University of Victoria

Department of Software Engineering

SENG 371 Software Evolution

Migrating to Cloud Infrastructure

Abstracting research-focused scripts and migrating local instances to cloud-agnostic deployments.

Misha R, Devlyn D, Grant H-M, Robbie T, Alec C, Juan C

Spring 2019

Table of Contents

List of Figures

1 Introduction

1.1 Purpose

This document is intended to analyze the migration patterns of research-based Python programs from single machine deployments to scaled cloud deployments and discuss the differences between programs created for local and remote development environments. A case study of a Python batch image processing program with a single third party dependency is abstracted to represent any of the common small Python-based programs used extensively across research communities.This abstraction is used to advise a recommended set of guidelines and steps to migrate similar projects from local instances to cloud-agnostic deployments.

Due to the focus of the given Python program on analyzing GIS data and satellite imagery, the SpatioTemporal Asset Catalog (*"STAC"*) data storage specification is considered. The specification is first overviewed, and then the tradeoffs and benefits are discussed to determine whether it is advisable to adopt as a standard in this field of research.

Additionally, this document aims to gather requirements for tools to assist and automate future migrations for other researches. focusing on abstracting the technical details for non-developers.

Finally, this report begins the design of a system to aid the transition from a Compute Canada implementation to a multi-cloud solution with a focus on longevity,stability as well as usability. Software evolution tools to assist with ensuring these software quality features are also discussed.

1.2 Project Scope

This project examines and compares the implementation and deployment of a single machine instance and a distributed cloud computing deployment, hosted on Compute Canada, of image processing programs. These implementations which were provided are abstracted and used to design a cloud-agnostic architecture.

2 Current Systems

This case study of the transition from a one machine solution to a cloud hosted solution on Compute Canada focuses on two currently existing systems. These two systems are used to gather and process raw geospatial data from Spectral BC. These two systems are analyzed to provide an abstracted reference architecture. They are described below:

2.1 One Machine

The One Machine implementation is comprised of three scripts; get data CODA fernanda.sh (get-data-CODA), loop folders unzip.sh (loop-folders-unzip) and test1.py (parser).

The first script, get-data-CODA, has a manual configuration inside of the script where the user must set up multiple parameters. These user parameters include:

- OS that the script is going to be run on
- Downloader script the get-data-CODA will call
- User identification fields that will be used to connect with the spectral BC website
- Sensor type
- Date range the user wants to retrieve
- Number of downloads running concurrently
- Number of download retries
- Coordinates of the earth download area
- Type of download

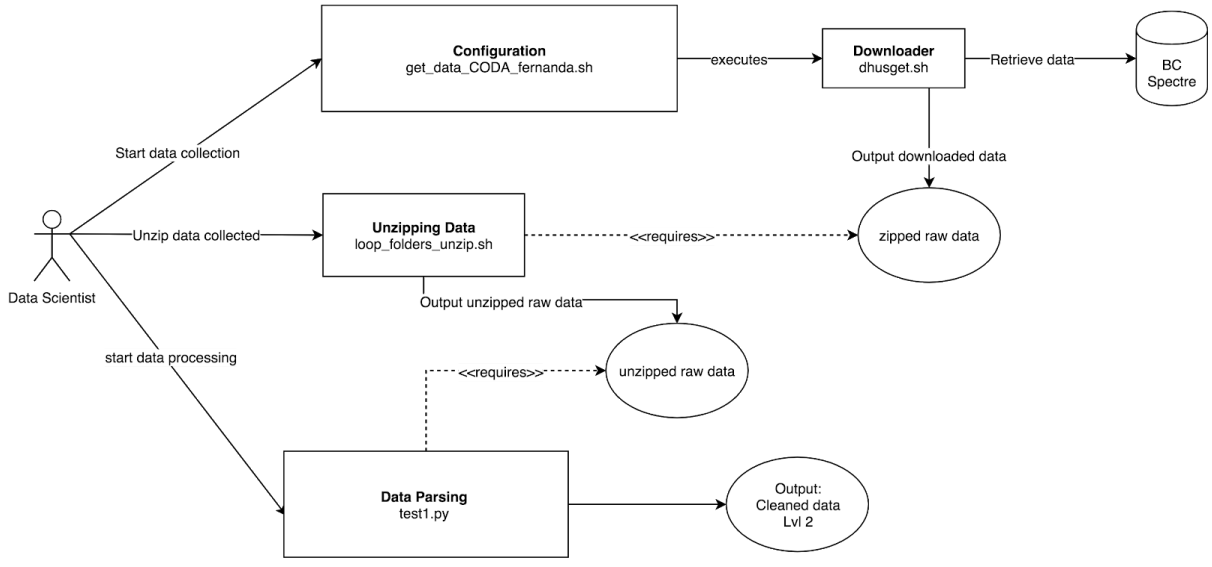

Figure 1: Data Flow Diagram of One Machine Version

Once configured, get-data-CODA uses the configuration to create the date-based file directory tree and calls a specified downloader script to fill each respective directory. By default, dhusget.sh, which downloads data from Spectral BC, is used.

The last two scripts are the loop-folders-unzip and the parser script which do not get called in any of the other scripts; they are both used manually. Loop-folders-unzip unzips all the data that has been gathered by the downloader. The parser traverses the unzipped files and corrects them with polymer, "an algorithm aimed at recovering the radiance scattered and absorbed by the oceanic waters from the signal measured by satellite sensors in the visible spectrum" [1], In particular, atmospheric correction such as light scattering from water, ice and other types of reflections.

get_data_CODA_fernanda.sh

- Selects the downloader and gathers the data from the Spectral data servers.
- The 'README' explicitly says only Linux/OSx, but inside the file windows is also listed as supported
- User settings should not be in the actual script.
- Password stored in plain-text, this needs to be securely stored.

Dhusget.sh

- Called from within get_data_CODA_fernanda.sh to handle the downloading of the data from Spectral.

Loop_folders_unzip.sh

- Loops through the folders of zipped raw data downloaded with get_data_CODA_fernanda.sh and Dhusget.sh and unzips them so they can be used in Test1.py
- Hard coded directory paths.
- There are no comments or documentation for this script.

Test1.py

- Calibrates Polymer and inputs the unzipped data into Polymer to handle the data cleaning
- polymer function run_atm_corr does atmospheric correction
- Calibration specifics are not documented.

2.2 Compute Canada

The Compute Canada implementation also consists of three main scripts and one install script; Fernanda_submit_job-_script_with_lock.sh (submit-job-with-lock), stub49a_1.sh (script-executor), script_-2018_one_folder.py (parser), and minimal_recipe (installer). The install script, minimal_recipe, handles the installation of all dependencies. The first of the main scripts, submit-job-with-lock, also has a manual configuration inside of the script where the user must set up multiple parameters. These user parameters include:

- Time duration for parsing
- Number of nodes to assign for the job
- Job name
- Account
- Amount of memory to allow the parser to use
- Notification configuration

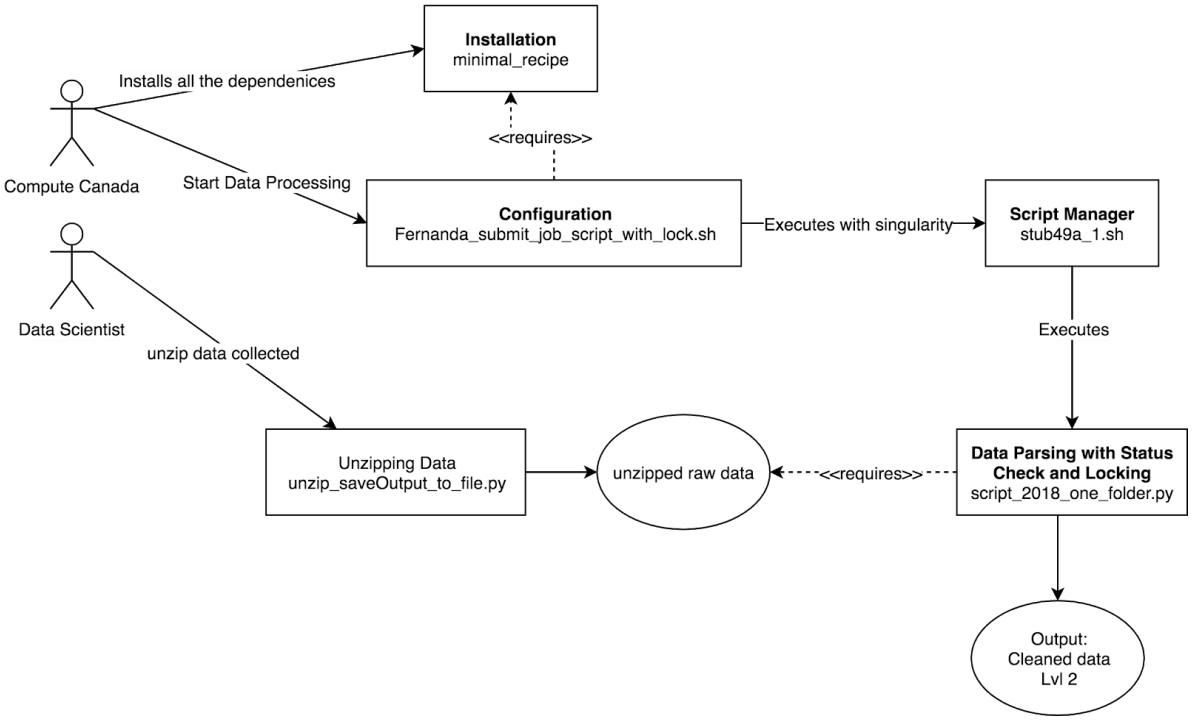

Figure 2: Data Flow Diagram of Compute Canada Version

Once configured, the singularity shell launches the script-executor, which changes the directory and starts running the parser. The parser then reads a text file and parses each line randomly, performing the same image, using polymer, correction as done in One Machine implementation.

Fernanda_submit_job_script_with_lock.sh

- Configure time frame, amount of nodes that are used to process the data, how much memory to allocate, and notification types.
- This script does not handle the locking of files as mentioned in the script name but is instead the version of the submit job script that calls the parser that has locking.

stub49a_1.sh

- Changes the directory.
- Runs the parser.

unzip_saveOutput_to_file.py

- unzips files and stores log of which ones were unzipped successfully
- Not evident how this step is called and if there is an intermittent missing process for how the data is retrieved
- The process for the recursive unzipping is not documented well

minimal_recipe

- This is run before running any script
- Installs dependencies and softwares using apt-get and pip3

script_2018_one_folder.py

- Reads 2018unzipfilelist
- If the random line is not in the completed or processing status , process image
- Keeps track of the status of files being processed
- Handles the locking and unlocking of files to minimize inefficient duplication of work on raw data

2.3 One Machine to Compute Canada Transition

The One Machine solution has been transitioned to Compute Canada and other systems have been implemented to attempt to deal with the limitations and issues that arise in the transition. The main changes and additional scripts include:

- The raw data unzipper, formerly loop_folders_unzip.sh now unzip_saveOutput_to_file.py, was changed to recursively unzip files.
- The addition of configuring Compute Canada, as found in Fernanda_submit_job_script_with_lock.sh, to maximize scheduled time. These configuration settings include a list of undocumented configuration settings.
- File locking was added to the main parser, script_2018_one_folder.py, to avoid collision with the multiple threads now handling the data processing. This locking functionality has not been entirely successful so the efficiency of the concurrency of the Compute Canada solution could use further improvements.
- The status of each file is kept track of in the main parser, script_2018_one_folder.py, incase the parsing script is descheduled partway through cleaning the data to avoid wasted processing time.

3 Reference Architecture

When comparing the One Machine and Compute Canada implementations side by side, the similarities become evident. This section proposes a reference architecture based on the similarities, while accounting for the differences.

3.1 Configuration

For the one machine solution, as covered in section 2.1, the configuration script sets where to download and store data via another script, creates a date based file directory tree, and runs the script to download data into the tree. For the Compute Canada System, as covered in section in 2.2, the configuration script mostly sets Compute Canada specific details.

Most configuration details are implementation specific, and therefore, do not belong in the reference architecture. One commonality between the systems is setting where data is retrieved, processed, and sent. This information is pertinent to the reference architecture.

3.2 Data Processing

For the one machine system, a script unzips all data into a tree, then Polymer is run over the files in the tree to correct all images within that data. For the Compute Canada system, as

covered in section 2.2, a parser goes through text files line by line then, similar to the one machine system, Polymer corrects any images within the data

Both systems first format the data so it can be processed, then correct images within that data with polymer. The Compute Canada and the One Machine systems are similar in this regard. The reference architecture should not be bound to polymer, all that matters is that data is formatted and corrected in a way that matches specifications.

3.3 System Structure

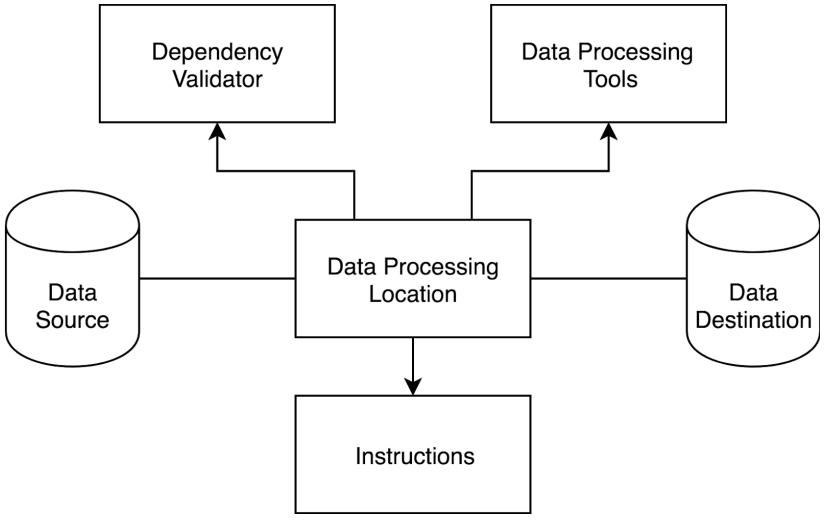

Figure 3: System Structure Reference Architecture Diagram

The purpose of this section is to break down the one machine system and the Compute Canada system into their similar and essential components and compile it into a single architecture.

3.3.1 Data Source

The data source for the program is an external repository of raw data from Spectral BC. It is used by both implementations of the program and is periodically updated by the provider. For any abstract data processing program there must be methods of retrieving data. These methods could be enacted manually, periodically, or through provider notifications. Cache is typically implemented as a hidden layer before the provider, and similarly, the attempts should be made to minimize the use of bandwidth and the provider's time.

3.3.2 Instructions

Instructions, or the program, are system agnostic and define how the data needs to be processed, based the user's needs. Instructions are needed to execute the process and manipulate the data. Different instructions will be more or less scalable depending on what processing is required. On the one machine system, the scalability of instructions is not relevant because single machine systems can't make use of parallelism. However, scalable instructions are critical to ensuring a cloud based system runs efficiently.

3.3.3 External Dependencies for Data Processing

Data processing tools are specified in the instructions, they help the user manipulate data. For instance, both the Compute Canada and One Machine systems use the third party image processing tool Polymer. It is imported as a third party Python dependency.

How dependencies are downloaded, imported, and used varies per architecture and per programming language. This enters the topic of package management, and several existing package management tools exist for Python to simplify or automate the management of these dependencies.

As with the instructions, the third party tools used depend on the data processing which is required by the user.

3.3.4 Dependency Validator

The dependency validator ensures that the instructions' dependencies are met, regardless of operating environment. For instance, the dependency validator must ensure that the correct version of Python is installed, for instructions written in python using Polymer, while pythons package manager will do the rest. The one machine system has no such validator, but the Compute Canada system does to some extent.

3.3.5 Data Processing Location

The data processing location is the network of machines used to process the data. For the One Machine system, this is simply the machine the implementation is run on. The Compute Canada system, as the name suggests, uses Compute Canada to process data.

3.3.6 Data Destination

The data destination is where processed data can be accessed by the user. For the One Machine System, this is simply the machine the system is run on. Similarly the Compute Canada system stores the data on the machine the system is run from.

3.4 Data Flow

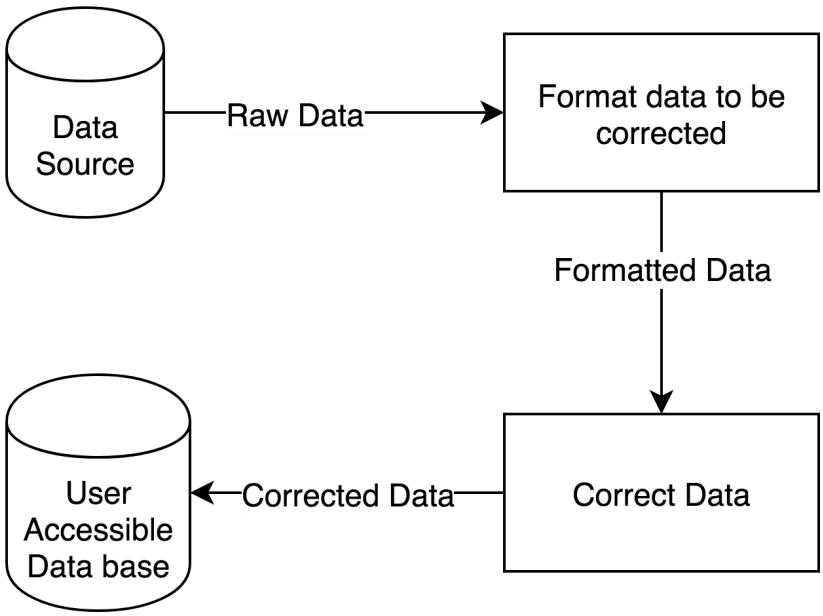

Figure 4: Data Flow Reference Architecture Diagram.

Within the context of the reference architecture, the required data flow is simple. Upon manual activation the system, data is pulled from the source,formatted, and corrected to specification. The corrected data is then saved where it may be accessed by the user.

3.5 Importance of CI and CD

Continuous integration (CI) can be an effective tool for combating the increasing complexity anticipated by the Laws of Evolution of Software Development. CI tools also help mitigate the decline in quality by running user-written and pre-built tests before each build is deployed. CI allows for developers to focus more on their code rather than testing manually or ignoring errors altogether. However, developers should not rely on CI to catch all errors as a build may have insufficient tests or not be considering all cases. CI is not a replacement for peer code review. The best way to use these types of tools is to combine a strong test suite that covers a sufficiently large scope with built-in tools, but most importantly, not forgetting to manually test the system afterwards as well.

4 STAC Transform Analysis

The tradeoffs in adopting the STAC spec are few, if any. STAC is a collection and extension of many well established standard technologies and best practices that have been designed to allow the widest range of devices and services access to data.

4.1 About STAC

STAC, the Spatial Temporal Asset Catalog [2], implements hierarchical organization and indexing of GeoJSON plain text files in a way that can be deployed purely statically. This is an incredibly well supported and simple way of providing data.

4.2 Standards in use

The base unit of data in STAC is a GeoJSON file. GeoJSON is a standardization of the layout of a normal JSON document. It's designed to convey coordinates, area, polygons, and other geospatial information.

The parent format is JSON, which is a normal plain text file that defines 'key:value' pairs of data. It is incredibly well supported in modern programming due to the proliferation of web technologies and browser scripting. An example of JSON and GeoJSON documents are provided below:

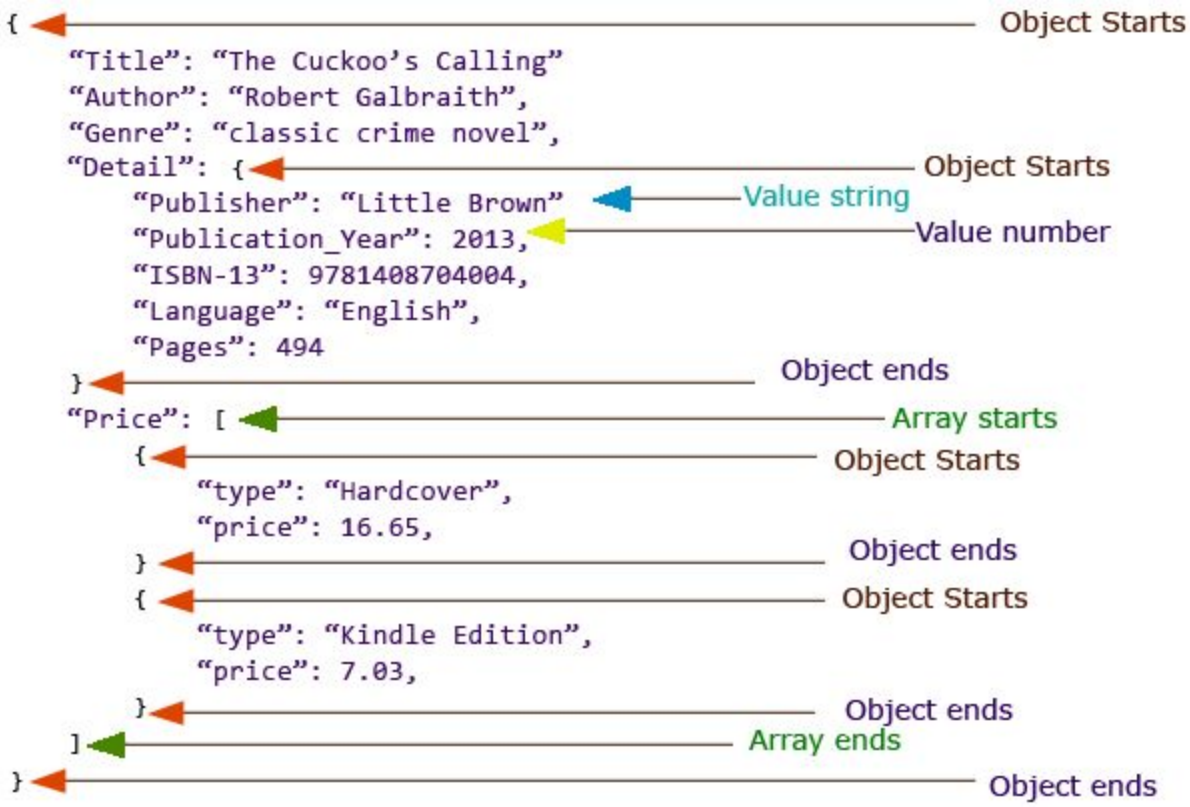

Figure 5: Example JSON document

Figure 6: GeoJSON document (right) and visualization (left)

GeoJSON is incredibly well supported by the GIS community and all modern software from enterprise software suites to small web editors. It's also distinguishable by people and easily human readable. Several applications exist, such as GeoJSON.io [3] seen in figure 6, which allow in-browser viewing and editing of GeoJSON files including drawing polygons and metadata.

4.3 Static files

STAC uses a static file structure for storing data on disk. When data is requested, the server does not need to perform any computation or generation of data to respond to the request. STAC provides extra static metadata describing the relation of any asset to neighbouring or related assets. This consistent API enables both people and bots to easily understand how they can travel from one asset to asset through collections or parent assets. For anyone familiar with REST APIs popularized in HTTP, this is very similar, but more consistently implemented.

STAC is not only static GeoJSON files. There is a lot of indexing, generation of metadata, catalogs, and cleaning/adding/removing structure to the assets. Thankfully this is well thought out and not trying to reinvent the wheel. The information is well usable and well worth the generation. Developers, users, and other services, are sure to save time when finding assets in what is otherwise a vast sea of data.

It's exciting how well designed STAC is. Its simplicity places it close to some minimal data storage concepts like Plain Old Javascript Objects (POJO) - STAC is attempting to disappear into an idea of how data should be stacked rather than trying to be a database application.

4.4 Plugins

Plugin system is also well thought out. There's a consensus on stages for a plugin's development and stability. This saves developers from adopting plugins not fit for production use. However, it seems most plugins in their ecosystem at this time are more conceptual rather than code. For instance, adding specially named properties to the GeoJSON in order to standardize a way of conveying information.

4.5 Adoption of a standard

Standards are hard to implement and popularize. When developing a standard that is meant to convey robustness and stability, is it best to follow what has been tried, tested, and well supported as previous standards.

Often, in a new project, a teams wish to use the latest cutting edge technologies. However, the Earth Data Store (EDS) needs to be a stable system, and new technologies frequently do not provide stability. STAC’s simplicity and use of existing underlying standards are reasons to trust it will provide the reliability and stability required.

The hardest part of a standard is having it reach the audience and become used. This is greatly simplified by building on layers of existing standards, but still requires work, time, and collaboration by a significant majority of a community to make it happen. However, STAC has had much success in being adopted and reviewed by major data and hosting providers such as NASA, Google Earth, and Amazon Web Services. This form of press is a way to help developers trust the success of the standard and promote its adoption.

4.6 Comparison to similar work

Data formatting is required for data driven project. This is largely due to a lack of standard data formats and the need to normalize data to begin analysis. Most code that deals with data is glue code to bridge and convert different formats.

There is not a good standard that fits every use case. Other projects have attempted to solve that, such as GraphQL. However, those abstractions are often dynamic, requiring a server to compute each request; they need a database; they have processing time strongly associated with the complexity of a request; and need a very significant amount of boilerplate code on clients to establish a connection to the server.

STAC appears very light in comparison. GeoJSON is like English. It needs no processing, even for searching or browsing catalogs, as once the data has been generated it is static and has negligible overhead aside from bandwidth.

STAC will not solve EDS' data storage issues. Even if STAC was a perfect specification, finding places to store and servers to access the sheer amount of data will be challenging. However, we fully recommend the STAC format should be used to simplify the data storage requirements of EDS.

5 Converting the Reference Architecture to Cloud

The reference architecture provided in section 3 details the required steps needed to abstract any general data processing scripts away from both one-machine or Compute Canada infrastructure. In this section the reference architecture will be used to convert the scripts for cloud deployment. Amazon Web Services has already looked into and documented how the STAC spec could be applied to work with AWS in their blog post [4].

For a cloud deployment we will be investigating Amazon Web Services (AWS) Lambda, hereafter referred to as Lambda. This platform allows for "serverless" functions designed for the developer to not need to consider the underlying hardware that runs their program. Our program is written in Python and uses external dependencies, which is supported by Lambda.

A serverless function is a zip archive of all of the code required to run the program, including its dependencies. This is very similar to the container architecture run on Compute Canada as a way of isolating dependencies for Python. Having all code in a single zip file has the added benefit of locking a program's dependency tree - there is no question as to what version of Polymer is being run, as its files are installed alongside the first-party code.

There exist several community maintained scripts to assist or automate in the production and deployment of these Lambda zip archives [6].

5.1 Data

To begin the transfer from the reference architecture to cloud, the data source and data management must first be addressed. In Lambda, a function must fetch the required data at runtime and only store the working files locally, not all files in bulk. This is already a major change in the order of operations for the program. Recall that the original program downloaded all data into a large file tree before running processing scripts on those files. In Lambda, multiple invocations of a function for parallel processing would initialize separate trees without the knowledge of one another. This also defeats the lockfile used by the Compute Canada program that ensured a node did not process an already processed file. In Lambda, this kind of shared filesystem is not available.

Amazon provides a service for shared file system storage called AWS S3. This is accessible by Lambda and could be used for the original data tree creation and lockfile system. However, AWS does not recommend this service for atomic operations. An S3 resource being accessed by multiple (read as thousands) nodes at once can be at issue. Instead, Lambda can use an atomic shared database such as AWS DynamoDB [5] for tracking such information. Thousands of Lambda instances could atomically mark which files need to be processed and which have been completed via DynamoDB. Similar tools exist for other cloud providers to satisfy this use case.

5.2 Execution

Lambda functions need a way of being triggered to run. This can be run manually from a researcher using the AWS web user interface or command line, or scheduled to run when a URL is hit, a text is sent, or a periodic timer runs out. There are many options.

5.3 Public user accessible processed data storage

As previously mentioned in section 5.1, Amazon provided a storage service called AWS S3 (Simple Storage Service) which is offers detailed customization for quotas, retention policies, encryption, and user permissions. It is also possible to create expirable links which provide temporary authorization to view a file or the latest file of a set. These options have been

developed over many years of working closely with data producers, such as our researchers, to ensure their features are sufficient for any use case.

6 Conclusion

By analyzing a case study of a migration of a data processing program from using a single machine to utilizing cloud computing, we've found similarities and differences of the two systems to understand how to streamline future migrations. Both systems can be described in terms of a data source, data processing instructions, a set of external dependencies, a dependency validator, a data processing location, and a data destination, as described in section 3. This reference architecture provides a guideline for thinking about how similar systems can be updated to use cloud computing technologies.

Additionally, we studied the STAC data format and judged that it could make a strong standard format. It builds upon used and trusted technologies and standards and hence, it would introduce minimal risk while providing many benefits.

This case study encountered several issues in the original design and transition from a one machine solution to Compute Canada, as discussed in section 2. These issues demonstrated the need for an adoption of tools for linting, continuous integration, easier deployment, and better accountability.

Furthermore, a platform to host similar data processing programs would benefit the correctness of the implementation and documentation of geospatial data processing code. This has yet to be implemented. A portion of such platform is proposed with the Simplified Execution Engine outlined in the following document.

7 Glossary

Term	**Description**
CD	Continuous Deployment
CI	Continuous Integration
Canada Compute Cloud / Compute Canada	This is a cloud infrastructure provided by the Canadian government for usage by Canadian researchers and scientists to execute performance demanding tasks

Distributed Cloud Computing	Cloud infrastructure that is physically distributed and duplicated in multiple areas of the world for redundancy and to allow a cloud application to move as needed. Often abstracted to appear as a single server and service.
One-Machine	Referring to the usage of a single machine to perform some computational task
Polymer	The given algorithm for analysis and migration. It is a geospatial image transformation library written in Python [1]
STAC	This is a standardized data format for earth observation data
The Platform/System	The system being described by this document

8 References

1. "HYGEOS - Polymer", Hygeos.com, 2019. [Online]. Available: https://www.hygeos.com/polymer. [Accessed: 10/02/2019]

2. STAC - SpatioTemporal Asset Catalog specification working draft, "stac-spec" on GitHub, 2019. [Online]. Available: https://github.com/radiantearth/stac-spec [Accessed: 26/02/2019]

3. GeoJSON.io web editor, 2019. [Online]. Available: https://geojson.io [Accessed: 26/02/2019]

4. "Keeping a [...] STAC Up To Date with SNS/SQ", Amazon Web Services Blog, 2019. [Online]. Available: https://aws.amazon.com/blogs/publicsector/keeping-a-spatiotemporal-asset-catalog-stac-up-to-date-with-sns-sqs/ [Accessed: 26/02/2019]

5. Amazon DynamoDB documentation, Amazon Web Services, 2019. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html [Accessed: 26/02/2019]

6. Examples of community resources for Lambda zip archive creation. [Online]. Available: https://stackoverflow.com/questions/30670957/creating-a-lambda-function-in-aws-from-zip-file [Accessed: 26/02/2019]