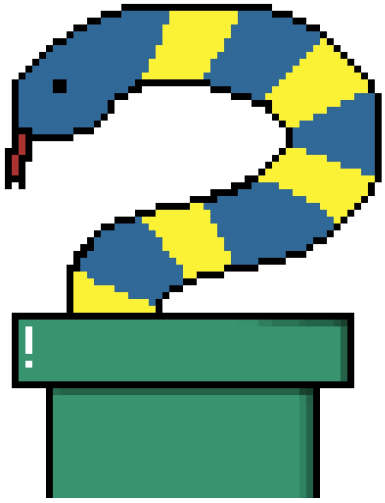

Pypline Design Document

Version 2.0

15 February 2019

Sponsor	Scott Akins Dr. Jay Laura
Mentor	Isaac Shaffer
Team	Pypline
Team Members	Nicholas Anderson Austin Collins Connor Schwirian Abdulaziz Zarie

1. Introduction

The mission of the United States Geological Survey Astrogeology Center (USGS) is to further our knowledge of the solar system through their research into planetary cartography, geoscience, and remote sensing. The center was founded to assist with lunar mapping and astronaut training for the Apollo program and continues its work helping chart and understand all planetary bodies in the Solar System. Among their responsibilities are the development of a software toolkit for working with planetary images called the Integrated Software for Images and Spectrometers (ISIS), participation in mission planning, and the archive of all NASA planetary image data.

USGS maintains the Planetary Image Locator Tool (PILOT) website to facilitate the distribution of planetary image data to the scientific community and the general public. The site allows for the selecting of images from the planetary database and utilizes their in-house processing cluster to apply desired ISIS processing tools onto the images. These tools convert archived images into useable forms that can be applied to a variety of use cases: scientific research, mission planning, and even high school science projects. This entire process is encapsulated in the Map Projection On the Web (POW) pipeline. Given the importance of the planetary images in scientific research and future NASA mission planning, their distribution is an area that needs to be made as user-friendly as possible. The current image processing pipeline lacks the flexibility needed by USGS, requires upkeep on pre-built scripts, and requires downloading of files for internal USGS staff.

Our solution will be creating a new website for user configuration of image processing workflows and the deployment of a new workflow scheduling software. The flexibility issues seen in the current pipeline will be solved through the elimination of prebuilt workflow scripts and creation of dynamically generated workflows. This will also allow our team to separate out the portions of the pipeline potentially requiring maintenance and allow for easier upkeep. Additionally, Internal USGS employees will be given an option to specify a folder path on the server to place the finalized output.

Our sponsors, Scott Akins (USGS IT Specialist) and Dr. Jay Laura (USGS Research Scientist) work in a number of areas for USGS, including general software development, spatial data analytics, and planetary data infrastructure. They have brought Pyline on to upgrade the current POW implementation, which lies between the user selecting images on the PILOT website and processing on USGS's processing cluster.

2. Implementation Overview

Pypline will replace the existing image processing pipeline with software to dynamically create job-specific workflows for execution on the USGS processing cluster. A new UI for presenting options to the user based on mission-specific recipes will allow for greater flexibility in workflow creation. The UI will allow a user to create, edit, and submit jobs over the web and work with the existing PILOT website. The user input will then be taken by the job-generating script to dynamically create a workflow unique to that user's processing job. The generator will be built as a library accessible via a RESTful interface and any additional future front ends. Management of submitted jobs and their execution will be performed by a new workflow management tool, Airflow, deployed at USGS. Airflow will continuously detect the creation of new user jobs, schedule them for execution on the processing cluster, and monitor their progress.

Pypline is designed to be easily maintainable for USGS IT staff and handle updates to the ISIS3 toolkit and new NASA missions. Our new pipeline is required to operate in a server environment that will be hosted by USGS. All of the provided components of the new pipeline will be containerized within Docker images and function independently. This will allow the new images to be integrated within the current Docker swarm at USGS.

3. Architectural Overview

The architectural build of the new pipeline is not only designed to perform faster than its predecessor but produce dynamic results. It achieves the best of both worlds by making use of the previous pipeline foundation while recruiting new processes to operate in a more efficient manner. To better understand how this design works as a whole, one must understand its individual moving parts and their specific functions.

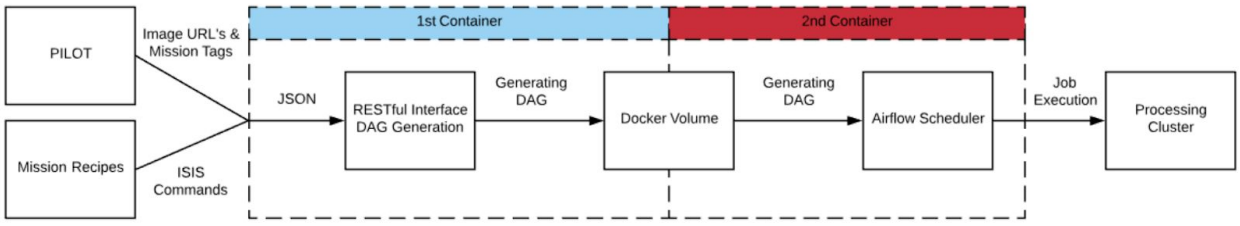

Figure 1. Architectural design of the main components and processes

The new pipeline is designed with a new user interface (UI) in mind that functions with the web-based search tool, PILOT. Our UI will accept the user selected image URLs and mission tags from the PILOT website. Additionally, it will parse current mission recipes and any future mission recipes stored in the same format. These represent the different ISIS processes that can be performed on the images from a specific mission camera. This is a key feature, as this process will present the user with the available ISIS commands, along with their respective parameters, allowing for customization of the image processing workflow. Once completed, this will result in a GET request to our API containing a JSON object that details the selected workflow processes.

The RESTful Interface serves as one of our primary focuses our product. A key feature of the RESTful interface is the flexibility offered by functioning solely through API calls. Our UI will have access to the interface but the command line API calls are able to run the pipeline even if the UI itself is excluded. The same workflows can be generated and executed though any future front ends utilizing the same API. This step is important because it creates a unique workflow for each image processing job. This process is performed by calling the Dynamic Workflow Generation library and the resulting output will be stored in the same folder utilized as input for the job scheduler. The process can then continue to the next step of the architectural design, dynamic workflow generation.

The Job Generator will take the generated workflows from the UI and convert them into a format compatible with the job scheduler. It manages this process by use of a library made up of different functions for the commands contained in the parsed mission recipes. Our team is responsible for designing the generator as a functioning library. The RESTful Interface and front end functionality depend on access the library. Processes running during job generation will be enveloped by the generator and will produce the feasible ISIS commands from the same library of functions that parse mission recipes. A key feature of generating workflows is that it is performed dynamically, meaning that generation relies on the processes and parameters selected by the user. Additionally, the library will also contain functions meant to successfully communicate with the job scheduler so it can run and parse workflows.

The job scheduler will then perform various management processes as jobs are created and completed. It will check for newly created jobs on a frequent basis. It will immediately parse and process any new job detected and then execute them on the cluster. An important feature of the new scheduler is the ability to allow USGS employees to monitor jobs as the process. Therefore, our job scheduler will utilize a UI that lists the workflows currently processing and can be selected to see the current

tasks and display the actual workflow. Additionally, images can be stored on the USGS internal server for employee access or for non-employee users to download the packaged images at their own desire.

Containerization will allow the different components of our pipeline to function concurrently in an isolated environment. All functionality in the pipeline has to occur containers to allow for proper hosting as USGS. The allows all pipeline processes to run on set stable environment even if the host machine changes.

4. Module and Interface Descriptions

The nature of our project as an image processing pipeline implies a structure of connected components operating sequentially. These components are inherently decoupled from one another, but maintain individual connections to other components.

4.1 User Interface

The user interface will function as an endpoint to our RESTful interface. It will provide the user with specific satellite functions and available parameters to manipulate the orbital satellite pictures in whatever shape the user desires. Afterward, since we are using Flask as the backend to our UI, it will take the custom user parameters and provide the restful interface with necessary information. In the end, the UI will get a new set of processed images which the user can then display or download.

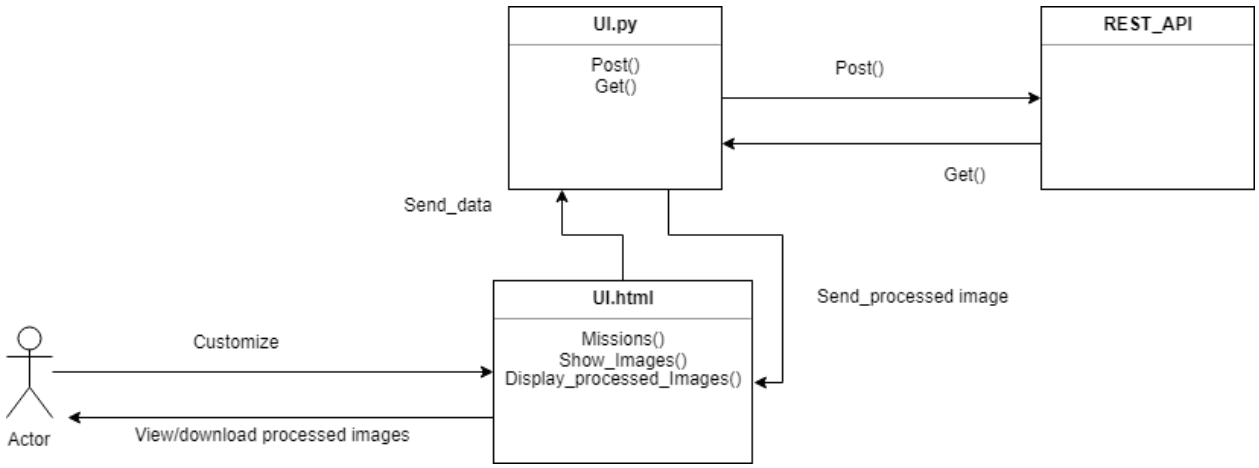

Figure 2. A diagram of the structure surrounding our user interface.

- **Missions()**
  This function will display the satellite missions currently inside the USGS archive for the user to choose from.
- **Show_images()**
  This function will display the images related to the satellite so the user can select the images that will be sent to the flask environment “UI.py”.
- **Display_processed_Images()**
  This function will display the processed images received from the api.
- **Post()**
  Post sends the images selected and mission specific command instructions to the api from flask for processing
- **Get()**
  Receives the output of processed images from the api.

4.2 RESTful Interface

The RESTful interface receives the user input from our UI endpoint as instructions to the satellite data and images they selected. It will convert these instructions into a JSON blob that will be sent to our job generator along with the images for processing. Afterward, it will function as the point in our system in which it transports the processed images back to our UI as shown in figure 2.

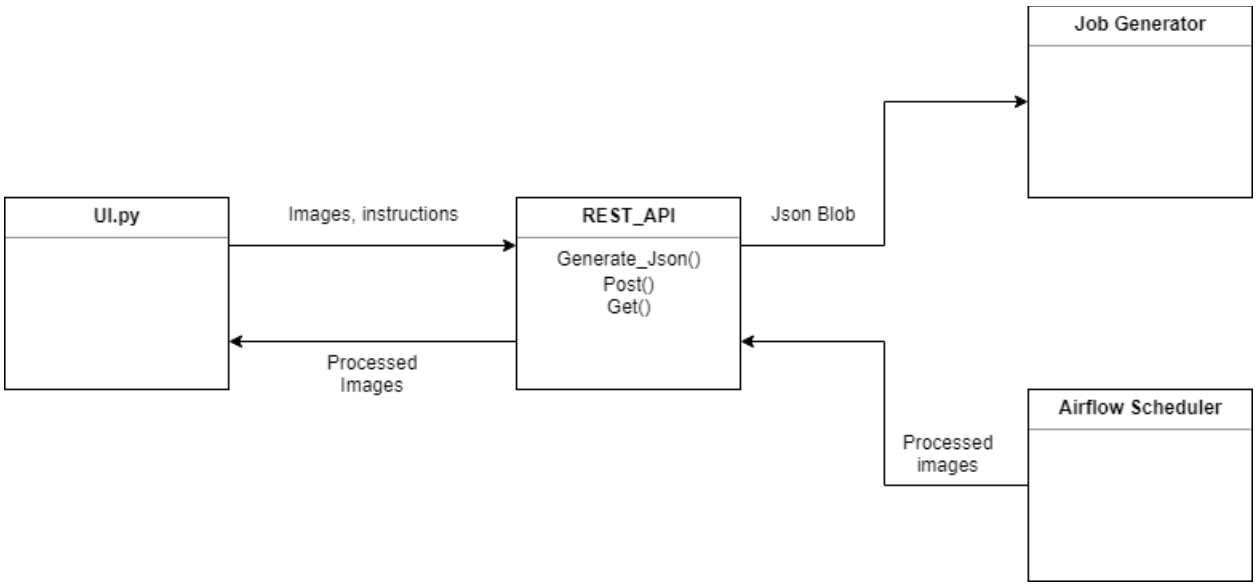

Figure 2. A diagram of the structure surrounding our RESTful interface.

- **GenerateJson()**
  This function dynamically generates a json blob from the instructions sent by the Flask endpoint.
- **Post()**
  The post function in the API will send the generated blob to the job generator along with the images.

- **Get()**
  The function will receive the images from the Airflow scheduler and report it to the UI or download it in the user's specific file directory

4.3 Job Generator

Our pipeline is powered by Directed Acyclic Graphs (DAGs). These DAGs are image processing instruction sets that are interpreted by our pipeline, which processes images based on these instructions. DAGs essentially encapsulate pipeline "jobs" created by users. These jobs are generated via our job generator. Our job generator is responsible for converting JSON input sent to our RESTful interface into an executable job. The generator exists as a callable Python library called by the RESTful interface. Figure 3 demonstrates the flow of the generator's operation.

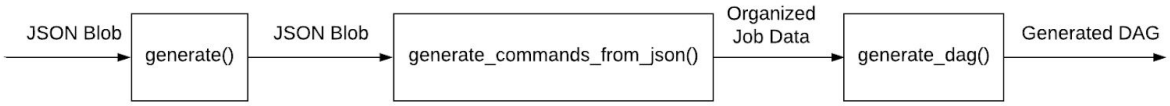

Figure 3. A function flow diagram of our job generator.

The primary entrypoint of the generator is the generate() function. This function is called by the REST API upon receiving a request. generate() takes JSON input, such as the JSON generated by the REST API, and utilizes the utility functions generate_commands_from_json() and generate_dag(). The functionality of the individual functions is as follows:

- **generate()**
  A driver function that initiates the actual processing of input and outputs generated DAGs to a specified location. This function also serves as an endpoint for the generator, essentially constituting a main function called by outside

services. This function accepts well-formatted JSON strings as input and does not provide any return.

- **generate_commands_from_json()**
  A utility function that parses JSON input and organizes the contained data into a more easily workable format. This function accepts well-formatted JSON strings as input and returns a well-organized data structure containing all information necessary for the creation of a job to its caller. This data structure is a list of job instructions.

- **generate_dag()**
  A utility function that parses the output of generate_commands_from_json() and converts the organized data into an executable DAG. This function accepts a list of job instructions as input and returns a well-formed DAG in the form of a string.

In this context, a well-formatted JSON string is a JSON string that adheres to the format of a processing job recipe. Processing jobs are formatted as follows:

```
{
   "mission": <mission_name,
   "tasks": [
       [<command1>, [[<parameter>, <value>], [<parameter>, <value], ...]],
       [<command2>, [[<parameter>, <value>], [<parameter>, <value], ...]],
       ...
   ],
   "output": <output_directory>,
   “images”: [<image1>, <image2>, ...]
}
```

4.4 Airflow Scheduler

The Airflow scheduler is responsible for continuously detecting the creation of new user jobs, scheduling them for execution on the processing cluster, and monitoring their progress. Airflow operates based on individual tasks combined into a complete workflow represented as a directed acyclic graph (DAG) within the program. In Pypline, the tasks will consist of individual ISIS commands that are chained together to produce the desired image.

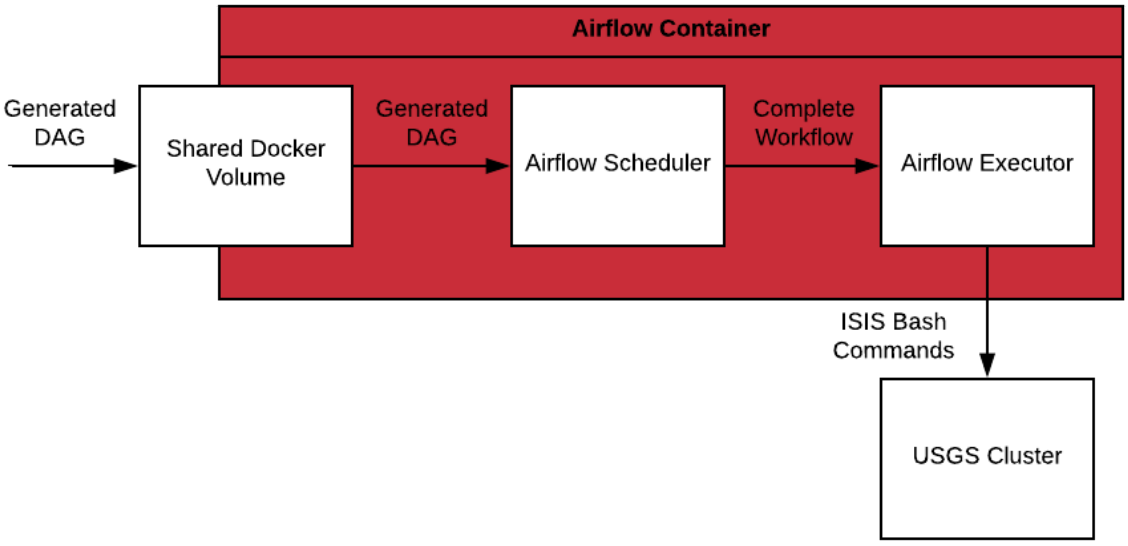

Figure 4. A diagram of the structure of our job scheduler.

The generator program will output all created DAGs to a shared volume between the UI docker container and the Airflow. On the Airflow side, this will be mounted within the Airflow directory at ~/airflow/dags. Once a new DAG is created, the Airflow scheduler will detect the new file and start the processing sequence. Airflow will attempt to parse the tasks contained within the DAG. Any syntax errors will halt processing and generate an error message in the log. The DAG will show in the web interface with a broken DAG icon. Any valid DAG will be then be scheduled for execution by the Airflow scheduler. The generator will be creating DAGs such that they will be picked up for immediate execution. To achieve this the default set of parameters are needed to ensure proper operation:

- **schedule_interval: @once**
  This parameter specifies that the generated DAG will only execute one time. Since each DAG is unique to a particular image and ISIS command sequence, there should not be a need to generate additional image copies based on it. However, any need to rerun the DAG can be accomplished manually via the web interface.

- **depends_on_past: False**
  This parameter indicates that there are no previous DAGs that are depended upon. Each DAG is a stand-alone operation and this addition prevents any blocking of the execution within Airflow.

- **start_date: datetime.today()**
  The Airflow scheduler triggers a DAG for execution based on start_date + schedule_interval. By default, Airflow schedules DAGs for execution the following day. datetime.today() will specify the beginning of the current date and in combination with the @once interval translate into immediate execution.

- **retries: 3 & retry_delay: timedelta(minutes=5)**
  To deliver reliable image processing without the need for intervention, each DAG tasks needs to be set with a number of small number of retries.

The default Airflow executor is the sequential_executor and will only run one task instance at a time. It is the only executor that can be used with sqlite which is the included database within Airflow. The executor's output to the processing cluster will be individual ISIS command line operations. The general ISIS command line format is:

```
program parameter1=value parameter2=value ...parameterN=value
```

Output images will either be moved by the final DAG task into a USGS user's folder or back to the PILOT website for user notification of image processing completion.

5. Implementation Plan

Development of our project is separated into five sections, UI, RESTful Interface, Generation, Scheduling, and Testing. Figure 4 is a complete representation of our implementation plan in the form of a Gantt chart. Broadly, our implementation plan can be described by two sections, development and documentation.

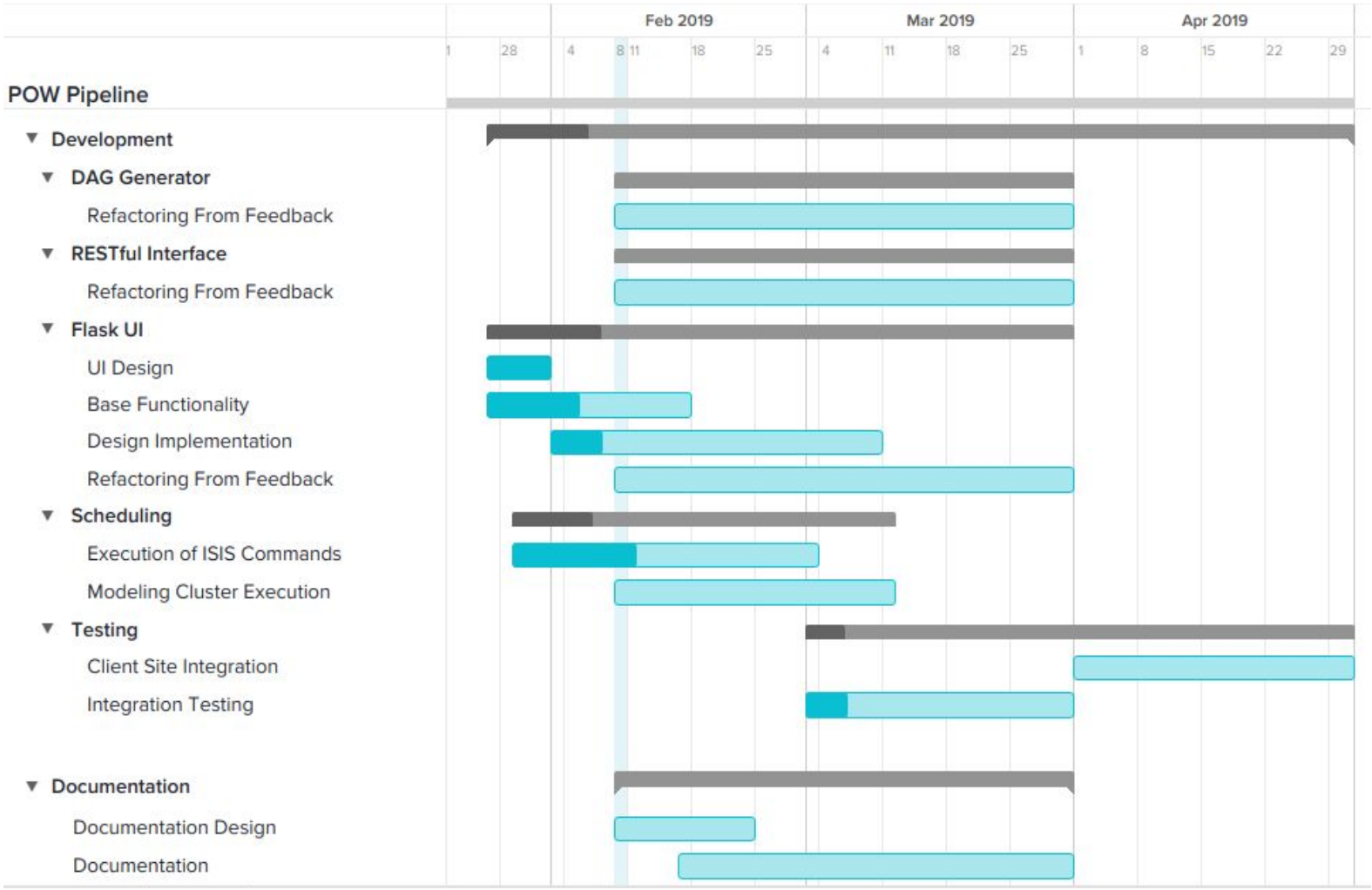

Figure 5. Our Spring implementation plan.

The development phase primarily consists of refactoring work completed during the Fall semester based on feedback from our client and can be separated into 5 subsections: our DAG generator, our RESTful interface, our user interface, our scheduler, and testing. During the Fall, we developed a comprehensive tech demo that encompassed near the entirety of our project.

We found this demo to be sound enough to be converted into our final product; with the exception of our user interface. However, following the initial development of a new user interface, development will be directed by feedback from our client concerning look and behavior. All other components have been developed according to client feedback from the outset of their Spring development cycles.

Following the completion of component development, we will begin the Testing phase. There are two sections to this phase: local and client-site integration testing. Integration testing involves ensuring our developed components perform as desired when interacting. This testing is performed through the operation of the pipeline as a whole. Client-site integration originally entailed testing how our product will perform on premise, ensuring we had developed a product immediately implementable by our client. However, due to the current direction of our product, client-site integration will now consist of demonstrating our completed product to our client.

The second section of our project is documentation. As our project will be taken over by our sponsors upon its completion, documentation is a vital part of its development. Well-developed documentation will allow our sponsors to easily comprehend our projects code and operation. Our plan is to document our project alongside its development. We believe this will allow us to best represent and explain our project.

Figure 6 describes how development work is assigned within our group. Due to the interconnected nature of User Interface and RESTful Interface development, two group members have been assigned to share the tasks.

Task	User Interface	RESTful Interface	Generation	Scheduling	Testing	Documentation
Team Member	Abdulaziz Zarie and Austin Collins	Abdulaziz Zarie and Austin Collins	Connor Schwirian	Nicholas Anderson	All	Connor Schwirian

Figure 6. A table representing the assignments of work in our project.

6. Conclusion

In addition to its scientific research, USGS serves an important role for the scientific community and the public by providing access to all of NASA's planetary image data. Distribution of the archived images is handled through the PILOT website maintained by USGS. However, the current pipeline lacks the flexibility needed to generate some of the workflows requested by its users and requires too many modifications in order to be kept updated with the ISIS toolkit. The new pipeline will provide a greater degree of customization by dynamically generating workflows that are unique to each processing

job. It will provide easier maintainability by requiring only minimal changes to the system to order in support of new missions and ISIS toolkit updates.

The architecture overview in this document details the connections between the major modules of the pipeline and defines the public methods for access. The user interface will function as the entrypoint to our RESTful interface and provides the user the ability to customize the image processing tasks. The RESTful interface will then convert the instructions into a JSON blob that will be sent to our job generator along with the URLs for processing. Our job generator is responsible for converting JSON input sent to our RESTful interface into an executable DAG. Airflow scheduler will continuously detect the creation of new user jobs, scheduling them for execution on the processing cluster, and monitoring their progress.

The logical separations between the modules allow for future modifications or additions into the system. DAG generation can be initiated via any front utilizing the RESTful interface API. Additionally, the Airflow scheduler can be extended to perform any new workflows by placing a new DAG in the processing folder.

Airflow, Docker, and Flask are new and innovative technologies that our team is excited to be working with. The pipeline technical prototypes are performing as expected and demonstrating the excellent integration potential of these technologies. Our team is confident that this design will meet the needs of USGS. We feel our product will help USGS better provide an important service to the public at large and look forward to the continued development of Pypline over the coming months.