Proof of Concept Implementation

https://github.com/BraidenCutforth/SENG371Project

See the project README.md for documentation!

Associated DevOps

There were four main tools that were used to ease maintenance and evolution. Firstly, Terraform was used to define all of the necessary cloud infrastructure in Terraform template files. Secondly, an automated testing file was added to perform unit tests. Thirdly, TravisCI was used to perform continuous integration, and hopefully at some point continuous deployment. Finally, Git and GitHub were used for version control and hosting respectively.

Terraform was used to define all of the cloud infrastructure. Terraform has several benefits that help with maintenance and evolution. Firstly, since all of the infrastructure is defined in Terraform files, it is extremely easy to manage and see infrastructure changes. Secondly, again since all of the infrastructure is defined in these files, all of the configuration can be seen by everyone, thus mitigating the 'bus factor'. Another benefit of Terraform is that infrastructure deployments are always consistent. When one person deploys a template, and another person deploys the same template the infrastructure will be identical. This stops issues around infrastructure differences (such as someone accidentally using a different lambda runtime). Finally, Terraform allows the application to be more easily setup in a CD environment, even though that isn't configured right now.

Unit tests were added to the project to help manage change as well. Unit tests are aimed at finding bugs in the code, unwanted regressions and unwanted changes. Although unit tests are not a catch all for bugs, they are definitely an important aspect to large projects. The unit tests are setup so that they are run for every PR and will display an error on the PR if a test fails.

TravisCI was used for continuous integration. TravisCI can also be set up for continuous deployment although that was not done for this project. TravisCI makes is extremely easy to run automated tests, package up your application, and run any pre-validation prior to deployment. It allows developers to validate their PRs on the fly as well. Continuous integration is an amazing tool for managing change, especially as a project grows larger.

Git and Github were used for version control. Git is the version control software and GitHub was used to host the repository and manage merging and code review. Version control is absolutely crucial for software projects of any size. It allows people to view changes before they are in the master code branch and it allows for rollbacks if something broken makes it into master. GitHub will take it a step further, making it extremely easy to view PR's and perform code reviews as well as see feedback from CI systems such as TravisCI. This tool makes managing change much easier, and is a crucial tool for any software project.

Overall, all of the above tools were used in this project to help ease maintenance and evolution of the system. Without these tools, managing changes to code and infrastructure would be a nightmare, and it would quickly turn to a mess of stitched together scripts and undocumented infrastructure.

Evaluation of the Design and Implementation

The proof of concept implementation is very evolvable and scalable, especially relative to the previous implementation. The previous single machine implementation was, well, meant for a single machine. It did not take advantage of any parallelization and would run for several days. Additionally, the code dependencies were not well managed, and simply put, the code was a bunch of scripts stitched together. The previous system was not very evolvable and had a lot of maintenance overhead in figuring out what was going on and managing all the random scripts.

The new system in contrast is very evolvable and scalable for several reasons. It is evolvable because the dependencies are well managed, the code is managed through version control, the infrastructure is in version control and the system uses CI. The new system is scalable because it is run in the cloud, and the way in which it was implemented allows for seamless scaling and parallelization.

The new system is managing dependencies using pip, the python package manager, and virtualenv. This allows anyone that clones the project to run a couple simple commands in order to have the same development environment as anyone else working on the project. This drastically reduces the chances of dependency conflicts, including in production. The dependencies present in the virtual environment are wrapped up and deployed with the application, again decreasing the chance of “but it worked on my machine”. Of course, no solution is perfect and issues may arise, but this solution covers the most common issues.

The new system is managed through version control as well. With version control, this allows easy change management. You can look back in history and rollback if necessary. You can review who made a specific change, and ask them why if need be. This also allows the opportunity for easy rollback if something goes awry. Using source control also makes certain quality control procedures really easy, such as code reviews. Code reviews greatly improve the overall code quality and maintainability as other users review code. Code reviews make sure that other people can understand the code and that the code is clean and logical. Code reviews depend on the team, so as long as the team cares about the evolution of the system then this will be a great tool.

The new system also manages all of the infrastructure configuration using Terraform templates. There are a few benefits to doing this, namely that the infrastructure is consistent, changes to the infrastructure are managed and the infrastructure does not depend on any one person. Since Terraform is used for infrastructure deployment, the infrastructure is always consistent with the template. Since the infrastructure is defined as code, every time a

deployment is made, given the same template, the exact same infrastructure is deployed. This makes the infrastructure consistent and easy for anyone to see what is actively deployed. Since the Terraform template is simply a text file, the template is stored in the repository allowing for easy change management. As with all other code, the Terraform template will go through code review and changes can also be reverted if things go awry. Finally, everything necessary to deploy the application will exist in the repository (aside from the cloud provider credentials). This allows anyone to deploy the application and stops the infrastructure from depending on a file that only exists on one person's machine. This drastically lowers the 'bus factor'.

The new system also uses TravisCI in order to do continuous integration and hopefully in the future continuous deployment. It is set up in such a way that testing and other validation will be performed on PR's and commits to master. This makes applying changes to the code extremely easy and allows for constant code testing. There is a single test function implemented right now, but from this template any number of tests could be added. This simplifies the development process and makes the new system significantly more robust for future evolution.

Finally, the new system is extremely scalable as it uses AWS Lambda, AWS's function as a service offering. Currently the application is configured such that, when a file is uploaded to a specific S3 bucket named queue it will execute a lambda function, passing in the name of the file as an argument. If multiple files are uploaded at once, several Lambda functions are created with the same code, and they run in parallel. The only limitations with Lambda are a 15 minute runtime, 3GB of RAM and 1000 parallel executions (parallel executions can be increased upon contacting AWS). This makes the scaling trivial to manage, as AWS does it without any additional configuration.

The system is also scalable in the sense that you could add additional Lambda functions, and even chain together multiple Lambda functions, possibly bypassing the runtime limit if it is possible to split your application into multiple functions. It is also possible to add independent functions to the system, so many different data science algorithms may be deployed independently. This allows the system to scale as the user base grows.

Overall, the system works as a good proof of concept and has many features that make it robust. The proof of concept was built in a way such that other people can take it and build upon it, possibly turning it into a full fledged system for performing data science operations.

Future Work

There are several things that can be done for future work in order for the new system to be fully functional. Firstly, the main data science library used, polymer, has not been implemented in the new system. Secondly, the system was implemented in AWS and should be ported over to Azure because of Azure funding support to the project. Thirdly, the system needs an interface in which people can properly execute their data science algorithm.

Fourthly, we could setup TravisCI to do continuous deployment. Finally, additional functions should be added to make the new system STAC compliant.

A significant aspect of this project is getting the bring your own algorithm aspect working in the cloud. From project 1, the single machine and Compute Canada versions of the code downloaded data and used the algorithm known as Polymer for analysis. Polymer seemed like a natural first algorithm to try when implementing our cloud proof of concept, but we were unfortunately unable to get it working. Researching alternate methods of including libraries like Polymer into this project in a working manner would be suggested future work related to this issue. Getting Polymer working is an important aspect to the project as it is the prominent library the project uses. With some additional assistance this could be possible.

Initially, the plan for this project was to use Microsoft's Azure for our cloud platform, and utilize Azure Functions. However, there were issues getting the functions running properly on Azure. The new system is implemented using python, since python is the most prominent data science programming language, and it was the language used in the single machine implementation. Azure functions currently have the python execution environment in preview. As a result, some of the functionality provided by Azure was not working correctly. Since the platform is currently unstable, the decision was made to move the implementation to AWS. In the future it would make sense to port the application back to Azure once the python implementation is stable. The move back to Azure makes sense since Microsoft is sponsoring the project through compute credits.

Another future feature of the system would be to add an interface where people may choose data from an existing S3 bucket and the associated data science algorithm that has been deployed. This would make using the system even easier for data scientists as most things would be handled for them behind the scenes. This is definitely a feature that would complete the system for continued use by many data scientists.

Another feature for the system that would be beneficial is setting up TravisCI to do continuous deployments. The reason this wasn't set up was that in order to deploy using Terraform using TravisCI was that we needed to implement remote state. Essentially, Terraform needs a state file to know what the state of the infrastructure is. To setup remote state was a bit more difficult that what we had originally anticipated. In the future, remote state could be setup and Travis could be set up to perform automatic deployments.

Finally, another aspect that this proof of concept does not cover is stack compliance. STAC compliance could be added in one of two obvious ways. Firstly, the data science algorithm specifically could be responsible for outputting data in the STAC format. This would pass on the responsibility to the data scientist to make sure their algorithms are STAC compliant. Another method that could be used is function chaining. How this would work is by having the data science algorithms output their data to a specific location, that another STAC compliance Lambda would retrieve data and make it STAC compliant. This would move the STAC logic out of the data science algorithms and move the logic for converting data into the STAC spec into its own separate function. This might make the system more maintainable overtimes, and is the suggested model if it makes sense to do so from a data perspective.

Overall, the new proof of concept system has been built such that it can be expanded upon. It has been built to handle many different possible changes and additional features. It has also been set up to help mitigate issues as the system grows larger. This will hopefully help the application as it faces the laws of software evolution.