AWS Cloudformation Infrastructure for NCAP

Taiga Abe

August 2019

1 Introduction

Amazon Web Services (AWS) is a massive, cloud based Infrastructure-as-a-service (IaaS) platform that powers many of the world's leading technologies. Given their wide user base and diversity of use cases, there are often many different ways to accomplish a single task. Here we will describe the way that that we have decided to implement a user-friendly, automated data analysis platform that hosts a variety of popular data analysis tools for neuroscience. To implement this platform, we will be using a few key AWS resources: AWS S3 for storage, AWS EC2 for compute, AWS IAM for user management, AWS Cloudwatch for monitoring, and AWS Lambda for serverless, event triggered data flow management. Of the many ways in which these resources can be configured to run together (see AWS Data Pipeline, for example), we have decided to work with the AWS Cloudformation resource, an interface that allows us to deploy pipelines programatically as collections of coordinated units with predefined interaction behavior. We have chosen this for several reasons. First, it gives us the flexibility to implement user interactivity and multi-step/branching pipelines that will be useful for certain workflows, such as parameter search for algorithms. Furthermore, it encompasses a variety of other pipelined compute solutions that we can include later, such as AWS Batch, Auto-Scaling Groups and AWS Data Pipeline.

2 Main Contributions

The main contributions of NCAP are twofold. First, we present neuroscience data analysis pipelines that make use of the Infrastructure as Code paradigm. This means that data pipelines are written in code at every scale from the software to the infrastructure. This approach leads to stereotyped behavior of the data processing pipeline at the largest scale [without necessarily cutting out the human as a checking mechanism]. Infrastructure as code makes a data processing pipeline much more reliable and automatic than the de facto norm in neuroscience. Second, we offer the resulting analysis pipelines as a service by hosting our infrastructure platform on the cloud, offloading the technical and financial cost of installing these algorithms from scratch off of the end user.

Together, the approaches of fully-coded, cloud based infrastructure maximizes the replicability of complex data analysis pipelines, and increases the efficiency with which these pipelines can be updated and extended.

3 TLDR

This section describes bare bones setup necessary to get a new pipeline up and running. Please refer to the rest of the document for details.

3.1 Prerequisites

You must first have an ami that has all necessary software downloaded, and has an executable file that automates running of all relevant code. Note that this executable will not necessarily be executed as the user; it may be necessary to manage permissions and environment variables in your bash script accordingly.

Additionally, please take your handler file, and include it in the `lambda_repo/submit_start` module so that it can be referenced. Please include a requirements file (pip style) in the `lambda_repo` directory as well. Finally, please download the aws sam cli to deploy pipelines from the command line. For references on deploying pipelines from the console, see the rest of this doc.

For a reference executable, please see below. The below assumes that you have the `ctn_lambda` repo and a configured DeepLabCut (with requisite packages in a conda environment) running on your ami already, and that new data will be directed to `auxvolume/temp_videofolder`.

```
#!/bin/bash
## CRITICAL to add cuda drivers to identifiable path.
source .dlamirc
## CRITICAL to allow for source activate commands. (okay not as critical
    as the first thing )
export PATH="/home/ubuntu/anaconda3/bin:$PATH"

## Activate the environment:
source activate dlcami

bucketname="$1"
part1="$2"
pathname="$(dirname "$(dirname "$2")")"
resultsname="$3"


sudo mkdir -p auxvolume/temp_videofolder
sudo chmod 777 auxvolume/temp_videofolder

## Download only mp4s:
cd ctn_lambda/transfer_utils
```

```
python Download_S3_single.py "$part1" "$bucketname"
    "auxvolume/temp_videofolder/" "mp4"

## Run deeplabcut analysis:
cd DeepLabCut/Analysis-tools

python AnalyzeVideos.py

## Activate script to pipeline analyses:
cd ../../ctn_lambda/transfer_utils

## Reupload everything except mp4 files.
python Upload_S3.py '../auxvolume/temp_videofolder/' "$bucketname"
    "$pathname" "$resultsname" "mp4"

## Delete temp folder
rm -r ../auxvolume/temp_videofolder

sudo poweroff
```

3.2 NCAP Template Config File

In the `cloudformation_pipelines` directory, you will see a json object file called `stack_config_template.json`. This is a blank template to work from. Please create a new directory for your pipeline, and copy this blank template into it. Use your favorite text editor to enter the following information:

```
{
    "PipelineName":"dlcmonitored1", ## Name of your cloudformation
        pipeline [not implemented]
    "REGION":"us-east-1", ## Region for your cloudformation pipeline
    "Lambda":{
        "CodeUri":"lambda_code", ## Don't change
        "Handler":"submit_start.handler", ## Specific handler function
            (start/stop)
        "Launch":true, ## Specify start or launch behavior [not
            implemented]
        "LambdaConfig":{
            "AMI":"ami-045deefcfb062fd2a", ## your ami id
            "INSTANCE_TYPE": "p2.xlarge", ## instance to deploy
            "REGION": "us-east-1", ## see above
            "SECURITY_GROUPS":"launch-wizard-34", ## security group to
                use when launching
            "IAM_ROLE":"pmd-s3-ssm", ## iam role that allows for ssm to
                send commands
            "KEY_NAME":"ta_testkey", ## secret key to use when connecting
                to instance for debug
            "WORKING_DIRECTORY":"~/bin", ## doesn't work
```

```
            "COMMAND":"ls; cd ../../../../home/ubuntu; bin/run.sh \"{}\"
                \"{}\" \"{}\"", ## Command to send to instance.
            "SHUTDOWN_BEHAVIOR":"terminate", ## routes poweroff
            "CONFIG":"config.yaml", ## not implmented
            "MISSING_CONFIG_ERROR":"We need a config file to analyze
                data.", ## not implemented
            "EXECUTION_TIMEOUT":180, ## timeout for lambda function
            "SSM_TIMEOUT":172000, ## timeout for ssm command
            "LOGDIR":"logs", ## name of logging subdirectory
            "OUTDIR":"results", ## name of output subdirectory
            "INDIR":"inputs", ## name of input subdirectory
            "LOGFILE":"lambda_log.txt" ## name of logfile subdirectory
        }
    },
    "UXData":{
    "Affiliates":[ ## One entry per lab
        {
            "AffiliateName":"carcealab2", ## name of lab's folder
            "UserNames":["user21","user22"], ## name of users in lab
            "PipelinePath":"", ## [not implemented ]
            "PipelineDir":"trackingfolder", ## [not implemented]
            "UserInput":true, ## [not implemented]- false => take output
                from the results folder of this group
            "ContactEmail":"The email we should notify regarding
                processing status." ## email to send notifications to.
        },
        {
            "AffiliateName":"testergroup2",
            "UserNames":["taiga2"],
            "PipelinePath":"",
            "PipelineDir":"trackingfolder",
            "UserInput":true,
            "ContactEmail":"The email we should notify regarding
                processing status."
        }
    ]
    }
}
```

There are still some features that have not been implemented yet- sending emails to users, a separate set of user buckets, and perhaps most importantly, checking for redundancy between user names across different pipelines, and handling the case of pre existing users. These are coming soon. The COMMAND field assumes that your executable is structured as in the above example regarding user input.

Once you have filled these out, navigate to the `template_utils` subdirectory, and run:

```
python config_handler.py path/to/your/pipeline/stack_config_template.json
```

You should see that this produces a `compiled_template.json` file in your pipeline stack directory. Then, run the following commands from inside your pipeline stack directory:

```
sam build -t compiled_template.json -m ../lambda_repo/requirements.txt

sam package --s3-bucket ctnsampackages --output-template-file
    compiled_packaged.yaml

sam deploy --template-file compiled_packaged.yaml --stack-name [name of
    your stack] --capabilites CAPABILITY_NAMED_IAM
```

You should then be able to view your new template under the cloudformation section of the console. Go to the outputs section to view generated user credentials.

4 Features

4.1 General Overview

4.1.1 Templates and Stacks

The most important concepts in AWS Cloudformation are templates and stacks. Briefly, a template is a JSON or YAML file that programatically declares all of the resources that you would like to be deployed together. Although we will be using a python template constructor, and may never need to write them directly, it is useful to know the structure of the baseline template too. Resources are declared as follows: have a stereotyped reference style, in the form of:

```
// JSON style
"ResourceName": {
    "ResourceType": str,
    "Properties": json object}

## Examples:
## An S3 Bucket
"PipelineMainBucket": {
    "Type": "AWS::S3::Bucket"
    "Properties": {
        "AccessControl": "Private",
        "BucketName": "dlcmonitored1"
    },

}
## A Lambda Function
"S3PutObjectFunction": {
    "Type": "AWS::Serverless::Function",
    "Properties": {
```

```
        "CodeUri": "../lambda_repo",
        "Description": "Puts Objects in S3",
        "Handler": "helper.handler_mkdir",
        "Role": {
            "Fn::GetAtt": [
                "S3MakePathRole",
                "Arn"
            ]
        },
        "Runtime": "python3.6",
        "Timeout": 30
    }
}
```

Each resource name must be alphanumeric. Note that each resource has two entries: a type, and a json object of properties. Each Resource Type declaration takes the following form:

```
service-provider::service-name::data-type-name

## Examples:
AWS::S3::Bucket (Declaring an AWS Bucket)
AWS::IAM::User (Declaring a new User)
AWS:Serverless::Function (Declaring a Lambda function)
```

The properties entry of each resource is entirely dependent on the resource type. Note that all resources accessible through cloudformation (which are most resources available on AWS) have a thorough Cloudformation Resource documentation as well.

Finally, in the Lambda function declaration, please note that under "role", there is a reference to another Resource "S3MakePathRole", not depicted here. This reference uses the GetAtt function to reference an attribute of another resource declared in the same template. There is some functionality to do computation inside templates to reference the properties of other resources: Notably we have Ref (reference name), GetAtt (reference attribute), Sub (substitute into string), and Select (get entry from list). Additionally, we can declare an additional "DependsOn" entry in our Cloudformation resources that ensures a given resource gets created after (and deleted before) another, to prevent resources being built with references to others that may not exist. In general, the computational capabilities of the template itself can be limited however, and it seems more efficient to relegate as much of computation as possible to the python template constructor, discussed later.

These templates are then compiled by AWS cloudformation into "stacks"; groups of resources that are defined to be deployed together. This has several benefits. First, as we saw above, there is no need for post hoc configuration of the resulting resources: Lambda functions can be linked to S3 buckets as event sources, users can be declared with access permissions to other resources in the

stack, ec2 instances can be set up with automatic monitoring. Second, resources can be deleted together, preventing pollution of the resource space and lingering costs (some complications here with Buckets- see Custom Resources below). Third, stacks can be developed as code: we can create test environments for stacks, pass environment parameters, and update with minimal interruptions: configurations on lambda functions can be changed, new users can be added with minimal (tbd how minimal) interruptions to existing workflow, examination of changes before deployment, and rollback behavior should something break. Finally, reused resource patterns can be modularized as helper stacks that can be referenced in a cloudformation template just like any other resource (as AWS::Cloudformation::Stack).

4.1.2 Console Interface

The most direct way to get started with the Cloudformation is through the AWS console. Cloudformation can be accessed just like any of the resources that it is used to manage. There are several example applications referenced in the setup guide that are useful here, as they come with templates that can be fed directly to the console. These have been downloaded in the `ctn_lambda` repo, and can be accessed under sam-app and aws-sam-ocr. These example implementations contain readme's which can also be accessed online:

`https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-getting-started-hello-world.html`.

Please note that you will see references to both AWS Cloudformation and AWS SAM (Serverless Application Model). AWS Cloudformation is a reference to the general Infrastructure-as-Code model used to build groups of resources together. AWS SAM refers to the specific use case where one of the resources in question is an AWS Lambda (Serverless) Function, and the suite of additional tools designed for development in this use case (see Testing)

Although everything else that we discuss here will be development through the console, it is worth noting that there are some unique and useful features of working with the console. In particular, upon uploading a template, there is a Designer that lets you visually check all of the resources that you have declared, and the dependencies and links between them (Figure 1). This can be very useful in cases where you have many resources and the dependencies between them are complex.

4.1.3 Direct Template Imports from Lambda

Another easy way to get started developing in Cloudformation is by making an import from AWS Lambda. If you go to an existing lambda function, you can click the "actions" dropdown menu, and hit "export function". Exporting as an AWS SAM deployment package will give you a self-contained code library, and downloading as an AWS SAM file will provide a .yaml Cloudformation template. You can then add resources to the resulting YAML template directly and redeploy your function into a stack with all of the benefits of doing so.

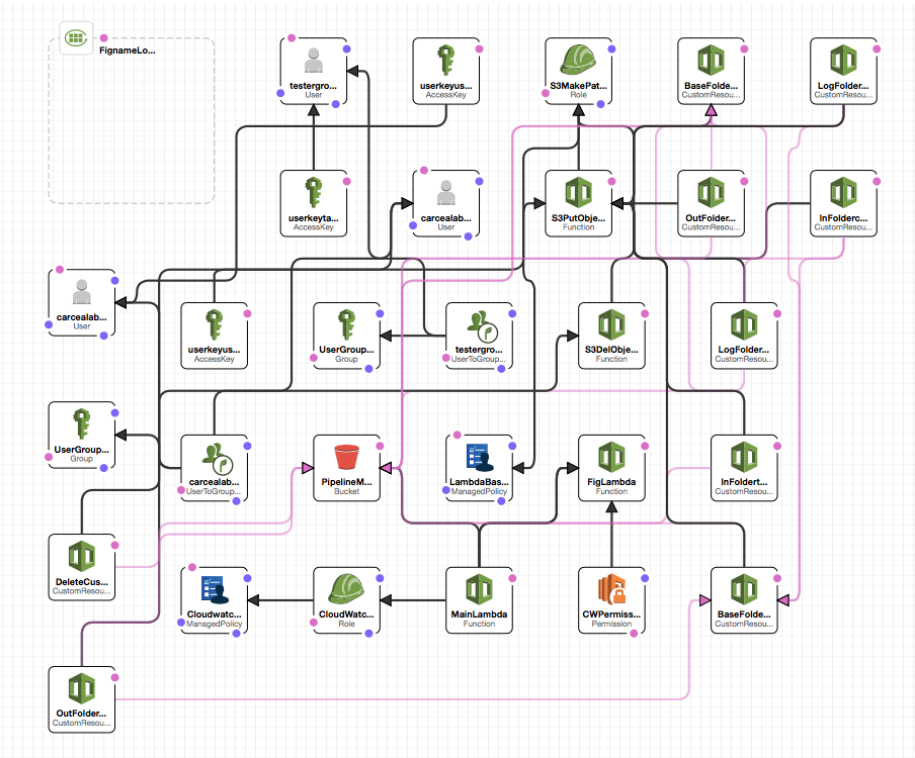

Figure 1: Designer rendering of template for DLC pipeline. Black arrows indicate linkages for functionality, pink arrows indicate infrastructure level dependencies (influencing build order of resources.)

4.1.4 Testing

One of these benefits of working with AWS SAM (Cloudformation for Lambda) is that there is a full framework to lambda functions as they interact with other resources in your stack locally, without first deploying them . To set this up, we need to install the AWS SAM CLI (different from the AWS CLI). Please find instructions to do so for your machine here: `https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install.html`. You will be asked to install docker as well.

Once you have a stack that contains a template referencing your lambda code, you can build a "deployment artifact" that allows for testing by running:

```
sam build -t template_name.json/yaml -m lambda_function_requirements.txt
```

Here, the requirements file contains pip packaged requirements of your lambda function declared in pip requirements style.

Then, you can manually trigger a call to your lambda function in a local execute environment (i.e. a docker container that mimics the lambda environment) by calling:

```
sam local invoke LambdaName [--event Event file] [--env_vars Environment
    Variables file]
```

Here LambdaName is the Resource Name given to your lambda function in the template. If your lambda function is triggered by an event [most are], you can generate a mock event files through the sam cli as well. Within the `ctn_lambda` repo, simulated events and local environment files are stored under `cloudformation_pipelines/template_utils/simevents`. Please add more as suits your specific use case. In some cases (simulating stack creation/deletion), it may be sufficient to trigger simulated events before ever deploying any resources. In others (triggering on uploads to a specific bucket), simulated calls can be rendered more expressive by first deploying a stack of resources, and referencing the deployed resources in the simulated events file. In order to deploy a resource, first package the deployment artifacts that were created with `sam build`:

```
sam package --s3-bucket ctnsampackages --output-template-file
    packagedtemplatefilename.yaml
```

This step will take your deployment artifacts (lambda function, libraries and template), and store them together in an s3 bucket (for our application, this is "ctnsampackages"). We will also write a secondary template file (default is yaml) that is functionally identical to the first, except that the lambda function will point not to the local lambda repo, but to the cloudformation bucket. The packaging process makes it easier to access existing stacks from remote locations, as the relevant deployment artifacts are located in the cloud, not on a local machine.

Finally, we can deploy our application to AWS by calling:

```
sam deploy --template-file packagedtemplatefilename.yaml --stack-name
    stack name --capabilities CAPABILITY_NAMED_IAM
```

The above states that we will deploy resources acconrding to our packaged template file (created in the previous step), the resulting stack name (note that if stack name already exists then an existing stack will be UPDATED by this step), and that Cloudformation has permissions to deploy named IAM roles when creating this stack (necessary for our application)

4.1.5 troposphere

Although writing JSON templates can be straightforward enough, there are still some serious limitations to parameter handling and programmatic declaration of templates (no iteration over a list, for example!) to warrant tools for working with these templates within a more powerful framework. One option for doing so is troposphere, a python package that mirrors much of the structure of Cloudformation. The documentation for troposphere proper can be sparse, but in general, if an AWS resource type AWS::SERVICE-NAME::DATA-TYPE-NAME exists in cloudformation, it can be imported in troposphere as troposphere.servicename.DataTypeName. These troposhere resource objects can be added to a template object by calling a method of the template object. The entirety of our use of troposphere can be found in the module `cloudformation_pipelines/template_utils/config_handler.py`. For more, please see the troposphere docs.

4.2 Ncap Template

The NCAP template uses troposphere to take a stack config file of relevant parameters (lambda function parameters, user parameters etc.) and generates a corresponding cloudformation template. The template parameters and usage are described in TLDR. The troposphere code that works with the stack config file generates the following resources:

- An S3 bucket to store data for the pipeline
- Group folders for each affiliate of the pipeline, Input/Output/Log folders in each group folder
- Group permissions to write into relevant input folders and download from relevant output folders only.
- Individual users in each group, Handler to add users to each group, and access credentials (access key + password) for each user
- A lambda function to handle ...submit.json file uploads and trigger EC2 instance spin up

- Lambda functions, custom resources and IAM Policies and Roles to handle folder creation and deletion
- A lambda function, Log Group, Policies and Permissions to handle automatic creation of cloudwatch events rules upon boot of a new EC2 instance.

4.3 Custom Resources

Worth discussing independently for headaches.

4.4 Automated Monitoring

Worth discussing independently for headaches.

4.5 TODO

- AMI updating
- Detailed logging
- handle preexisting users
- handle output routing
- a second front end layer for users. Developed as an independent stack. Discussed in meeting 8/14 regarding implementation details. submissions,uploads,downloads
- make sam command line deployment refer to config
- programmatic deployment for non test cases
- cleanup of cloudwatch rules, volumes, launch wizards -parameter handling – limitations