ESIP NCPP Brief on cohort study – A closer look at the phases

Exploring Cloud backends for the new OGC Environmental Retrieval API

By Steve Olson and Shane Mill
NOAA National Weather Service and NODD Collaborator

Agenda

1. Background
    a. Data Silo Problem
    b. OGC and Environmental Data Retrieval (EDR) API
    c. ESIP NCPP Program
2. Goals and Objectives for cohort study
3. Phase 1 Key Findings and takeaways
4. Phase 1 Results
5. Demo
6. A Look Towards the Phase 2 Work
7. Questions

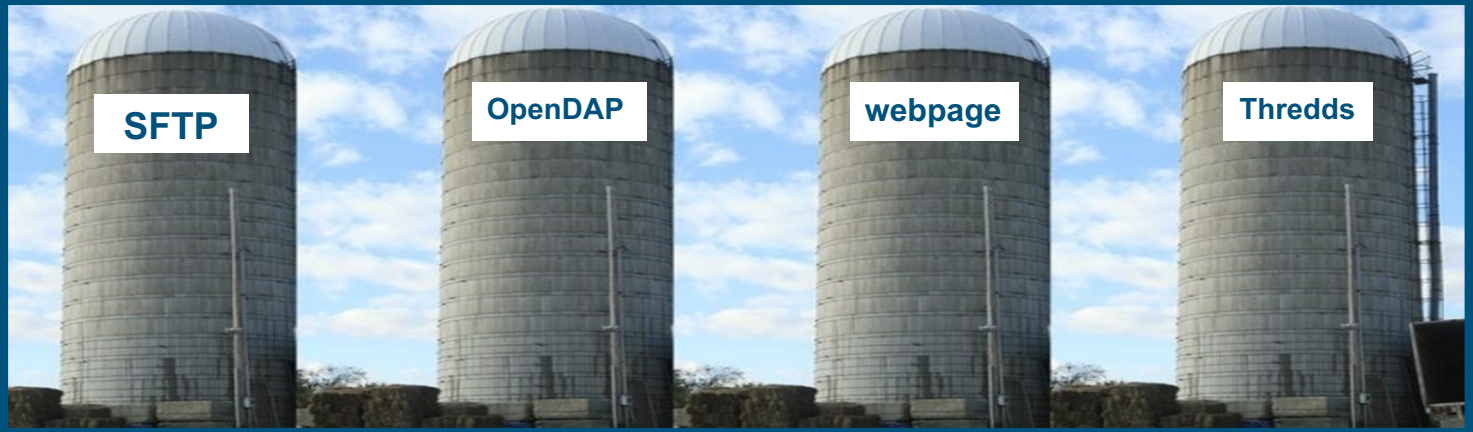
Data, Data, and more data … What do I do?

“Data-Silo” Issue

So many Formats

Data silos are isolated groups of data that stymie data storage, discovery, and use

So many Systems & Sources



Silo’d services impact both users and providers

Impacts to users

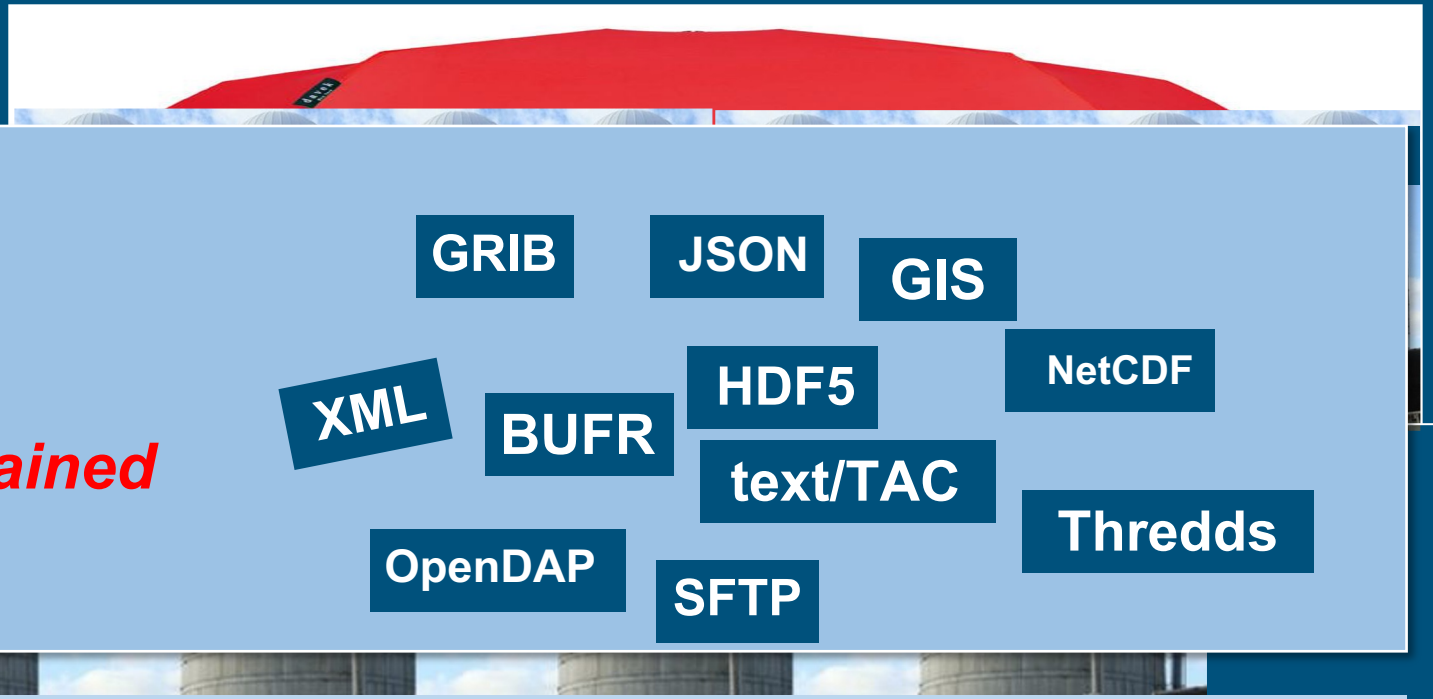
- Requires users to post-process data
- Users may be forced to download more data then desired
- Users may be forced to download in a less than desirable format
- Insufficient metadata that describes the data
- Service doesn’t support machine to machine communications
- Insufficient notifications regarding new data … Leads to overloading of service due to demand

Impacts to providers

- As cloud becomes more and more prevalent, there are costs to disseminate and store data
- Typically less flexibility with these types of services
- Providers are forced to impose rate limiting controls to keep demand/needs in check

These types of services don’t lend themselves to FAIR principles

EDR-API Breaks Down the Data Silos

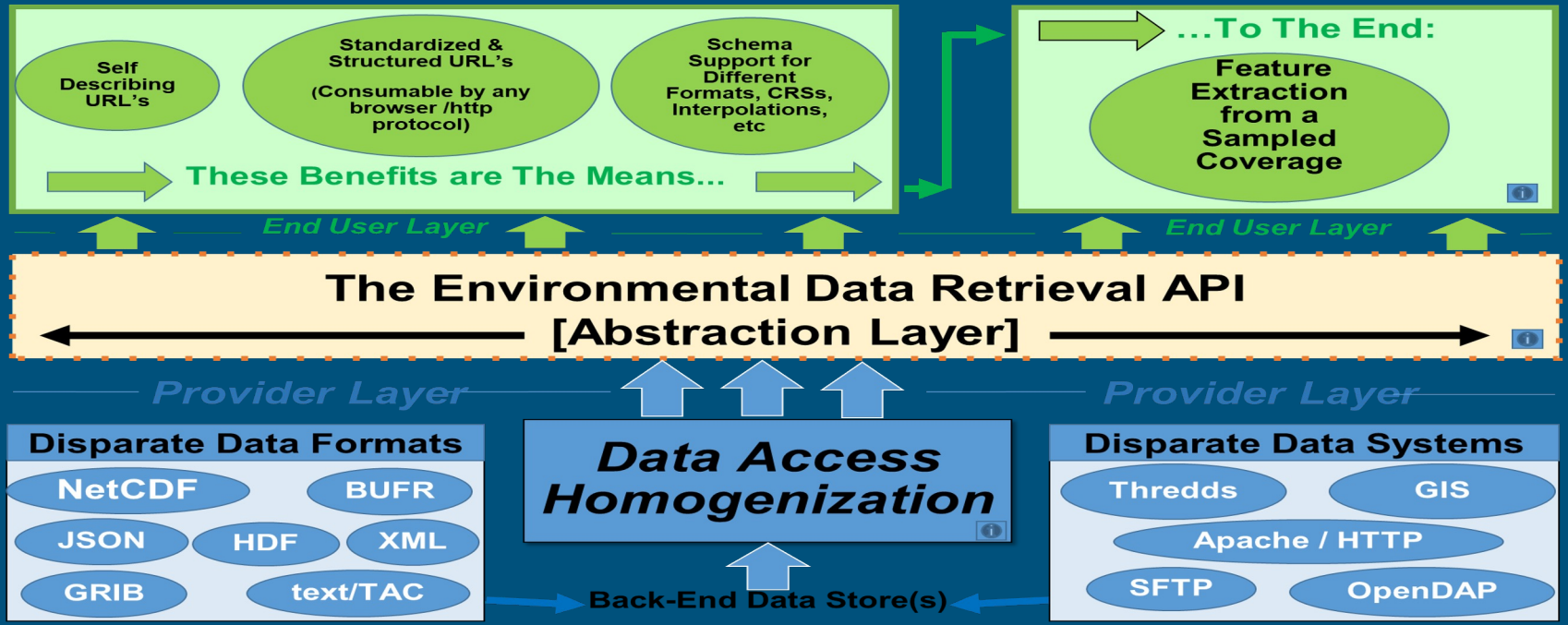


EDR-API
<u>Homogenizes</u>
and
<u>Integrates</u>
the Data contained in the Silos

GRIB
JSON
GIS
XML
BUFR
HDF5
NetCDF
text/TAC
Thredds
OpenDAP
SFTP

"It provides a uniform internet interface and platform where disparate data formats, systems, and sources are all integrated and combined into web accessible structured endpoints

OGC Environmental Data Retrieval (EDR) API

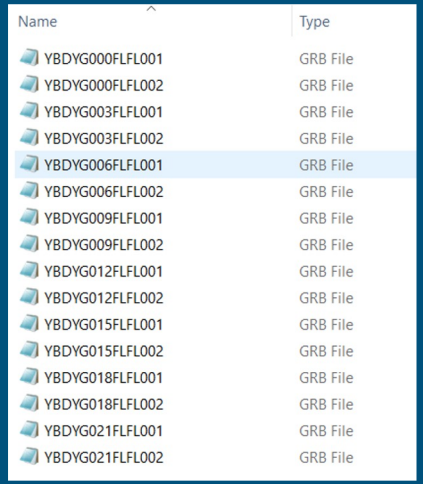


EDR lowers the bar of entry for users, hiding away the complexities of application software, data systems and data formats.

EDR-API Feature Types

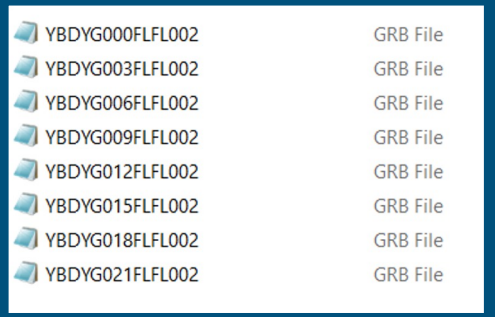
Items

- Retrieve data for point/position identified by a name rather than coordinates.
- `http://data-api-c.mdl.nws.noaa.gov/OGC-EDR-API/collections/KWBC_WAFSHZDS_BLENDED_CB_0P25/instances/00z/items/region_1`



Locations

- Items Retrieve a feature by identifier. The coordinates in the feature could be used to create an EDR query. The feature could also be a previously stored query. Compatible with OGC API - Features - Part 1: Core.
- `http://data-api-c.mdl.nws.noaa.gov/OGC-EDR-API/collections/KWBC_WAFSHZDS_BLENDED_CB_0P25/instances/00z/locations/region_1?f=GRIB&parameter-name=cb_extent`



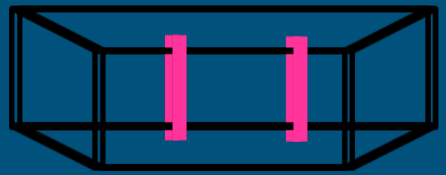
EDR-API Features Demonstration

Watches, Warnings, Advisories disseminated as OGC EDR-API locations:

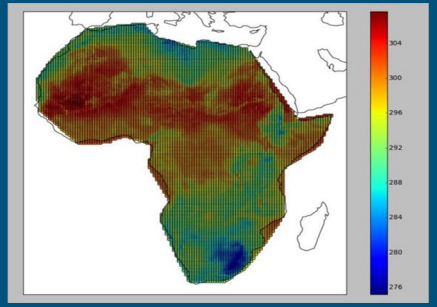
Locations:
http://data-api-c.mdl.nws.noaa.gov/OGC-EDR-API/collections/wwa_active_severe_weather_statement/locations/severe_weather_statement_KTWC_2022-09-12T16:25:00-07:00?f=text/cap
http://data-api-c.mdl.nws.noaa.gov/OGC-EDR-API/collections/wwa_active_severe_weather_statement/locations/severe_weather_statement_KTWC_2022-09-12T16:25:00-07:00?f=json

EDR-API Query Types

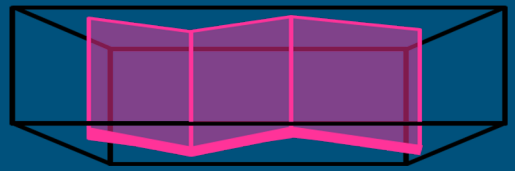
EDR allows users to query and request a feature from METEOROLOGICAL <u>SHAPES</u> (Position/Point, Multi-point, radius, area/polygons, Cubes, Trajectories, corridors and instance).



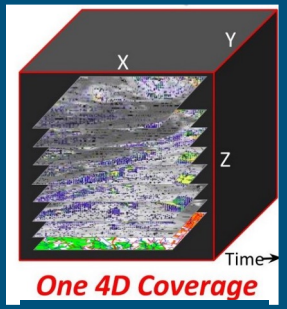
Vertical Profiles



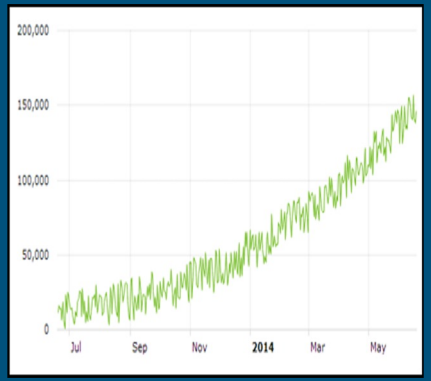
Area



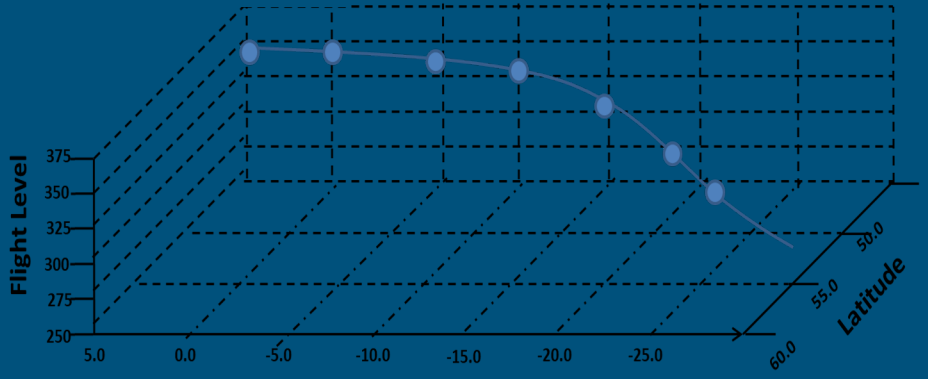
Vertical Cross Section



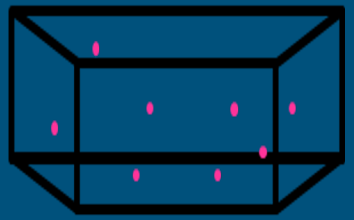
Cube



Trajectory

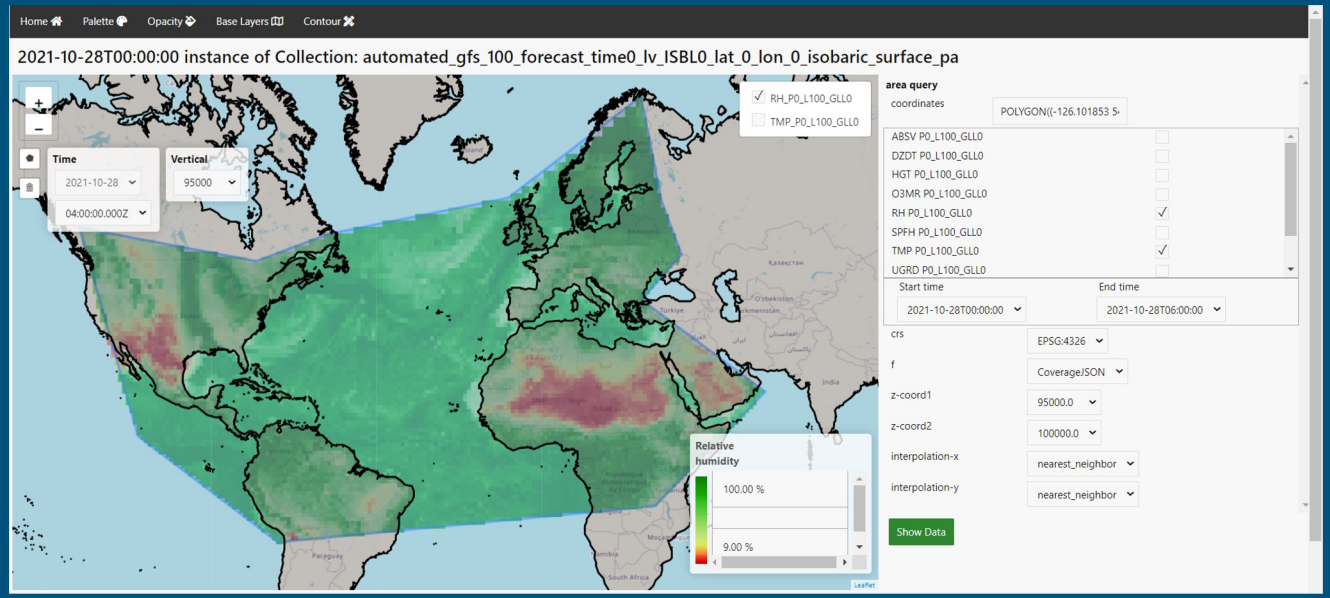


Corridor



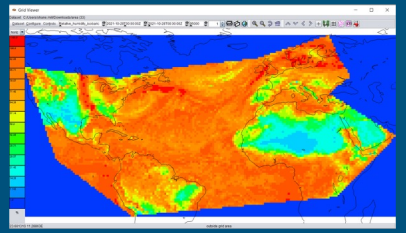
Multi Point Collection –> Multi Time Series

EDR-API Query Demonstrations

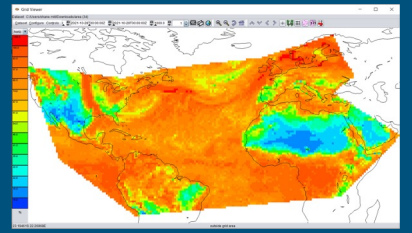


- Ability to query by a user defined polygon.
- Ability to change the color palette
- Ability to view multidimensionality of the data (multiple times, multiple vertical levels).
- Ability to add other base layers such as land/sea borders.
- User pane allows user to construct their EDR-API query.
- CoverageJSON is rendered in the browser, but user is also given the ability to download other formats such as **GRIB, NetCDF, and Cloud Optimized Geotiff**.

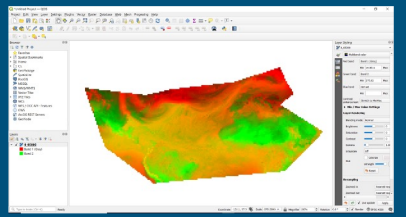
Visualization with other tools:



ToolsUI Visualization of GRIB



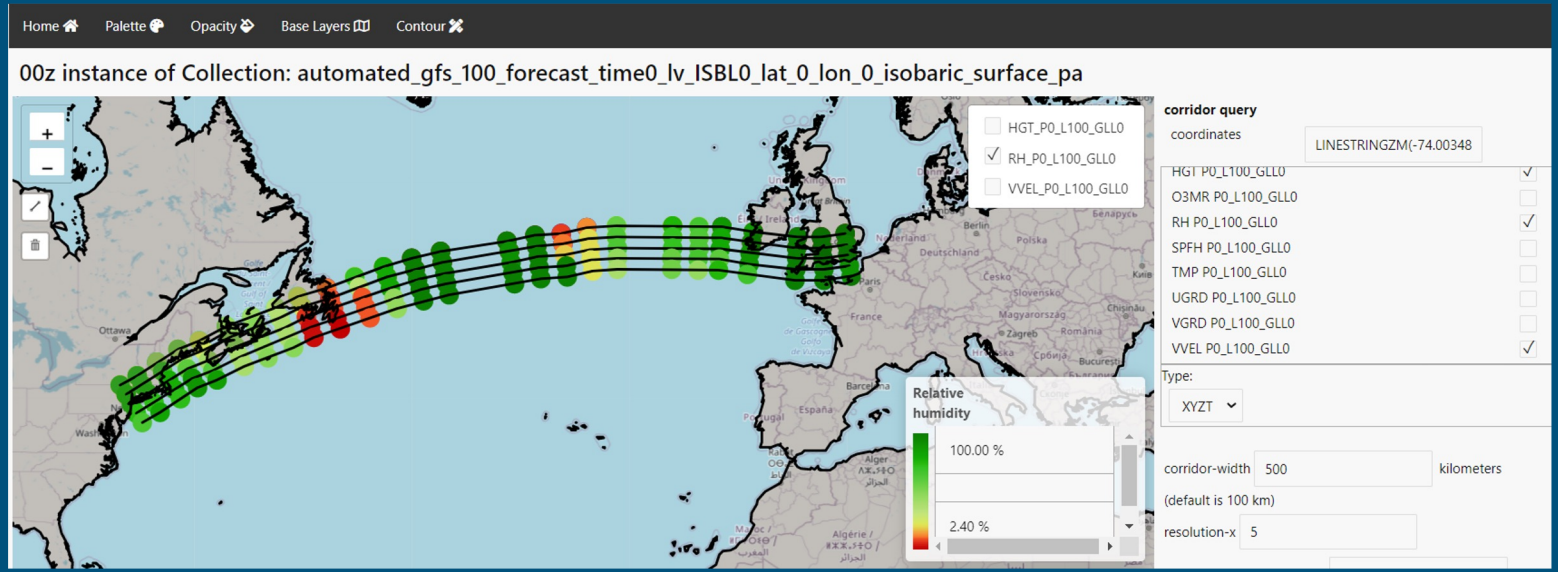
ToolsUI Visualization of NetCDF



QGIS Visualization of COG

EDR-API Query Demonstrations

User Interface Experience:



- Ability to query by a user defined corridor.
- Ability to change the color palette
- Ability to view multidimensionality of the data (multiple times, multiple vertical levels).
- Ability to add other base layers such as land/sea borders.
- User pane allows user to construct their EDR-API query.
- CoverageJSON is rendered in the browser, but user is also given the ability to download other formats such as NetCDF.

ESIP NCPP and the OGC EDR-API

- Early prototyping efforts in the NWS EDR API server implementation (**EC2 instance**) focused on the integration and homogenization of disparate data systems, data sources and data formats, ways to automate that process, and the discovery, dissemination and visualization of that data to users.
- A number of NOAA Line Offices (LOs) are now interested in exploring the use of the OGC EDR API as a dissemination and visualization mechanism for their applications.
- ESIP NOAA Cloud Pathfinder Program (NCPP) provided a perfect environment for us to explore a cloud implementation for the new OGC Environmental Data Retrieval (EDR) API

ESIP NCPP and the OGC EDR-API

Two main focal areas for study:

1. Explore and assess what an operational implementation of the OGC EDR API could look like in the Cloud
2. Prototype a Cloud Native environment that connects OGC geospatial APIs with NODD data. The goal is to assess the most cost effective and performant EDR API environment for the dissemination and visualization of NODD datasets

ESIP NCPP and the OGC EDR-API

Several Phases for this work

- Phase 1 as multiple parts
  1. Assess options for scalability and caching in a baseline serverless environment
  2. Connect EDR API with S3 NODD buckets of data to enhance user experience
  3. Expand autoscaling, explore CSP agnostics alternatives to CSP specific toolings and assess options for supporting asynchronous processing and responses
- Phase 2 is future work and will be described later in this presentation

Early Phase 1 Work – Establish baseline serverless configuration that promoted operational needs (Introduce scalability, caching)

Goal : Enhance both frontend and backend aspects to our single server EDR API implementation

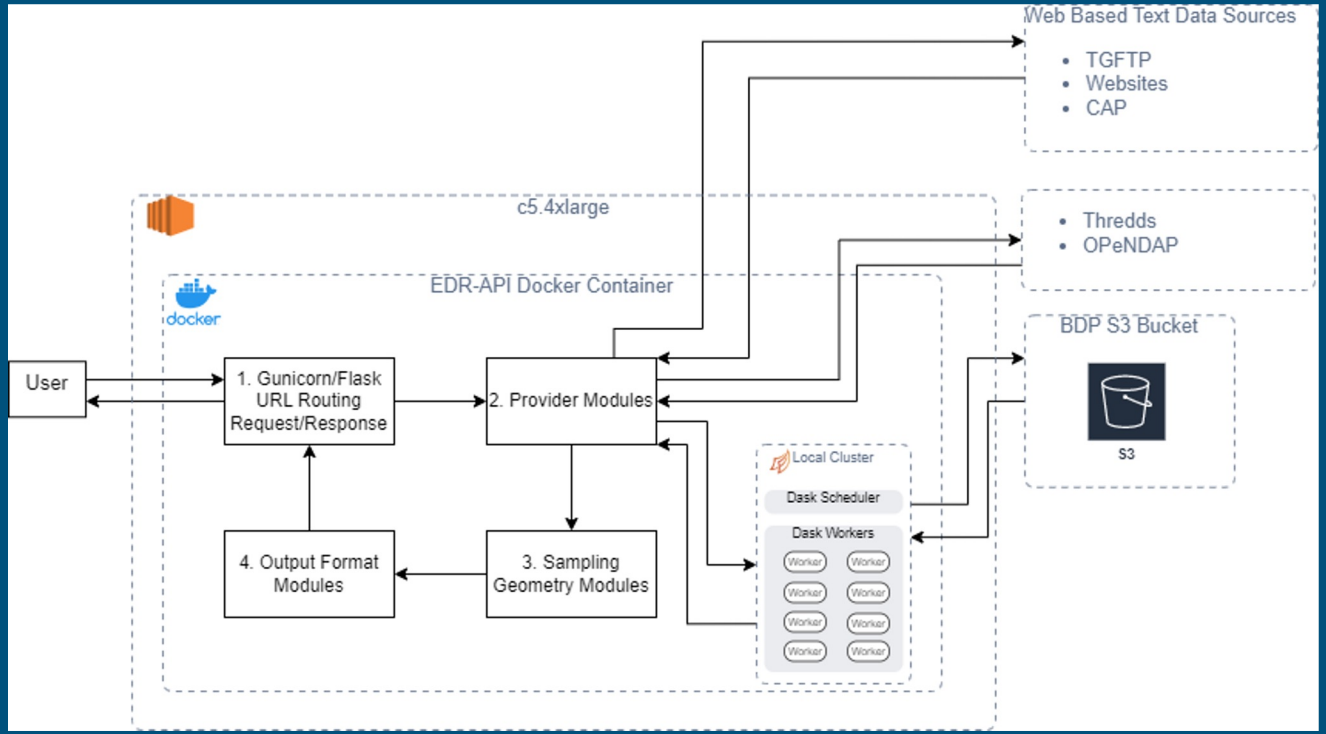
Front end Enhancements:

- Implement **AWS CloudFront** with caching behaviors. On a cache hit, the response is returned directly to the user much quicker. On a cache miss, the request is forwarded to **AWS API Gateway**.

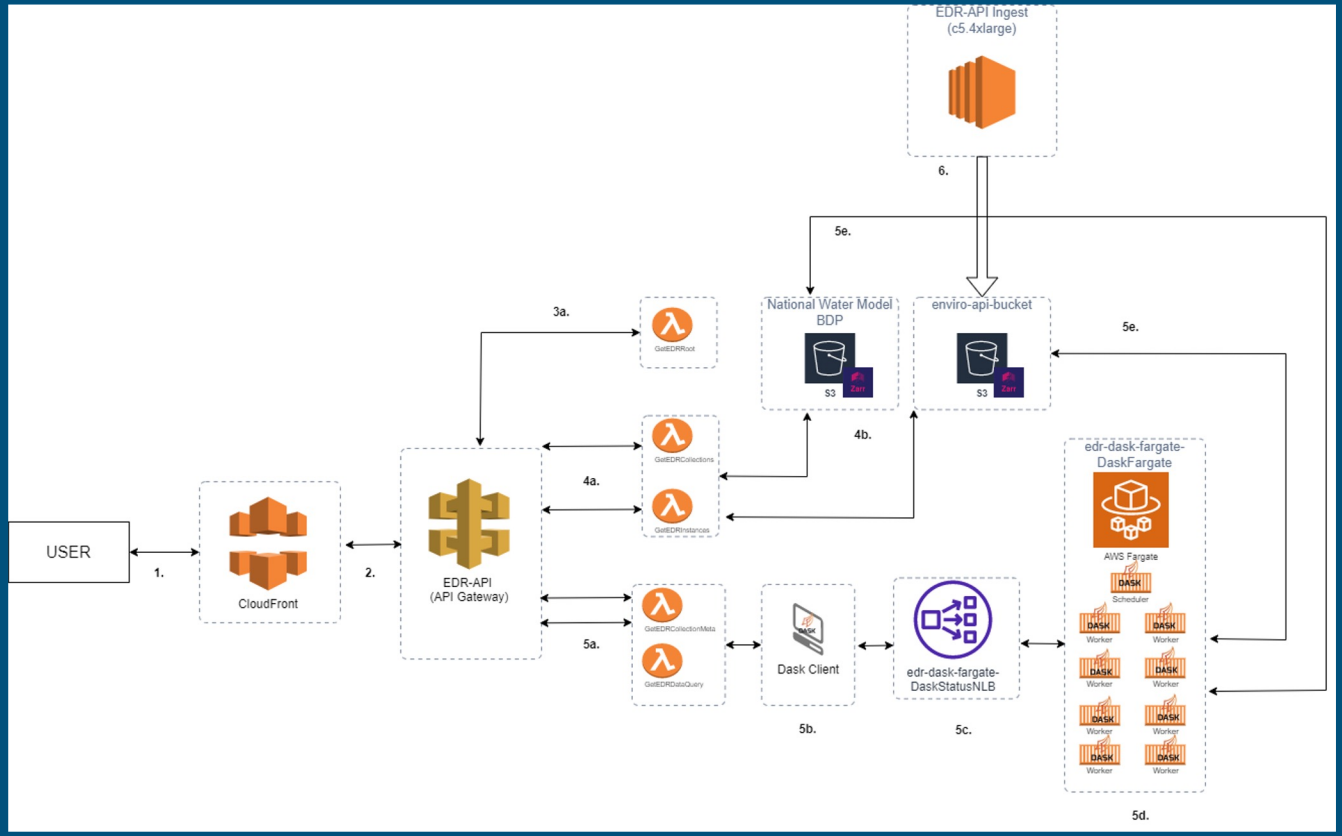
Back end Enhancements:

- Use **AWS API Gateway** which allows the API functionality to be separated into serverless functions (**AWS Lambda**). Created **AWS Lambda** functions for each aspect of the EDR-API schema (root, api, conformance, collections, instances, data queries). Within data queries, separate **AWS Lambda** functions were created for sampling geometry types (position, radius, area, cube, trajectory, and corridor).
- Implement a **centralized Dask Cluster** using **AWS ECS Fargate**. A connection to a centralized **Dask Cluster** from **AWS Lambda** facilitated horizontal scalability.

Single Server Architectural Design and workflow



Serverless Architectural Design and workflow



1. User accesses API through CloudFront. On cache miss request is directed to API Gateway.
2. Bases on path, API Gateway invokes a Lambda function.
3a, 4a, 5a: Lambda functions are invoked, some of which return static responses, some return response based on S3 bucket contents, some of which use Dask to process data and return a response to the user.
5b,5c,5d: For Lambda functions that use Dask client, the centralized Dask cluster based on ECS Fargate is accessed using the DNS name of the Network Load balancer through the Dask client within the invokated Lambda function.

Fully annotated diagram: [EDR AWS Architectural Diagram]

Summary of key findings and takeaways

1. For Deployment, **AWS Cloudformation** very useful in configuring/automating deployment of AWS resources for EDR-API
2. On frontend, several lessons learned with **AWS Cloudfront**
    a. Need caching behaviors that capture similar API requests
    b. Need to limit **AWS Cloudfront** cache invalidations in order to control costs. More cost effective way is to set the time to live (TTL) header on the cached objects through the caching behaviors
    c. Overall, **AWS CloudFront** had significant impact on performance on a cache hit. For the National Water Model position query for one point at one time, a request to the backend takes 11.28 s versus 53 ms for **AWS CloudFront** cache hit

Summary of key findings and takeaways (Cont'd)

3. On backend, there were several key takeaways
   a. Use of **AWS API Gateway** promoted quick implementation of the OGC EDR-API specification because of its ability to load an OpenAPI definition.
   b. Use of **AWS CloudFormation** template allowed for automation and to directly tie EDR-API endpoints to specific **AWS Lambda** functions within the **AWS API Gateway**
   c. While **AWS ParallelCluster** supports a centralized **Dask cluster**, we were unable to connect to the **AWS ParallelCluster** from a **Dask client** within an **AWS Lambda** function.
   d. **AWS Elastic Container Service (ECS) with Fargate** was much better option**.**
   e. Performance can be impacted through memory allocations at the **AWS Lambda** function level. The key is right sizing memory by EDR-API endpoint.
   f. Performance can also be impacted through CPU and memory allocations within the Dask workers in **AWS Fargate.** The key is to balance the allocations for performance and cost effectiveness.
   a. **AWS API Gateway** has 30 sec timeout. This limited the type and size of successful EDR API requests/responses

Testing methodology

1. Test 3 types of implementations
    a. Server East - c5a.4xlarge (EC2), Serverless East and Serverless West
2. Dataset collections used during testing
    a. National Water Model (NWM) and Global Forecast System (GFS) 1.0 degree model
3. Testing was done covering
    a. Different dimension sizing for:
        i. Spatial coverage, Vertical and Temporal ranges
    b. With and without CloudFront
4. Testing of the load of 10, 50, 100, and 500 concurrent users
5. 34 total tests covering the different collections, sampling geometry types, and dimensional coverage
6. Apache JMeter was used to perform load testing and capture metrics for different levels of concurrent users and different timespans.

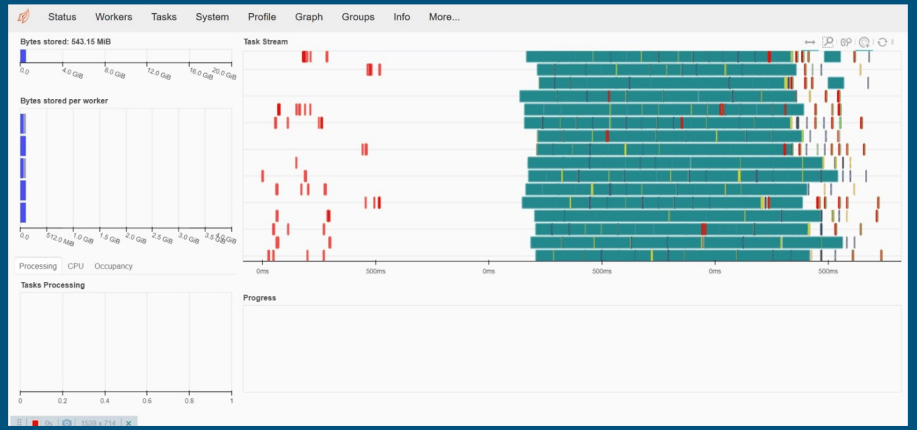
Key Findings of Test Results

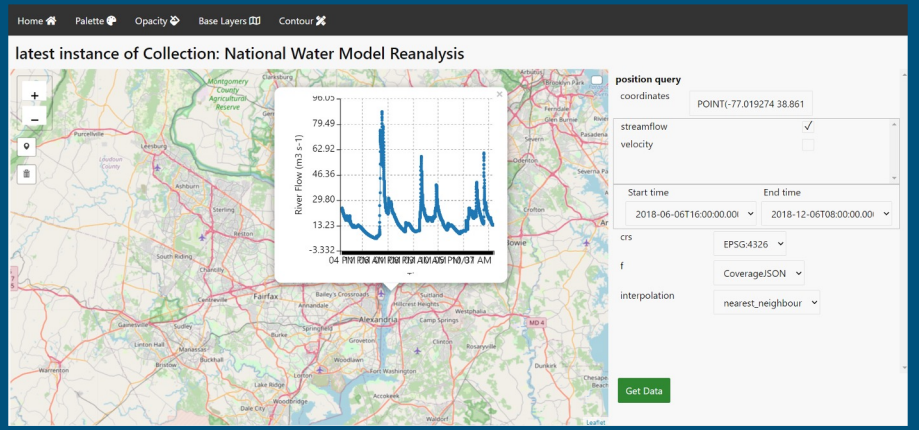
1. Test results for NCPP Serverless East and NCPP Serverless West followed similar distributions, although NCPP Serverless East was slightly faster overall (we should note that testing occurred in Arlington, VA).
2. Both Serverless environments were responsive and consistent in the results once load was increased to 50 concurrent users. Once concurrent users were increased to 100 concurrent users, there was some degradation in performance. Once concurrent users were increased to 500 users, we began to see some failed requests.
3. The Server environment was less consistent in performance, and truly suffered in responsiveness once load was increased to 50 concurrent users and for larger data queries. Overall, the Server environment saw worse overall performance compared to the serverless environments.

Demonstration of this capability and what it means

- Role of the EDR API User Interface
- OGC APIs as building blocks and connection with NODD
- EDR API is an example of one project meeting the NOAA data strategy goals of leadership, open data, capabilities, and collaboration

National Water Model Demo (AWS Serverless environment)





- When a request is made and CloudFront determines a cache hit or miss. On a miss, API Gateway invokes the appropriate lambda function to make the query.
- Within the lambda function, the Dask client accesses the ECS Fargate Dask Cluster to optimize the loading of the Zarr data chunks. Using Xarray, the data requested is selected and returned to the user in CoverageJSON, which can be visualized in Leaflet.

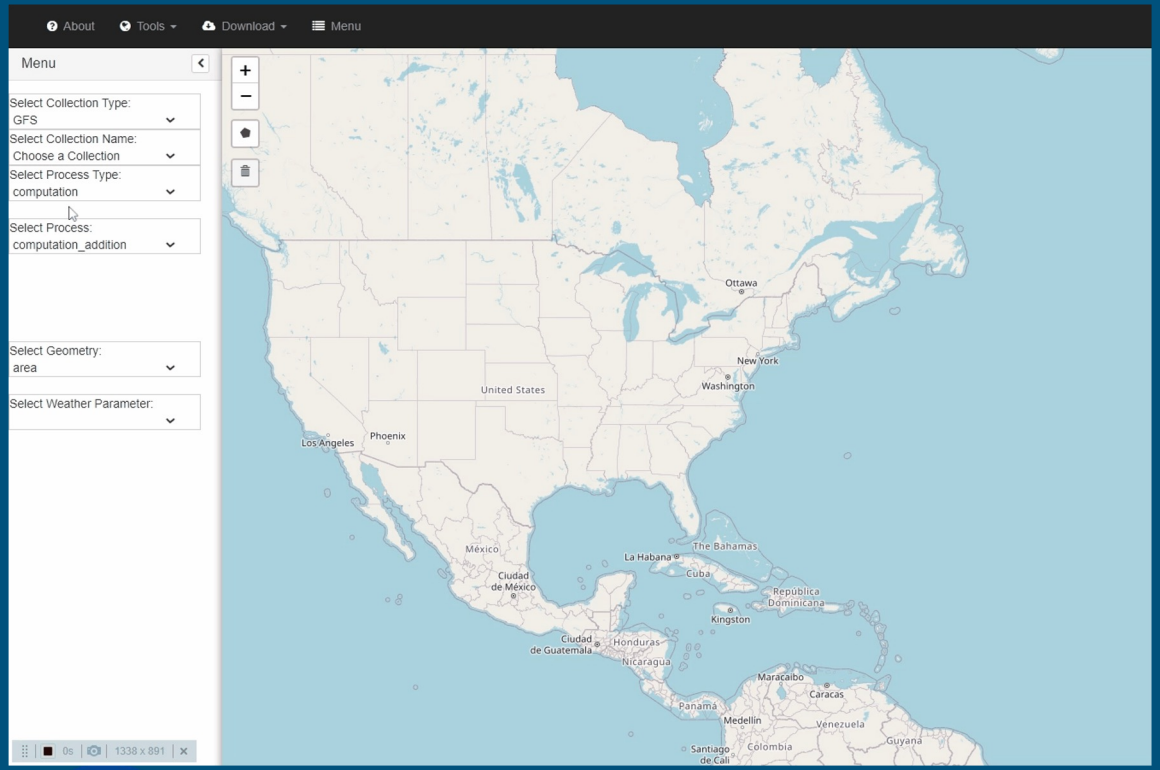
NOAA

ESIP
MAKING DATA MATTER

Connecting EDR API with NODD Data: OGC API - Processes and OGC API - EDR Integration: Wind Streams

- This demonstrates the ability to take the U and V components of the wind from an EDR-API response and return wind streams using OGC API - Processes based on user selections of geospatial area, time, and vertical level.

Using OGC APIs as building Blocks - Chaining of Processes



OGC EDR API and Processes API Integration Client

- With recent development updates to the client, we are now able to chain processes together, with the OGC EDR API as the originating data source.
- In this example, a user selects an EDR-API collection, a weather parameter, a geometry, and a time range. The user can conduct a summation of the data over a time range using OGC API - Processes.
- Given that result, the user can then conduct a threshold selection using OGC API - Processes based on the previous result.

NOAA

SIP
MAKING DATA MATTER

Next Steps in our Phase 1 work

1. Focus on assessing and expanding frontend and backend autoscaling approaches
2. Explore CSP agnostic alternatives to CSP specific toolings (Work in progress)
3. Exploring direct access to data in a cloud optimized manner (Work in progress)
4. Assess options for supporting asynchronous processing and responses (Future work)

1. Assessing and expanding autoscaling approaches

Frontend Autoscaling

- We originally investigated the use of **API Gateway** invoking **Lambda** functions (**Serverless approach**).
- We then investigated an alternative approach using an Application Load Balancer tied to an autoscaling group for the “T” (General Purpose) class of ec2 instances (**Server Based approach**).

Motivation

- We saw limitations with **API Gateway** (30 second timeout) and **Lambda** (Payload response size).
- We wanted to compare the performance and cost of the two different approaches.

1. Assessing and expanding autoscaling approaches (Cont'd)

Backend Autoscaling

- We originally investigated the use of **ECS** with **Fargate** (**Serverless**) to implement a Centralized Dask Cluster (the scheduler and associated workers as **ECS** Tasks)
- We then investigated the use of **ECS** backed with EC2 (**Server Based**) Container Instances (ECS tasks associated with EC2 instances rather than **Fargate**) to implement a centralized Dask Cluster.

Motivation

- While **Fargate** performed well upfront, we wanted to evaluate cost/performance of alternative approaches.
- **Fargate** does not support the connection to a Lustre FSx FileSystem, and we had a desire to test performance of Lustre vs. **EFS**.
- The investigation of an EC2 Backed **ECS** Cluster moves us closer to a Cloud Agnostic Solution.

Key Findings/Takeaways from autoscaling approaches investigation

1. Frontend
    a. We found that using an **Application Load Balancer** tied to an autoscaling group of **EC2** instances overcame the limitations of **API Gateway** (30 second timeout) and **Lambda** (6MB payload limit).
2. Backend
    a. We found that cpu and memory allocations need to be sufficient for performance but not over allocated to manage costs. This was the case for both **ECS Fargate** and the **EC2 Backed ECS** Cluster.

Key Findings/Takeaways from autoscaling approaches investigation (Cont'd)

Horizontal and Vertical Autoscaling of EDR API are impacted by the backend storage choice (**S3**, **EFS**, Lustre)

Backend Autoscaling

a. We found that we were able to mount to a Lustre FSx file system using the **EC2** backed **ECS** Cluster, and found performance to be faster than **EFS** and **S3**.
b. We found Lustre to be more predictable than **S3** in terms of cost because **S3 GetObject** costs can quickly accumulate while Lustre costs are fixed.

Frontend: Next steps in autoscaling approaches investigation

- We found that the Server based environment had less limitations than the Serverless environment in delivering EDR-API queries, so our next step would be to expand the Server based environment
  - Continue to experiment with appropriate ec2 instance types
  - Adjustments with the **Application Load Balancer**
  - Role of Pub/Sub and Notifications (**SQS**, **SNS**, etc.)
  - Role step functions may play in breaking a large request into separate smaller requests

Backend: Next steps in autoscaling approaches investigation

- Role of asynchronous processing and asynchronous requests.
  - Pre-signed URL's sent back to users to access payload from an **S3** bucket for responses.
- Centralized Dask Cluster Approaches
  - So far, we have investigated a Dask Local Cluster, an **ECS** Cluster with **Fargate**, and an **ECS** Cluster backed by **EC2** Container Instances.
  - With the **EC2** Backed **ECS** Cluster, an autoscaling group contains a group of **EC2** instances. We have found a need to extract the scheduler instance out of this autoscaling group for the following reasons:
    - To prevent the Scheduler instance from getting terminated when scaling down
    - To assign a separate CPU/Memory allocation to conserve costs
    - A desire to not include scheduler instance cpu/memory metrics for the Worker autoscaling group
  - Research alternatives to **ECS** such as **EKS** (Kubernetes), Lithops, etc.

3. Exploring direct access to data in a cloud optimized manner

Kerchunk and direct access to data

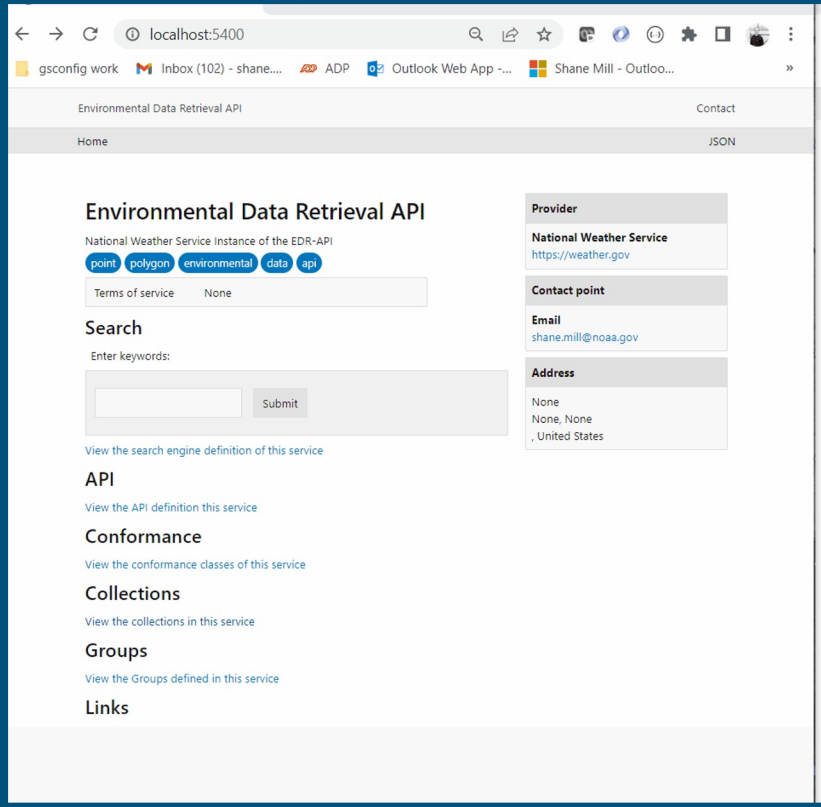
- Kerchunk allows direct access to data stored in a variety of formats (NetCDF, GRIB2, GeoTIFF, etc.) stored in CSP object storage (such as **S3**).

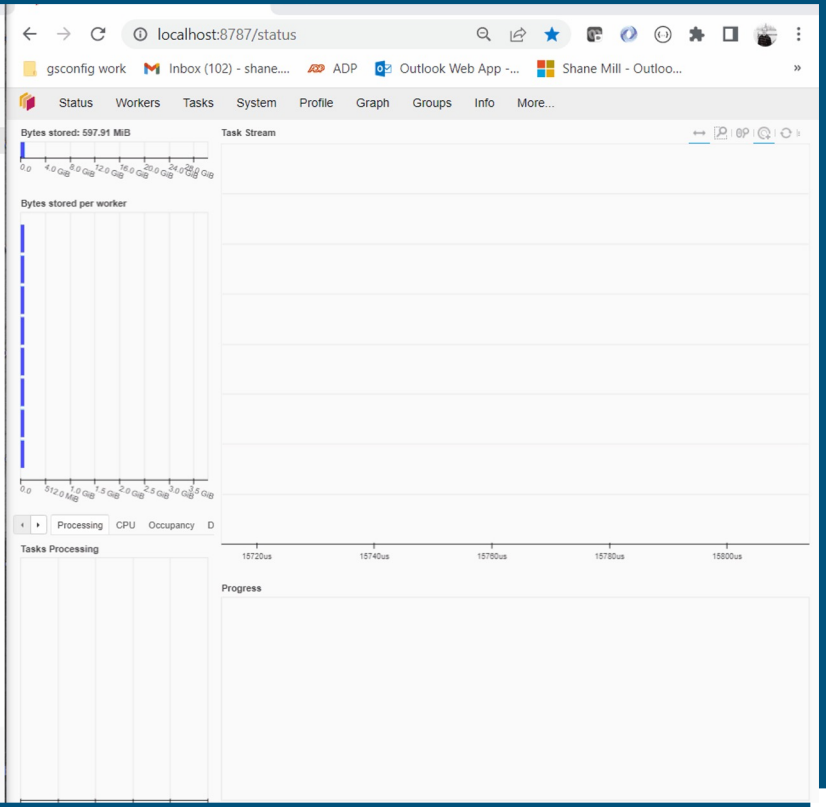
Role Kerchunk plays for EDR API

- Kerchunk has the potential to allow a consistent interface between the OGC - EDR-API and a variety of existing data stores, such as data stored in NODD buckets.
    - Reference files created by Kerchunk can allow the EDR to read the different data stores as a consistent data structure (as an Xarray dataset object using the Zarr engine).

Kerchunk can provide a consistent direct connection to growing data in the cloud, which can contribute to an ecosystem that facilitates FAIR principles for data access

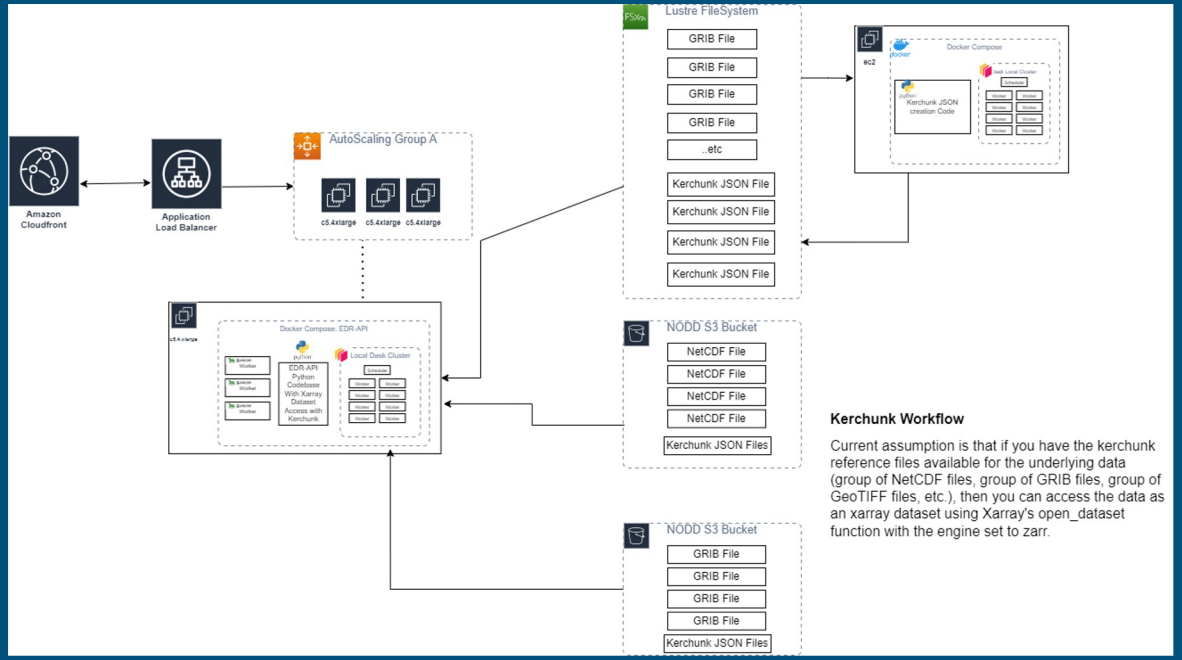
Early Prototype of Kerchunk with the EDR-API





- Inspiration and guidance came from this article written by Peter Marsh.
- Demonstrates remote access to HRRR data using Kerchunk and Intake-Xarray.

Potential EDR/Kerchunk Architecture



- When the Kerchunk reference files have been created to accompany the underlying data (such as the HRRR GRIB files), the data can be accessed directly from a single intake catalogue.
- An alternative to accessing the data through an intake catalogue is accessing the **S3** bucket and reference files directly with fsspec.
- The EDR-API then interfaces with the data through xarray-datatree where each EDR collection is a single xarray dataset object.
- We have not yet constructed our own Kerchunk reference files, but with the data access through the intake catalogue, we can envision a consistent approach to accessing different buckets of data through the EDR-API.

NOAA

ESIP MAKING DATA MATTER

Next Steps for Kerchunk and Direct Access to Data

- Connecting to HRRR data using the intake catalogue was a great first step in demonstrating a direct connection to data in a cloud optimized manner and that the use of this technology can be integrated with OGC API's.
- Currently, we are only supporting the Position query for the HRRR data, but it would be great to expand that to all of the EDR queries such as Area, Cube, Radius, Trajectory, and Corridor.
- We are also only supporting a JSON output, but the EDR-API allows for a variety of output formats, so we could expand this to CoverageJSON, NetCDF, GRIB, COG, etc.
- We could create Kerchunk reference files for a specific NODD bucket of our choosing to show the same exact data access workflow as the HRRR example.

A look ahead to Phase 2 Work

Complete Assessment/Investigation and establish best practice/guidance for implementing this service

- Completing frontend and backend autoscaling assessment
- Complete Testing and evaluation of alternative methods for a Centralized Dask Cluster
- Continuing our research and investigation into Kerchunk and taking advantage of methods to directly access data in a cloud optimized way.
- Defining and implementing an asynchronous approach to time consuming queries or large payload responses

Operational Functionality Development and Testing

- Support the use of User Authentication/Authorization to backend EDR API resources
- Implement a Pub/Sub framework and approach for users to subscribe to and be notified when data is available
- Support a message queue service (**SQS**)
- Implementation of Caching (Both CSP and Cloud Agnostic)

Questions?