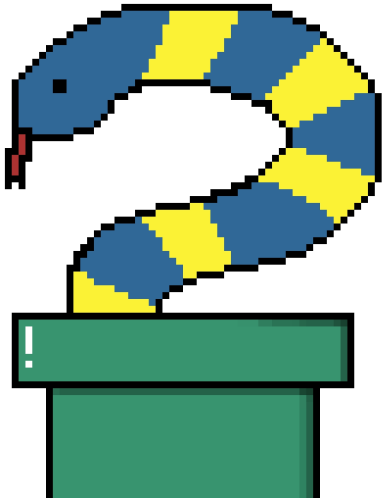


Pypline Feasibility

9 November 2018

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 USGS Astrology Center (USGS) is to further our knowledge of the solar system through their research into planetary cartography, geoscience, and remote sensing. Their responsibilities include 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 archival of all NASA planetary image data. USGS maintains the Map Project On the Web (POW) website to facilitate the public and scientific distribution of these images. The site allows for the selecting of images from the planetary database and utilizing their in-house processing cluster to apply desired ISIS tools onto the image.

Our clients, Scott Akins (USGS IT Specialist) and Dr. Laura Jay (USGS Research Scientist) have brought us on to upgrade the current processing pipeline that lies between the user selecting images on the POW website and processing on the POW cluster. Currently, after selecting the images, the user is presented with prebuilt options for modifying the images. These options utilize a set of difficult-to-maintain python scripts that offer only a static subset of options. The images selection and options are then handed off to an external scheduler within the processing cluster.

Pypline will replace the existing POW pipeline with software to dynamically created image processing workflows to be utilized by a new workflow management tool deployed at USGS. Airflow, our recommended management software, allows for workflows to be broken down into the individual processing steps and submitted as python scripts that are unique to each image processing job. The included UI will allow for viewing submitted jobs as directed acyclic graphs (DAG) providing easier monitoring and troubleshooting of generated workflows. Finally, USGS is requiring containerization of all components of the final solution to allow for easier maintenance post project completion. This document breaks down the client request and analyzes what technology stack will be needed in its development.

2. Technological Challenges

In this section we provide a high level view of the major components of our overall solution, how they will integrate, and the technical challenges we foresee in their implementation. The challenges are as follows:

- **Containerization**
  USGS requires delivery of all software as container images. Containers provide OS level virtualization by bundling an application and all runtime files needed for its execution together into a single image. Because of the isolation between containers, our chosen software will need to support a method of exchanging data between multiple container instances.

- **Workflow Management**
  This project requires us to deploy a workflow management tool to represent and execute ISIS pipeline jobs. The management software will need to include an internal scheduler to ensure correct execution of the of the workflow or provide a easy method of integration of an external scheduling solution. Additionally, the software must provide a means of monitoring the submitted jobs and ideally a dashboard that allows for job management within the UI. Since the workflows will be generated dynamically it will also need a way to recognize and execute newly created workflows.

- **Dynamic Workflow Generation**
  USGS hosts data collected from a wide of instruments that were deployed over decades of missions and stored in spacecraft specific formats. Due to this variety, our workflow generator will need the ability to parse configuration files that determine how to correctly convert the image into an ISIS standard format and what compatible tools can then be applied. This data will be combined with imputed processing steps from the user to dynamically generate a job specific DAG. These unique DAGs will then be submitted and processed by the management software. Our main technological concern is finding a language that offers a well defined library for parsing files and lends itself to easy post-project maintenance by the client.

These challenges present three distinct problems to be solved by three equally unique technologies. Below is our analysis of a number of possible solutions for each of these problems.

3. Analysis

The entire purpose of this document is to justify our choices for the technologies we will implement to complete this project. This section will present our arguments concerning various criteria for selecting the technologies we choose to implement. Our arguments are based on rigorous research into the various technologies listed below and what would make them appropriate choices for their associated requirements.

3.1 Containerization

3.1.1 Introduction

Containerization represents a major step forward in both portability and performance in comparison to past virtualization tools. Application containerization allows for deployment and isolated execution of applications through OS-level virtualization without the overhead of running an entire virtual machine. Applications are made highly portable by bundling their specific runtime components needed within the container itself. This allows for applications with different library version requirements to be run from the same host machine and preventing application breakage from updates to the underlying OS. Our project seeks to take advantage of containerization for these reasons to add to the efficiency and usability of our final project. One of the main challenges of leveraging multiple container instances is breaking of the containerization to allow the exchange of data between parts of the overall solution. Our solution will need to allow for method for passing the created workflows between the container with our DAG creation software and the scheduling server itself.

3.1.2 Alternatives

The choice of container options is largely dependent on the underlying OS they will be deployed from. We limited our search to those with Linux support which is currently being run on USGS servers. Our team only wanted to consider well developed projects and further limited our field of choices to those applications with a high adoption rate. Our team found the field of Linux containerization to be dominated by Docker with most sources estimating 80% of containers using the software. Rkt, developed by CoreOS, aims to provide better container security and is the only competition with a sizeable deployment.

- **Docker**
  One of the major benefits of Docker stems from its dominance in the industry. Docker images can either be built by hand or downloaded from the Docker Hub. Docker Hub provides Docker Certified images to provide standard libraries on which to build Pypline. While the team has not decided on using prebuilt images or not, it allows for the option during development. During our client meetings, USGS detailed the extensive use and familiarity with Docker within the organization and strong preference for the platform. One of the concerns with Docker is that the daemon runs with root privileges and spins off new containers as sub-processes. If an attacker breaks out of the container it can obtain system-wide root level access.

- **CoreOS rkt**
  CoreOS introduced rkt in 2014 as open source alternative to Docker with the aim to increase containerization security. rkt implements the app container (appc) specification while maintaining Docker image support. rkt achieves its additional security goal by removing the need for a daemon process to launch containers. In comparison to Docker, rkt does not require root privileges to create containers, preventing attackers from gain root privileges. An additional advantage of the daemonless architecture is the ability to upgrade without stopping containers that are already running. The main disadvantage of using rkt for this project is the infrastructure change that would be required by the client. The appc specification is not currently supported by the Docker engine, currently in use at USGS, and would require rkt to be run in parallel by the client.

3.1.3 Chosen Solution

To help with a decision between the alternatives, both containerization platforms were installed and used to execute multiple Alpine Linux containers. Installation and setup was simple for both platforms and allowed a prebuilt Alpine Linux container image to be downloaded from each of the alternative's image repository. Docker accomplished the image build and execution in 10 seconds with rkt requiring 13 seconds for the same task. Docker maintained its slim lead when using a larger application for testing. The execution of a nginx server container taking 1:05 and 1:19 for Docker and rkt respectively. Differences in performance between Docker and rkt while running multiple instances of the containers proved to be negligible. Using each solutions built in monitoring tools, memory utilization was 2.5 MB per nginx instance for both applications. As seen in Figure 1, both solutions will provide the needed containerization and offer fast runtimes. Our team decided to move forward with a recommendation for Docker in our project. While the security improvements with rkt were interesting, they

did not outweigh benefits of USGS familiarity with Docker. The applications were too similar to provide us with a compelling reason to suggest the USGS run an additional container management platform.

Figure 1. Containerization Comparison

	Client Familiarity	Containerization	Fast Build and Run	Strong Security
Docker	X	X	X	-
rkt	-	X	X	X

3.1.4 Proving Feasibility

To prove the feasibility of Docker as our containerization technology, we have built a Docker image that represents the environment all possible Docker containers will implement. We then ran said Docker image on multiple machines to ensure consistent performance of this image. Docker allows for the mounting of either external directories on the host machine or Docker managed drives called volumes. Multiple containers can be configured to mount the same directories or volumes. We tested this feature by mounting a host OS directory in two Docker containers. Then verified files could be created from one container that were accessible in the other instance. We are confident this feature will allow us to break the container when needed to pass data between the overall components of our solution.

3.1.5 Conclusion

Containerization represents an exciting and relatively new technology and its integration into our project will bring a number of positives to our solution. We believe Docker to be the most appropriate choice for our project, for reasons beyond our client's familiarity. Further, Docker will serve as a more than adequate environment for any technology we may select for both workflow management and dynamic DAG generation.

3.2 Workflow Management

3.2.1 Introduction

The main task within the Pypline project is to dynamically generate a list of instructions to be applied to a planetary image or list of images as chosen by the POW user. Once generated, that list must be put in the form of a workflow that is then processed and submitted to POW cluster.

3.2.2 Alternatives

Several software options were found that could possibly meet the client's needs. Airflow is an open source workflow management system developed by Airbnb and is currently undergoing incubation under The Apache Software Foundation (ASF). Luigi is an open source Python module developed by Spotify to build complex batch jobs to provide workflow management. Pinball is a workflow management platform developed at Pinterest.

- **Airflow**
  Airflow was found to have the easiest combination of tools for defining the workflows that are in use at USGS. Airflow allows for workflows to be submitted as a directed acyclic graph (DAG) made up of individual tasks. While the DAG represents how to proceed through a workflow, operators are used to describe a single task within it. Airflow includes a large list of operators by default, which will allow their inclusion of common tasks, such as Bash commands, directly into DAGs without the need for their creation during the project. Each of these workflows can be visualized in graph form and then monitored or edited on the fly via the included web interface. Additionally, DAGs can be viewed as tree view spanning across time, providing a quick overview as to which process in a DAG is producing a bottleneck. Workflows are executed via the scheduler running within Airflow. Airflow was found to offer some advantages during the testing phase of new code as well. DAGs can easily be rerun either from the beginning or from one specific task onwards via the web interface. Additionally, Airflow allows for DAGs to be directly submitted without having to wait for the scheduler to schedule them. In our testing, installation was completed via the pip installer and our team found writing simple DAGs using the included Bash operator to be a straightforward process.

- **Luigi**
  Luigi includes many of the same features as Airflow but lacks some of the polish we found in Airflow. Luigi includes a simplistic webui for monitoring of workflows but the available views are limited to only an overview of tasks and simple dependency graph. It does not provide additional views such as a task duration or a list of task variables that Airflow provides. Importantly Luigi does not include a complete scheduler as part of the project and users have to rely on cron for job execution. To achieve parallelization of jobs, pipelines first must be divided into sub-pipelines and given to separate cron processes. While not impossible, parallelization in Luigi requires additional coding that is handled automatically by

the Airflow's built-in scheduler. The team also found Luigi lacking some features that aid in the testing of new code. Determining why a job failed requires finding and examining the cron log file for that particular run. Luigi does not include a way to directly submit a pipeline, instead requiring it to be picked up by the scheduler. Workflows have to be manually started by the user and do not include a method for re-running previous jobs. Luigi only maintains a small library of task definitions that can be reused with workflows. This will require additional code to be written for any task required by the Pypline project as compared to Airflow. Luigi was tested using an installation from pip and then by using a pre-built Docker image from Docker Hub. Writing tasks to execute Bash commands was similar in difficulty to Airflow.

- **Pinball**
  Pinball was found to have the least maturity of all the options considered. Documentation for the project was found lacking with major sections missing. This was a concern not only for the development of this project but also the future maintainability by the client. The project is slow moving with only 3 commits in the last 30 days and currently lacks any major adopters. During our testing period, the pip install was broken, preventing hands-on evaluation. Based on the documentation we were able to find, Pinball does offer a web interface and scheduler with comparable to the features found in Airflow. However, due to these issues with installation and documentation, the team determined that Pinball involved too much risk to be considered for the Pypline project.

3.2.3 Chosen Solution

Airflow was initially required by the client in the project description but our team found they were open to alternatives. Through our research, we found Airflow to not only fulfill all of the project requirements but was the best management software alternative. Airflow allows for a dynamic set of individual tasks to be combined and submitted as a DAG for processing fulfilling the main client requirement. The DAGs are written in Python, which is currently the main language in use at USGS, and will allow for additional mission cameras to be added into the system post project completion. The separate scheduler offered by Airflow allows USGS fine control over the processing jobs sent into their cluster. Additionally, it aligns with future plans at USGS to leverage the scheduler set up in the Pypline project to be used for future task scheduling. Airflow's superior dashboard allows for visualization of currently in process and scheduled jobs as well as providing pre-built reports and metrics for the client. Airflow additionally has built-in support for monitoring for the presence of new DAG files and launching their execution. Luigi, by comparison, has no central process to automatically trigger a job to

launch. Pypline would need to recreate this functionality during the project since jobs will be created dynamically. As seen in Figure 2, Airflow was the only option that fulfilled all of our evaluation requirements.

Figure 2. Workflow Management Comparison

	Dynamic Workflow Support	Dependency Checking	Dashboard	Scheduler	Alternate Executor
Airflow	X	X	X	X	X
Luigi	-	X	-	-	-
Pinball	-	X	X	X	-

3.2.4 Proving Feasibility

During the evaluation of these products, our team has been able to setup Airflow within a Docker image. Small DAGs executing bash commands were written and successfully sent to the Airflow server. The next step in prototyping will be creating Airflow tasks that define a selection of tools within the ISIS toolkit and combined into a handwritten DAG. Finally, a dynamically generated DAG will need to be created and processed by the Airflow server as part of our technical demonstration.

3.2.5 Conclusion

Through our research we have found that Airflow is the only real choice for workflow management in our project. A large portion of the reason for this is its compatibility with dynamic DAG generation.

3.3 Dynamic Workflow Generation

3.3.1 Introduction

An important aspect of the above-described workflows is that they will need to be dynamically generated. In order to achieve that, we will be developing a script that will take input--involving the files to be pushed through the pipeline and the operations to be performed on those files--and generating a workflow (DAG) based upon that input. This input will come in the form of a JSON object and DAGs will need to be written to a new file. Therefore support for these actions will be an important factor in choosing a language. Additionally, our group considered code simplicity, client familiarity, and performance when analyzing programming languages.

3.3.2 Alternatives

We originally decided on three possible choices for implementing a script to the end of Dynamic DAG generation, based upon our experience with different programming languages throughout our academic careers. Python, C, and Java all represent possibilities due to their ubiquity and unique attributes.

- **Python**
  Python is a muti-paradigm programming language that sees use primarily in scripting and API design applications. The language has seen large success due to its focus on readable syntax and popularity in open source circles. Python was found to represent both the easiest to implement and best-supported tool for implementing dynamic DAG generation. Python has powerful, well-documented libraries for both the parsing of JSON and file I/O included out of the box. These libraries will be paramount in parsing recipe input and writing generated DAGs. The Python standard library also contains tools for string formatting and manipulation, among others, which allow Python to ease the process of generating output in a standardized format. DAGs will be generated in such a standardized format. Additionally, Python's strict styling requirements allow for more readable code, with fewer developer enforced styling principles. Finally, Python is in use in a variety of projects maintained by the USGS. This would allow our client to better support a Python-based generator when compared to a Java-based generator, as shown in Figure 3.

- **C**
  C is a general purpose, imperative programming language with a focus on mirroring machine-level actions. Such a focus grants the language a great deal of speed when compared to other modern programming languages. However, this speed comes at the cost of abstraction offered by the other languages our group has analyzed. A lack of abstraction translates to greater difficulty in implementation. This is the primary and most important drawback of C. The tools provided by Python and Java are simply not present in C. Where there is an entire library dedicated to parsing JSON in Python, a solution would have to be manually implemented in C. Alongside performance, C is in use for many projects maintained by the USGS. This would allow our client to better support a C-based generator when compared to a Java-based generator, as shown in Figure 3.

- **Java**
  Java is an object-oriented programming language that has seen a great deal of use in application development, especially in enterprise environments. Java was found to represent a middle ground between Python and C in terms of most categories. As described in Figure 3, Java has extensive library support for JSON and file I/O. However, Java is not a scripting language. An object-oriented programming language, such as Java, requires far more code to accomplish a task when compared to a scripting language, such as Python. This added overhead leads to inherently verbose code that can be difficult to read.

3.3.3 Chosen Solution

Python is our language of choice for creating the script to dynamically generate DAGs. Python represents the most powerful choice for both parsing information and dynamically generating output to a file. Additionally, Python's syntax is inherently the simplest and most readable of the languages we analyzed. Python does suffer somewhat in terms of performance, being the slowest language analyzed. However, while it was an attribute we considered, our group decided performance is less important than the categories that Python excels in. Instead, we prioritized library support and code simplicity as we felt these were most important for our script.

Figure 3. Dynamic Generation of Workflows

	Library Support	Code Simplicity	Client Familiarity	Performance Focus
Python	X	X	X	-
C	-	-	X	X
Java	X	-	-	-

3.3.4 Proving Feasibility

In order to prove the feasibility of Python as our scripting solution, we implemented a test script that took in sample output and generated file output.

3.3.5 Conclusion

Our dynamic DAG generating script is largely the workhorse of our project. We are confident that Python is the right choice for the script due to its overwhelming success in most of the criteria we set. We believe Python will best aid the success of not only our project, but any future implementations of our solution.

The integration of these technologies will be the crux of our project and where the majority of its complexity lies. As such we have outlined the particular challenges of integration below.

4. Integration

Each technology that we have chosen will serve in a principal aspect of our project, and as such have been selected because we believe them to be the best solutions at our disposal. However, it is vital that these technologies also are able to interact well with each other.

4.1 Integrating Docker

Docker is already being utilized by USGS and we can leverage their existing process for adding containers into the swarm. The portability of Docker, due to its reliance on a pre-built image, will ensure that our integration will not run into problems with differing runtime environments or lacking dependencies.

Docker can run multiple containers at the same time for a given host but our project only requires two. These containers run directly within the local machines kernel which allows for more containers to be running compared to a virtual machine. Our container will be loosely isolated to allow for a secure connection which is exactly how we will be able to run both containers simultaneously.

This means it is separate from the rest of the system and will require the user to map ports to specific files of what the user prefers to be or not to be inside the environment. Therefore, our two containers will result in greater compatibility of the given recipe and DAG generation. This shouldn’t be looked over as the Dockerfile will behave the same way it runs.

4.2 Integrating Airflow

Integrating Airflow provides a number of benefits that can be reused at the developers discretion. It creates a web server that can be used to render views and a metadata database to store models. This in turn allows for access to databases while being able to properly connect them. Likewise, an array of workers will be used and will adhere to respective dependencies. Airflow also provides fundamental libraries and abstractions that will simply allow the client to continue off our work for stretch goals and features.

Integration of Airflow will center around how to allow execution of individual ISIS tasks to the processing cluster within the USGS. The purpose of Airflow is to schedule and facilitate workflows as DAGs of given tasks. This can be done from the command line interface which provides a multitude of operations to be performed on a DAG. One of which is called *run* and is used to run a single task instance. It uses positional and named arguments and can be paused, unpaused, tested, etc. This is in light of supporting development and design of applications while being able to test functionality along the way.

Additionally, Airflow will be configured to allow for monitoring and workflow metric retrieval by the USGS.

4.3 Containerizing Airflow

To start, Docker and Airflow will obviously have to be installed. Then an image will have to be pulled from the puckel/docker-airflow repository followed by entering the command *docker pull puckel/docker-airflow*. This will install the Docker image that can then run Airflow in a Docker container.

The next step is to create a running container so Airflow is running on the users machine. This is done by using the command *docker run -d -p 8080:8080 puckel/docker-airflow* and allows access to the UI.

If the previously stated steps have been executed correctly the next step is to begin running DAGs. A DAG is created by defining the script and adding it to a DAG folder within the directory. However, it is not advised to add them directly to the DAG folder due to the lack of a text editor and more importantly the user cannot build the image the container runs on from the Dockerfile.

This is where ‘volumes’ come into play as they allow the user to share a directory with the machine being used and the Docker container. Thus, anything added to the container will also be added to the directory. From here a volume can be designed with intention of mapping the directory which contain the DAG definitions.

Once the DAG has been copied to the local machine it can then be tested by the operations of the command line interface which will return the success or failure of the individual tasks.

4.4 Integrating Python For Dynamic DAG Generation

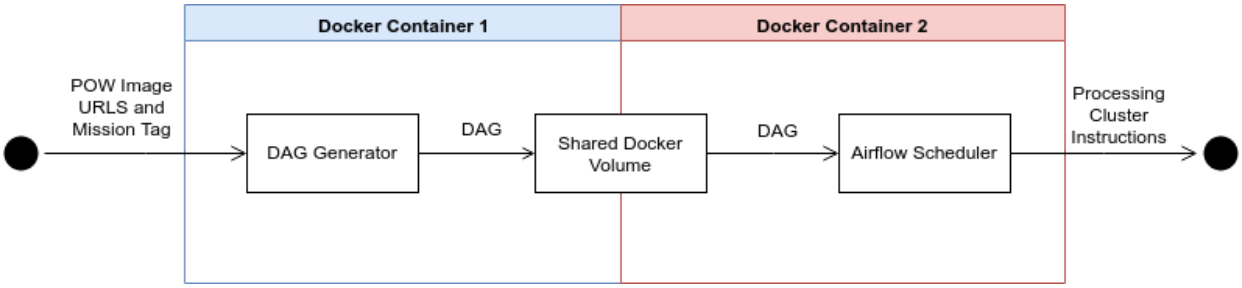
Integration of Python will revolve around running our developed script in a Docker container, which mirrors running it in a standard terminal window. This does not represent any particular challenges due to the simple nature of this script as a black box within our greater project. The Dockerfile can be written to fetch and include any additionally required libraries needed by the script for recipe parcing and DAG generation.

4.5 Overall Integration

These technologies will be integrated together in our final product in a fairly simple pattern, briefly described in Figure 4. As stated above, both Airflow--specifically the the Airflow scheduler--and our dynamic DAG generating script will be located and run in Docker containers. We will implement the technologies in two separate containers. These containers will need to communicate for the sole purpose of sharing files. As such, they will share a directory mounted to the containers.

Airflow's included sequential executor will provide a method for testing of generated scripts throughout the project. When moved onto USGS servers, near the completion of the project, a customized executor designed to interact with the processing cluster will need to be created.

Figure 4. Integration Diagram



5. Conclusion

Chosen Technologies:

- Containerization - Docker
- Workflow Management - Airflow
- Dynamic DAG Generation - Python

In addition to its research, USGS provides an invaluable service to scientific community and public through its distribution of planetary imagery. The current pipeline is in need of replacement to improve the user experience by allowing greater flexibility in submitted image processing workflows and provide USGS with better methods for the inclusion of future mission data. The purpose of this feasibility report is to consider to major technologies to include within the new pipeline and the challenges we foresee in their implementation. The Python language provides the necessary libraries for the file parsing and script generation needed by our team to create a dynamic workflow generator. Our team is confident that Airflow will provide USGS with the best workflow manager for visualization and execution of submitted jobs. The included scheduler combines the needed ability to recognize newly created DAGs, as well as a means for their submission to the cluster. Finally, the new pipeline will be able to integrate within the already existing USGS infrastructure through its delivery as containerized Docker images. Docker's volume management and mounting options provide a method for overcoming the challenges of working with the isolation inherent to containers. Overall, our team is confident that the chosen technologies will allow for delivery of an vastly improved image processing pipeline to USGS.