Octo-Docs

Final Report

Dr. James Palmer

Garrison Smith
Peter Huettl
Kristopher Moore
Brian Saganey

x _____

5/10/18

Table of Contents

1. Introduction

Octo-Docs is a team that was formed with a goal to improve how software development groups create, edit, and interact with comments in their projects. The members of team Octo-Docs are Garrison Smith, Peter Huettl, Kristopher Moore, and Brian Saganey and we are working on creating a new documentation management system called CrossDoc. The project is sponsored by Dr. James Palmer, who first proposed that commenting is in need of an improvement.

Software development teams currently face the problem that their documentation is highly dependent on their project's codebase. This dependence results in documentation that is hard to view or edit externally, particularly by non-tech savvy individuals. The globalization of software development groups means that not all developers working on a particular codebase speak the same language, and when this is coupled with the fact that software development is one of the biggest modern industries, it becomes apparent why this is an issue. This is why many modern teams have groups dedicated to the localization of the work environment. Employees such as this are often not familiar with current commenting practices and have a hard time combing through the codebase to find the language to change. The longer this comment management time takes, the more time and money a team is using not developing their product.

Octo-Docs and Dr. Palmer aim to fix this with CrossDoc, a commenting system that connects external comment stores to a codebase. By simply referencing external comments, this new comment system will provide a great deal of flexibility and improvability to the standard commenting system. Improvements such as distinct comment categories, an adapting comment history, and user-specific comment modules will not only solve the initial commenting problem but also provide an adaptable canvas to solve future documentation issues. Dr. Palmer, as the Associate Director of Undergraduate Programs at NAU, is particularly interested in this product for its implications for teams and even individuals working on their own projects. As such, this technology works well in an educational environment, and the categorization of comments can be used to help beginners and experienced developers alike.

2. Process Overview

When CrossDoc was first presented to us as a team the first idea was to understand what the purpose of CrossDoc is and who is going to use it. CrossDoc is an improved commenting system used by software developers and technical writers and understanding this allowed us to then start building the project. The first step was outlining what programming language that we plan on using and what communication tools that we were going to use to build this product and work together as a team to do so. The initial plan that we eventually stuck with was having google docs for our documentation drive so that all documents, graphs, and presentations could be worked on collectively and could all be updated in real time. The software was going to be shared in a git repository where we would be able to upload all of our website software, programming software, and some documentation on how to use the product. Then we need to make sure that we kept track on our progress on certain tasks and assignments that were assigned on a team basis and the tool we used there was Trello which is a nice way to organize all of our tasks and split them up evenly throughout the team. This step allowed us to have the table set for the project itself and make sure we were using these tools for organization and management purposes.

The next step in our project was to understand what technologies we were going to use during the process. This included the main website for our documentation, the general programming language that we are going to use in this project and what other technologies the project may need as we continue to build the project. The first step was getting the main programming language which as a group we decided to use python. Python allowed us to have an easy to use installation process for new users and python also allows for a plethora of libraries that we can use when developing our product. For the website, we decided that we would use the basic HTML, CSS, and JavaScript to create this website to allow us to have a website that can be resilient to future programming languages.

When we were going in depth with the product we also found out that there were 4 main text editors that need to coincide with CrossDoc. These text editors were Vim, Emacs, Sublime, and Atom, and each one of these text editors had their own languages that we had to use to implement into CrossDoc itself. These technologies helped us continue the development of CrossDoc and allowed us to do research and better understand what is happening in the backend of our product.

The final step that we did was making sure that the team was organized and tasks were split up evenly as well as making sure everyone understood what was happening every week. The process that helps assist us with this was making sure that everyone had contact information with everyone and that there were weekly meetings that would happen so that we could sit and discuss for a couple of hours what the next week was planning for us. Another idea that happened rarely but was helpful when it did was making sure everyone knew each other's schedule and so that we could have emergency meetings in case the week's tasks got challenging. The idea of keeping our team organized as well as understanding each team members strengths and weaknesses so that when it came to certain development that we could utilize these strengths and help increase the weaknesses to make them better for the future. This process was changed ever so slightly as the year went on but as the year went on each team member was working just as hard as anyone else was and each team member always felt comfortable to come to the team when they would get stuck on a certain part of the development phase. This shows the general overview of our process through this project and shows how we tackled certain tasks throughout the year.

3. Requirements

In order to deliver a product that matches this solution vision, we have developed a collection of functional, non-functional, and environmental requirements. These requirements will act as a guide to ensure that we are providing a product that actually solves the problems presented and demonstrates value to the user. Some of these requirements are higher level abstractions of ideas we intend to implement, while others are low level, actionable items. All of these requirements fall into five key requirement domains.

1. *Implementing core functional data storage capabilities*
2. *Creating an intuitive and easily adoptable system*
3. *Expanding the functional capability of comments and comment storage*
4. *Supporting the scalability of a team environment*
5. *Incorporate powerful tools for experienced users*

Each one of these key requirements is part of how CrossDoc works as a whole and how the system itself will function throughout the user experience. In addition, each one of these domains can be broken down into subcategories of functional requirements and how each functional requirement works with CrossDoc itself.

3.1. Functional Requirements

Domain 1: *Implementing core functional data storage capabilities*

The core guiding principle of this domain is ensuring **CRUD** operations [6]. As such, the four primary requirements for this domain are:

1.a. *The ability to **create** comments*

1.b. *The ability to **read** comments*

1.c. *The ability to **update** comments*

1.d. *The ability to **delete** comments*

Each of these functional requirements is key to CrossDoc's basic operations and serve as the baseline functional assumption made in other requirements. By following CRUD principles, we are able to ensure persistent storage of comments and provide additional basic features that expand on traditional commenting.

Domain 2: *Creating an intuitive and easily adoptable system*

To ensure user adoption of our product, we have a domain of requirements targeted specifically at ensuring that our system is easily adoptable and intuitive. Initial user adoption is one of the largest challenges to any new software product, particularly one that is developer facing. We have identified a *difficult initial installation process* and *lack of documentation* as two of the main contributors to this lack of user adoption. As such, our two high-level functional requirements for this domain directly target these two concerns, while their respective low-level requirements provide distinct, actionable development goals:

- **2.a.** *Providing an easy installation processes*
  - **i.** *Intuitive installation process through package managers*
- **2.b.** *Ensuring the product is debuggable and well documented*
  - **i.** *Develop command specific help text*
  - **ii.** *Provide identical command detection*
  - **iii.** *Graceful error handling and response*

Domain 3: *Expanding the functional capability of comments and comment storage*

Alongside providing a seamless transition into using CrossDoc is distinguishing our product from traditional commenting. We will accomplish this meaningfully expanding the functionality present in traditional commenting systems.

- **3.a.** *Implement comment scoping*
- **3.b.** *Integrate dynamically toggleable comment sets*

3.c. *Provide external commenting capabilities*

- **i.** *Create and insert comments from the command line*
- **ii.** *Modify comments directly from the comment stores*

Furthermore, CrossDoc uses an external wiki storage to help create sets and stores for users. In addition, more experienced software developers can also use this idea to create sets and stores in the command line itself and then later can be accessed and used in the external wiki storage too. Moreover, a user can then separate their comment sets and stores and have them be organized in a fashion so that user can easily toggle through each comment.

Domain 4: *Supporting the scalability of a team environment*

CrossDoc is a product that is targeted at both individual developers, and software development teams. To provide value to these target markets, CrossDoc needs to support the large-scale operations required in a team environment in a way that is flexible enough to not sacrifice the small-scale uses of the product. As such, this domain contains requirements focused on the **scalability** and **portability** of CrossDoc. The two main high-level requirements within this domain directly target the two principles respectively:

4.a. *Flexible and modular comment storage*

- **i.** *Support for multiple distinct comment stores in a project*
- **ii.** *Flexible classification of comment types*
- **iii.** *Comment store hierarchy*

4.b. *Easily share documentation data between users*

- **i.** *Support for wiki comment storage*
- **ii.** *Provide project-specific customizable configuration files*

Domain 5: *Incorporate powerful tools for experienced users*

Because CrossDoc is a software development tool, it is important that the product can be optimized to improve workflow efficiency. This domain focuses specifically on this aspect by outlining requirements that improve the user's interaction with the product. The following functional requirements demonstrate exactly this:

- **5.a.** *Implement command shorthand alternatives*
- **5.b.** *Provide text-editor specific command hotkeys*

The tools that CrossDoc provides allows for experienced and inexperienced users to interact with CrossDoc this could range from creating alternative hotkeys for technical writers as well as having easy to use command lines for experienced developers. For example, when a developer calls **create_comment** they can instead call **cc** this will allow CrossDoc to expand to experienced and nonexperienced developers.

3.2. Non-Functional Requirements

In contrast to the functional requirements, the non-functional requirements are used to structure what CrossDoc will be. To ensure the system is capable of providing all the domain requirements along with the original design goals, it must fulfill the non-functional requirements of Maintainability, Performance, Reliability, Security, and Usability.

Maintainability: *Ensuring future development capabilities*

In order to ensure future development capabilities, we want to implement the idea of **Domain 2**. We want to make sure that if CrossDoc is easy to adopt and easy to use then we can make it easy to maintain as well. In addition, CrossDoc systems will be implemented with detailed source documentation to assure swift comprehension of original coding practices and provide a modified solution when necessary.

Performance: *Facilitating fast and efficient use*

For the CrossDoc system, the performance of the system is important because CrossDocs goal is to be easy to use for users and no user wants something that does not perform well. The goal to make sure that the performance is at its utmost is to make sure that there are plenty of unit testing and plenty of integration testing. We want to make sure that the user's experience with CrossDoc is fast and reliable. In order to do this, we want to make sure there is not a lot of stress on the backend command line program and to make sure that the CrossDoc commands are well responsive from the text editor to the wiki and vice versa.

Reliability: *Creating a stable, functioning system*

The reliability of CrossDoc is key to making sure that users can adopt the system and understand how easy it can be used. We want to make sure that we have reliable error handling and making sure that the command line parser is doing what it needs to do. To make sure that this is defined and works we will do rigorous testing for CrossDoc and make sure that we have all integration testing working and that the main functionality of CrossDoc is working.

Security: *Preserving the security of documentation*

The security of CrossDoc is broken down into two sections which are Authentication and Authorization. In the Authentication layer, the system will check for valid user credentials before allowing access to the respective comment stores. The Authorization layer of the system will only allow users with proper access levels to utilize respective tools. The access levels will consist of: Administrator, Editor, Viewer. These security protocols will allow for users to work in their own team environment and make sure that it supports a scalable system for these end users.

Usability: *Providing an intuitive and easily adoptable system*

In order to create a truly intuitive system, it first must be a system that is inherently designed for usability. The system should be efficient, effective, easy to use, and ultimately have a small learning curve. For CrossDoc to comply with this requirement it must provide users with support documentation such as readmes, operation documentation, and help functionality for all of its application facets and plug-ins. Additionally, CrossDoc must be designed in a user-friendly format, such that the operations associated are intuitive to an individual unfamiliar with the system. Using these metrics we will perform a number of test runs of CrossDoc with volunteer users, to record information about their experience, to determine the overall usability of the system.

3.3. Environmental Requirements

Alongside our functional and nonfunctional requirements, there are also several environmental requirements. These are requirements that were provided by the original problem statement and the several outside software integrations. Due to the sparse nature of these requirements, they span multiple distinct domains. This section will be a brief overview of what the environmental requirements are and how they affect our other requirements. One of the first environmental requirements is integrating for universal systems and making sure that our code runs on all popular text editors. This can be used through the easily adoptable domain level requirement. In addition, we want to make sure that there is a Git integration with the system itself and make sure that software developers can feel comfortable with the feel of CrossDoc and not feel that CrossDoc is a hassle to work with. Finally, another environmental requirement that CrossDoc is going to use is that it should have good documentation for end users. This will allow the final users to be able to upgrade our product to there specific requirements and allow CrossDoc to further expand into the future.

4. Architecture and Implementation

The architecture of this project is to have a user access CrossDoc through a plug-in within their text editor. Then the user will be able to use the plug-in to access the command line interface. This is where they will have access to all the command tools and the comment stores within their project. The user accessing CrossDoc is our way of describing the front-end of our project and the command line interface where the user can access the command tools and the comment stores will be the back-end of this project. Figure 1 illustrates the broad picture of what the architecture will look like at the highest level, however, within each box is a subset of tools and interfaces that the user can use while using our project.

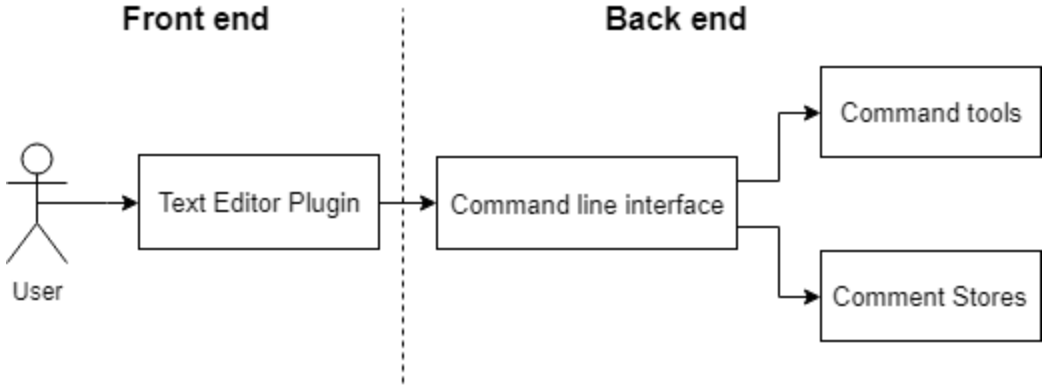

Figure 1: An architectural diagram of the user interaction flow.

The interaction the user will have with CrossDoc is to download a plugin through their text editor. The goal is to make sure that whatever plug-in the user is using it should be a quick and easy process for them to download CrossDoc within the text editor. This is how the front-end of the project will work and once the user has access to CrossDoc, they will be able to use all the functions present in the command line program. Once CrossDoc is downloaded on the text editor, the user will then have access to the back-end of CrossDoc. The purpose of the back-end is to have centralized command line interface that branches off onto the tools and functions that the user will be using.

Once the user has access to the command line interface, they will initialize a project and begin using the CrossDoc repository to create, read, update, and delete comments within their project. This repository will be a general repository where all their creations will go to. In addition, they will have a localized file that they can also update that will be shared with the general repository. Figure 2 demonstrates this connectivity.

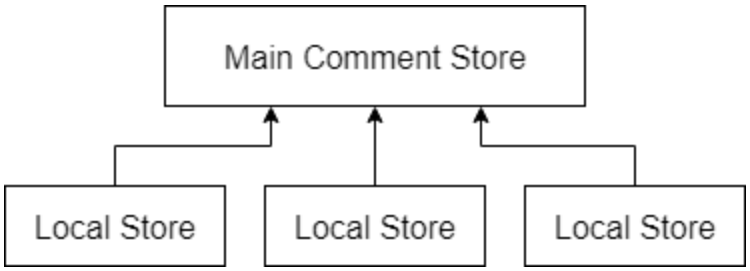

Figure 2: A diagram that outlines comment store inheritance.

Each comment store will also have a set with it too that can categorize where to find what comment for the project. This will allow users to have an organized method so that they are not struggling to find a comment within their project. The command line tools are essentials for the users to have so that they can access these comment stores and with their command line interface and manipulate them into anyone they might need to for the project. However, the command line tools are more of a modular/interface description which will be discussed later. The main focus of the architecture of CrossDoc is to make sure that our project can stay up to date with the text editors that come out as well as making sure that the idea behind CrossDoc remains relevant.

When a user uses the plug-in for CrossDoc on their text editor the goal is to make sure that it spans across all available text editors and can easily be updated as time goes on. This will allow CrossDoc to be used in future projects as well as become adaptable to the changes that happen through programming in computer science. This shows the overview of what the architecture for CrossDoc is and how it can continue working as the future comes along.

4.1. CrossDoc Repository

The CrossDoc Repository is the instance of a CrossDoc project. This repository will often correspond with a Git repository, but also functions separately to denote a particular directory of code as referencing CrossDoc comments. A CrossDoc repository can be initialized within a particular project using the `cross-doc init` command. Within this module of the architecture, there are submodules that comprise the functionality. Submodules such as remote and local comment storage, comment sets, and a CrossDoc configuration file that stores persistent information for a particular repository.

This architectural module can be seen as the lowest level of interaction with the system. A user could manually create comment stores, sets, and tweak their configuration file to support the CrossDoc system, but the repository will primarily be interfaced with the *text-editor plugins*. In terms of the larger architectural system, the CrossDoc repository will be initialized by the *CrossDoc command line program*, and the repository directly holds the references and connections to both the local and remote file stores.

4.2. Command Line Program

In our plugin, we want to make sure that the user is able to use all the commands that they need to use in order to fulfill a requirement within their project. Most of these commands are used within the command line interface where the user brings up a command prompt and will be able to use all the basic functionalities of CrossDoc within the command prompt.

Figure 3 demonstrates the connectivity of the elements in the CrossDoc repository module. Although the Repository object only includes a reference to the Config object, the distinction is important as the repository includes the concept of a system instance. Although a single user will only interact with a single repository, this object should be versioned in a project, and as such, is distributed to all developers who have pulled a CrossDoc project's source.

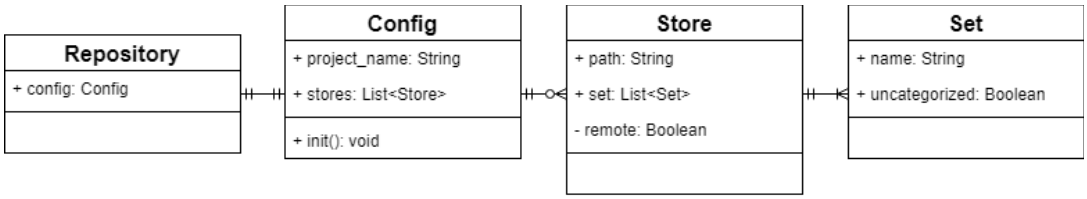

Figure 3: A UML diagram of the CrossDoc repository submodules.

4.3. Text Editor Plugins

For CrossDoc to be easily integrated into a programmer's professional work, it must be compatible with the major programming environments. The first version of CrossDoc will ship with unity between the Command-Line Parser which serves as the heart of CrossDoc functionality, and four major text editors: Atom, Emacs, Sublime, and Vim. The text editor plugins act as a bridge between the User and CL-Parser. They provide editor tools and functions for the user to document with CrossDoc easily within their environment. Although these editors may server the same sub-system role, due to their implementation differences and APIs each Text Editor plugin is its own system that must be developed, integrated, and tested independently.

Figure 4 demonstrates the objects present in this architectural module. Users interact with a selection of text editors that we have created plugins to support. This text editor plugin tool directly communicates with the command line tool to fetch and utilize the CrossDoc functionalities. An in-depth analysis of the text editor objects is below.

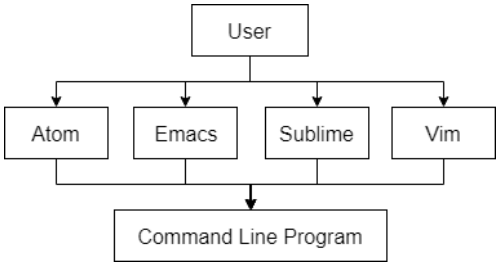

Figure 4: An architectural diagram of the text editor plugin interaction flow.

4.4. Final Architecture

The interconnectivity of the CrossDoc architecture is demonstrated in Figure 5. Each high-level module is surrounded by components that make up the module's implementation. The connection between these main modules is unidirectional in the direction of lower level functional abstractions. This architecture synchronizes well with the use case of CrossDoc as a program. Users will often interact with the *text-editor plugins* which serve as the user interface to the *command line program*, which in turn, manipulates the state of the *CrossDoc repository*. In short, the CrossDoc system can be used and manipulated at each of these high-level modules, but the more abstracted the interaction becomes, the more architectural modules will be utilized. The user could theoretically manipulate all comment storage state by simply editing the text and JSON files that make up the CrossDoc repository, but this architecture creates the most accessible and usable developmental flow.

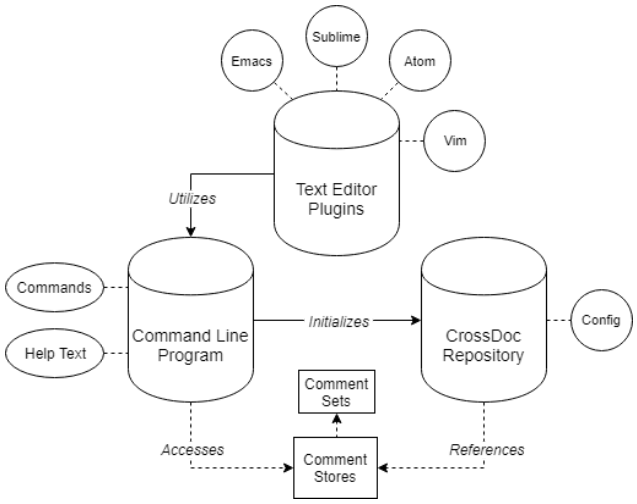

Figure 5: A high-level diagram of the CrossDoc module architecture.

Ultimately, we implemented CrossDoc using an MVC style architecture. This Model-View-Controller paradigm allowed us to ensure the modularity of the system while providing an easily expandable and maintainable core program. This meant that our descriptive and prescriptive architectures were identical, which simplified the development process.

5. Testing

To ensure we deliver upon the functionalities for CrossDoc and establish a consistent and comfortable user experience, we must put our Software through rigorous System and Usability tests. To test the core system functionality of CrossDoc's Model (Command-Line Program), we utilized Python's "unittest" library, to create 124 equivalence partitions to test and cover all of the commands within the program. Additionally, these unit tests follow through in 100% of all branch conditions within conditional functions in the Command-Line Program. This coverage is crucial to assure that CrossDoc will run as expected in all cases of user input and that in no case will the program not have a proper response. Alongside our CL Program, our View (Text Editor Plugins) must be tested for proper integration with the Model. To accurately test integration within the four unique text-editor plugins, without developing a suite of testing commands built specifically for each, we chose to run the System tests through the Text Editor Plugins using a commands class built into the CL Program. By creating a chain between the TE Plugins and the CL Program for initiating the system testing commands, we can test integration while running the already created unit tests.

Our second phase of testing revolves around the user experience, our usability testing took place with two groups of our user base, Software Developers, and Technical Writers. During testing of the Software Developers, our main focus was to see if CrossDoc's interface through the text editor plugins felt like an extension of the programming experience and not a burden. For the Technical Writer phase, the goals for testing were similar but within the Remote Store editing system (Wiki), we wanted to see if the users could not only edit the comment base remotely correctly but also see if the interface was intuitive and user-friendly. These goals reflect a key requirement of CrossDoc, as a well-crafted user experience drives retention. The feedback received from these usability tests gave differing views on the integration of CrossDoc into the software development process. We then created new conditionals for text-editor commands to more consistently anchor to the current viewed comment, with varying mouse axis locations.

6. Project Timeline

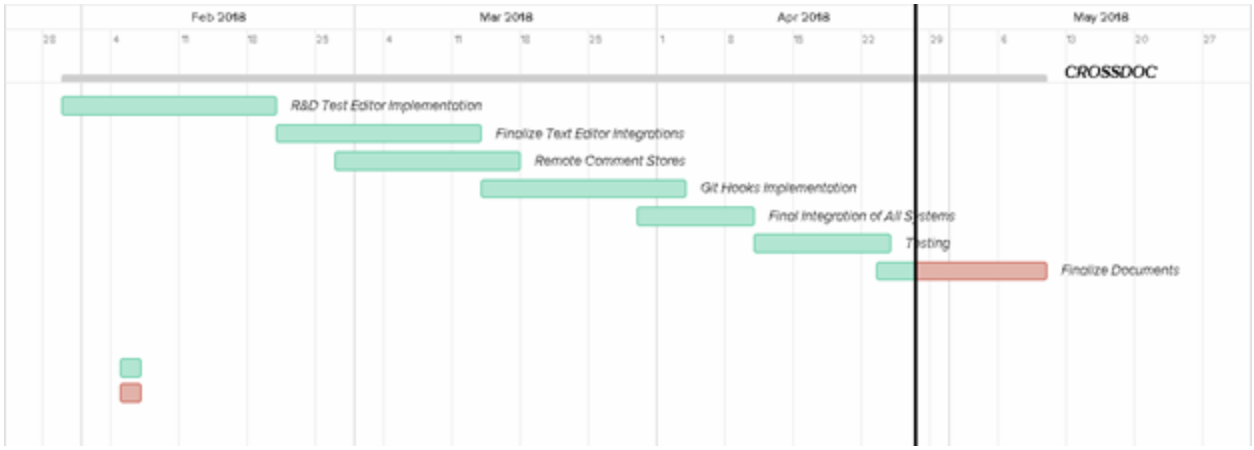

Figure 6: CrossDoc development timeline (Gantt chart).

For the development of CrossDoc, we first spent some time doing research and development for the four editors (Atom, Emacs, Sublime, and Vim). If we could prove the four editors work with CrossDoc command-line tool we knew we could create a plugin that could work with the majority of text-editors. Once we finished the R&D for the text-editors we started creating plugins for the four text-editors. We each were assigned our own text-editor to make a CrossDoc plugin for. While we were creating the CrossDoc plugins for the text-editors we started the create a Wiki (Remote Comment Store). As we were finishing up the Wiki and text-editors plugins we started to implement the Git Hooks with CrossDoc. Our next step was the final integration of CrossDoc to the text-editors. Our last step was testing the final product.

7. Future Work

As we were implementing the CrossDoc storage systems we realized that there was no way of knowing if a user's comments were up to date with the external comments stored remotely. It would be nice for another development team to have a mechanism that flags out-of-date comments. Another upgrade for CrossDoc would be integrating documentation generators (Doxygen and Javadoc). Having CrossDoc work with documentation generator would improve user's experience by expediting software documentation. Also, it would make developers more likely to comment with CrossDoc.

Lastly, it would be important to improve CrossDoc performance to improve users' experiences. Currently, CrossDoc is written in Python and we believe that if we convert CrossDoc to another high-performance language it would improve CrossDoc performance. Some of the languages that we believe would improve CrossDoc are C, C++, and D.

8. Conclusion

As reflected upon earlier, the current software development environment lends itself to problematic documentation. Such documentation can become bloated, overwritten, or in some cases nearly illegible due to its format. Additionally, due to the nature of commenting in the source code, the documentation is highly coupled with the codebase making it impossible for outside contributors (see Translator) to access without direct source access.

The solution created by Octo-Docs and Dr. Palmer is CrossDoc, through our system users are able to apply additional functionalities like CRUD and Remote Access editing to their now de-coupled documentation. This newfound flexibility within the documentation empowers Software Developers and Technical Writers alike to organize, view, and work within a comfortable development environment while utilizing the extended systems.

In review, CrossDoc was designed in an MVC style architecture, chosen specifically for the modular components of the Model, View, and Controller. As these systems utilize and don't rely on, one another we can modify and extend particular systems (such as the Text-Editor Plugins) without affecting the Model and its implementation. This is important as it allows CrossDoc's Text-Editor implementations to change alongside their adapting text-editors.

To implement such a system, we developed a robust command tool, referred to as the Command-Line Program. That is responsible for the back-end commands of CrossDoc, using it we can view and edit the CrossDoc Repositories both remote and local. Further, we created four unique Text-Editor plugins for Atom, Emacs, Sublime, and Vim. These serve as the user interface for the CL Program and are adapted specifically to their base environment. This is specifically important as the Text-Editors spread four programming languages and API's so integration into one format for CrossDoc was important.

Lastly, teams that utilize CrossDoc to extend their documentation are able to expand their documentation system for better readability, superior communication through documentation, and a remotely accessible encapsulated comment base. This system has many advantages for the user but cannot be quantified by one particular metric, like our solution our clients are unique and some aspects of the current documentation problems affect them more than others. In one software development team, CrossDoc saves countless hours of pouring through blocks of commenting for relevant information. In another the encapsulation of the remote comment editing allows Technical Writers to translate the documentation without modifying the source code. CrossDoc provides the platform needed for the enhancement of software development documentation.

9. Glossary

Comment Store: CrossDoc's comment data storage entity (locally or remotely hosted).

Comment Set/Category: CrossDoc's user-defined categorization system for similar comments.

Command Line Program: CrossDoc's Python program that is distributed through pip.

pip: Python Package Index; A python package management system.

PEP8: The style guide for Python code that denotes proper syntax.

CRUD: Create, read, update, and delete; The four basic functions of persistent storage.

MVC: Model View Controller; An architectural pattern that decouples major components.

Atom: A hackable text-editor for the 21st Century.

Emacs: A family of extensible text editors.

Sublime: A sophisticated text editor for code, markup, and prose.

Vim: A highly configurable text editor for efficiently creating and changing text of any kind.

10. Appendix

Hardware: We developed CrossDoc using Windows computers. Our development systems were relatively diverse beyond this, seeing as how the program had no minimum spec requirements.

Toolchain: The first, and most important, tool in the toolchain was Python due to the fact that the command line program was written in it. Specifically, this tool utilizes Python 3 features. On top of this, pip is responsible for the distribution and installation of the program. All other libraries were Python core libraries, so no other packages need to be installed. To facilitate coding standards and improve code readability, we followed the PEP8 style guidelines with the exception of rules E111, E114, E121, E128, and E129. We found these rules specifically to be intrusive and detrimental to the readability of the code, but we found the consistency the other rules provided to be beneficial to development. To simplify the utilization of these standards, several team members used a Sublime plugin called Anaconda. This plugin automates the checking of PEP8 standards. We also heavily utilized Git and GitHub to version and distribute our code among team members.

Setup: Firstly, new developers should install or ensure that they have already installed Python 3+ and pip. The developer will also find it beneficial to have Git installed on their machine. From here, the next step of setting up the development system would be to pull the code from the GitHub repository. Now, in this repository, the developer can run `pip install . --upgrade` to locally install the CrossDoc package. Now the developer can call CrossDoc commands or install the individual text editor plugins. The Atom, Emacs, Sublime, and Vim packages are hosted in the GitHub repo and can be installed like any other package on the respective editor.

Production Style: The usual iterative development process in CrossDoc from this point would be to make a change to the command line tool and r-install it locally by once again running the pip install command. A local installation avoids having to deploy minor revisions to pip. Then, once a feature is added or a bug fixed, a new version tag can be added and the new command line program can be upload to the pip package system by following the traditional upload system. More details regarding this can be found in the NOTES.rst file in the Git repository.