Future Work will be covered in detail. There is also a Glossary and Appendix at the end to use as a reference. The Glossary is a list of defined technical terms used throughout the document and the Appendix is a guide for developing the Git-OSS-um tool further.

Process Overview

From the very beginning of Capstone, planning has been one of the most important elements of development. The same can be said about software development in general as well. The most straightforward path to a successful application is built upon a strong foundation created by robust planning.

The first half of our development process for this product was almost entirely composed of planning and requirements acquisition. We began by analyzing the problem at hand with our client, taking many notes of the issues with the current state of the art as it pertained to Open Source Software and newcomers to this industry. After much discussion with our client, we had a robust list of requirements and were ready to move on to how we would satisfy all of these requirements.

In order to figure out how to satisfy all of the requirements, we did a lot of research on common technologies used for things that had similar characteristics as our list of requirements. The key requirements that shaped our search for the right technologies were that it had to be a web-based application, we had to mine data from GitHub, and we must display interactive graphics to the user that would be based on the data mined from GitHub. These main points served as a solid guide for our technologies research. What we came up with after a couple weeks of research and discussions between the team as well as the sponsor, we came to a final plan that we would be using Python to mine the data and host that as the backend of the application on one server, then we would host the frontend of the application in which the user sees where the graphics and data would be displayed. Connections between the frontend and backend would be made by creating our own REST API to send data back and forth. This marked the end of the research phase and it was time to write our Technological Feasibility report, proving that our design concept would work together and that we were making the correct decisions on the technologies we would be using for this solution.

At this point, we got through writing the entire document on how these technologies would work without any issues and we were all set to make the product using our concept. We made the decision to create a very simple prototype that performed all of the main functionality that would be in the final product using a different concept just to maintain simplicity and prove that the main functionality would work just fine. Instead of using the separate servers for frontend and backend for the demo, we decided to just use a single development framework called Django that is built upon the scripting language Python. The demo worked great and included much of the core functionality that the final product would make use of. After the demo was completed and shown, we spent the time between Fall and Spring semesters learning the technologies that we had decided to use for our product.

At the beginning of the Spring semester, we met as a team and made a major game changing decision for our development plan. As we discussed our previous plan more, we realized that we were highly overcomplicating the solution for this project and found ourselves asking “Why don’t we just use Django like we did for the demo instead of the separate servers and REST API?”. We didn’t come up with a good answer to this question, so we decided to scrap the previous plan and go with Django in full force. Now the plan was to build the backend using Python (like the original) but within the Django framework as well as build the frontend with Django since it had the capability to do so. This reduced complexity by having all of the code base in one framework, allowing us to use technologies that we were used to (Python, HTML, CSS, Bootstrap), and getting rid of a point of failure by putting the entire product on a single server rather than have it on two separate instances and relying on communication between the two.

Now that we had our new plan in place, we had spent the first few weeks reworking our idea and it was time to implement it. We started out by getting our demo copied over to a new project on GitHub and began from where we left off on the demo. Since we had much of the backend in place already in Python, we needed to get the frontend application built with all of the pages that we would need. Once the necessary pages were completed, we continued adding backend functionality that worked hand-in-hand with the frontend such as displaying the list of repositories in our database, filtering through those repositories, and generating graphics for each repository based on data mined from GitHub for any repository in our database.

January and February came and went, at this point it was March and time to show our Alpha prototype which had all backend functionality implemented and most of the frontend completed. After the Alpha demo was shown, the rest of our Spring semester was comprised of refinement of both frontend and backend based upon user feedback as well as testing the application via Unit Tests, Integration Tests, and Usability Testing. This testing and refinement led us to the final deployment of the application and getting our sponsor’s stamp of approval.

This was our development process at a glance, now let’s take a more detailed look at the product itself, starting with the Requirements for the product.

Requirements

Our requirements acquisition method mainly came from weekly meetings that we had with our sponsor. As meetings progressed, we eventually created a Requirements Acquisition document that heavily details the necessary functionality that our product must have to be accepted by our mentor. After meeting with our sponsor throughout the first semester, we have created two functional requirements with many sub-requirements attached and three non-functional requirements that are summarized in the sections below.

Functional Requirements

As stated above, there are two separate functional requirements along with many sub-requirements that are implemented. The requirements are to let the end-user login and view data for a specific repository from GitHub and have administrative management capabilities. From our requirements specification document, the first requirement is larger than our second just because of the necessary functionality that is present to fully implement the requirement.

Our first functional requirement of allowing an end-user to login and view data for a specific repository from Github has extensive subsections because of the technology and functionality that was implemented. Three major subsections of this requirement include a search and filter for repositories, request based mining from end-users, and the analysis of data and display of interactive graphics.

The first subsection of search and filter has been implemented as part of our “Mined Repo” page on the web application. The search is simply a textbox where end-users can type in a repository to see if it has been mined by our system. The filter is a set of fields where a user can fill the selections out to fit the necessary criteria. The second subsection is a request based mining form, where once a user is signed up, they can request a repository to be mined if it is not already in our system. When the request is made, it will be sent to the admin portal to be accepted or denied for mining. The final subsection is the analysis of data that is mined. This is represented for individual repositories as a table with data and three interactive visualizations where end-users can explore data. An additional part of this subsection is to cross compare two or three repositories for further comparison and exploration.

The second functional requirement is to allow administrative management capabilities to certain end-users. This has been implemented defaultly through Django administration functionality. If we alter certain parts of the portal, we can access and change permissions of data and users by the admins. This functional requirement also has four additional subsections.

The first subsection allows the admins to view the database as a collection of users and mined repositories, the second subsection allows the pausing and restarting of mined repositories in the form of blacklisting repositories by an admin. The third subsection is the configuration of database updates, this has been done through frameworks such as Celery and RabbitMQ to automatically update repositories on a daily basis, instead of manual updates by the admins. The final subsection is the granting of permissions from admins to other users. Now that the functional requirements are summarized, the non-functional requirements will be explained.

Non-Functional Requirements

Non-functional requirements are stated as a requirement that can be used to judge the system as a whole instead of specific behaviors. Our non-functional requirements ensure that our system will remain operative and useful to our sponsor and end-users. The three main non-functional requirements that we have obtained are response times of data within one minute, usability of the application, and scalability.

Our first non-functional requirement allows a quick response from our application to display the necessary data and graphics to our end-users. We found this non-functional requirement as a core part of the whole application because, if the data is not displayed to end-users in a timely manner, then they may become uninterested and leave the site. To solve this, we have functionality to update our visualizations every thirty seconds so that they can be loaded quickly.

The second non-functional requirement is usability. To create an application to help newcomers into open-source software, our application must be usable if it is an end-user’s first time on the application, or hundredth time. To solve this, we have created a warm, friendly, and simplistic user interface to limit end-user confusion on navigation. Our user interface comes in large images and text to make the layout simpler for end-users to navigate, along with a central navigation bar to locate our pages in the application.

The third and final non-functional requirement is scalability. Our client expects, as time goes on, that more repositories will be mined from end-users, therefore, increase the size of our database and take up more resources on our server. Our solution to this is through our hosting service, Digital Ocean, which provides scalable solutions to our server to handle more mining requests and accept more tasks from end-users.

Now that we have explored the requirements of our application, we will move forward and explain the architecture and implementation of our final product.

Architecture and Implementation

Throughout the life-cycle of Git-OSS-Um's development, the team had to learn a great deal about several technologies to ensure that everything worked properly. Within this section we will cover in-depth the architecture of the product in order to paint a general picture of how the software works regardless of the changes it may experience in the years to come.

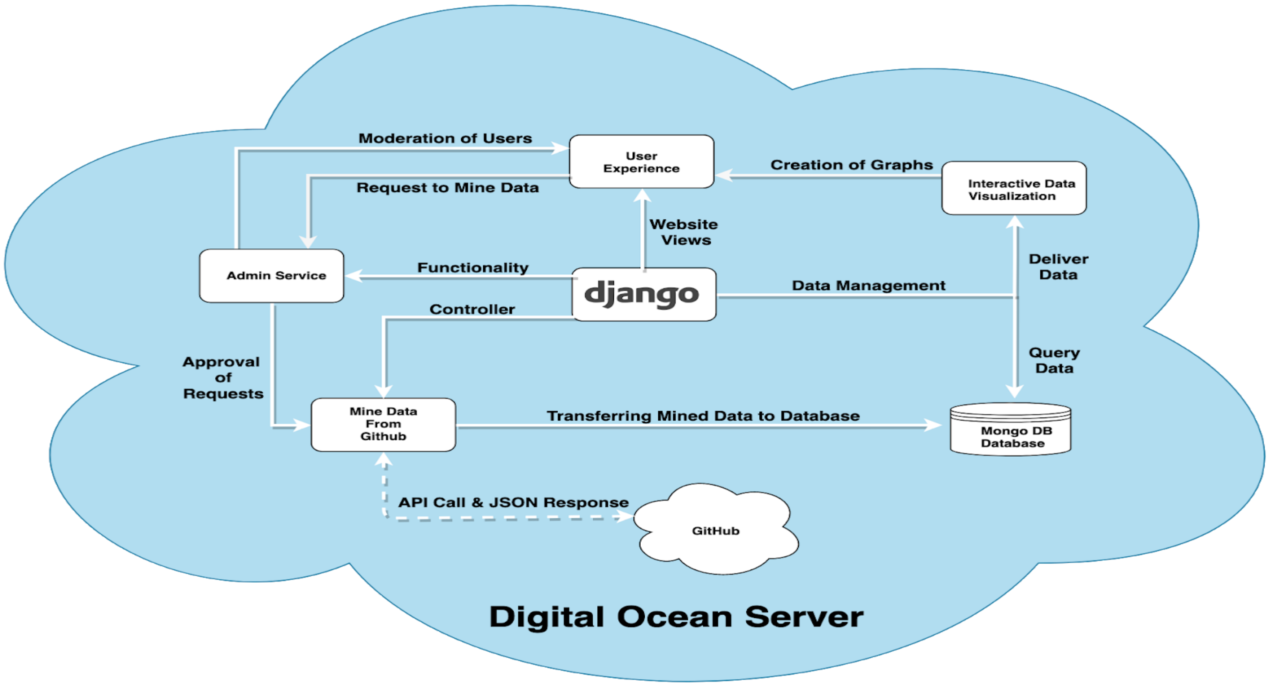


Figure 1: Architectural Overview of System Components

Starting off with Figure 1, we will describe the high level architecture of our system, and the jobs which it will perform. At the center of the diagram is Django, this is because Django is the framework the team had chosen to develop the dynamic web application. Another note here: Git-OSS-Um is hosted on a Linux-based Digital Ocean

Server (Ubuntu 18.04 to be specific). The core experience of the web application comes from Django, all HTML, CSS, and form interaction is controlled by Django. The user will have access to the website, may create an account and submit requests as covered within the User Manual we have written, and the request will be forwarded to the administrator. The administrator will make a decision to either approve or deny the request. If the request is approved, the system's mining script will kick off and make requests to GitHub's API and store all of that JSON information that is returned within our MongoDB database. From there, the system will process that information from MongoDB and generate interactive graphics will Plotly. After this, the cycle continues from the beginning. With this high-level overview introduced, we will take a closer look at each of the components in order to understand the finer details of the software.

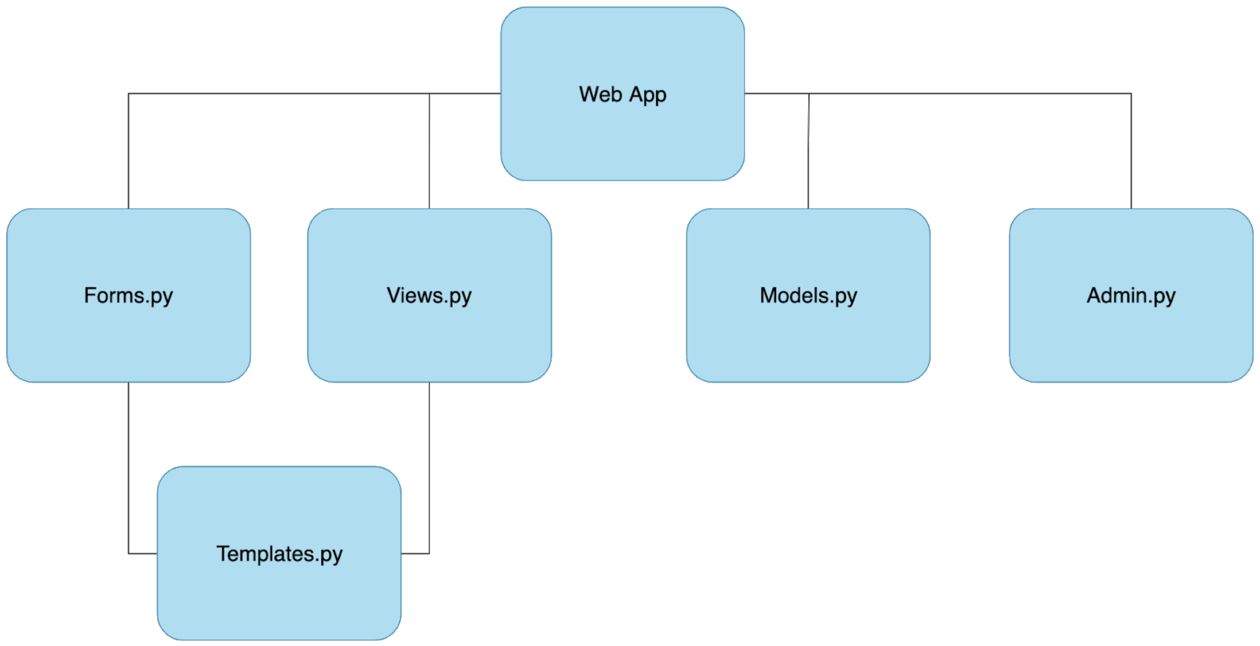
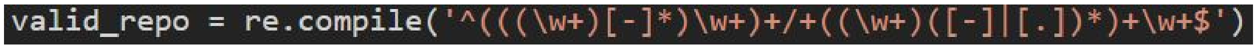


Figure 2: Software Architecture with Django

Within Figure 2, one can see that the Web App can be broken down into several core Python files. The most important files that control the user experience are *Forms.py, Views.py* and the *Templates* folder. It is the responsibility of the Views.py file to contain the logic for each web page and what context will be passed to the corresponding template file. The Forms.py file contains all of the logic for requesting a repository to be mined, comparing repositories, and the filter system the user uses to search for repositories. We will quickly discuss these features below:

- **Submitting a Mining Request:** When a user submits a request to mine data from GitHub several things happen. Primarily, the string which they typed is cleaned of any potential SQL/HTML/CSS/Javascript code to protect our site, and is piped through a filters to ensure the repository meets our standards. The filters are summarized in the following bullet points.



```
valid_repo = re.compile('^(((\w+)[-]*)\w+)+/+((\w+)([-]|[.])*)+\w+$')
```

Figure 3: Regular Expression to Validate Repository Names

- The repository will first be checked against the regular expression depicted in Figure 3 to ensure a repository is written out as a valid GitHub repository name. To summarize, the regular includes all words followed by a dash ending with a word, followed by a forward slash, followed by any number of words, then dashes or dots, and ending in a word.
- A request will then be made to the GitHub API searching for the string that was given to make sure such a repository actually exists on GitHub.

- If all checks have been passed thus far, the form will check to see if the repository has already been mined.
- If the repository has not already been mined, the form will check to see if the repository has already been requested.
- If the repository is in the process of being mined, the system will then check to see if this is true.
- If the repository has been blacklisted by the administrator, this repository will then be denied.
- Lastly, the form will check to make sure the repository being requested has at least one pull request.
    - A list of all violations will be collected and passed raised as a Validation Error and displayed properly on the form in red letters.

- **Viewing a repository:** After the administrator approves a requested repository for mining (see “Approval/Denial Flow” Section) the user has the ability to view various information about the given repository and any other repository within our database. The list of repositories is generated within the Views.py file by querying the SQL database and passing these names and images as context to the repos.html page for the user to interact with. When the user selects a repository for viewing there are several different components which they will see on the following web page. These include an image of the repository, a table containing small statistics about this repository, a bar chart displaying the pull request distribution of the project, a line chart depicting the number of unique individuals contributing per month since the project’s inception, and a final line chart displaying the number of pull requests per month since the project’s inception broken down by category (open, closed-merged, and closed-unmerged). Note: Within the visualizations.py file, all of the necessary information for obtaining tabular information takes place. To see a more in-depth description of how these statistics were calculated, see the “Data Pre-Processing” subsection of the “Admin Approval/Denial Flow.” The following are basic descriptions of how the information is being passed around the system:

- **Repository Information Table:** The table itself is written within the mined_repo_display.html file using Jinja2 context tags. Each of these tags are denoted in the following way: “{{ This is a Variable Name}},” and {% This is a Conditional Statement %}. Within visualizations.py information is collected from our SQL database that was created during the data pre-processing phase, post-mining of a repository and passed as a Python dictionary as context to this Jinja2 context within the template file.
- **Interactive Visualizations:** For all interactive visualizations, the system makes a simple call to our SQL database to obtain the HTML, CSS, and Javascript code that builds the corresponding chart which the user will see. This was a necessary design choice to prevent the user from having to wait lengthy period of time if we were to generate these graphs on the fly every time a user wanted to visualize a repository. Some repositories contain thousands of pull requests, and the need to recalculate all

necessary statistics on the fly would put unnecessary strain on the CPU and force the user to wait longer to view a given repository. Because of this design choice, the user will never be expected to wait more than a second or two to view any and all repositories within our system.

- **Comparing Repositories:** If a user would like to compare repositories, a few things take place within Views.py as well as the corresponding templates (repos.html, mined_repos_display_2.html, & mined_repos_display_3.html). The checkboxes the user clicks on are written directly as an HTML form within the “repos” div in the “repos.html” page. All repository names and images are passed to this HTML page via context (the repository image is {{ value.1 }} while the repository name itself is passed as {{ value.0 }}. Once the user presses the “Compare Repos” button, the logic within Views.py takes over. Given it is a POST request, the form will verify that the number of checkboxes is between 2 and 3, sending a Javascript alert if this is not true. In the event that these are correct, an HttpResponseRedirect is returned with the proper url. The url for comparing repositories is quite simple:
    - Comparing two repositories: gitossum.com/repos/compare/<owner1>&<repo1>&<owner2>&<repo2>/.
    - Comparing three repositories: gitossum.com/repos/compare/<owner1>&<repo1>&<owner2>&<repo2>&<owne32>&<repo3>/.

        - Once this redirect goes through the generation of the comparison table takes place within the visualizations.py file and is passed via context to “mined_repos_display_2.html” and “mined_repos_display_3.html” in the same fashion that the one-way table is generated.

- **Filtering Repositories:** Because of the complex nature of filtering repositories, the feature was implemented as a form within the Forms.py file. There are five filters the user is able to manipulate within this form, the mechanics of which are written within the filters.py file:
    - Search Box: This filter utilizes the search query as a regular expression to parse repository names within the repos collection in MongoDB. All repositories that have a name that matches the regular expression will be returned in a set.
    - Language Checkboxes: This filter is unique as it is dynamic. Every time someone visits the Mined Repos page, a list of all of the languages is passed to the form to auto populate the checkbox fields the user will interact with. This was a design choice, as forms typically consist of static fields, however this information needs to be easily altered in an automated form. In order to obtain this information from MongoDB, we made use of Mongo’s built in aggregation function to not only return each language within our database, but to also count the number of occurrences so the

user may see this information on the front end. Therefore what is returned is a tuple containing the language and its corresponding number of occurrences within the NoSQL database.

- Minimum Pull Requests Textbox: This filter option makes use of the aggregate function within MongoDB to group every repository with its corresponding number of pull requests and return a set of all repos whose pull request count is less than a specified value.
- Maximum Pull Requests Textbox: This filter option makes use of the aggregate function within MongoDB to group every repository with its corresponding number of pull requests and return a set of all repos whose pull request count is greater than a specified value.
- Has Wiki Page Checkbox: This filter makes use of the aggregate function built into MongoDB to group all of the repositories in our database with their corresponding value for “has_wiki.” These repositories are then returned as a set.

Once all of these filters have been applied the backend will have a list of sets (one set of repositories per filter). This list of sets is then passed to one final function whose sole purpose is to make use of a reduce and lambda function within Python to obtain the intersections of all sets. These intersections represent the application of all filters selected by the user, and are returned as a simple list. This list of repositories is then passed via context to the “repos.html” template where they will be rendered for the user to interact with.

With an understanding of the mechanics behind the user experience, we will now move forward and discuss the mechanics behind the Administrator experience. Figure 4 depicts the flow of the administration process on a base level.

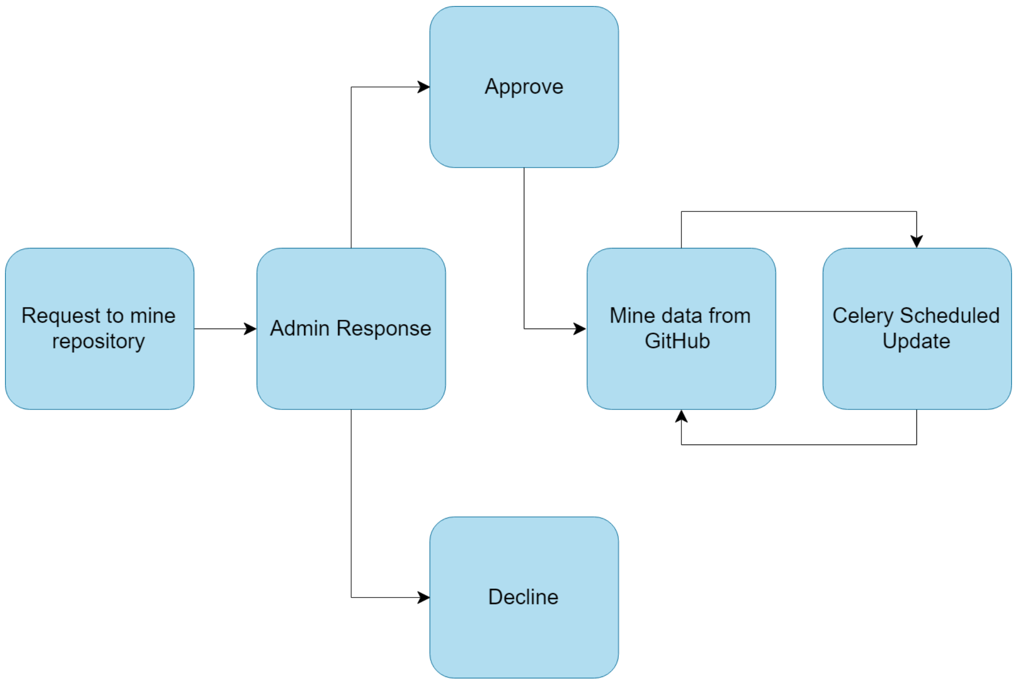


Figure 4: Data Mining Flow and Administration Process

The administrator experience can be broken down into four top level components: Asynchronous Collection with Celery & RabbitMQ, Storing Information within MongoDB, Data Pre-Processing/Data Visualization, and Scheduling Tasks.

- **Celery and RabbitMQ for Asynchronous Collection:** Upon the acceptance of a repository by the administrator for mining so begins the use of Celery and RabbitMQ for the mining process. We made use of these technologies as they are of great value when it comes to accomplishing "tasks" asynchronously. All asynchronous tasks are written within the "tasks.py" file, and the initial setup of

Celery as it pertains to Django is written within the “celery.py” and the “settings/base.py” files. When the selected repositories are approved, Celery will take over the mining process. This mining process can be quite complex when viewed altogether, so we will break it up into smaller, more easily digestible chunks:

- Celery accepts the “mine_data_asynchronously” task as its primary function.
- A request is made to GitHub to obtain a paginated list of all of the repository’s pull requests, and this list is then “batchified.” All mechanics for this batching process is written within the “/mining_scripts/batchify.py” file. Essentially what takes place is batching of pull requests into predefined chunks, specified as a constant within batchify.py, and these batches are kept track of within a collection in MongoDB (see the “Storing Information in MongoDB” subsection). For example, if a repository has 23,432 pull requests with a corresponding batch size of 1,000 pull requests, we would need to process 24 total batches (23 batches of size 1,000 and 1 batch of size 432).
- Once the pull requests have been batched, these batches are “initialized” within MongoDB. This means a collection for the repository we are mining is created to keep track of the batch size the was defined for this batch, the total number of batches needing to be processed, the total number of

    successfully processed batches, and the total number of attempted batches.

    - Once initialized, the backend will iterate over every batch, and send them into the RabbitMQ queueing system where they will wait for their turn to be processed.
    - For each batch being processed, the system will iterate over every pull request and make requests to GitHub to obtain the raw JSON file which describes the given pull request through the use of the PyGitHub open source library, and will be stored within the NoSQL database (MongoDB).
    - As this iteration continues, the system will increment the stored value of attempted and completed batches.
    - When the number of attempted batches is equal to the total number of batches, this means the repository is ready to be “visualized” (see “Data Pre-Processing”).
        - This process happens for every repository that was approved by the administrator.

- **Storing Information in MongoDB:** All storage of mined JSON files will be accomplished through the use of the PyMongo open source library. The JSON files which we mine contain many fields that can be accessed for our systems specific needs. Figure 5 demonstrates the two primary collections used to store mined information from GitHub.

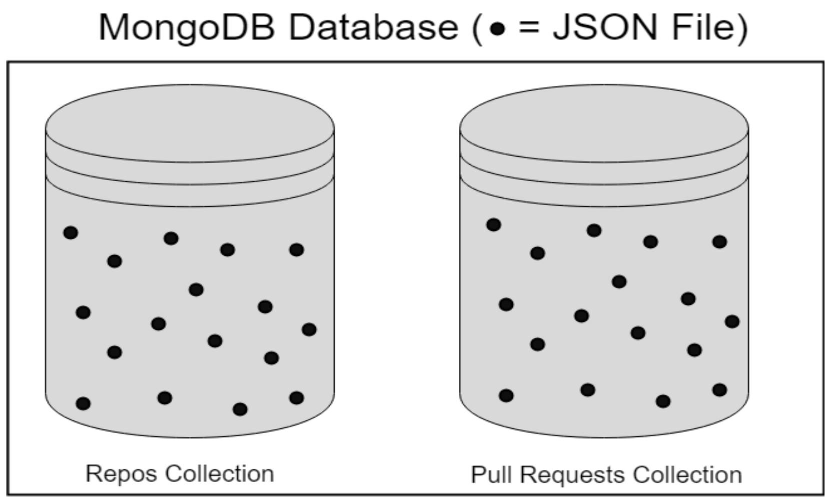
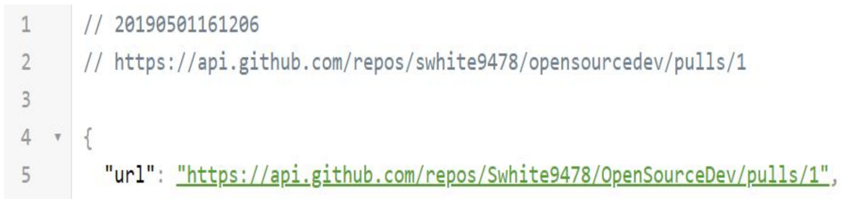


Figure 5: MongoDB Database Collections

The MongoDB database is comprised of 3 total collections:

- Repos: The repos collection contains all of the "landing page" JSON files for every repository that Git-OSS-Um mines (within MongoDB the title of this collection is "repos"). This collection is used to locate information regarding a specific repository (the date it was created, last updated, its license, etc.). Our primary means for identifying what JSON file we want to look at within the repos collection is by using PyMongo to find a JSON file whose "full_name" attribute contains a matching regular expression based on the name of the repository we would like to find. What is returned is the full JSON file for that given repository.
- Pull Requests: This collection is responsible for storing all pull requests that Git-OSS-Um will mine (within MongoDB the title of this collection is "pullRequests" written in camelcase). Because this collection will contain thousands of pull requests and inserting new JSON files is easy, we need

to have a unique way to filter out all pull request JSON files belonging to a repository of our choosing. To do this, the system looks specifically at the "url" attribute of the pull requests and locates pull requests whose url attribute contains the full name of the repository. This is done through the use of regular expressions. For example, if I wanted to locate all of the pull requests for the repository "Swhite9478/OpenSourceDev", I would look at all pull request JSON "url" attributes for "Swhite9478/OpenSourceDev." Figure 6 depicts what a full pull request url looks like within our MongoDB database.



```
// 20190501161206
// https://api.github.com/repos/swhite9478/opensourcedev/pulls/1

{
  "url": "https://api.github.com/repos/Swhite9478/OpenSourceDev/pulls/1",
```

Figure 6: Pull Request URL in a MongoDB Collection

- Pull Batches: As discussed within the previous section ("Celery and RabbitMQ for Asynchronous Collection"), we make use of this collection to keep track of the pull request batches we are mining (within MongoDB the title of this collection is "pullBatches" written in camel case). This collection is simply used to track the progress of the repositories we are mining. We know that once the number of attempted batches = the total number of pull request batches = the total number of processed pull requests batches, that our system is ready to "visualize" the repository.

- **Data Pre-Processing & Visualization:** Recall, once a repository has finished mining, the total number of attempted batches = total number of completed batches = total number of batches (see “Celery and RabbitMQ for Asynchronous Collection”). Through the use of Celery, Git-OSS-Um will check every 30 seconds to see if there are any newly completed batches (see “Scheduling Tasks”). At this point, we will cover the two necessary steps when it comes to obtaining the information that the end user will see after we have mined all of a repository’s data: pre-processing and visualization.
- **Data Pre-Processing:** Before being seen by the end user, every repository will be “pre processed”. By this, we mean that Git-OSS-Um will locate all relevant JSON files, make use of Pandas and Numpy to manipulate the data and pass this information to Plotly which will handle the graphing. Git-OSS-Um handles pre-processing within the “visualizationModelExtraction.py” file. The system makes use of PyMongo to query all pull requests belonging to a specified repository and pulls out each variable of interest. Once this information is completely gathered we store these values within our SQL database for quick extraction in our visualization phase.
- **Visualization:** The major component of visualization is preparing the interactive graphics for the user to see. In order to accomplish this, we make use of Numpy and Pandas to group dates of pull request based on month and pull request category, then take these data points and use plotly to handle all the Javascript,

HTML, and CSS for the graphics. The raw HTML, Javascript, and CSS is then stored in the SQL database for future use. This is a design choice because regardless of the number of pull requests that comprise a repository, the system will be able to make a simple query to grab the already generated graph instead of constructing the graph on the fly with potentially thousands of pull requests. This increases the speed of the application substantially. The goal was to do the heavy lifting only once, then make the information quickly accessible by our system.

- **Scheduling Tasks:** All scheduling of tasks is handled using Celery and RabbitMQ. As mentioned previously, all tasks are written within the tasks.py file. In order to facilitate scheduling future tasks, Django-Celery is an installed application for this django project, allowing the admin to quickly and efficiently schedule a future task through an intuitive GUI. At this point, two major tasks are scheduled on the server:
    - Visualize Repo: This Python task checks every 30 seconds for new repos that are ready to be visualized. In order to accomplish this, the script checks to see if any repository names are within the repos collection of MongoDB that are not already documented within the SQL database. If it finds a repository that fits this criteria, all of the pre-processing (described in the section above) takes place and the repository will be seen by the user.

- Update All Repos: This task is specified by the admin and at this point all repositories are updated every 12 hours (Midnight and Noon). This task will make use of PyGitHub to make requests to the GitHub API for every repository within our database, calculate the number of new pull requests that we have not yet stored within MongoDB, and collect only these new pull requests. This prevents the system from simply recollecting all of the information it has already gathered. On top of collecting new pull requests, all of the graphics and tables are updated as well and stored within the SQL database as discussed above.

As you can see, Git-OSS-Um is comprised of a lot of backend technologies. Throughout its life cycle, the product has undergone major changes to get to the state in which you are now familiar. To begin, the team first wanted to make the product with a separate front and backend but ultimately decided to combine these into one simple web app. This was for a number of reasons, including our limited knowledge of how to build a REST API, React, and the overall idea of needing to keep track of two repositories. With the current solution we were able to work better as a team and have all of the files in one location. As discussed above the mining system relies on batching pull requests to determine when to visualize a repository. This design choice was chosen over basing completion off of the total number of pull requests because some pull requests may be inaccessible, therefore the total number of mined pull requests would never equal the total number of possible pull requests. Also, this batch system allows Celery to more accurately swap out processes during mining, so the server's RAM usage will not be overloaded. If all of the pull requests were mined in one fell swoop, more and more RAM will be utilized until the point where there is none left and the program crashes in the middle of it mining data. Lastly, the team almost used D3.js to visualize our data instead of Plotly. The more that we looked into this technology, the less it made sense to use it if we were moving away from having a separate front and backend. The syntax for creating these plots in Javascript was also way more complicated than the team was willing to put up while we had much more to take care of. All of this technical knowledge would not be useful if we did not test our software. Within the following section we will take a closer look at how Git-OSS-Um was tested.

Testing

Below are explanations of how we will be testing our product. We have three main forms of testing: unit tests where we will examine individual components of our software to see if they work as intended, integration testing where we check if all of our components work together as intended, and usability testing, where our users will test the product and provide us with feedback. A more detailed explanation of our testing methods are available in our Software Testing Plan document located on our team website.

Unit Testing

Our unit testing will examine individual components of our software with correct and incorrect inputs. Our unit testing mainly consists of our backend technologies so that we can focus on the correct output for our web application.

Our unit tests have been organized into “test suites”, where individual components are grouped together to test a main piece of functionality. The reasoning behind this is to have the necessary unit tests and to organize these units into a class so that the suite can be run to test all the individual units. For example, the RepoLandingPageTestSuite class suite is responsible for testing the mining, storing, access of data from a GitHub repository. This suite would then execute the tests, and if successful, would mine a repository that has no record in our MongoDB database and exists on GitHub. Additional functionality would include the location of a JSON file and the attributes from that file along with error checking if a repository is already mined and to check if there are multiple occurrences of a single repository in our database.

Despite our unit testing covering a large amount of our software, it cannot completely cover our application as a whole. Other frameworks such as Celery, our asynchronous task manager and automated repository updates are handled separately by the framework or the end user. Our unit testing can only go so far, so we have implemented another layer of testing for integrating all of our components so that all of the functionality works together as intended.

Integration Testing

Our unit tests may cover individual components of our software, but our layer of integration testing would ensure that the front-end and back-end software work as intended. Our integration testing will focus on the compatibility of back-end information to the front-end so that users can successfully navigate and perform tasks on our application.

We have implemented a black-box form of testing, where testers have no knowledge of the code being used but follow a path to make sure the correct output is displayed or to see if the application is working as intended. The form of black-box testing would help us flush out any issues that we may have while integrating all of our components and suites together. We have also performed black box testing with a “bottom-up” flow of testing, where we track the flow of data from the lowest point to the highest point possible to flush out any issues with integration.

An example of an integration workflow would be the process of mining a repository. By tracking how a mining request flows through our application, we can see if there are any flaws of integration. We can track the files that have specific functionality, starting from the beginning of the process to the end to see if the flow of data is correct. For example, a request is made for mining, if the request made it to the admin portal and is accepted by an administrator, it would mine the data until complete, once complete the mined repository would show up on our front-end “Mined Repos” page, and when clicked, would show all necessary information. This example would be a successful flow of data from the lowest level to the highest. If the flow of data is interrupted between any of the steps, we would know where the failed integration is located by the premature ending of the flow of data.

Our integration testing is one step above our unit testing and checks that the flow of data is complete and correct as outlined in the example above. To continue our testing plan, we will move forward to our usability testing where actual end users will test the application and provide us with feedback on our product.

Usability Testing

From the time we had our product live on a server, we have had our sponsor as well as other users using our application. This gave us about a month and a half of usability testing by default, but it wasn't structured. To further our usability testing, we submitted to the International Review Board (IRB) a research study proposal to perform a series of experiments where one group of users will perform a set of tasks using our application and another group will be using GitHub to perform the same tasks. The results from both such as time per task and number of questions asked per task will be compared. The users will then be given a survey after the completion of all tasks and asked to give feedback on our application's usability and user interface design.

Due to the nature of the lengthy review process that research proposals are required to go through, time ran out on us before the IRB was able to accept the research proposal. This made it impossible for us to conduct our formal study of our application. That being the case, we decided to have a few colleagues use our application and give us feedback on the user interface and usability to give us a little extra user input.

Since our research study was not accepted in time for it to be conducted by us personally, the study will take place next semester with Dr. Steinmacher and his new research students that he brings on next semester, so our efforts for the study are not going to waste, the study is just being put on hold for the time being.

Project Timeline

Over the past year, we have focused on gathering requirements from our sponsor and developing our product. We have split requirements gathering and development into two separate semesters to align with the capstone schedule. Below are summaries of each schedule.

Semester One Schedule

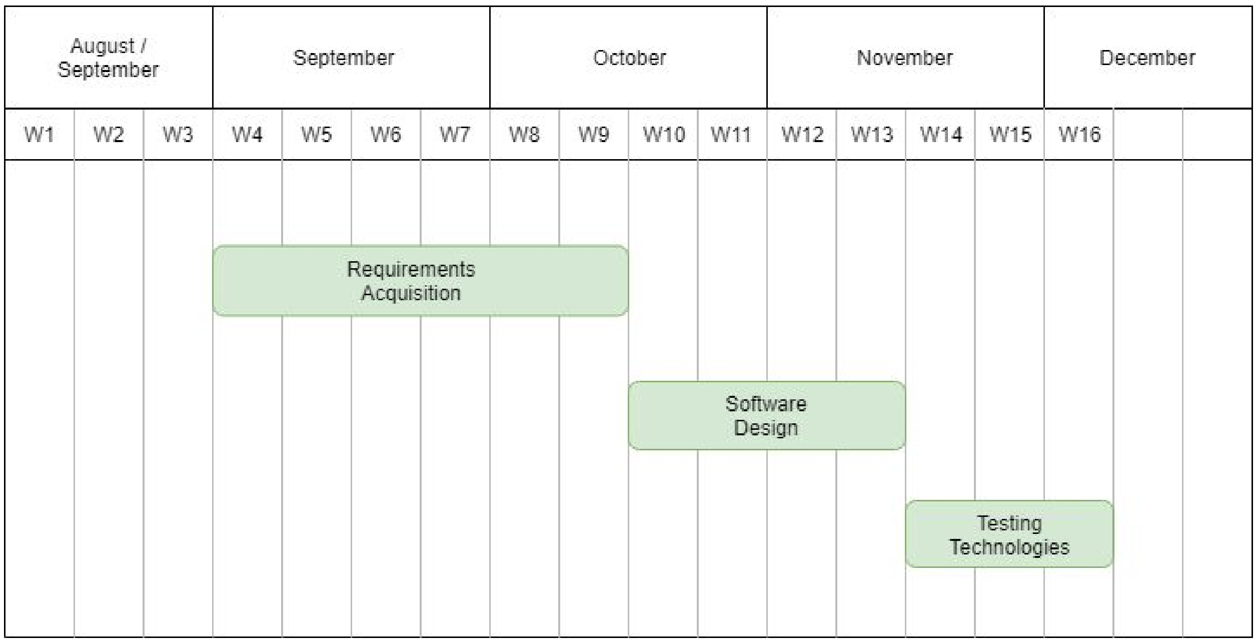


Figure 7: Semester One Schedule with Milestones

Above in Figure 7, is a basic Gantt chart of the first semester schedule. As briefly stated, the first semester was scheduled to acquire requirements from our sponsor. As the semester move forward, we eventually begin to design our software based on the requirements acquired.

When we designed our software, we would describe our technologies we would use, and implement the technologies into a functioning prototype. The gathered frameworks during this time would be changed at the beginning of the second semester. Overall, the first semester outlined our requirements, layed out the initial design and technologies of our product, and ended with a functional prototype that highlights our requirements, design and proof of concept.

Semester Two Schedule

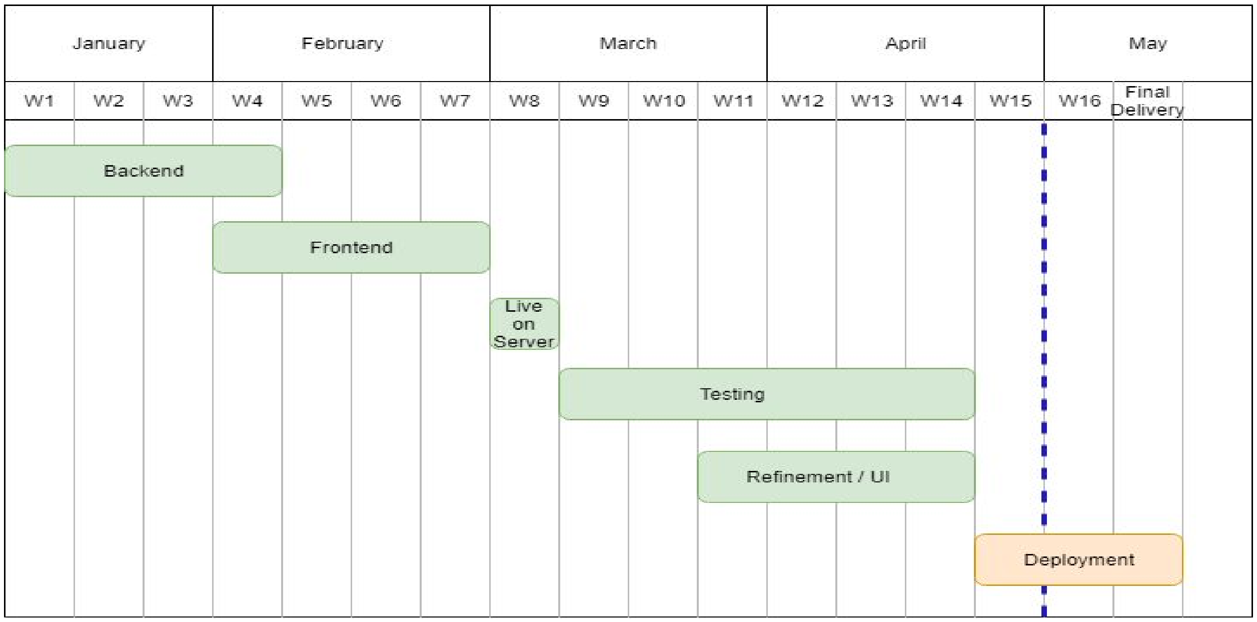


Figure 8: Semester Two Schedule with Milestones

We are currently near the end of semester two, denoted by the dashed, blue line in Figure 8. As of now, we are in the deployment phase for our product, the software is functional and complete.

At the beginning of the second semester, as stated above, we had a shift of technologies so that we can better accommodate our product and requirements. With this change, we begin to develop our backend technologies such as the mining and storing of that information. When this is complete at the beginning of February, we set to work on the front-end part of the application to create a friendly, warm, and easy to use interface for our users. When the back-end and front-end were complete, we pushed the application to our linux-based server at the beginning of March. Throughout most of March and April, we begin the testing of our product and the refinement of flaws that we have found with testing and user interface improvements. Finally, the last milestone of the second semester is the deployment of our product to our sponsor along with the necessary documentation.

Future Work

Although we love Git-OSS-Um in its current state, we do have some ideas that could be implemented to make it even more “OSS-um”-er! If possible, the team would like to see the separation of the front and backend as we originally planned because this would allow us to make use of a heavier frontend framework such as Angular or Node and take full advantage of the frontend tools these frameworks provide. There is also potential for Git-OSS-Um to expand the data that it collects and analyzes. At the moment the system only collects pull requests and repository landing pages, and this could be expanded to include commits, comments, and users. The purpose for this would be able to draw graphs connecting users to other repositories, allowing someone to see a visual representation of what other repositories people are contributing to that have also contributed to the repository you are currently viewing. Machine learning could be utilized to analyze the comments users receive on pull requests (per comment: 0 = neutral language, 1 = positive language, -1 = negative language). This would allow our users to get an even greater understanding of the culture of a repository. Since commits are “smaller” than pull requests (i.e. several commits can be within one pull request) they could offer us even greater avenues to explore.

Conclusion

Our project’s main goal is to help newcomers in open-source software find a repository that they would want to contribute to. Our client, Dr. Igor Steinmacher tasked us with building a web application that mines information from repositories on GitHub and to display that information in a meaningful way, along with automatic updates so that admins will not have to do it themselves. Dr. Steinmacher plans to use our application as a tool for his continued research into open-source projects.

In order to fulfill the needs for Dr. Steinmacher’s research, we have created a web application that assists newcomers in open-source software to find a project that they would want to contribute to. Some of our best features from our application is the display of simple and easy to read statistics and visualizations. This data will assist Dr. Steinmacher to further his research and find trends in open-source software.

Our application will assist Dr. Steinmacher's research and decrease the time for finding trends in repositories. As an admin, Dr. Steinmacher can make the necessary changes for mining repositories and permissions for users so that they too can assist him on research. As explained above in "Future Work", the additions of new functionality such as sentiment analysis for comments and repository projection with machine learning algorithms are future steps for our application. These new changes could greatly impact the future of the application, as end-users can see a prediction of where a repository would go in the future.

The overall project in the past year went relatively smooth. Most of the assignments and tasks were split evenly between the three of us and were done on time. Our weekly, iterative process of requirements and development has made our product progress at an optimal pace. The organization of the team, tasks, and weekly meetings made for a smooth experience both for the project and the class. Overall, the Capstone class and the project flow were great and we have succeeded in creating an optimal product for our client over the year.

Glossary

- Asynchronous:
    - Where data can be transferred at irregular intervals, instead of a steady stream of data.

- Celery:
    - Open-source, asynchronous task queue and scheduling. Used to schedule tasks for mining purposes.
- CSS:
    - Cascading Style Sheets, alters the presentation of web pages. Styles web pages to look “pretty”.
- Data Mining:
    - The retrieval and examination of data to find useful information.
- Digital Ocean:
    - Cloud service provider. Our web application is hosted through a linux-based server through Digital Ocean.
- Django:
    - An open-source, Python based web framework. Uses the model-view-controller architecture.
- GitHub:
    - Hosting service for open-source projects. Github is the platform where we are mining data.
- Gunicorn (for server):
    - Python web server gateway interface that is lightweight on host server.
- HTML:
    - Hypertext Markup Language, the standard language for creating content for web pages.

- HTTPS (within our future work):
    - Hypertext Transfer Protocol Secure, an extension of Hypertext Transfer Protocol. Adds additional layers of security for the web application.
- Integration Testing:
    - Where individual testing is combined into groups so that there are no faults between the integrated units.
- Jinja2:
    - Templating language for Python. Modelled after Django templates, designer friendly as well for easy implementation. Allows for dynamic elements in templates.
- JSON:
    - JavaScript Object Notation, data files that is easy for humans to read and for computers to parse. Similar to dictionaries in Python.
- Model-View-Controller:
    - An architecture pattern that divides a user interface to allow code reuse and parallelization.
- MongoDB/NoSQL:
    - Document-oriented, noSQL database. Used to store data that is mined from GitHub.
- Nginx (reverse proxy for server):
    - A reverse proxy for our server. Retrieves resources from other servers and is returned to us.

- Plotly:
    - Open-source graphing library for Python. Used to generate our interactive visualizations from mined data.
- PyGitHub:
    - Open-source library that handles the GitHub API for Python applications, such as our product. Makes mining of data more simple.
- PyMongo:
    - Open-source library to work with MongoDB using Python.
- Python:
    - High-level programming language, is used to create web applications in Django.
- RabbitMQ:
    - Open-source broker software. Used with Celery to organize the communication between tasks and scheduling.
- Regular Expression:
    - Also called a regex, which is a sequence of characters used for searching and as a search pattern.
- Repository:
    - Where the data of open-source projects are stored and managed. The information we mine is located inside repositories.
- Root (Linux):

    - The default account that has access to all available commands and files on a Linux or Unix operating system.
- SMTP (Simple Mail Transfer Protocol for mining email service):
    - Used for our email service, a communication protocol for electronic mail transmission.
- Template:
    - An HTML file that uses Django template language. This creates the visible web pages seen in Django web applications.
- Ubuntu 18.04:
    - Open-source, long term service operating system from the Linux distribution.
- UFW (Firewall for our server):
    - Uncomplicated firewall, a default firewall in Ubuntu. Designed to be easy to use and manages netfilters.
- Unit Testing:
    - Where individual testing of components are validated. Makes sure the individual component is working as intended
- Usability Testing:
    - Testing the product with the users. Users give direct feedback of how the system works.

Appendix A: Development Environment and Toolchain

Within the following section, we will lay out several need-to-know pieces of information when it comes to developing for Git-OSS-Um. As it pertains to the specific technologies that will be discussed, links will be provided for the reader's own personal development.

- **Hardware:** The most basic of components to know about Git-OSS-Um is that it is run on a linux based Ubuntu 18.04 server. This server has 4 GB of ram and 64 GB of memory, and makes use of 2 CPUs. This software can be developed using a Windows Machine, Macintosh, or the Linux operating system. (For reference we specifically used DigitalOcean's One-Click MongoDB Droplet). There are no other hardware requirements in order to begin developing software. It is however, easiest to develop locally on an ubuntu machine.
- **Toolchain:** Once you have the appropriate hardware, there are several pieces of software that are 100% necessary to be able to develop software for Git-OSS-Um. For readability purposes, these software components will be written out in list form below:
    - **Editor:** This is really more of an aesthetic choice for the developer, but for documentation purposes let it be known that the original developers of this software made use of Visual Studio Code (https://code.visualstudio.com/download) to code Git-OSS-Um. Some

other recommendations for text editors are Atom, Notepad++, or Sublime. If you want to get really challenging feel free to use Vim, or Emacs to edit directly via a console window.

- **Django 1.11:** Django is the heart of Git-OSS-Um because it integrates with all other technologies that are listed. It is what runs all of the Python files and makes the web application work properly.
- **Python 3.6:** The team made use of different versions of Python 3 (>= 3.6.0 but < 3.7) in conjunction with our virtual environment to run Django. The reason we used these versions of Python is because they support f-strings, which are used frequently throughout the software. We avoided Python 2 because it is becoming less likely to be supported in the near future.
- **Virtualenv 15.1.0:** This is a package that needs to be pip-installed on your machine. Its main use is to create a virtual environment that will encapsulate all of the pip-dependencies that the project needs to run.
- **Requirements.txt:** This text file contains all of the necessary dependencies for being able to run Git-OSS-Um. It will be covered more in depth within the setup subsection.
- **MongoDB 4.0.2:** MongoDB is the NoSQL database that Git-OSS-Um uses to store and access all of the information which we mine from GitHub. Without this software, the system will not be able to mine or store JSON information.

    - **RabbitMQ:** This is an external broker which is used to queue Celery tasks. It is necessary to be able to use Celery for asynchronous task scheduling.
    - **Supervisord 3.3.1:** This is a package on the linux server that is necessary for easily creating Celery processes. Without it, you will need to go through the process of writing daemon scripts to allow Celery to run in the background.
    - **Nginx:** This is the reverse proxy which the system uses in order to redirect gitossum.com to our server. This is necessary to manage traffic to the website.
    - **Gunicorn:** Gunicorn is used to handle all of the different requests within the application.
- **Setup/Production Flow:** Follow these instructions in order to set up Git-OSS-Um on a new production server and begin development:
    - Obtain a new Digital Ocean MongoDB, Ubuntu 18.04 Droplet at https://marketplace.digitalocean.com/apps/mongodb
    - With a brand new server, you should clone the Git repository for Git-OSS-Um. This can be done by going to https://github.com/swhite9478/git-oss-um. If you do not have access to the repository you may request it from Igor Steinmacher at igor.steinmacher@nau.edu.

- Next follow these instructions to get rabbitmq-server set up on your machine: https://computingforgeeks.com/how-to-install-latest-rabbitmq-server-on-ubuntu-18-04-lts/
- With RabbitMQ installed, you should now setup supervisor with a Celery process for Celery. NOTE: When you are setting up Celery worker conf file, make sure you use the following command: “command=/home/git_oss_um/env/bin/celery -A web_app worker -l info --concurrency=1”. This will ensure that this asynchronous task will not be run concurrently. This is necessary so all pull request batches will finish as expected.
- Pip install virtualenv to be able to set up the virtual environment.
- Change into the Git-OSS-Um directory and create a virtual environment by using the command “virtualenv -p 3 env”. This will setup a virtual environment where we will store Git-OSS-Um’s dependencies.
- Once it is created, activate the virtual environment by typing “source env/bin/activate”. To deactivate the virtual environment all you have to type is “deactivate”.
- Once activated, you will need to install the dependencies for Git-OSS-Um. To do this, within the Git-OSS-Um directory type “pip install -r requirements.txt”. This may take a while, as all necessary requirements are installing.

- Within the mining_scripts folder, create a file called config.py. Within this file you need to name and fill out 3 variables in order for the system to mine data:
    - EMAIL_ADDRESS = "gitossum@gmail.com"
    - EMAIL_PASSWORD = "Django123"
    - GITHUB_TOKEN = *personal github token. Follow the following link to create a github token: https://github.com/settings/tokens*
- Next, you should type the following within the /Git-oSS-Um/web_app/ directory to make sure your copy of Git-OSS-Um has a working SQLite database: “python manage.py makemigrations user_app”, to prepare the migrations, “python manage.py migrate” to migrate the database, and finally “python manage.py createsuperuser” to create your admin account.
- Finally, it is time to setup Gunicorn and Nginx to allow the website to run. Follow the instructions at https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-ubuntu-18-04 to get Nginx, and Gunicorn on your machine. Make sure to substitute the proper directories within this tutorial.
- Congrats! You now have Git-OSS-Um set up on your new digital ocean droplet!
- You may begin making changes to files and see these changes reflected instantly by running “python manage.py runserver”

- As you make changes to files on the server, if you want these changes to take effect on the production server you will have to restart the Gunicorn process using the following command: "sudo systemctl daemon-reload && sudo systemctl restart gunicorn && sudo systemctl status gunicorn".

Thank you for your interest in Git-OSS-Um! We hope this guide has helped you understand our software on a deeper level, and that may contribute to keep this product growing!