Evolution… to the Clouds!

SENG 371 Project 2

April 10, 2019
Gregory O'Hagan V00836165
Andrew Rose V00884894
Jamie St Martin V00868359

Part 1: Proof of Concept Overview

Three Python scripts are put to work in the application. Together, they form a useful snapshot of the way the entire application works:

1. frontend.py: This script sets up a web server that allows users to submit jobs through a basic UI. It also does some basic error-checking on the inputs.
    - Each job consists of a python script file, a dockerfile, a data description file in JSON format, and the requested number of nodes to spin up.
    - Once accepted, jobs are assigned a unique job ID and passed on to the manager.
2. manager.py. This script continuously lists for jobs sent by Frontend.py. When a job is received, it builds the docker image, searches for the data requested data, spins up the requested number of containers, and sets them to work processing the collected data.
    - Once the manager completes these tasks, it immediately returns to listening for more incoming jobs.
3. data_access.py: This script is trawls the CBERS database to filter out relevant STAC Items and download their useful contents: an image (thumbnail), the image coordinates, and the image time point.
    - The CBERS database is filtered by an input JSON file that contains a two-coordinate bounding box (a geographical rectangle defined by two opposite corners) and a timespan represented by two ISO 8601 timestamps.
    - The proof of concept only trawls the CBERS MUX camera's catalog for brevity's sake, but is easily extensible to the other CBERS cameras simply by starting the traversal one level higher in the CBERS catalog. Whether this extension would work *efficiently*, however, is another matter; see section 3.2.
    - "Useful contents", which are downloaded from the CBERS database, are:
        1. a STAC Item's "thumbnail" JPG image, stored in the /input/images directory
        2. a STAC Item's bounding coordinates and timestamp, stored in the /input/metadata directory in JSON format

      Again, the project could easily be extended to trawl and process more information, but this was unnecessary for the proof of concept.
    - Also, a limit of three Items per processor container (see section 3.1) has been implemented. This puts a cap on the amount of data that the proof of concept will have to trawl and process, since calculations that took minutes or more would be unnecessary to show that the system works. (In particular, the numerous HTTP GET requests required by data_access.py take a lot of time.)
4. blur.py and shrink.py. These two scripts, used as test input data, process the stored thumbnail images as placeholders for whatever data-processing algorithms the end

users of the final product may want to use. Both scripts are built around the PIL (Python Imaging Library) “Pillow” fork.

- blur_thumbnails.py applies a simple Gaussian blur to each stored thumbnail image.
- shrink_thumbnails.py reduces the thumbnails to 100 by 100 pixels. (In other words, it shrinks them to actual thumbnail size; strangely, the so-called “thumbnail” images downloaded from CBERS are all around 730 by 695 pixels.)
- Both processor scripts dump their output to the /output folder on their respective containers.

Part 2: Dev-Ops

We used a wide variety of version control, configuration management, dependency management, and other services to assemble and maintain this project.

First up is Git, our version control software of choice. Git was, first and foremost, used for its obvious purpose of version control, as well as sharing files between the members of the team. In combination with Docker and docker-compose, it was used as a configuration dependency management system; with the most recent Docker images stored in Git, a fully up-to-date version of the proof of concept could be pulled and run at any time.

Were this a professional project or simply a larger project, we would have used GitHub Webhooks for continuous integration and deployment. Whenever a new version of the project is pushed to the master branch, GitHub would then automatically update the server with the latest version of the software.

Pip, the ubiquitous Python package management system, is used to ensure that all packages are up-to-date and ensures that we don’t need to manage dependencies directly within our docker containers, which ultimately contributes to our configuration management.

Flask, the common Python web framework, is used as a proxy server for Gunicorn. Flask is well-suited to scaling, but doing so requires the implementation of some of its more advanced features, including packages (to replace modules) and blueprints. So a suitable system is in place, but some refactoring will likely be in order when scaling up. A load balancer will also be necessary to distribute requests between multiple server instances once the system encounters more frequent demands than it can handle.

Another element that would be included in a professional setting is continuous integration. In a professional setting, centred around a more complicated product, it would be completely necessary to add automated testing, perhaps using Travis, CircleCI, or Jenkins.

Finally, a brilliant version control system of sorts is Google Docs, in which this document was written. The power to collaborate simultaneously on a document has become taken for granted by university students everywhere, but it is worth noting how effectively the tool streamlines the documentation process.

Part 3: Evaluation of Design

In theory, it is possible to use parallelization to run scripts that process many files independently in the time it takes to run the script over a single file. While our minimum feasible runtime is never better than a few minutes, our design proves that this performance can be practically achieved using cloud computing on datasets of any size.

This prototype is a step between a single machine or Computer Canada script and a general cloud solution. It currently only scales across threads and not between compute nodes, but is set up in such a way that the architecture can be deployed to and take full advantage of the cloud.

3.1 Parallelizing Processing

Using our solution on a cloud service provider allows to us scale faster and larger than either a single-machine-based or Compute-Canada-based system. The core of the design is how the work is split into containers. In our single-computer prototype, this allows for full utilization of all the processor threads available, but each of these containers can be run independently on a separate processor container. A specific script to be run on many files has these files split into a number of sublists, and each of these sublists is fed into a container along with the script. Each container runs on a separate processing node, taking advantage of the full power of cloud computing.

There is a small setup cost for each container along with a minimum runtime that is charged, meaning that a balance needs to be struck between parallelizing the processing and not being wasteful with the number of nodes, but this can still parallelize a batch of any size of images down to a few minutes for short-running scripts. For long-running scripts, this can process the entire dataset in the time it previously took to process a single input file.

3.2 Data Access

Most of our data is expected to come from the CBERS database, which is stored on AWS. Network traffic to and from this to compute nodes on the same cloud service provider as the storage is typically free, and AWS is no exception here. While our current system only targets satellite data, the same approaches can be taken to any easily searchable database. As the system evolves, adding new databases is a potential area for growth.

Currently, we are traversing a satellite image catalog, which is not feasible when scaling to large numbers of searches across a massive database - the trawling process would simply take too long. However, this is where the STAC API protocol comes in; if a database was compiled using a searchable protocol such as this, the data could be searched and accessed in a way that supports scaling to the massive scale necessary for a satellite image repository. In this case, most of data_access.py would become superfluous.

Data output from users' scripts will be temporarily also stored on the cloud under our proposed system. From here, the user can download it or run it through additional data processing. This minimizes the amount of data egress from the cloud, minimizing costs to us, and by extension the users. This temporary data does need to be managed and deleted by us, but allowing cloud storage of our user's data increases the flexibility and longevity of our system.

3.3 Front End

Like the rest of our system, the front end is built from containers. This makes updating and porting it as simple as possible. If our service grows past what a single front-end server can support, scaling our front-end up is as simple as running multiple nodes behind a load balancer, ensuring it can grow with the rest of our system.

Our front end was built this way to keep it as modular and portable as possible. The amount of processor time spent on our front-end versus our compute nodes is expected to be very small. This makes the time wasted keeping our front-end up when no jobs are incoming a very small expense. This could be reduced by implementing the front end on a serverless platform. However, this comes with lock-in to a specific vendor, and moving away from containers adds complexity to managing and evolving the front end - these can no longer follow the same processes used for the rest of the system.

Moving away from Compute Canada gives us a chance to offer a much smoother user interface. Users need to modify their scripts to accept input files from the command line, but can otherwise run anything on our system that they could run on a local machine. We can offer a number of pre-built minimal Docker images that cover most of our users' needs while still allowing them to use custom containers when needed.

Because we handle data access and scaling as part of our system, users don't need to worry about implementing scaling in their own code. This will prevent some of the inefficient use of resources seen in some Compute Canada code. Keeping our users' scripts focused on the actual processing they want to do will make maintaining their code easier as well. In this way, we can not only offer the parallelization itself, but also minimize the effort required for our users to take advantage of it.

3.4 Evolution

Our entire solution is deployed in containers. This makes it modular and portable. User scripts also run in containers, which further improves this, allowing us to run any user script (provided they can package the required dependencies in a container).

Maintenance is also simplified by using these containers, as we can pull up-to-date dependencies when building these containers. Even with this, containers can be set up with specific versions of packages to support legacy user code. This greatly reduces the amount of maintenance we need to do in-house while both staying up-to-date and keeping full legacy support. Proper security measures (as discussed in section 4.2) would ensure that users running legacy code don't compromise the security of our system.

In terms of in-house maintenance, the modularized containers keep the system easy to maintain and update. However, scope creep is a major concern; modular, scalable computing has many possible applications, and care needs to be taken when adding features not to complicate the basic system too much. To stay relevant in the field of data processing, growth is needed in terms of the amount of data content and/or the number of features. But while our system has potential for both, growing too quickly is one of the largest risks to this system's ongoing success. Good management is required to combat this.

Part 4: Future Work

While this prototype proves that the concept is feasible, there is still much work to do. Here are the immediate next steps.

4.1 Migrating to the Cloud

As our current prototype is offline, the work we have done so far is cloud-agnostic. However, this does need to be moved to a cloud service provider to provide the scaling we need. As the CBERS database we are using is hosted on AWS, it is recommended to use their EC2 platform during implementation to avoid excessive data ingress/outgress costs, but another platform could be used if it could also store the database.

Each Docker image started in our prototype should be converted to a new compute node being spun up on the cloud. This provides a basic model for near-limitless scaling. If system demand grows enough to require it, the monitor and front-end nodes could be duplicated and placed behind load balancers.

4.2 Security

Security wasn’t a primary concern when building this prototype, but to turn it into a professional service, security would have to be taken very seriously. Docker is not built for security, but there are steps that can be taken to fix this. We are aiming to offer near-infinite scalability on demand at a low price point; this could be used by attackers both against our system and to launch external attacks. Both problems need to be addressed.

The first way to ensure the containers are secure is to build a wrapper around each container. This could monitor and log everything into and out of the container, ensuring it only has access to what it is intended to. Implemented properly, this would confine the damage an attacker could cause to the inside of their container, which is exclusively their own data and processes.

A second solution is to trade Docker for a different containerization software designed with security in mind, such as Singularity. Some additional security will still no doubt be required for our service, but this would reduce the amount that needed to be developed in-house. The major cost of using a lesser-known container software is potentially being limited to a smaller audience.

Monitoring and vetting users is highly recommended. While it would increase overhead, it helps protect against people creating “throw-away accounts”, which could be used to launch a single attack and then be discarded.

Tracking usage by user and time, as mentioned in section 4.3, would also help identify attack attempts that do take place.

4.3 Business Model

For this service to be a viable business product, some monetization model is needed. Relying exclusively on government grants doesn’t offer long-term stability, and even if these are being used as subsidies, a method to limit the resources a specific user can use is still required. A system needs to be developed for authenticating users and linking job requests to specific users.

First, we need a way to bill users. The service’s cloud expenses are linked to that of our provider, and the major four costs for it will be:

- Processing time (both script run-time and a “tax” for spinning up each node)
- Cloud storage (temporary storage for output data)
- Data outgress (downloading output data)
- Data ingress (uploading custom data as part of scripts)

We could bill our customers in much the same way, increasing the cost of each of some (or all) of these by a certain percentage to keep it viable and to cover the costs of storing the satellite data. The slight cost increase also allows us to offer trials of this service, similar to the "free tier" on some cloud service providers. Certain features, particularly those requiring human interactions or review, could have additional fees associated with them. Any government grants could then be used as a percentage discount or given as a certain amount of credit to some (or all) users.

Matching the cloud service provider has a number of benefits. To begin, it keeps our billing transparent to our users and most closely matches the service's expenses. This rewards well-written scripts and ensures that the service isn't responsible for bills accrued by poorly written ones. It also keeps billing easy to modify when the cloud service changes its own prices. Furthermore, the revenue generated can be used to continue developing, improving, and maintaining the service.

Monitoring in this way offers a way of limiting how much a specific user can use. While the purpose of this service is to offer near-limitless scaling, there are situations in which it is good to limit how many resources a user can consume in a given time period. A common use case for this is that if a script runs for much longer than its expected run-time (such as if it enters an infinite loop), it can be safely terminated without completing. Another common use case is to ensure a certain user's costs stay within a certain budget.

Detailed monitoring has some additional benefits. It could be used as part of the security system, as it provides the groundwork for identifying when and who was responsible if a fault occurs. The analytics produced can also be exploited by users to improve their scripts and by developers to improve the system.

Appendix A: RACI Charts

Included here is a breakdown of the tasks, their deadlines, and the assigned team members. This was updated as needed throughout the project, and served to track progress through the project.

Task	w1	w2	w3	w4	R	A	C	I
Basic server for users to submit jobs		X			Jamie	Jamie	Andrew	Greg
Update demo interface with any requirements that arise as a result of logic layer development				X	Jamie	Jamie	Greg	Andrew

Fig. A1: UI layer RACI chart

Task	w1	w2	w3	w4	R	A	C	I
Build Docker images from dockerfile		X			Greg	Greg	Jamie	Andrew
Protocol for taking down Docker containers once they finish			X		Greg	Greg	Jamie	Andrew
Protocol for splitting processing between multiple Docker containers				X	Greg	Greg	Jamie	Andrew
Splitting CBERS data into discrete parts that can be passed into a container				X	Jamie	Jamie	Greg	Andrew
Protocol for storing output data from containers				X	Andrew	Andrew	Greg	Jamie

Fig. A2: Business layer RACI chart

Task	w1	w2	w3	w4	R	A	C	I
Feasibility of accessing CBERS database	X				Andrew	Andrew	Jamie	Greg
Accessing CBERS data demo script			X		Andrew	Andrew	Jamie	Greg
Method for storing the output data from Docker containers				X	Andrew	Andrew	Greg	Jamie

Fig. A3: Data access layer RACI chart