Octo-Docs

Technological Feasibility Analysis

Dr. James Palmer

Garrison Smith
Peter Huettl
Kristopher Moore

x _____

10/21/17

Table of Contents

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 team is comprised of Garrison Smith, Peter Huettl, and Kristopher Moore 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, as a whole, 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 heavy integration of code and comments 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; This is why many modern teams have groups dedicated to the localization of the work environment. In traditional commenting, these helpful code descriptors are buried deep within the code, and may often be confusing to find and edit.

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 problems.

This document covers the technological feasibility of CrossDoc's capabilities by addressing several technologies that are key to its development. We begin by giving an overview of these technologies and the problems that comprise the feasibility of the project in the section titled "Technological Challenges." After presenting the issues we can properly address them in the "Technology Analysis" section by providing various solutions to the issues, and evaluating them based on their value to CrossDoc. By clearly presenting the challenges and our options, we can then clearly connect our proposed solutions in the "Technology Integration" section in which the synergy between the solutions and our project will be made clear.

Technological Challenges

There are several major technological challenges that CrossDoc will need to overcome, and in this section, we will introduce the three primary concerns. Challenges such as finding a portable and adaptable programming language that can generate an easily accessible executable for all major operating systems, creating or finding a dynamic command line argument parsing system to handle input from a wide range of text editor plug-ins, and determining the best package manager systems to distribute our program. Each of these challenges presents a clear and distinct hurdle to be overcome. By researching these decisions now, we can simplify the implementation phase of CrossDoc and thus improve developmental iteration time.

The first challenge we face is finding a **flexible programming language** that we can use to create the executable we need and that works well for development in a team environment. In this paper, we are going to analyze the feasibility of two languages that we have specifically narrowed the choices down to; Python and C/C++.

Another challenge we will encounter is creating a **functional dynamic command-line parser** with the flexibility to handle direct input and support multiple text editors through plug-ins. Integration support for Atom, Emacs, and Vim are our current focus, along with creating a parser that allows for the fast and easy creation of plug-in adaptors.

The goal of utilizing **package management systems** is to reach the most possible end-users and provide them an easy avenue to integration into our system. These systems allow developers to easily install new programs, often using a command line interface, but only for packages (otherwise known as programs) that are uploaded to the system. By keeping a catalog of where and how to install a program, these package managers can greatly simplify the installation process. Because we plan on prioritizing adoption of our platform, providing an easy installation through various package management systems is one key method of doing just that. We plan on conducting research to determine which package managers are the best fit for our product.

Technology Analysis

Understanding the primary challenges we will face is only the first step in evaluating technological feasibility. By first clearly stating the problems, it is easier to address potential solutions, which is what we will focus on in this section. In order to ensure our product is written in a portable language, has a flexible command line interface, and is available on readily available package managers, we plan on analyzing a wide variety of options.

In this section, we will research a wide variety of options to solve the problem, analyze the research, and select the most viable solution to the problem at hand. With a particular solution selected, we can then test and prove the feasibility of an implementation using this technology. By carefully analyzing and mindfully filtering our options, we will end up with a development plan that most accurately fits our project.

Selecting the Right Language

The issue that we are facing or in a sense the challenge is that we want to be able to have a programming language that is adaptable to all environments and making sure that the program is going to execute well. In addition, making sure that the language we choose is flexible for us programmers and as well as the users. The two main languages that we are researching are Python and C. Although we, as a group, are familiar with either language, we are interested in finding the language that best fits the project's requirements.

Potential Languages

This sections goal is to select between the two languages either **Python** or **C**. Python is a very extensible language that features many standard and custom libraries [14]. Python also will make this project much easier to manage in regards to readability which is something we are looking for in this project. Alternatively, C provides a lot of flexibility in regards to program manipulation

[16]. For example, we will be able to have many different files and know exactly what our code is doing when we have to make different libraries for our program. In the following sections, we will weigh the pros and cons of each language, and determine which best fits our project.

Benefits of Python

Right now we have two languages that we want to mess around with, Python and C which is what our challenge is as well. The focus is to determine what makes Python better than C because we are leaning towards a Python approach for our program but we know that C has a lot of free roaming in regards to programming the code [17].

In the case of Python's, most programmers know and are aware of the simplistic behavior that python offers and that python is the most understood language in a programming sense[18]. Python also has many embedded systems that could make the programming for our project seem less tedious and allow us to explore these libraries that Python has so that we can focus more on the project features and not the tedious stuff that we would have to do with C [18]. Additionally, the improved readability will help detect respond to bugs in a timely manner.

Benefits of C

In the case of C, we know that C is compact and has a faster runtime for programming languages. C also has an extremely fast development time as well, which can lead to the efficient interpretation of the code when it runs through a compiler [19]. For example, C can write code in a much smaller size compared to Python, such as a while loop or conditional statements for higher level programming [17].

In our project, we did come to a conclusion that C can give us many unique functions within the code base but python will allow us to take advantage of the embedded systems that python has to offer and still allow us to program what we need to program for higher level performance [17]. In the future, the end-goal might be to migrate to C, but at this moment Python seems to be the best option for our project [19].

Functionality (1-5)	**C**	**Python**
Readability	**3** - C language is difficult to understand and Read [17]	**5** - Easy organization and readability for programmers [14]
Embedded Tools	**4** - Offers 95% of embedded systems and tools for programmers [17]	**4** - Same deal as C for embedded programs, however, Python has libraries for our project [17]
Development	**5** - Development in C more flexible in regards to functionality purposes [16]	**4** -Organizational purposes and developing in python can be easy for programmers [14]

Prototype Feasibility

At this moment the main prototype that will be displayed is some basic command line functions that will be created to provide the command line activities for a user. Right now this prototype is still in development but with the basic ideas of creating a file that contains that main functionality for each command line and then uses that Python embedded systems to create these functions we could have them organized and made so that if we were going to transition to C that it would not be a difficult process to switch later down the line. However, the basic functions that we have created using Python have been a lot easier to make based on the *object-oriented programming* language style that Python has to offer and the access to many libraries that are used within the basics of the Python language.

Creating a Command-Line Parser

To support the goals of the CrossDoc project, we must implement a system to handle operations from any level the end-user may choose. The system must function using command-line processes as well as actions executed by text-editors like Atom, Emacs, and Vim. To do effectively this we must create the most efficient parser for all use-cases.

Possible Implementations

The biggest choice in the parser's design is whether the system should be designed in a monolithic or modular capacity. The monolithic approach creates a parser that is tightly coupled and can draw off of other aspects of the code. A modular approach results in a parser that is less cohesive, and communicates to other parts of the code through a set of standardized procedures.

Choosing the Best Approach

The decision between a **monolithic design** and a **modular design** has been constantly debated upon for various systems. The final choice ultimately lies in their effectiveness within the requirements of the given system. monolithic designs tailor themselves better to systems that are less likely to need additional functionality after release. Additionally, the monolithic approach reduces redundancy in systems that frequently use and access the same information. Conversely, modular design is more appealing for refactoring existing tools, adding functionality, and overall testing and readability. For CrossDoc, a modularly designed parser is the most effective design due to its maintainability, flexibility for adding in new supported editors, and more efficient readability.

Parser Tech Demo

Since our command line parser needs to be flexible in its base form as well as with a text editor, our technological demo must work within itself and at least one text editor. To prove its ability we will draft the parser and all of its needed operations, then create adapter plug-ins that will serve as the bridge between the parser's interface and the text editor. For the demonstration, we will show functionality between the parser and a plug-in for the Atom editor.

Comparing Package Managers

In the big picture, the challenge is to get our product in the hands of the most end-users possible. More technically, we are looking to solve this challenge by selecting a few major package management systems on which we will upload our program. This specific solution is in a position to solve the big picture problem by seamlessly connecting specifically with the subset of the market that we are targeting. CrossDoc is a product that improves in the documentation process for individual developers and teams. Both parties use package management systems to install and update useful programs. By fitting into these ecosystems, we will be putting our product in the best position to succeed in the target environments.

Available Systems

The unfortunate developmental hitch to using package management systems is that they are inherently self-focused ecosystems. And because there are multiple operating systems and multiple package managers per operating system, we are looking at the problem that there are many separate ecosystems that separate users utilize. Therein lies the challenge for us; which package management systems should we add our program to.

In order to narrow down our selection to a short list of systems, we must first start with an extensive list of possibilities, and begin our process from there. Our initial list of viable package managers is:

- **Chocolatey** - *Windows* [1]
- **Homebrew** - *Mac* [2]
- **Nix** - *Unix* [3]
- **Advanced Packaging Tool** - *Debian/Ubuntu* [4]
- **Python Package Index** - *Python Environment* [5]

With a general understanding of the timeline we have available to us, we will likely have the time to integrate with **2-3** package management systems. Although every system on this list is valid and has users, we are going to need to narrow the field down to the best few contenders for our specific project.

Manager Criteria Analysis

In order to simplify presentation and provide structure to the research, we will be judging the package manager alternatives on **3** main criteria; *Ease of Development*, *Popularity*, and *New-User Friendliness*. *Ease of Development* is how difficult an integration into the service would be, and is also directly correlated to how well-documented a package manager is. This data point is particularly important when considering our project's deadline and overall timelines. *Popularity* represents how adopted a manager is for its particular system. By targeting systems with the most users, we can expand the potential reach of our program. This data point will be scaled to accurately represent the entire market and not a specific OS. This may unfairly represent managers which have a large market share within their OS, but their OS is not particularly popular in the market as a whole. *New-User Friendliness* is a criterion that will not affect our development process, but a crucial one nonetheless. Again, in order to attract the most potential users, we need to be on platforms that are easily accessible so that we have room to grow and gain new users.

Analyzing Ease of Development

The first of the criteria to be reviewed is *Ease of Development*. In order for a package manager to score highly in this section, it must have easily **accessible general documentation** on uploading new packages, **language-specific documentation**, and a **flexible install process**. By analyzing the systems on these criteria, we can easily quantify how well they each fit with our product.

The first package manager to analyze is **Chocolatey**. Chocolatey's GitHub account has been around since 2011, so the project has had almost 7 years to develop and expand [6]. This is evident when viewing their documentation wiki [7]. Chocolatey has a wide array of features, and all of them are well documented on this page.

Next up, is **Homebrew**: the MacOS package manager. Much like Chocolatey, Homebrew has extensive documentation for every aspect of their system, and from every perspective (Users, Contributors, and Maintainers) [8]. One such documentation page is their Python-specific author guide [9].

Nix is the next package manager to address. Nix provides a "Package Manager Guide" [9] on their website that outlines every aspect of their system, including both how to use it, and how to create new expressions. Nix also has a GitHub documentation file detailing the creation process for Python scripts, similar to Homebrew, which would greatly streamline the process [11].

Advanced Packaging Tool is a tool for managing packages specifically on Debian systems. Unfortunately, although APT is a very popular Linux, the documentation is dispersed across various websites and pages. Without a centralized repository for implementation info, it will be very difficult to debug issues we run into while integrating our project.

Python Packaging Index is a Python-specific package manager, which means that our implementation with PIP relies heavily on whether or not our program is written in Python. For the sake of analysis, we will assume while analyzing PIP that we are using Python. Python Packaging Index has a centralized source for documentation regarding packages [12].

Analyzing Popularity

The next important criteria to analyze is a system's *Popularity*. Popularity is difficult to quantify, but is a good indication of a manager's pulse is the number of packages hosted on the platform. This does not directly correlate to the number of users, but can be used to gauge the interest in the platform as a whole. This, combined with ratings on software rating sites, and open source contributions will be used to measure a package manager's popularity. All package numbers and open source contribution numbers were updated on October 26th, 2017.

Once again, we will begin by analyzing **Chocolatey**. Chocolatey's homepage features a counter of both unique, and total packages on the platform [1]. Through this, we know that Chocolatey has *5,351 unique packages* and *39,743 total packages*. Also, in "chocolatey-coreteampackages" , the GitHub repository [13], there are *61 contributors* with a total of *4,853 commits*.

Homebrew is another package manager that is actively contributed to through open source. Through Homebrew's GitHub page [15], we can see that Homebrew is very active with many different contributors and commits across all of their repositories (the primary repository alone has *14,361 commits* from *579 contributors*). There are about *~6500 formulae* listed on Homebrew formulae package listing page [20].

Unlike the previous two managers, **Nix** is not open source but does still have publicly accessible data. Nix claims on its homepage [3] that it has *nearly 6,500* packages on its service. **APT** contains an extensive library of over *29,000 packages* that include "everything from the Linux Kernel to games" [22]. Similar to Nix, APT is not open source. Finally, **PIP** boasts a package count of *120,293*. PIP contains easily the most packages of any of the alternatives (even combined).

Analyzing New-User Friendliness

The last criteria on which to judge is *New-User Friendliness*. This category is more subjective and can be analyzed through the attempted installation of use of the managers. A user-friendly manager will provide a clear user installation guide with easy to follow package fetching steps.

Firstly, **Chocolatey** features a prominent "Install Now" button on its homepage [1], instantly directing the user to the information required to install the product, and this page features easy to follow installation instructions.

Homebrew is similar in that it prominently features the single line installation command on its homepage [2]. Homebrew's installation process is simple and intuitive, with an interactive command line interface during the install.

Nix is a package manager that shares its name with many other products, and as such, user installation instructions are not as easy to find as the two previous managers. That being said, Nix features a simple single line installation script on its homepage [3], similarly to Homebrew.

APT is perhaps the simplest user installation for those users who are already running on a Debian system because it comes pre-installed. That being said, the initial user barrier of Debian systems like Linux may be a factor.

Finally, **PIP**; which is installed by simply running a single python command. This is an intuitive process for users who are already familiar with the Python language.

Analysis Results

All-in-all, each package manager has their pros and cons, but for the sake of selecting a most viable candidate, we have quantified the value of each alternative to our product on the three main categories listed. Below is a table displaying the final results of each manager with respect to their competitors on the topic.

Functionality (1-5)	Chocolatey	Homebrew	Nix	APT	PIP
Ease of Development	**4**	**5**	**5**	**2**	**5**
Popularity	**4**	**4**	**3**	**4**	**5**
New-User Friendliness	**3**	**4**	**4**	**5**	**4**

Based on our research and the final scores of each package manager, we plan on creating implementations for both **Homebrew** and **PIP**. Creating implementations for these two options give us the broadest coverage of the market while still proving feasible to implement

Proving Feasibility of Implementations

We plan on presenting the feasibility of these technologies by creating an implementation first in PIP due to its broad market coverage and well-documented implementation instructions. This implementation will be fed the external source of the test code, likely hosted on GitHub, that can then be hosted for users to install on the PIP platform. This should provide ample proof that a full implementation using CrossDoc is feasible.

Technology Integration

The technology we have considered to use while integrating our project ranges from languages to libraries, to package management and with these challenges we have come up with a plethora of solutions that we could use to help out with these issues. Many options present different challenges within themselves as well as have pros and cons between these solutions. Laying out each solution and comparing which solution can improve the issue and/or fix it could help us better understand where we should be when it comes to prototyping our project. When we look at each solution in depth we can also come to a justification on which solution can be better integrated within our system but as well as not ruling out the other options in the end.

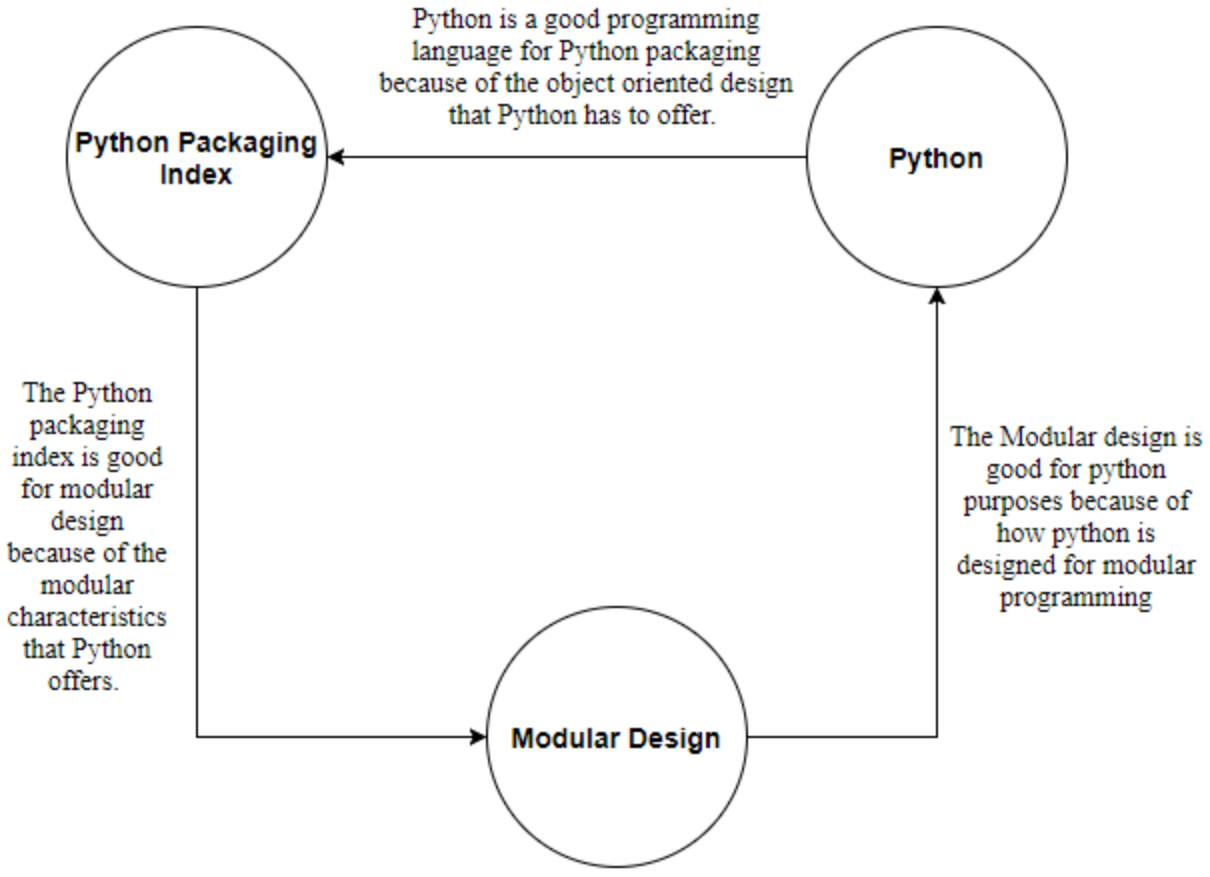
The first challenge that we ran into was deciding what type of programming language we should use when integrating our project. This challenge is a priority because it brings us one step closer to starting our prototype later in the project. The solutions that came up for the challenge was two different programming languages, **C** and **Python**. Both languages were discussed above and the key solution for our challenge was to go with Python because it is easily accessible, readable, and it has flexible libraries that C does not cover for our project.

The next challenge that we face is parsing through a command line interface and how we want to parse through this interface. The two main solutions that we discussed, regarded how we wanted to parse through the command line interface, were **monolithic design** and **modular design**. Both of these different designs have their benefits for our project as well as some more challenges that we might have to face as well. However, the best approach for the parsing design is to go with modularly designed parser it is the most effective design due to its maintainability, flexibility for adding in new supported editors, and more efficient readability.

Finally, the last challenge that we discussed was about what type of packaging system that we want to use in our project. Packaging systems are made so that when a user gets our product it can be easily accessible or at least easily executable from a file. The technology for packaging systems have evolved greatly over time and when we looked at the solutions for the packaging systems we recognized that there was going to be many options for this challenge which makes

this challenge one of the biggest ones. Making sure that we choose the correct package system depends on how easy it is for end users which in the end is our main goal to get this project for everyone. The solutions that we had in mind are **Chocolatey, Homebrew, Nix, Advanced Packaging tool,** and **Python Package Index**. However, the solution that seems to work the best in our case is the Python Package Index or **PIP** for short. The reason why is because since python is a program that can work on all operating systems than the packaging system could also be universal for our project and the end-users as well.

Below is a graphical representation of the integration between our solutions and how each one interacts with each other. Furthermore, the solutions we chose happened to work out perfectly with each other and have the best integration between each other. The information for this is in the image below.



Conclusion

Commenting in the modern age is in dire need of enhancement, the highly coupled nature of comments and their codebase leads to a multitude of problems for software development teams. CrossDoc will correct this by providing additional functionality and flexibility to standardized programming environments.

This feasibility analysis was created to assess distinct technological hurdles, and provide solutions to them. Further, it serves as a research platform for the solutions to evaluate the optimal choices for CrossDoc implementation.

Challenge	Proposed Solution	Confidence Level
Programming Language	**Python**	**90%**
Command Line Parser	**Modular Design**	**95%**
Package Management	**PyPI, Homebrew**	**98%**

In summary, our feasibility analysis has yielded multiple solutions to our core challenges. We have concluded that utilizing Python, designing our command line parser in a modular method, and deploying with the PyPI and Homebrew packaging systems are the most effective choices for CrossDoc.

Sources

[1] - Chocolatey Homepage (https://chocolatey.org/)

[2] - Homebrew Homepage (https://brew.sh/)

[3] - Nix Homepage (https://nixos.org/nix/)

[4] - APT Homepage (https://help.ubuntu.com/lts/serverguide/apt.html)

[5] - PIP Homepage (https://pypi.python.org/pypi)

[6] - Chocolatey GitHub Stats - (https://api.github.com/users/chocolatey)

[7] - Chocolatey Docs (https://chocolatey.org/docs)

[8] - Homebrew Docs (https://docs.brew.sh/)

[9] - Homebrew Python-Specific Docs (https://docs.brew.sh/Python-for-Formula-Authors.html)

[10] - Nix Docs (https://nixos.org/nix/manual/)

[11] - Nix Python-Specific Docs (https://github.com/NixOS/nixpkgs/blob/master/doc/languages-frameworks/python.md)

[12] - PIP Docs (https://packaging.python.org/tutorials/distributing-packages/)

[13] - Chocolatey Packages Repo (https://github.com/chocolatey/chocolatey-coreteampackages)

[14] - Python Docs (https://www.python.org/)

[15] - Homebrew GitHub User (https://github.com/Homebrew)

[16] - C Docs (http://www.learn-c.org/)

[17] - Python v.s C (https://www.activestate.com/blog/2016/09/python-vs-cc-embedded-systems)

[18] - Python pros (https://www.infoworld.com/article/2887974/application-development/a-developer-s-guide-to-the-pro-s-and-con-s-of-python.html)

[19] - C pros (https://www.invensis.net/blog/it/benefits-of-c-c-plus-plus-over-other-programming-languages/)

[20] - Homebrew Package Listing (http://formulae.brew.sh/browse/a)

[21] - Monolithic and Modular Architecture (https://books.google.com/books?id=SyHWBgAAQBAJ&pg=PA94#v=onepage&q&f=false)

[22] - APT Debian Wiki (https://wiki.debian.org/Apt)