A Report on the Deployment of Numerical Weather Prediction Services in the Amazon Cloud

Don Morton

Boreal Scientific Computing

Fairbanks, Alaska USA

06 July 2022

Executive Summary

The primary goal of this project was to investigate potential ways that cloud resources can be useful in a NWP environment, with special emphasis in leveraging already-existing DTC NWP Docker containers. Many of the issues in using cloud resources for NWP were deemed unexplored, and a deliverable was proposed to build a NWP client-server-like environment, with all services launched on-demand in the AWS cloud. By pursuing this, it was hoped that we could begin to "get our feet wet" and begin to build a foundation for further work.

The idea was to implement each of the WRF components - ungrib, geogrid, metgrid, real and wrf - as independent, standalone cloud services, loosely coupled in flexible workflows by having all outputs staged in S3, positioned to serve as inputs for other components (for example metgrid service output would be stored in a designated S3 location, and this location would be provided to the real service as required input).

The good news is that the basic implementation was successfully constructed and documented, and for the persevering user should be accessible as alpha software. The basic objectives of the project were met, but, as is often the case, this drove many questions and problems. Much was learned, and before a next iteration takes place there will be some critical evaluation of what was good and what wasn't.

This final report begins by discussing historical motivations for the project that influenced the design goals, followed by many of the tradeoffs considered in realization of the ideal vision versus what might be more realistic. The next section of the report outlines the actual implementation and usage (which is discussed in more detail in an accompanying online HOWTO), and summarizes the good, the bad, and the ugly, presenting ideas on a possible next iteration of the work.

Introduction

The inspiration for this activity dates back to the initial surge of the Internet in the early 1990's, when NSF funded five supercomputing centers and participated in the development of the vision of a "computational grid" in which, among other things, scientists would be able to launch complex computer simulations from their local desktop through an interface that would hide the details of how and where it was being executed. The computational grid infrastructure exploded over the years, and NWP codes like MM5 and then WRF became widely used, but they were always challenging for many to use. As someone who used to present WRF tutorials to local scientists and students, I was impressed with the great talent and enthusiasm for performing NWP simulations for a large variety of purposes, but dismayed at how quickly participants got distracted by their own work as soon as the tutorials were over, and lost retention of the skills they had gained. This impressed upon me the need to make the NWP simulation process easy and intuitive for the typical Joe/Jane Sixpack atmospheric scientist. Without it, most of them would never have the opportunity to perform the modeling activities that excited them. In 2008 I had a student build a web interface for running simple WRF simulations on the supercomputer at the Arctic Region Supercomputing Center, but this was highly experimental, and was restricted due to various security concerns, potential unavailability of resources when needed, and was limited in its workflow flexibility.

Additional inspiration for the activity came from increasing experiences in the development of real-time and research WRF workflows for a wide variety of scenarios. Although often somewhat similar, each scenario was different enough from others that they required custom code within the workflows. It started to become apparent that thinking of WRF simulations (and their various pre- and post-processing actions) might be more efficiently thought of as a collection of loosely-coupled, distributed components that could be interconnected in various ways depending on the scenario. Some of this was explored and implemented within a DTC visitor project.

Finally, another driving inspiration has come from requests by scientists to help with the deployment of on-demand, custom WRF simulations for disaster response planning. For example, volcanic eruptions may spew large amounts of ash into the sky, severely disrupting aviation activities, and the ability to focus a sudden set of forecasts on such an event may aid in the reduction of airspace that needs to be closed for safety reasons (see Figure 1).

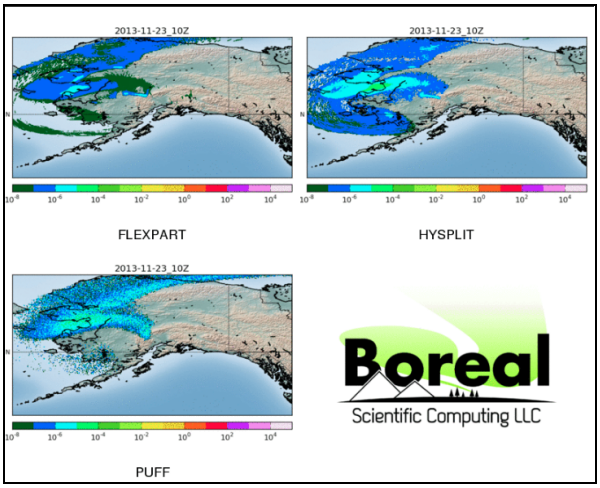

Figure 1. **Multimodel volcanic simulation**

Or, hints from a global forecast of potentially damaging storms in an area of interest might prompt the need to generate more targeted, higher resolution forecasts for a few days, which might guide the decision of staging resources before a storm event occurs. What these and many other scenarios have in common is the need to rapidly assemble a set of important, custom numerical forecasts and launch them with the expectation that they will run immediately to provide timely results. During this time there will be a need for an unprecedented amount of computational resources and then, once the event is over, there may not be another need for weeks or months.

So, taking all of the above into consideration, what we have is a need for atmospheric and computational scientists to be able to enjoy the realization of the “computational grid” vision in which custom NWP simulations can be easily launched as needed with the expectation of a reasonable turn-around time, without needing to worry about where the simulations are running or the details of how they are running. For most of my career, realization of this kind of need required access to supercomputers for extensive development, testing, and the occasional real-world deployment. Given the high costs of these supercomputing resources it was always necessary to run them at as high a capacity as possible and not let them sit idle. This meant users competing with each other for the limited resources, making the above vision very difficult to realize.

Now, with the availability of on-demand computational resources that are paid for only as-needed, and the ability to construct a wide variety of custom computing environments, it seems that all of the pieces to implement the vision are available to anybody with a credit card, and the real effort in making all of this work is to build the “plumbing” that puts these pieces together in a workflow as needed.

Hence, the driving objective of this work was to make a first attempt to build a set of loosely-coupled NWP services, available on-demand for custom scenarios, that could be connected together in flexible workflows to achieve research and operational objectives.

One of the greatest challenges in this work has been the desire to build cloud services for tasks that are not suited for the classic client-server paradigm that cloud proponents advocate. Whereas many of the client-server cloud implementations are RESTful and/or are short and bursty, the NWP services often need to run for minutes or hours. It is unreasonable to consider implementation of individual transactions of such length, so we need to think in terms of non-blocking, asynchronous processes in which the actual launch of the service is considered to be a short, synchronous transaction, which is then left to its own devices (I like to think of this as being analogous to shooting off a space probe - once it launches, for the most part the best we can do is monitor it and make some remote adjustments), maintaining the ability to be queried by outside clients.

The work was intensely tedious - not what was hoped for, but not a total surprise. There are a number of fine points in the AWS and Docker combined environments that make things that look like they should be straightforward not so straightforward. Dealing with these often-undocumented oddities took a great deal of time and investigative effort, requiring code to assume much more robustness that I had originally hoped would be built into the AWS environment.

Ultimately, the project succeeded in its goal of building standalone services for each of the WPS/WRF components, but with many caveats and questions to serve as a foundation for a next iteration. In this report I outline some of the key points of the implementation, and I summarise with a discussion of The Good, The Bad and The Ugly, with some thoughts on future directions. Some extensive user documentation has been drafted, available at *https://borealscicomp.com/NWPAAS/doc/*

Implementation Overview

A vision of the NWP loosely-coupled services paradigm is illustrated in Figure 2,

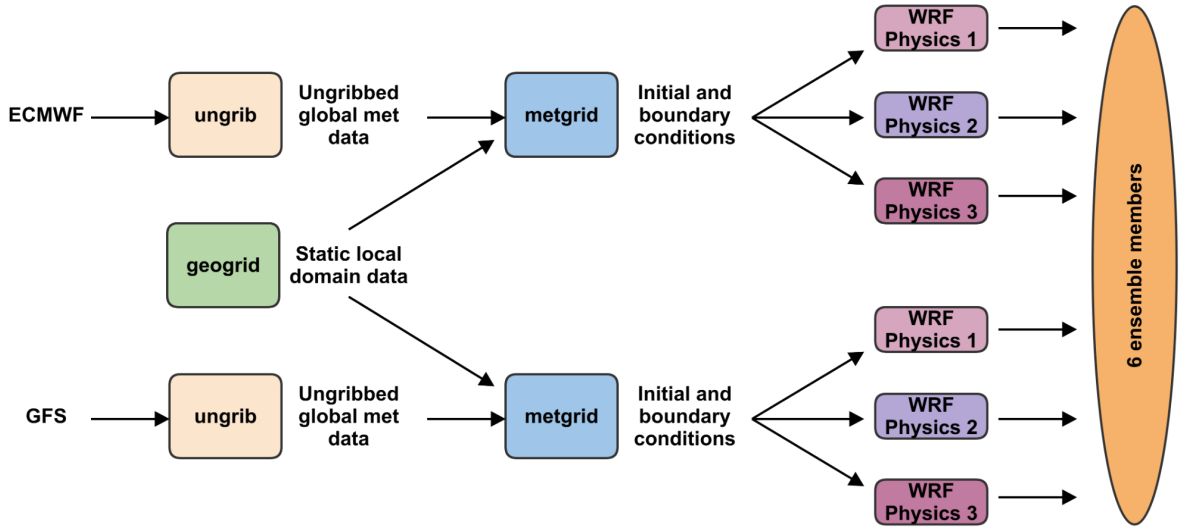

Figure 2. **Loosely coupled services workflow**

depicting the use of standard WPS/WRF services for creating an ensemble forecast generated from two different gridded initialization datasets and an assortment of WRF physics options. Each service is a standalone entity and can be run in isolation for development, debugging, testing, or to facilitate various workflow configurations. Coupling between the services occurs through specification of inputs and outputs which, in an Amazon Web Services (AWS) environment, we would locate in the Simple Storage Service (S3). Figure 3 depicts a single NWP component service in the AWS cloud

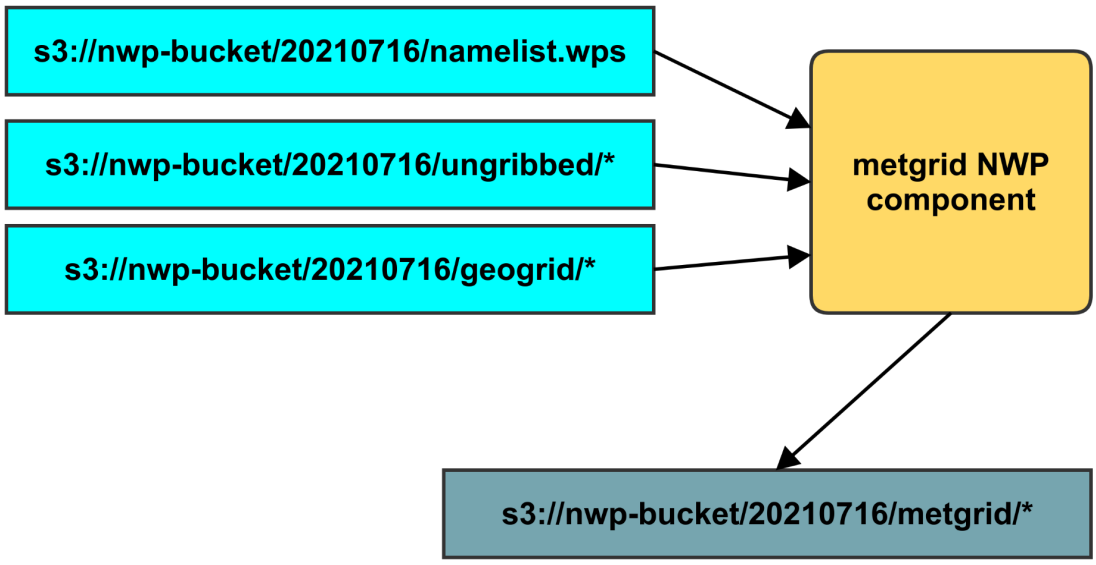

Figure 3. **Single NWP component service in AWS cloud**

in which the service, once started, expects to find all necessary input in the specified S3 locations, and expects to stage its output to another S3 location, which will then be available as input to other services as needed.

An idealised implementation (Figure 4) would have had these services invoked via RESTful interfaces, where all requests and responses are driven through HTTP via Uniform Resource Identifier (URI - analogous to a URL) requests.

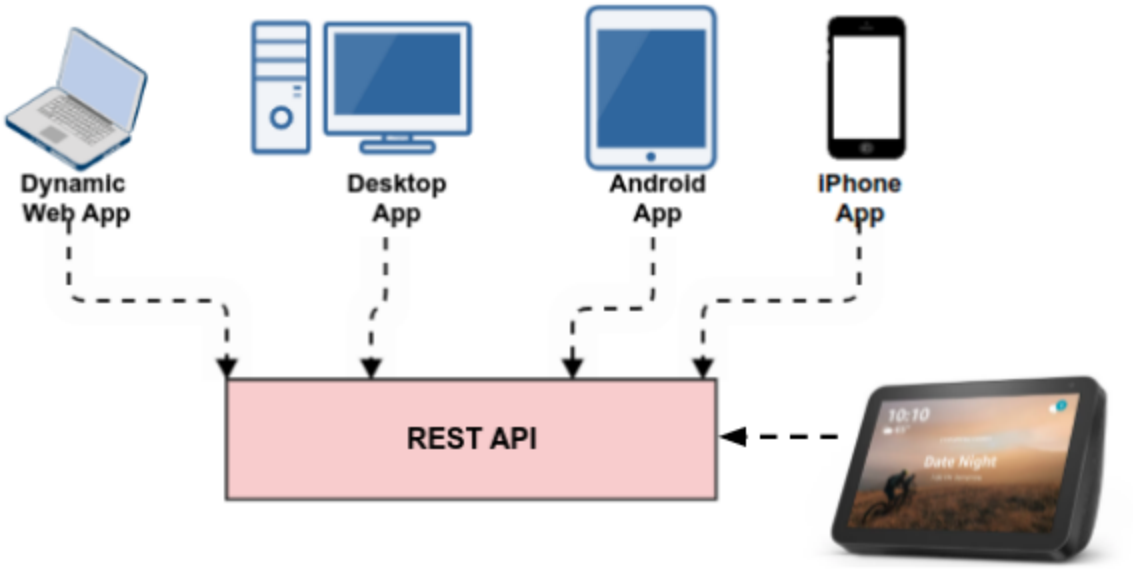

Figure 4. **Idealized RESTful implementation.**

This is a useful and popular paradigm to use for typical cloud services, and advantageous in that a well-designed set of RESTful endpoints allows for a variety of devices and programming environments to participate as clients, merely by issuing the standard URIs and handling the standard responses. Under such a paradigm, an idealized - but unrealistic - RESTful interaction for the metgrid service might look like Figure 5

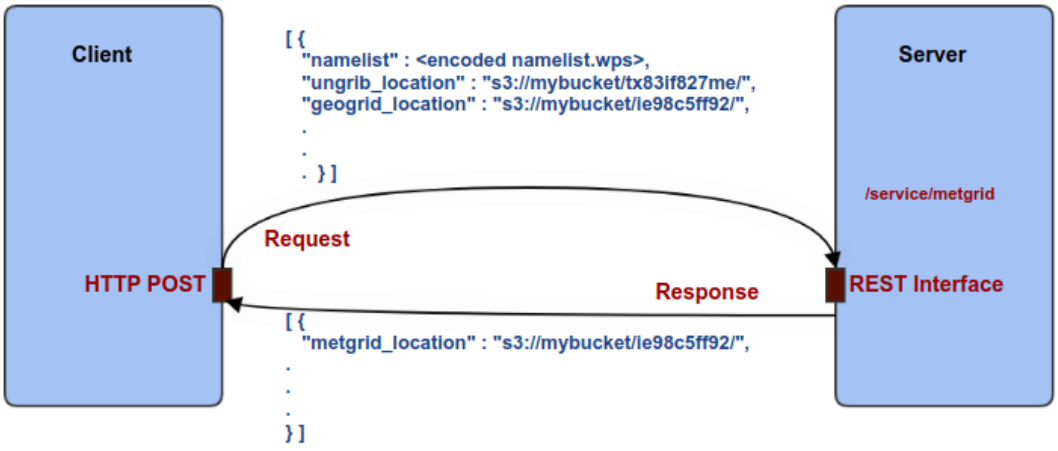

Figure 5. **Idealized RESTful metgrid interaction**

and a hypothetical workflow (Figure 6) might be created simply by programming the appropriate chain of requests and response handling

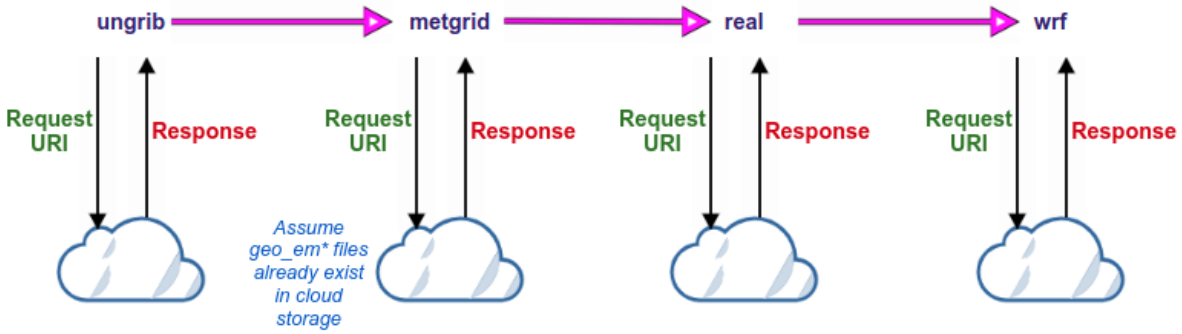

Figure 6. **Hypothetical workflow of chained requests**

Unfortunately, as hinted above, the large-scale, long-running NWP services do not immediately fit the assumptions of a RESTful model. The primary issue is that NWP services will typically take a long time to run (they can even take a long time to launch, due to their size), and so it isn't plausible to maintain a single TCP connection between client and service while waiting for launch and completion. There are far too many things that can go wrong during this duration of minutes or even hours, and trying to synchronize to a safe state after loss of a connection or service would be very difficult. Also, it's not feasible to simply leave services running forever so that they're readily available when needed - this is expensive and wasteful, and not in the spirit of the on-demand burst-processing that we want from the cloud.

Again, I equate the process of launching and executing an NWP service in the cloud with that of launching and using a space probe. Once the NWP service is launched, it's out there "somewhere" with only an IP address as a "tether." Simple events that might seem somewhat trivial on a local machine - problem finding an input file, syntax error in a namelist, insufficient disc space or memory - can result in the entire remote process dying, or needing to be recovered somehow. One of the most frustrating events that can take place is for an hours-long service to complete, only to fail somehow as it tries to stage output files to an S3 location. Finally, we also need to consider complete and reliable termination of a rogue service, lest we end up paying for it for a long period of time, without even realising it.

It should be emphasised here that these are the "little" problems that many developers don't address immediately. When they emerge on local systems, they can often be corrected through bug fixes, but when they emerge on remote systems - on those "space probes" they become exponentially more difficult to handle. But, we need a way to understand when problems creep in, and some means to deal with them, even if dealing with them means an informative, graceful termination of the process.

A fundamental design goal, then, has been to find a way to launch our service and then, like a space probe, have a way to communicate with it while it's working, so that we can, at a

minimum, query it at any given time to gain current situational awareness. In effect, this means we are launching, overall, a long-running asynchronous process, but always interacting with it in a synchronous, short-message, client-server way. This requires a robust and complex client environment to not only make the short requests, but to coordinate and make decisions based on what it learns about the current state of the remote process.

The idealised vision of the implementation (see Figure 7) - which was deemed to be too complex to build at this time - has most of the complexity of the “backend client” running as a process (a service, in fact!) inside AWS, accessible through the AWS API Gateway, which would allow external clients to drive it through properly-formed URIs.

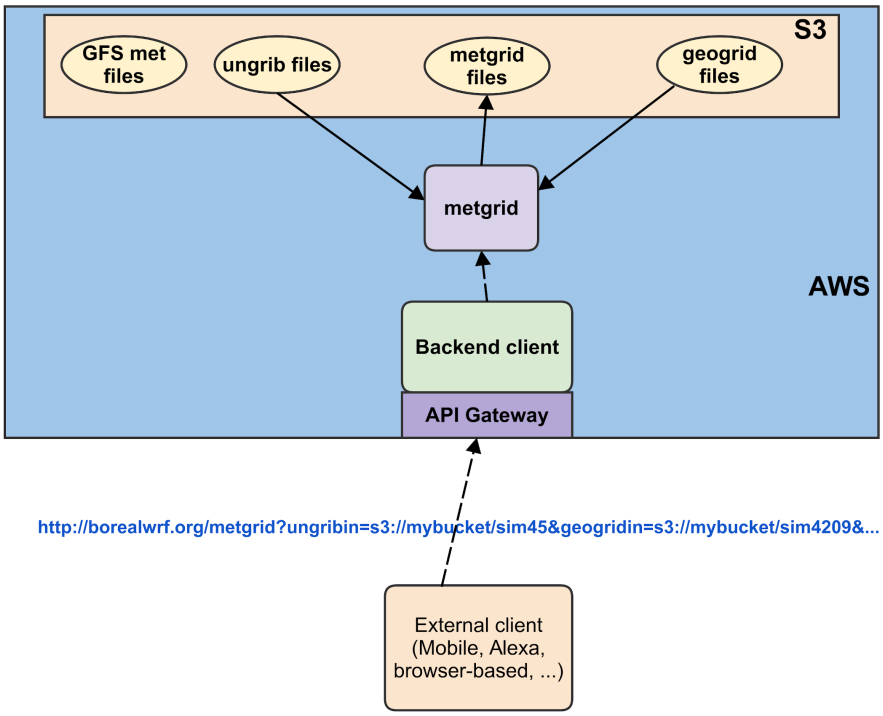

Figure 7. **Idealized implementation of NWP services**

The backend client, inside of AWS, would take care of the launching of the NWP service, as well as the ongoing synchronous communication with the service as it executes, possibly for hours. The external client, then, would be somewhat simpler, relying primarily on the generation of properly-formed URIs to make requests, through the API Gateway, to the backend client inside of AWS, and to deal with responses. Presumably, the external client would behave - in a simpler way - like the backend client in that it would need to start the NWP service, and would then need to check on its status now and then.

This two-client system would be cleaner from the perspective of someone building the external clients, but was deemed overly complex for immediate implementation, especially in a scenario

where so much is still exploratory. Therefore, it was decided to pull the backend client fully outside of AWS (Figure 8)

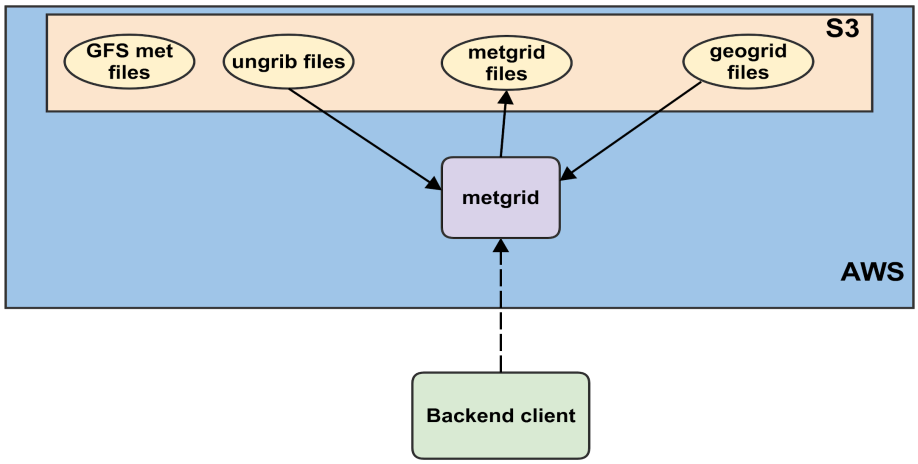

Figure 8. **Implemented backend outside of AWS**

In this scenario, a single, complex, external client is responsible for all direct interactions - including launch, query and termination - with the NWP service. This approach was also deemed preferable because it puts all of the client work - so much of it tedious and exploratory - in one place on a local development machine, making it more efficient for extensive iterative development.

The Implementation

Due to the time and resource requirements, the use of “serverless” types of AWS resources was not practical, and it was recognised that in some form, we would need to deal with management of EC2 instances in one form or another. Although the use of Docker wasn’t a part of the earliest visions, the fact that DTC had already embarked on a program to provision publicly available Docker images for the full suite of NWP preprocessing, modeling and postprocessing components, influenced the ultimate decision to base the AWS NWP services on these Docker containers running in EC2 instances. AWS provides the Elastic Container Services (ECS) environment for managing collections of containers within EC2 instances, and there are plenty of tutorials and examples that guide new users through the process. However, again, the NWP needs aren’t the same as the “typical” synchronous activities for which ECS is designed. In retrospect, the use of Docker containers and the ECS environment of AWS offered a number of advantages, though it was tedious to map our NWP needs into this not-perfectly-robust environment.

The general approach for implementing a NWP service has been to think in terms of a single ECS task running in an EC2 instance. The ECS task is defined by two or more Docker containers linked and invoked in specific ways as defined in the task definition. Figure 9 is a

representation of the ECS task for the geogrid NWP service, containing three Docker containers (most of the NWP services will require only two containers):

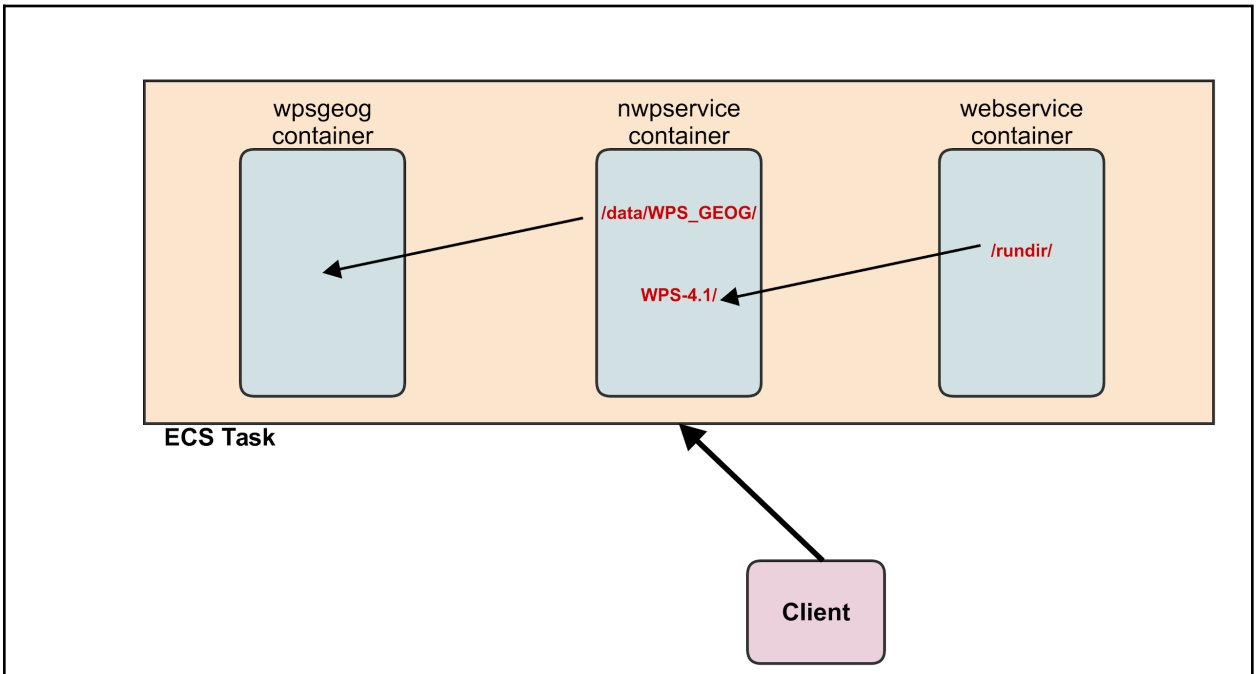

Figure 9. **ECS task consisting of three Docker containers**

- The nwpservice container manages the processes for setting up and running the geogrid.exe WPS application. Almost all of the container content is provided in the publicly available DTC WPS/WRF container, and a little bit of additional functionality was added to this base
    - A Python program which, when invoked with appropriate arguments, will facilitate the fetching of needed inputs from specified locations (primarily S3), specific configuration of the runtime environment, execution of geogrid.exe, and staging of the output to specified locations (primarily S3). Additionally, the extra layer of support sets up a monitoring and logging environment that other processes can query in order to gain situational awareness on the progression of the service.
- The wpsgeog container is also provided publicly by DTC and is used without any enhancements. It is a data container, storing the large collection of geog files needed by geogrid. Within the ECS task definition itself is a specification to link the /data/WPS_GEOG/ directory of the nwpservice container to this data. The ability to bring this container in when needed (which is only for the geogrid service) allows us to keep the storage requirements of the other services substantially reduced.
- The webservice container is an entirely custom container running a simple CherryPy webservice, with a directory that the task specification links to a specified directory in the nwpservice container. The idea is that URI endpoints in this container will generally

result in an examination of files within the nwpservice container, and use that information to form a response.

Some design philosophies

- The services should be as “dumb” as possible. The philosophy is that if a potential problem is encountered, the client should be able to know about it and react accordingly. At least at this early stage, adding in logic and intelligence to potentially correct problems would lead to great code complexity. Put another way, policies are determined outside of the containers, and the policies that have been made outside are implemented within the containers through choice of arguments passed to them.
- Like any service, we should always assume that they can fail at any time. The question then becomes, “how does the client know that something failed and how will it respond?” Particularly in the case of paid-for cloud resources, it is of the utmost importance that we don’t wait endlessly for something to happen, only to finally give up because we gave up waiting. And, we must be able to adjust our workflows so that we don’t try to proceed beyond the point of failure.

With our approach of a single ECS task composed of NWP container(s) and a webservice container, we provide the client with a communications link to determine what’s going on in the task. When everything is working smoothly, the webservice container has file access to the run directory of the NWP container, and our enhancements to the NWP containers include the generation of a JSON file with log entries after each step of the service routines. Additionally, the NWP service retains the WPS/WRF logs, as well as files containing memory and cpu usage measured during execution.

In the big picture, the client launches the ECS task, learns the IP address of the EC2 instance that it’s running on, and begins to periodically query the web service on a known port number (Figure 10). The web query communication is the only way that the client will communicate - indirectly - with the web services, and it’s solely in “listening” mode, trying to understand the status of things. The client makes HTTP requests such as that shown in Figure 11 (please note that the localhost hostname in the URLs reflect a local testing mode and, in deployment, would be the IP address of the remote EC2 instance running the ECS task). Each NWP service has a set of well-known (to the client) tasks (e.g. UNGRIB_METFILE_STAGE, UNGRIB_OUTPUT_STAGE) with states that a task can be in (e.g. RUNNING, COMPLETE) and these are the most important elements of a query response. Others, such as log entry time and comments may be used for more advanced client responses. For example, by looking at the time of the last log entry the client can make a decision as to whether it has been waiting for an unacceptable amount of time. In this way, the client is making a decision instead of forcing that complexity on to the service itself.

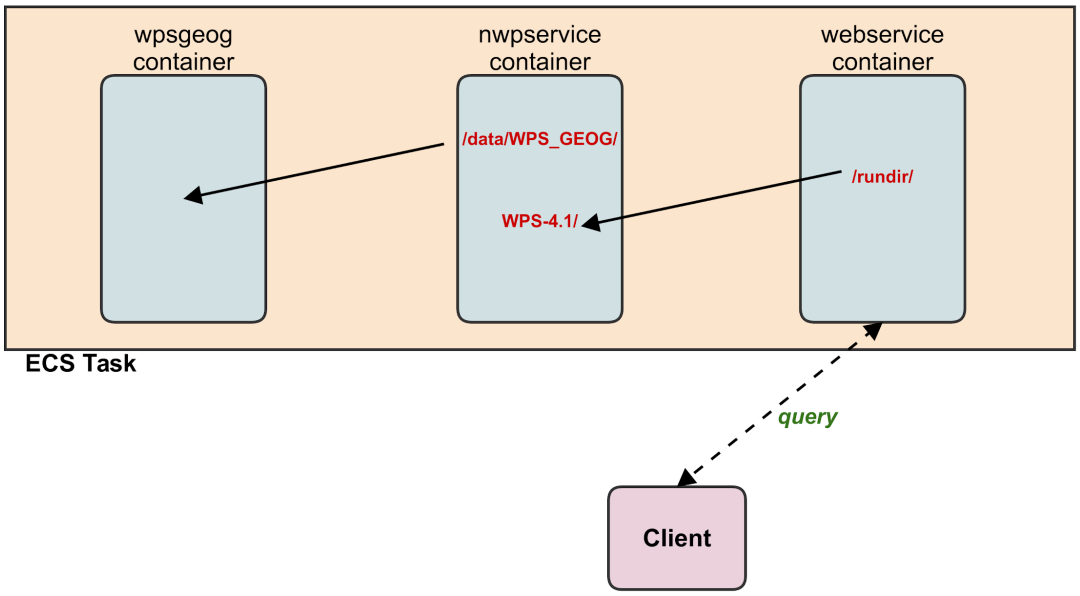

Figure 10. **HTTP query is sole method for communicating with ECS task**

Sample queries to the web service

Request URL: http://localhost:8080/status_log


```
'status_log': [{'state': 'RUNNING', 'status_report_time':
1596849541.762668, 'messages': ['Started UngribService...'],
'task': 'UNGRIB_PRECHECK'}, {'state': 'SUCCESS',
'status_report_time': 1596849542.757152, 'messages': ['namelist_wps
is present...', 'mettype is valid...', 'metfiles subdir
created...', 's3output PUT test completed...', 's3output DELETE
test completed...'], 'task': 'UNGRIB_PRECHECK'}, {'state':
'RUNNING', 'status_report_time': 1596849542.757803, 'messages':
['Started metfile staging...'], 'task': 'UNGRIB_METFILE_STAGE'},

...

{'state': 'RUNNING', 'status_report_time': 1596849548.646723,
'messages': ['Staging ungribbed files to
s3://boreal-temporary-01days/ungrib_test_20200806060000_73b11cff-9a
e6-4abc-8a12-fafb51626810'], 'task': 'UNGRIB_OUTPUT_STAGE'},
{'state': 'COMPLETE', 'status_report_time': 1596849601.78797,
'messages': ['Staging ungribbed files to s3 completed BUT NOT
VERIFIED'], 'task': 'UNGRIB_OUTPUT_STAGE'}]}
```

Figure 11. **Response of entire service status logs from a client query**

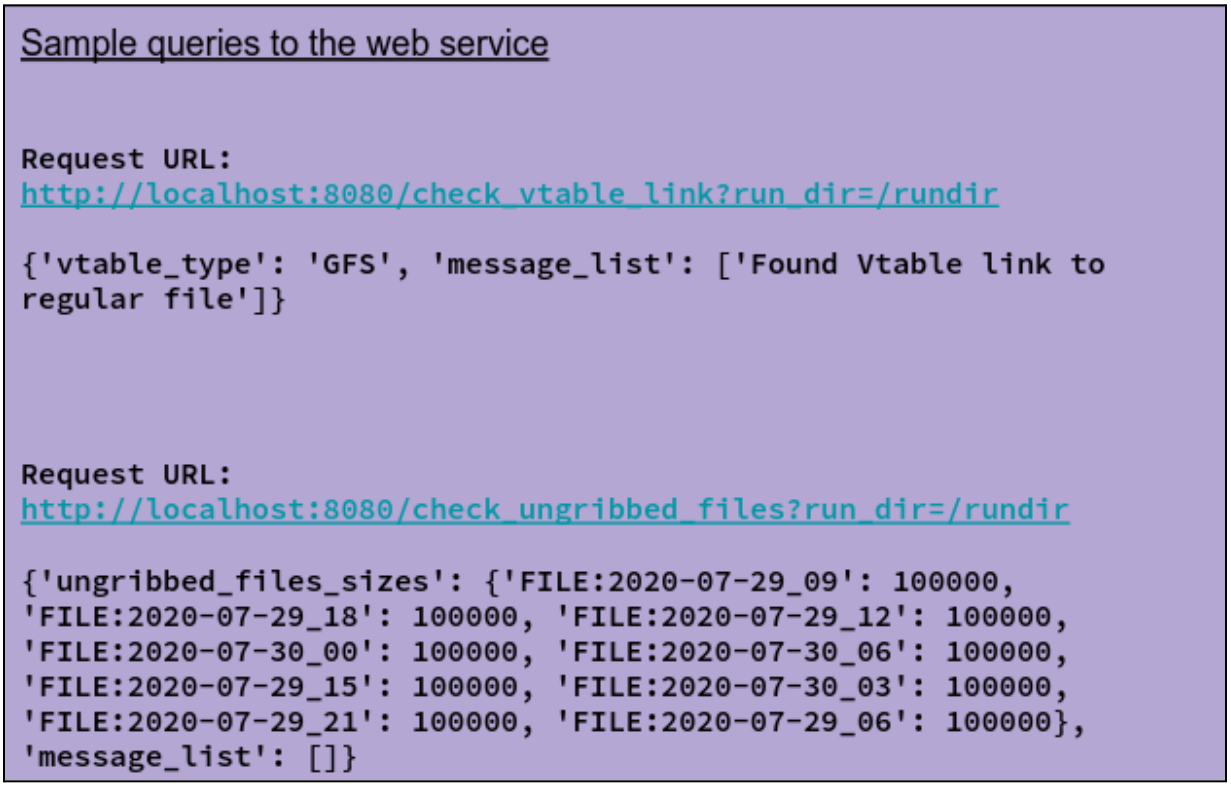

Figure 12. **Responses from client query of NWP service (testing mode)**

The client, through its periodic queries (Figure 12), can eventually find that the expected output files have been produced and staged somewhere in S3, where they become available for a downstream stage. At this point, the client will typically choose to tear-down the AWS resources, but again, this is a policy decision, up to the client, not the service.

The design of this approach is intended to provide at least some situational awareness for the client, no matter what happens.

- The web service container is set up in the ECS task to run, “no matter what.” Even after the NWP service has finished, as long as the ECS task has not been removed, the web service should remain intact. This means that the client can query the last status log entry repeatedly. The client can determine if the service finished successfully and, if not, the entry will provide the time of last update as well as the last task and state. If this entry is not updated after a reasonable number of period queries, then the client may decide to give up and kill the resources - a policy decision made by the client based on the “dumb” activities of the server.

- If something goes wrong with the entire ECS task, then the webservice would not be able to respond to queries as expected, and this at least provides potentially useful information to the client, allowing it to decide whether to give the service more time, or give up and delete the AWS resources.

So, although not perfect, the design allows us to achieve the goals of keeping the NWP service as simple as possible, by having it focus only on running a basic task and reporting as it does so. We avoid the complexity of having the service decide on policies (which have a way of changing over time) in the event that something goes wrong, and choose to have it simply report it in a way that a web query can reveal the event to a requesting client, and then the client can decide what to do.

A broader view of this loosely-coupled client-service interface is illustrated in Figure 13

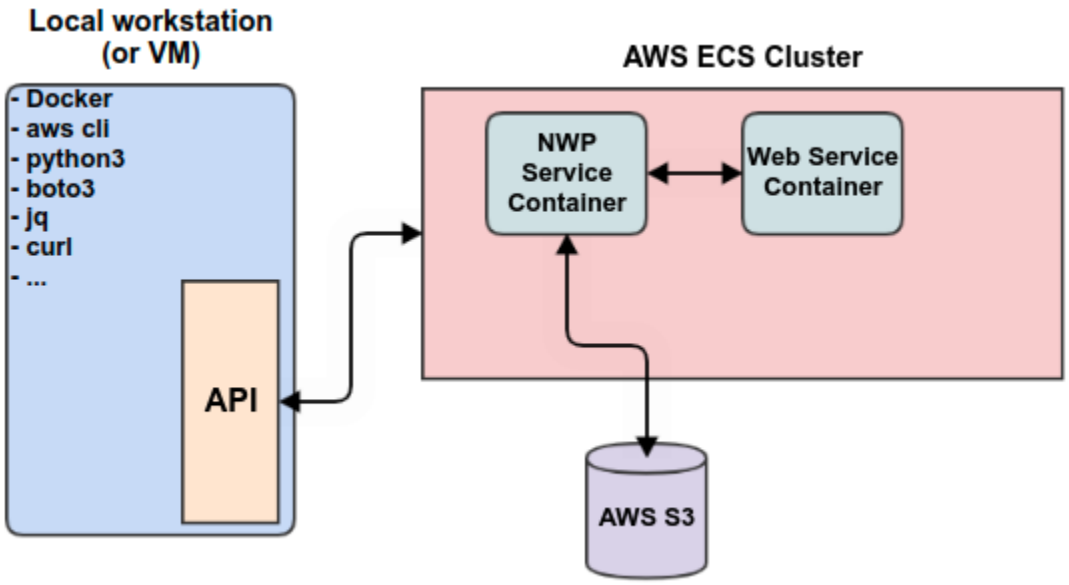

Figure 13. **General overview of the loosely-coupled client-service interface**

The client, as currently implemented, has few requirements - generally a Python 3 environment with boto3 for Python access to AWS resources and ***jq*** and ***curl*** for interacting with the JSON request/reply environment of the webservice. The Docker and awscli client resources are not needed for driving the NWP services, and are in the client software for development and debugging.

Through a Python-based API, client programs can launch the entire service structure in AWS, with general queries to the web service taking place within the API layer, not by the client application itself (though this is possible, if desired). An example of a no-frill “nothing will go wrong” Python prototype is provided below. It’s somewhat tedious, in that we need to ensure

beforehand that the various AWS resources have been created - things like S3 buckets, security groups, etc., and that the images for the Docker containers are available in Dockerhub.

```
#!/usr/bin/env python3

import pprint
import sys
import traceback

import api.geogrid_task

NAMELIST_WPS_PATH = "/home/dtcuser/git/nwp-as-a-service/wps_wrf/src/workflow_drivers/namelist.wps"

S3_INSCRATCH_BUCKET = 'boreal-temporary-01days'
S3_OUTPUT_BUCKET = 'boreal-temporary-01days'

NWP_DOCKERHUB_IMAGE = 'borealscicomp/wrf_wps_services:2021_06_27_v1'
WEBSERVICE_DOCKERHUB_IMAGE = 'borealscicomp/simwebservice:2021_04_08_v1'
WPS_GEOG_DOCKERHUB_IMAGE = 'dtcenter/wps_geog'

EC2_INSTANCE_TYPE = 'm5.2xlarge'     # 8 vcpus, 32 GiB

VPC_ID = 'vpc-7f99af17'
SUBNET_ID = "subnet-7099af18"    # us-west-2c
SECURITY_GROUP_ID = "sg-0306ce1350270e173"  # Allow ssh (22) and http (8080) inputs
TASK_HTTP_PORT = 8080

INSTANCE_PROFILE_NAME = 'ecstest1'

AWSLOGS_GROUP = 'boreal-dtc-cloudwrf'

LOG_LEVEL = 'INFO' # This is for client-side

def main():

    print('Creating GeogridTask...')
    Geogrid = api.geogrid_task.GeogridTask(
                    namelist_wps_path=NAMELIST_WPS_PATH,
                    s3_scratch_bucket=S3_INSCRATCH_BUCKET,
                    s3_output_bucket=S3_OUTPUT_BUCKET,
                    subnet_id=SUBNET_ID,
                    security_group_id=SECURITY_GROUP_ID,
                    task_http_port=TASK_HTTP_PORT,
                    instance_profile_name=INSTANCE_PROFILE_NAME,
                    docker_image_nwp=NWP_DOCKERHUB_IMAGE,
                    docker_image_geog=WPS_GEOG_DOCKERHUB_IMAGE,
                    docker_image_web=WEBSERVICE_DOCKERHUB_IMAGE,
                    awslogs_group=AWSLOGS_GROUP,
                    ec2_instance_type=EC2_INSTANCE_TYPE,
                    log_level=LOG_LEVEL
    )
    print('Geogrid init completed - ready to go...')

    setup_info = Geogrid.setup()
    print('Geogrid setup() completed...')
```

```
    start_run_info = Geogrid.start_run()
    print('Geogrid start_run() completed...')


    Geogrid.monitor_until_complete()
    print('Geogrid monitor_until_complete() completed...')

    output_info_dict = Geogrid.output_info()

    print('Output information dict:')
    pprint.pprint(output_info_dict)
    print("\n")
    sys.stdout.flush()

    cleanup_status = Geogrid.cleanup_aws_resources()
    print('Cleanup of AWS resources completed...')

if __name__=="__main__":
    main()
```

Excerpts from the log of the local client process are displayed below. We start with extensive error checking of various arguments and setting up the data structures which can be set up quickly, trying as much as possible to catch any errors before we get into actually trying to launch something in the cloud. In this example, it takes approximately four minutes from the time we start an EC2 instance to the time that we actually start running geogrid. Polling is implemented for any of the actions that take more than a couple of seconds so that we can "mask" our overall asynchronous activities behind a series of synchronous calls which are better suited for client/server interactions.

```
$ ./geogrid_demo_nofrills.py
Creating GeogridTask...
2021-06-28 02:00:48,021  - INFO - awsclient.py:credentials_check:101 --> success: True
2021-06-28 02:00:53,699  - INFO - geogrid_task.py:_check_dockerhub_image:1402 --> Found docker image:
https://hub.docker.com/v2/repositories/borealscicomp/wrf_wps_services/tags/2021_06_27_v1
2021-06-28 02:00:54,530  - INFO - geogrid_task.py:_check_dockerhub_image:1402 --> Found docker image:
https://hub.docker.com/v2/repositories/dtcenter/wps_geog
2021-06-28 02:00:55,135  - INFO - geogrid_task.py:_check_dockerhub_image:1402 --> Found docker image:
https://hub.docker.com/v2/repositories/borealscicomp/simwebservice/tags/2021_04_08_v1
2021-06-28 02:00:55,587  - INFO - geogrid_task.py:__init__:635 --> Creating ECS cluster:
2021-06-28-02-00-55_dot_135328_geogrid_task
2021-06-28 02:00:56,472  - INFO - geogrid_task.py:__init__:651 --> ECS container instance AMI id
(linux2): ami-0a51409a409fbc030
2021-06-28 02:00:56,472  - INFO - geogrid_task.py:__init__:669 --> Setting up task volume def for
wpsgeog container...
2021-06-28 02:00:56,472  - INFO - geogrid_task.py:__init__:705 --> Setting up task defs for nwp
container...
2021-06-28 02:00:56,472  - INFO - geogrid_task.py:__init__:803 --> Setting up task defs for webservice
container...
2021-06-28 02:00:56,778  - INFO - geogrid_task.py:__init__:866 --> Created taskdef_arn:
arn:aws:ecs:us-west-2:325425669423:task-definition/2021-06-28-02-00-55_dot_135328_geogrid_taskdefinition
:1
Geogrid init completed - ready to go...
.
```

```
.
.
2021-06-28 02:00:56,782  - INFO - geogrid_task.py:setup:1076 --> Starting EC2 instance, which will
become an ECS container instance...
2021-06-28 02:00:58,102  - INFO - geogrid_task.py:setup:1091 --> Waiting for EC2/ECS instance startup.
Timeout secs: 180
2021-06-28 02:01:28,571  - INFO - ecs_functions.py:ecs_instance_wait:386 --> 30 seconds --> EC2
instance_state: running
2021-06-28 02:01:28,971  - INFO - geogrid_task.py:setup:1110 --> Waiting for a container instance ARN
before proceeding.  Timeout secs: 180
2021-06-28 02:01:58,999  - INFO - ecs_functions.py:container_instance_arn_wait:503 --> Waiting for a
container instance ARN, 30 seconds...
Geogrid setup() completed...
2021-06-28 02:01:59,780  - INFO - geogrid_task.py:start_run:1154 --> Waiting for task startup.  Timeout
secs: 240
2021-06-28 02:02:30,085  - INFO - ecs_functions.py:task_wait:632 --> 30 seconds --> task_status: PENDING
2021-06-28 02:03:00,483  - INFO - ecs_functions.py:task_wait:632 --> 60 seconds --> task_status: PENDING
2021-06-28 02:03:30,885  - INFO - ecs_functions.py:task_wait:632 --> 90 seconds --> task_status: PENDING
2021-06-28 02:04:01,334  - INFO - ecs_functions.py:task_wait:632 --> 120 seconds --> task_status:
RUNNING
2021-06-28 02:04:01,339  - INFO - geogrid_task.py:start_run:1169 --> Task started successfully.  Task
ARN:
arn:aws:ecs:us-west-2:325425669423:task/2021-06-28-02-00-55_dot_135328_geogrid_task/d3a68800322646a68224
def7118ede01
Geogrid start_run() completed...
.
.
.
2021-06-28 02:04:01,460  - INFO - geogrid_task.py:monitor_until_complete:1210 --> Service status: time:
2021-06-28 02:03:56 UTC, task: GEOGRID_RUN, state: RUNNING
2021-06-28 02:04:31,613  - INFO - geogrid_task.py:monitor_until_complete:1210 --> Service status: time:
2021-06-28 02:04:02 UTC, task: GEOGRID_OUTPUT_STAGE, state: COMPLETE
2021-06-28 02:04:31,613  - INFO - geogrid_task.py:monitor_until_complete:1215 --> run() is complete...
Geogrid monitor_until_complete() completed...
Output information dict:
{'geo_em_filelist': ['geo_em.d01.nc', 'geo_em.d02.nc'],
 'geo_em_filelist_bytes': [814819, 962241],
 's3bucket': 'boreal-temporary-01days',
 's3keyprefix': '2021-06-28-02-00-55_dot_135328_geogrid_task_output'}
.
.
.
2021-06-28 02:04:31,744  - INFO - geogrid_task.py:cleanup_aws_resources:1317 --> Deregistering taskdef:
arn:aws:ecs:us-west-2:325425669423:task-definition/2021-06-28-02-00-55_dot_135328_geogrid_taskdefinition
:1
2021-06-28 02:04:32,142  - INFO - geogrid_task.py:cleanup_aws_resources:1332 --> Terminating ECS
container instance: i-076d5998c50f2a61b
2021-06-28 02:04:43,275  - INFO - ecs_functions.py:instance_termination_done:457 --> 10 seconds -->
instance_state: shutting-down
.
.
.
```

When complete, a console view of the AWS bucket (Figure 14) reveals the new geogrid output, which, by referencing via its S3 URI, is now available to other processes (like metgrid).

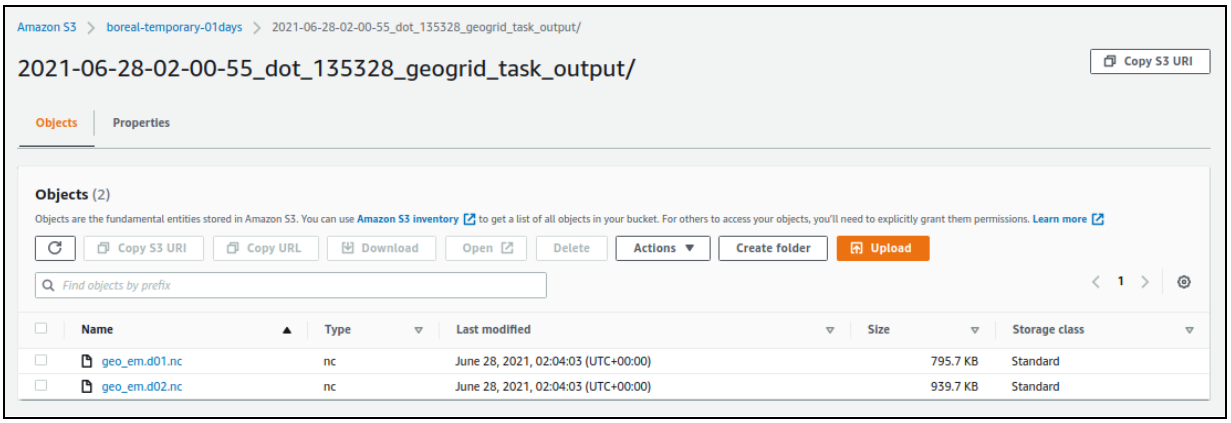

Figure 14. Screen shot of AWS S3 bucket containing newly-produced geogrid output

An alternative view of the system is represented in Figure 15, as the informal software stack. The “no-frills” test driver displayed above is a member of the upper “WPS/WRF Workflow Drivers” application layer, and utilises the GeogridTask Python API module to carry out the low-level details in communicating with the many AWS resources that constitute this geogrid service. Either directly, or through a set of “helper” Python modules, an EC2 instance is started in the cloud, loaded with necessary “plumbing” to build the collection of containers and their coupling, and then start the process of execution and monitoring. The application layer workflow drivers never deal directly with the AWS or Docker details, other than to define some of the resources to be used in the user’s AWS account - these, too, could be “hidden” from the user but are currently accessible for development purposes.

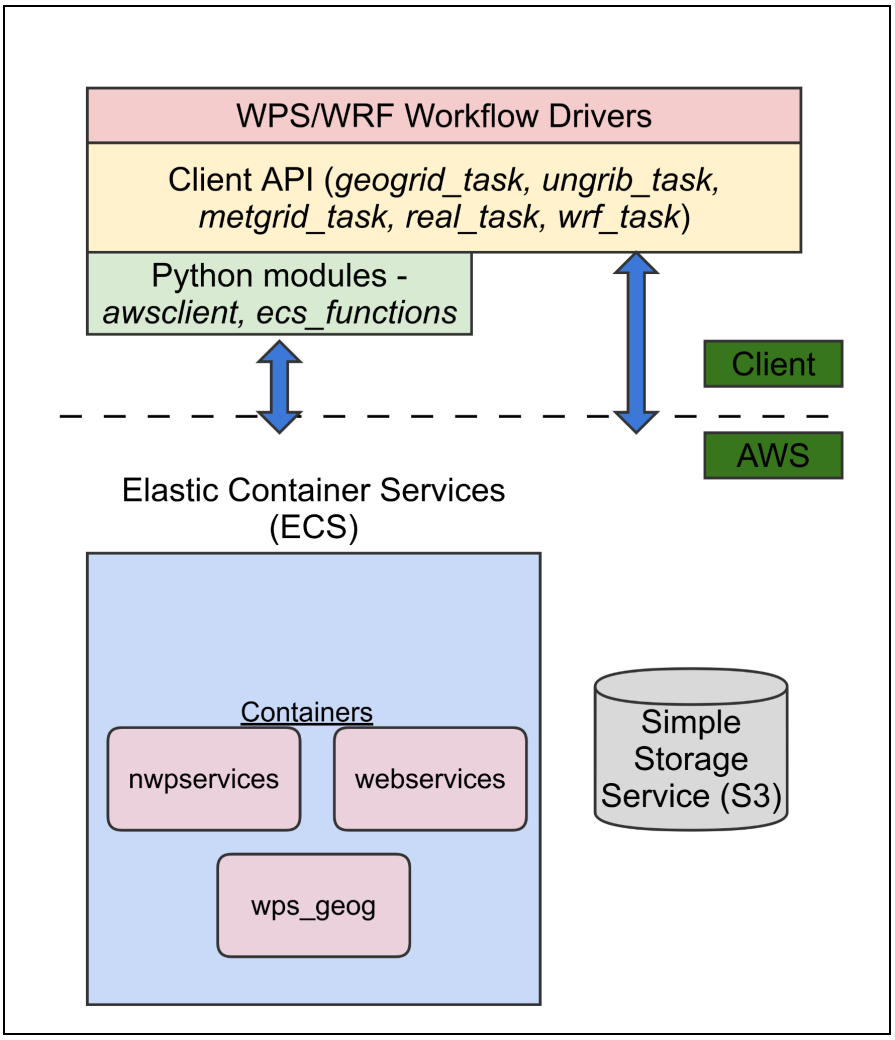

Figure 15. NWP services software stack

To expand upon all that the client APIs have to do, we enumerate the key tasks that the GeogridTask performs in order to launch the service, set it up, execute, monitor and terminate

- *__init__()*
  - Check AWS credentials
  - Using namelist.wps, estimate RAM and disc resource requirements
  - Check that selected EC2 instance type will support requirements
  - Check that the specified S3 resources have correct accessibility
  - Check that specified AWS resources (e.g. subnet, security group,...) are accessible
  - Check that the specified dockerhub images are accessible
  - Set up AWS ECS cluster and task definitions for the containers and shared volumes

- *setup()*
  - Launch the EC2 instance with parameters so that it ultimately becomes an ECS instance
  - Wait (via polling) for these things to fully start up
- *start_run()*
  - Launch the “task” (the coordinated set of containers) in the ECS instance
  - Wait (via polling) for the task start to be confirmed
- At this point, the job is off and running in AWS, and all we can do is wait and, if we choose, monitor its progress by querying the webservice running in the task
- *monitor_until_complete()*
  - Iterate until specified time out (current default is 86,400s)
    - Query the webservice container via properly-formed URI
    - Wait for completion or failure
- *output_info()* - retrieve listing of output files and their location
- *cleanup_aws_resources()* - make sure we don’t leave things running

Upon successful completion, the application layer driver has access to products of gegorid, and can use this information to drive another service, for example metgrid:

```
Output information dict:
{'geo_em_filelist': ['geo_em.d01.nc', 'geo_em.d02.nc'],
 'geo_em_filelist_bytes': [814819, 962241],
 's3bucket': 'boreal-temporary-01days',
 's3keyprefix': '2021-06-28-02-00-55_dot_135328_geogrid_task_output'}
```

Testing

Given the very high level of complexity, test-driven development has been a necessity, especially in this experimental environment. In reality, I estimate that more time was spent in developing the tests (before the code, and during code development). Although intended more for unit-testing, pytest was used to drive a wide variety (well over 100) functional tests. Many of the tests are applied to local running containers so that service components can be tested quickly, without the latency and complication of AWS deployment, but there are also a number of tests for each of the WPS and WRF components. The following illustrates a session from the early part of the project

```
[dtcuser@localhost tests]$ pytest
================================= test session starts ==================================
platform linux -- Python 3.6.8, pytest-5.4.3, py-1.9.0, pluggy-0.13.1
rootdir: /home/dtcuser/git/dockercloudwrf/wps_wrf/tests
collected 102 items

test_services/test_geogrid_system_after_aws_run.py .....                          [  4%]
test_services/test_geogrid_system_after_local_run.py ....                         [  8%]
```

```
test_services/test_nwp_standalone_geogrid_service.py .                              [  9%]
test_services/test_nwp_standalone_geogrid_service_after_aws_run.py ..               [ 11%]
test_services/test_nwp_standalone_metgrid_service_basictest001.py .                 [ 12%]
test_services/test_nwp_standalone_ungrib_service_after_aws_run_basictest001.py .    [ 13%]
test_services/test_nwp_standalone_ungrib_service_after_aws_run_with_realtime_aws_gfs.py . [
14%]
                                                                                    [ 14%]
test_services/test_nwp_standalone_ungrib_service_basictest001.py .                  [ 15%]
test_services/test_nwp_standalone_ungrib_service_with_realtime_aws_gfs.py .         [ 16%]
test_services/test_standalone_webservice_geogrid.py .........                       [ 25%]
test_services/test_standalone_webservice_ungrib.py ............                     [ 37%]
test_services/test_ungrib_system_after_aws_run_basictest001.py ........             [ 45%]
test_services/test_ungrib_system_after_aws_run_with_realtime_aws_gfs.py ........    [ 52%]
test_services/test_ungrib_system_after_local_run_basictest001.py .......            [ 59%]
test_services/test_ungrib_system_after_local_run_with_realtime_aws_gfs.py .......   [ 66%]
test_services_utilities/func/test_awstools.py ..................                    [ 84%]
test_services_utilities/unit/test_namelist_wps.py ................                  [100%]

============================ 102 passed in 1802.98s (0:30:02) ============================
```

User documentation

A preliminary “HOWTO” has been created for guiding a somewhat knowledgeable user through the process of setting up a local environment for interaction with AWS cloud, interacting with AWS cloud, itself, and launching and running simple prototypes of the NWP services in the cloud.

https://borealscicomp.com/NWPAAS/doc/

The document is not intended to be a step-by-step cookbook guiding the novice in each step from start to finish, but for a user with some experience in Unix and in AWS usage, attempting to provide an outline, with examples, of the necessary steps.

- Emphasis is on the use of the client-side programs, with provision of examples
- Contents
    - Overview
    - System requirements - relatively simple, but I’m providing access to a VirtualBox image of a CentOS 7.7 environment with the necessary (and additional) programs
    - Setting up an AWS IAM user id, a couple of S3 buckets, and some resources and policies needed for success
        - This is not a tutorial on AWS setup and use. It’s just some instructive guidance with lots of graphics, with the assumption that the user has a little bit of experience with AWS
        - A set of simple programs and procedures to test that the AWS setup is

sufficient
    - Instructions on retrieving a tar file with the necessary client software and examples
    - Assumes that specified Docker images are available in Dockerhub (they are!).
    - Examples and guidance for
        - Setting up and running an individual NWP service in the cloud
        - Setting up and running a full WPS/WRF workflow in the cloud

Developer documentation

Because this project was exploratory in nature, attempting a proof-of-concept - what works and what doesn't work - rather than a mature software product, the various codes are stored in a gitlab repository in a somewhat "experimental" environment, and not really suitable for public access at this point. However, developers who are potentially interested in some of this are welcome to contact me for access and guidance.

Included in the repository are

- A chef environment for creating a VirtualBox Virtual Machine with all software needed for development and testing (this VM is available, already created, as described at *https://borealscicomp.com/NWPAAS/doc/UserHOWTO/RequirementsForSetup/PythonClientEnvironment.html*
- Source codes for clients, NWP services and test drivers, as well as an extensive set of tests for local and remote assessment of functionality

```
.
├── src/
│   ├── clients/
│   ├── Dockerfile.nwp_service
│   ├── Dockerfile.web_service
│   ├── dockerpush-nwpservice.sh*
│   ├── dockerpush-webservice.sh*
│   ├── nwp_services/
│   └── workflow_drivers/
├── tests/
│   ├── configtest.sh
│   ├── test_clients/
│   ├── test_services_aws/
│   ├── test_services_local/
│   └── test_services_utilities/
└── utilities/
    └── s3_massdelete.sh*
```

- The expanded src/ directory:

```
├── clients/
│   ├── api/
│   ├── awsclient.py
│   ├── docker_functions.py
│   ├── ecs_functions.py
│   ├── namelist_input_wrf.py
│   ├── namelist_wps.py
├── Dockerfile.nwp_service
├── Dockerfile.web_service
├── dockerpush-nwpservice.sh*
├── dockerpush-webservice.sh*
├── nwp_services/
│   ├── geogrid_service.py*
│   ├── metgrid_service.py*
│   ├── real_service.py*
│   ├── ungrib_service.py*
│   ├── utilities/
│   ├── webservice_geogrid.py*
│   ├── webservice_metgrid.py*
│   ├── webservice_real.py*
│   ├── webservice_ungrib.py*
│   ├── webservice_wrf.py*
│   └── wrf_service.py*
└── workflow_drivers/
    ├── configtest.sh
    ├── geogrid_demo_nofrills.py*
    ├── geogrid_test_driver.py*
    ├── metgrid_test_driver.py*
    ├── namelist.input
    ├── namelist.wps
    ├── real_test_driver.py*
    ├── real_test_driver_v0.1.py*
    ├── real_test_driver_v0.2.py*
    ├── real_test_driver_v0.3.py*
    ├── recent_time_namelists/
    ├── singleservice_configtest.sh
    ├── testing_namelists/
    ├── ungrib_test_driver.py*
    ├── wps_test_driver_v0.1.py*
```

```
    ├── wpswrf_test_driver_v0.1.py*
    ├── wpswrf_test_driver_v0.2.py*
    ├── wpswrf_test_driver_v0.3.py*
    └── wrf_test_driver.py*
```

I re-emphasise that there is currently no overall, cohesive documentation for the developer beyond my hundreds of Google Docs pages of tedious, personal notes.

Summary

The goal of the project was to explore the somewhat unexplored concept of NWP-as-a-Service in the cloud by building a prototype that would allow for demand-driven custom simulations in the AWS cloud environment. Functionally, this was a success - with an already-existing AWS account, correct namelists, and access to gridded model initialization data (GFS met data is provided within AWS S3), we can quickly launch a custom, on-demand workflow. Although testing has been limited to the "typical" WPS/WRF workflow, the pieces for more complex workflows are there. These workflows are driven by relatively simple driver programs on a local client, interacting with the AWS cloud through low-level APIs. But, there are a number of areas of potential concern, and none of them are really surprising.

A first area of concern is that it takes time to "power up" an on-demand service in the cloud. These services have the potential to be long-running with large amounts of data, so we can't rely on the popular "microservice" tools that AWS makes available. For each NWP service that we choose to launch, an EC2 instance needs to be created in the cloud, then a container environment needs to be started within the EC2 instance, and then the containers need to be accessed from Dockerhub and started up. This is all done seamlessly through the client API, but it can take 3-4 or minutes for all of this to happen, before we can actually start running the service. For a long-running service, like wrf.exe, the 3-4 minutes may not be much of a hardship, but for shorter services, like geogrid.exe, the start up time can be much longer than the actual execution time.

By creating a single service that completed the full WPS/WRF workflow in one container instance, we would only incur this start-up cost once and it would likely seem negligible over the course of a full workflow. It wouldn't be very difficult to create this service, but it would seriously disrupt the paradigm of using a collection of loosely-coupled services for the creation of flexible workflows. For example, in many cases we may just want to run ungrib.exe to preprocess a large number of GRIB files for a number of simulations, and facilitating this through a single, monolithic service is not impossible, but becomes more complex.

This issue was anticipated at the start, but at this early stage it was deemed important to give higher priority to the ability to launch custom, on-demand workflows, and less weight to optimal performance. In many applications, these workflows are launched in a batch atmosphere and a

few extra minutes from startup time may not be an important issue. In other situations it will be, and in such cases we might create the larger, monolithic services discussed above.

Perhaps the greatest problem in the implementation - and, it too is not a surprise - was the large number of “unexpected” problems that crept up in the AWS Python API, *boto3*. Although this is a comprehensive set of interfaces, usually working very well, there are a number of annoying, sporadically occurring problems that can lead to hours and days of debugging time and resolution. One example is that of copying an S3 object (for example, GRIB files for initial input) into a container. The boto3 operation is generally expected to be blocking - the function call doesn’t complete until the copy is complete, yet now and then this doesn’t behave properly, and there seems to be confusion in the community about whether this really is supposed to be blocking or not!

One of the “holy grail” desired features was that a user would be able to easily run a simulation in the cloud, absent any direct interaction with cloud environments. The idea was that they would be able to go to a web page and simply specify simulation parameters and make the request, even if they didn’t have an AWS account. This paradigm, however, is quite problematic. First, “somebody” obviously needs to pay for the resources being used, so a complex front-end meant to facilitate payments and accounts would need to be added in to this framework, much as users have Netflix accounts and payment options to access resources in the cloud. This becomes even more problematic in that one needs to worry about how to handle user complaints about crashed simulations (that still have to be paid for), etc. Hence, at least in these initial stages it made more sense to require users to have their own AWS accounts, running the simulations on their own paid-for resources. But this meant that users needed to dive in deeper to the AWS world than we really would like.

My self-assessment of the work performed is that the goal of creating a layered architecture to hide the many tedious details of AWS deployment was successful, and the API facilitates the creation of custom workflows for a wide variety of scenarios. This was a primary design goal, and was meant to serve as the foundation for future work - once completed, these components would be moved behind RESTful calls to support interaction with the simulation environment solely through HTTP-like requests and responses, making them highly portable across programming languages and environments.

I’m not happy with the immense complexity of the code I constructed, but I don’t know that this could have been greatly simplified. At one point I engaged in informal conversation with an NCAR developer who was attempting similar things with a more lightweight approach. It was clever and lean, and adhered to many AWS best practises, but it didn’t appear to be as robust as I felt it should be. In a demo provided for me, the developer ran into a couple of glitches that required diving manually into the AWS environment, and I felt like this would be unacceptable for the type of user I had in mind. And, that’s the tradeoff - there are so many things that can go wrong in a typical WRF simulation, and the potential problems increase greatly in an AWS environment where the actual work is being done on resources located elsewhere, with no reasonable way for a non-technical user to access the remote resource in order to understand

and resolve the problem. Meanwhile, these resources are accruing charges. So, as is often the case in software deployment, lightweight interfaces can be created under the assumption that things will go well, but if we need something more robust, we need ways to protect the user from getting a simulation “stuck” somewhere out there, continuing to cost money.

So, goals were accomplished, many anticipated surprise problems were addressed, and it seems that an initial prototype worthy of future work was produced. Many decisions were made early in the project without a huge degree of confidence, but a recognition that “some” decisions needed to be made, and then lived with for the duration of the project. The completion of this project is a logical point to step back and evaluate what might have been done differently, and put some thought into how we can retool to further realise the long-term goals of easy access to custom, flexible NWP workflows in the cloud.

Future directions

The ability to easily launch custom, flexible NWP workflows follows on from the motivations of the early “Grid” days, in which it was envisioned that users would be able to access computational methods for doing better science, without getting bogged down in the technological details. The dream has yet to be realised, but it needs to be done, and this work will continue.

One of the fundamental goals is to make all of this infrastructure resilient to the huge number of anticipated and unanticipated problems that may creep in. Without this resilience, users are forced to dive into technological details when things go wrong or, even worse, not even be aware that things have gone wrong. This should all be handled in back-end cloud processes, hidden behind well-defined REST-like interfaces. With the RESTful approach, front-end programmers need only form the RESTful requests and handle the responses in the language and environment of their choice. This paradigm is well-supported in cloud environments like AWS, but the real challenge is in adapting long-running and large on-demand services into the mix.

Now that many of the low-level details have been explored and addressed, this is an opportune time to step back and think about cleaner, more “best practices” approaches of doing things without sacrificing the required resiliency. For example, the AWS CloudFormation services may present a more logical and cleaner method than I employed (with low-level Python clients) to launch and configure the cloud resources for a given NWP service.

In the long run, it is desirable to go much further than the standard WPS/WRF environment explored in this project. In addition to data assimilation (GSI), GRIB production (UPP), verification products (MET) and graphics production, other rising NWP models should be accommodated. It’s a long and intimidating vision, but ultimately it’s hard to get around the basic premise that “somebody needs to do this.”

Activities during the project

Although no physical visits to Boulder occurred, in some ways the Covid environment encouraged greater and more frequent outreach than maybe I would have pursued in normal days. In addition to several virtual meetings held with the DTC team, the following presentations were delivered virtually

- Morton, D., J. Wolff, K. Fossell, J. Halley-Gotway, M. Kavulich, M. Harrold, "Implementation of WRF as a Service in the AWS Cloud", in UCAR Software Engineering Assembly Improving Scientific Software Conference, Boulder, Colorado, 22-26 March 2021.
- Morton, D., J. Wolff, K. Fossell, J. Halley-Gotway, M. Kavulich, M. Harrold, "Building NWP Cloud Services to Ease the Task of Running Custom Simulations," National Weather Association 2021 Annual Meeting, Tulsa, OK, 21-25 August 2021.
- Morton, D., J. Wolff, M. Harrold, K. Fossell, J. Halley-Gotway, M. Kavulich, "A Docker / Amazon Web Services Implementation of Numerical Weather Prediction Services," in American Geophysical Union Annual Meeting, New Orleans, LA, 13-17 December 2021.
- Morton, D., J. Wolff, M. Harrold, K. Fossell, J. Halley-Gotway, M. Kavulich, "A Python API for Launching WRF Service Components in the AWS Cloud," in 102nd American Meteorological Society Annual Meeting, Houston, TX, 23-27 January 2022.

Acknowledgements

I would like to gratefully acknowledge the staff of the Developmental Testbed Center (DTC) for funding, inspiration and assistance, as well as access to a number of resources. The containers described in the online documentation at

https://borealscicomp.com/NWPAAS/doc/UserHOWTO/

were created by using the NWP containers originally created by DTC. I just added a little bit of extra functionality to them. More information is available at the DTC *Numerical Weather Prediction (NWP) Containers* project page

https://dtcenter.org/community-code/numerical-weather-prediction-nwp-containers