Maximilian Cunningham (V00877737)
Quynh Hoang (V00871183)
Jared Larter (V00865624)

Minimum Effort 2

SENG 371 Project 2

2019-04-10

RACI chart and Progress Reports

RACI Chart				
	Maximilian C.	Quynh H.	Jared L.	Yvonne C.
General Setup				
Set up tool access (i.e. Travis, Github, AWS)	C	R/A	A	I
Construct a RACI Chart	R	R	R/A	I
Set up Containers on AWS	C	R/A	A	I
Requirements Elicitation				
Conduct Discussion with Stakeholder	R	R	R	C
Establish Clear Goals	R	R	R	I
Establish Scope	R	R	R	I
Create Criteria and goals document	R	A	C	I
Divide Task Into Manageable Sections	R	R	R	I
Design and Prototyping				
Review Architecture Design	R	R	R	I
Identify Necessary Elements of GUI	R	R	R	I
Design GUI Prototype	R	R	C	I
Prototypes	A	C	R	I
System Development				
Develop Control Script	R	A	C	I
Construct Documentation	A	C	R	I
Implement Processing Algorithm	A	R	C	I
Construct Database	R	A	A	I
Implement GUI	C	R	A	I
Quality Assurance				
Unit Testing	R	A	C	I
Acceptance Testing	C	A	R	I
Alternate Path/Edge Case Testing	A	C	R	I
Documentation				

Ensure that code had adequate comments for understanding	C	R	R	C
Outline Project 2 written document	R	C	R	C
Write Proof of Concept Implementation	R	C	C	C
Edit Proof of Concept Implementation	C	C	R	C
Write DevOps and Tools	R	C	C	C
Edit DevOps and Tools	C	C	R	C
Write Evaluation of the Design and Implementation	R	C	C	C
Edit Evaluation of the Design and Implementation	C	C	R	C
Write Future Work	R	C	R	C
Edit Future Work	R	C	R	C
Deployment				
Hand in Final Project	C	R	A	A
Project 1 presentation	R	R	I	I
Project 2 presentation video				

Progress Log	
March 25	Official start of project development phase
March 31	Opening of Github repository
April 9	Begin work on project final documentation
April 10	Completion of documentation; Completion of program code and tests; Submission of the project

Proof of concept implementation

Project accessible at: http://mineffort.herokuapp.com/home

Repository accessible at: https://github.com/asianqtip/SENG371Project2

Our program is a simplified prototype of what we believe some of the core functionality of an Earth Data Store Observation Platform would be to serve as both a proof of concept for the idea as well as an exploration of the potential concerns and details associated with such a project.

The prototype itself consists of an application hosted in Heroku. This application is further hosted within a Docker container, with the front-end managed by a Flask API and simple HTML pages. The application consists of several pages, each dedicated to a core functionality. There is the 'Home' page, which shows the user a news feed of algorithms, complete with descriptions, and permits the user to run an algorithm from an available pool to use on a data file of choice. There is the 'Results' page, which shows the user the results of operations that previous users have done. There are the 'Upload Dataset' and 'New Algorithm' pages, which allow the user to upload an image file or algorithm, respectively. Next, there is the 'Account' page, which covers basic user account operations. Finally, there is the 'About' page, which provides a brief description of the prototype. For more of a walkthrough on the specific functionality of the program, links to a video on the subject can be found both here and in the repository for the project.

Part 1: https://drive.google.com/open?id=1QVLl8FdpIeJubv9ryo2QPtxhomGb-kgR
Part 2: https://drive.google.com/open?id=1Rqpl6rFAtXzcE2dKf_Lr725yXKtD97mg

DevOps and Tools

As mentioned in the previous section, the tools we chose were Docker, Heroku, Github, and Travis; for DevOps specifically, we took advantage of Github and Travis. Our primary method of version control was Github, with Travis as our primary method of continuous integration. When deciding what tools we wanted to use, our group unanimously agreed that version control would be necessary for optimizing the development of our project, so we chose to use Github, as it is a popular, well recognised and, most importantly, free web-based version control hosting service centered around Git - something with which all of us in Minimum Effort have had prior experience. We then chose to use Travis for continuous integration, due to it being not only free but highly compatible with Github and by extension, easy to use. With the combination of these tools, we were able to efficiently expand upon components of our main, shared codebase and then seamlessly integrate them back in, all without having to waste significant man-hours attempting to merge our changes.

It should be noted that this blend of Github and Travis was not our original set-up. In fact, at the beginning of our work on this project, we did not have a method of CI. Our codebase was still hosted on a Github repository; however, all merging was done by hand. This was not so a major issue in the beginning, as the majority of the code was written by a single individual, with the rest of the team more focused on design, management, and architecture. As the project started to evolve and grow, it was determined that we would need a better solution so that

multiple individuals could interact with the codebase, and so that as the code became more complex, it would continue to be sustainable.

In a similar vein, our project does not use a configuration management tool. While it does have many clear and well-documented benefits, we deemed it unnecessary, given the scope of the project was to create a minimalistic prototype, with a team of three people over the course of only a few weeks while each team member was also heavily involved in multiple other projects. As a result, we determined that the benefits of using configuration management would be outweighed by the added complexity it would add to the setup of our relatively small project.

Ultimately, our tools were chosen primarily because they were the most economical and familiar choices that also happened fit our specific needs. Though the functionality of all the tools we used was small, as we did not take full advantage of everything they were capable of, the combination of tools is more that enough to manage the maintenance and evolution of our project within the intended scope and to serve as the foundation for a much larger, more complex system.

Evaluation of the design and implementation

Though this proof of concept model is not the most advanced design created in all of history, it does serve a very important purpose; specifically, it shows that image related calculations and algorithms can be effectively run on a cloud platform. While it is not as complex as previous codebases that utilize image processing techniques such as running the data from the earth data store on a single machine, it is considerably more scalable.

Probably one of the largest contributing factors to the scalability of this proof of concept relative to previous models is its use of a Docker container for the program. This is very important as the use of a container makes it easy for the code to be run in a cloud-based environment, making it incredibly more powerful. Unlike most individual computers which are generally limited to single or double digit amounts of RAM, when using a cloud platform, it is possible to access hundreds of gigabytes of RAM at a time, meaning that a much larger amount of data can be run at the same time. This means that anything that needs to run can be done much quicker. The added advantage is that it reduces the dependency of the program on other computing resources, giving the program the ability to gain access to all the power that it needs at a moments notice. This could be further expanded if a serverless cloud service is used, making it so that the individual running the program could grow to use as much computational resources as needed, while still only paying for the cost of the computations themselves.

Beyond simply having the ability to be used on a cloud platform, this program has a few notable advantages over other similar programs; in particular, the use of a continuous integration tool. Using continuous integration makes it far easier to develop and improve upon a code base. This is because of the many helpful features that CI provides make code easier to compile, easier to build, easier to test, and change. Continuous Integration has the important feature of making it possible to only compile the required or changed sections of a program. The result of this is that it is much faster to do individual builds on a section of code, meaning that it can be much more quickly tested and implemented. These features, combined with the features of a configuration management tool (which we elected not to use due to the simple nature of our proof of concept) allow for a different kind of scalability. Unlike designing for a cloud-based implement, which makes it so that a product can scale the amount of data that can be processed, taking advantage of CI in development makes it easy for future developers to expand and improve upon code that has previously been developed with the result of creating a far more flexible product. This is because it could quickly evolve and be adapted to incorporate new features as they are required.

In the case of our project, there are a few drawbacks to the toolset that we used that would probably need to be improved upon before we are able to make a truly scalable solution for data scientists. This is however not an enormous issue as these issues could be relatively easily changed in a letter design as they have not been totally cemented due to the proof of concept being a relatively simple system.

Something that could have been done that would possibly work to improve our system in order to make it more scalable and easier to evolve would be to take full advantage of Travis CI. Travis is a very useful and effective tool, however, in the context of our project it was mainly used for its unit testing functionality, allowing us to easily run tests on our code were it to need more development. This could possibly have been improved upon by using Travis as a built tool as well, making it more efficient. As it stands, our builds were completed using the Heroku app which served to complete the task well enough.

Future Work

Future work that could be done on this system falls into two primary categories: the work that could be done to improve this specific prototype, and the work that could be done using this prototype as a stepping stone. It is necessary to look at this project from both perspectives, as covering just one will not portray the full picture of how this system could evolve.

Being that it is only an early proof of concept design, there is a very significant amount of work that could be done to improve this project. Chief among these would be to expand the functionality of the project. Based on our idea for what a system of this kind could look like, our proof of concept covers only the very basic functionality of the project. As a result, there are many features that would be excellent to implement in code, which would allow us to create a fairly robust and diverse system that data scientist could use to either run their data in the cloud or to access already calculated results, saving both time and money. As well, work could be done to polish the current prototype without expanding functionality, such as UI improvements and expanding the unit tests.

As for the work that could be done using the proof of concept as a prototype, we could take from the concerns and details that the prototype highlighted. These would be data storage and accessibility, security, algorithm interactions, user types, tools, and overall evolution. As far as data storage and accessibility goes, the prototype has shown that having as much data as a possible in one connected database would make it significantly easier to evolve over time as the primary data source would be one well-understood, accessible unit that could be treated as a foundation for future development. This solution would be no means be perfect -

not all data would be of the same type, so it would either need to all be converted to one single type (i.e. STAC) for convenience or else be left with multiple copies of different types, both of which would come with pros and cons. As well, future work would need to establish a standardized organizational hierarchy to make navigating and querying the data easier. Additionally, an improved method of uploading and downloading data to allow various degrees of contribution and security for public and private organizations would be a a very useful feature. Finally, all of this information would need to be made publicly available so that anyone who wishes to develop an algorithm for this system would know how to develop their algorithm to connect with the data. This ties into our next concern: security. Various organizations might wish for many different levels of security for their data and intellectual property; as such, it would be beneficial to use a system that allows these organizations the freedom to privately host their data or algorithms on the system's server or download copies of public data and algorithms to perform operations on private servers. Another security concern would be the possibility of an invalid or malicious algorithm, though this falls more within the next area of concern - algorithms and their interactions with the system, it would be a significant concern and thus safeguards would need to be put in place to stop this from happening. There are also a number of ways that the algorithm and algorithm interaction system could be improved. The user might want to create algorithms in any computing language, so the system would either need some way to run any kind of language, or, more realistically, have a publically available list of the languages and versions available, with a clear explanation of how the code must be configured to interface with the system. As well, the system would need some way

to keep a collection of public algorithms and their descriptions for use by data scientists who do not want to create their own algorithm and want to instead take advantage of already existing resources.

This leads into the fact that for even the simplest expansion of this system, there would be multiple user types: algorithm developers, data scientists, and general consumers, just to name a few. Some market analysis would need to be performed to understand more specifics about what each user type would want from such a system and what kind of interface would work best for the various types of users. This would allow the system to evolve to better fill the needs of its users, in accordance with the fifth and sixth laws of software evolution. For every main feature of the system and every tool used by the system, a proper cost-benefit analysis should take place to determine the path forward. Though most of the decisions for the prototype were made in a more speculative manner of simply trying to find the cheapest option, for a much larger system that would require a much larger development effort, investing in good DevOps, cloud platforms, and other tools would be essential to build a system that is resilient and evolvable (the specifics will vary based on the implementation). Finally, the most important lesson to take away from this prototype is to plan for evolution. This prototype had only minimal planning for coping with evolution, and yet, despite the relatively small scale of the prototype, it saw significant changes and evolution from the initial draft to the point it has currently reached. If a future system is to be built in line with the concept of the prototype, then there must be contingency plans in place for dealing with changes and mitigating the harms of evolution as the product grows and matures.