Zarr Storage Specification Version 2: Cloud-optimized persistence using Zarr

Status of this Memo

This Request for Comment (RFC) provides information to the NASA Earth Science community. This RFC does not specify an Earth Science Data Systems (ESDS) standard. Distribution of this memo is unlimited.

Change Explanation

This document is not a revision to an earlier version.

Copyright Notice

Abstract

This document designates the Zarr Storage Specification Version 2 as a cloud-optimized file format convention for NASA Earth Science Data Systems.

Suggested Citation

Newman, D. J. (2024). Zarr storage specification version 2: Cloud-optimized persistence using Zarr. NASA Earth Science Data and Information System Standards Coordination Office. https://doi.org/10.5067/DOC/ESCO/ESDS-RFC-048v1.

Table of Contents

1 Introduction 2
1.1 Motivation 2
1.2 Evidence of Implementation 3
2 Zarr Structure 4
2.1 Metadata 6
2.2 Data 7
3 API 7
3.1 Implementations 7
3.2 Relevant libraries and extensions 9

4 Interoperability and Applicability Considerations 11
5 Future versions of the specification 11
6 References 12
7 RFC Authors' Addresses 13

1 Introduction

Zarr is a community project to develop specifications and software for storage of large N-dimensional typed arrays. A particular focus of Zarr is to provide support for storage using distributed systems like cloud object stores, and to enable efficient I/O for parallel computing applications [1].

1.1 Motivation

One of EOSDIS' primary motivations for migrating NASA Earth Science data holdings to the cloud is to enable data-adjacent computing capabilities at scale for all. By data-adjacent we mean that the data is close, network-wise, to the compute you bring to bear on that data.

With the rapid explosion of data volumes, and the corresponding explosion in the compute necessary to produce science from that data, it is of paramount importance that users in the cloud can access cloud-hosted data in an efficient and performant manner. Current EOSDIS archival data formats are optimized for file system storage rather than cloud object storage. Consequently, capabilities offered for accessing and using those datasets are optimized for traditional file systems on local storage.

Given the different latency profiles of cloud and local storage, a cloud-optimized format should facilitate minimizing the number of read operations whilst also minimizing the volume of data accessed.

Since EOSDIS' traditional formats, such as HDF4, HDF-EOS, HDF5, NetCDF-3, and NetCDF-4[1] are optimized for storage, they produce large files. Quite often, a user will only need a small percentage of the file to do their analysis. In order to determine which part of the file they need, in the case of NetCDF and non-optimized HDF (Hierarchical Data Format), they need to read a significant percentage of the file. When accessing the file through a distributed network rather than a local file system this pattern is the opposite of the optimal case of minimizing reads and volume of data accessed.

A cloud-optimized format needs to support the following,

- High-throughput distributed I/O
- Parallel processing at the process and thread level

[1] Explanation of versions of netCDF - https://docs.unidata.ucar.edu/nug/current/netcdf_introduction.html

- Chunking - the ability to split data into manageable, addressable parts along specific dimensions. This allows the user or client to make performant decisions on what data users access within a data store.
- Chunk compression - the ability to compress individual chunks rather than the dataset as a whole so that decompression of data does not require access to the entire dataset.

One such format, optimized for object stores such as AWS (Amazon Web Services) S3 (simple storage service) [3], is Zarr. Zarr is specifically useful for the earth science domain as it supports large, multi-dimensional data arrays and is compatible with a variety of existing data science tooling such as Dask [4] and Xarray [5].

The Zarr data format utilizes the following features, in addition/accordance to the items list above, to achieve cloud-optimized data access,

- Random access with comparatively few communications
- Consolidated metadata. Other existing archival formats tend to place their metadata throughout the file space, requiring large or multiple reads to obtain the metadata.
- Chunking across any dimension
- 1:1 file/chunk ratio[2] to ease parallelism. Other existing archival formats use single binary files and tend to provide library support that is thread-safe rather than thread-optimized.
- Chunk-based compression

Zarr also features several other benefits that are not cloud-specific, such as,

- An open specification
- A defined governance model
- A hierarchical data structure
- Metadata in a common plain-text format (JSON). Other existing archival formats tend to describe their metadata in binary formats.

1.2 Evidence of Implementation

1.2.1 Pangeo Forge

The goal of the Pangeo Forge project [6] is to make it easy to extract data from traditional data repositories and deposit them in cloud object storage in analysis-ready, cloud optimized (ARCO) formats. It uses the Zarr format to achieve that goal. It has produced numerous datasets in the Zarr format on AWS S3 using various earth science sources such as NASA, NOAA, DOE, The Australian Bureau of Meteorology and the UK Met Office. It is an open source, community-driven effort that has integrations with NASA EOSDIS' CMR for finding canonical NASA earth science datasets as inputs for Zarr arrays.

The Pangeo Forge Python SDK provides the components for generating Zarr arrays from archival data stores in a variety of formats (such as NetCDF and HDF) and community-built recipes based on those components.

[2] This is true for version 2 of the Zarr specification. Version 3 of the Zarr specification supports sharding. See section 5 for more information on this.

1.2.2 Giovanni in the cloud

The Geospatial Interactive Online Visualization And Analysis Infrastructure (Giovanni) [7], developed at NASA's GES DISC, is using Zarr to provide time and area averaging services for GES DISC data. They achieve this in a performant manner by using a single grid-cell time series service based on the Zarr storage format in AWS S3. GES DISC currently holds 1.82 TB of data in Zarr format, representing six variables with a projected 20 TB of data representing 2,000 variables in the GES DISC archive.

1.2.3 AWS Open Data Registry

There are a variety of Zarr arrays related to Earth Science data in AWS' Open Data Registry [8]. One, oft-cited, example is the Multi-Scale Ultra High Resolution (MUR) Sea Surface Temperature (SST) dataset [9] which provides a Zarr array for PO.DAAC's global, gap-free, gridded, daily 1 km Sea Surface Temperature dataset.

1.2.4 OGC Zarr spec 2.0

The Open Geospatial Consortium has endorsed Zarr 2.0 as a community standard.

1.2.5 Zarr arrays and dynamic imagery generation

The access performance of Zarr arrays opens up the possibility of imagery services providing functionality without persisting intermediate assets. Historically, imagery services such as NASA GIBS [10] have generated persistent stores of static imagery from archival data to provide WMS and WMTS services. Zarr storage in the cloud allows the possibility of generating the imagery required for their services on-demand from a Zarr array.

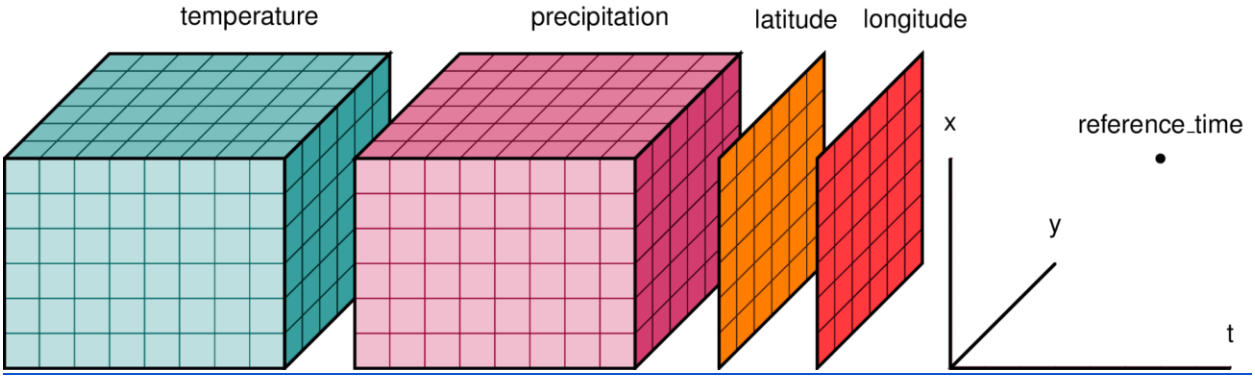
This architectural change is also being pursued by NASA IMPACT's VEDA [11] to publish and visualize NASA's Earthdata Zarr archives and deliver interoperable APIs for its data stores to support dynamic data visualization and storytelling.

It should be noted that the conventional Zarr chunk size stored for analysis (~100MB or more) is deemed too large for in-browser transformation and visualization. The upper limit of request sizes that can be reasonably fetched by a browser is ~10MB. This leads to a disparity between two very popular use cases: analytics and visualization that cannot be solved by a single Zarr array in version 2 of the specification. However, the ZEP 0002 extension (Sharding: see section 5.1) can resolve this issue and accommodate both access patterns in a single dataset.

2 Zarr Structure

Like HDF5, the Zarr format is designed to store an arbitrary number of array variables of arbitrary dimensionality in a chunked, binary format, along with arbitrary metadata. Unlike HDF5, a Zarr array is not a single entity but, conceptually, a key/value interface with read, write and delete methods. In the case of a file store implementation, each variable is contained within a single directory that contains the variable's metadata files, in plain-text JSON, and a set of compressed files representing the chunks of the variable's data.

For example, measuring temperature and precipitation over a spatial and temporal area could be visualized as follows,



[3]

Figure 1: Visual representation of multidimensional data

Each block in the first array would represent a temperature at a given spatial location (expressed in latitude and longitude) for a given time. The second block would represent precipitation in the same manner.

It should be noted that a Zarr array would represent either an entire collection, in EOSDIS parlance, or a sizable subset of a collection rather than a single granule/file. The hierarchical nature of Zarr allows the user to only interact with the parts of that array they are interested in.

The Zarr format consists of metadata in the json format that enables simple and efficient random access to a 'Zarr array' which contains a set of compressed, binary data objects arranged in a manner that matches the chunking scheme described in the metadata. This allows each chunk to be accessed, decompressed, and analyzed in isolation.

[3] https://xarray.dev/

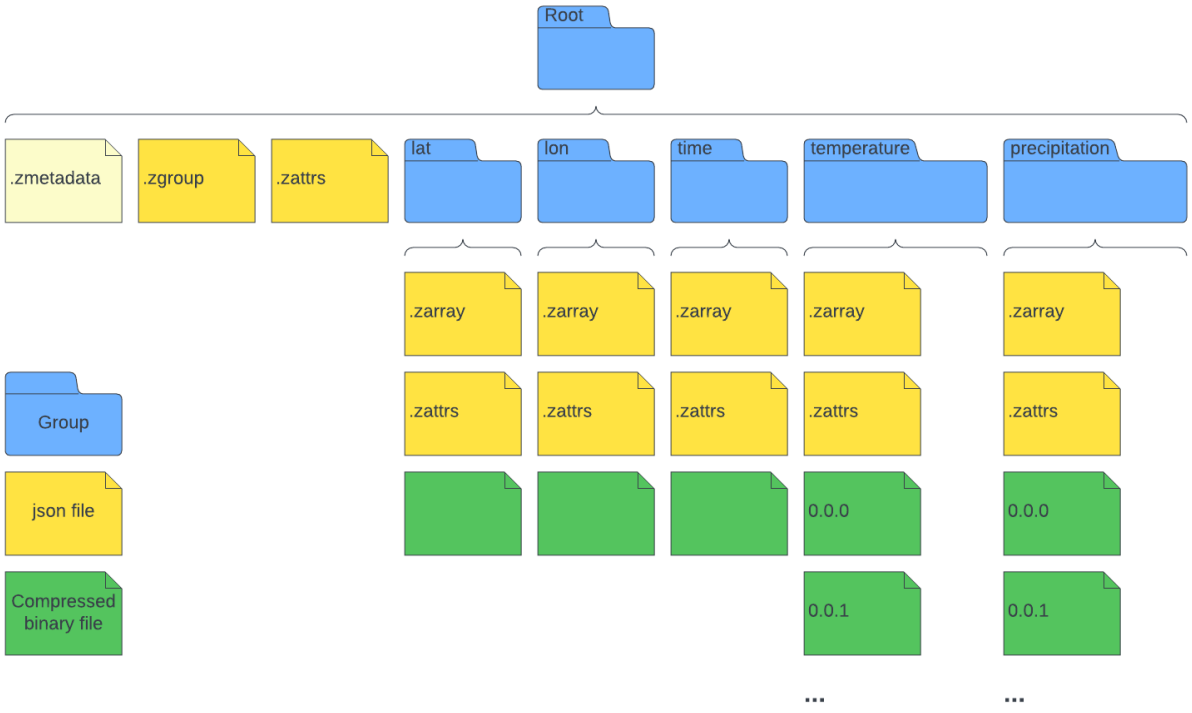


Figure 2: Visual representation of Zarr data layout

2.1 Metadata

Information about the structure and composition of the arrays in a datastore is represented in several json files in the root and groups of the store.

2.1.1 .zarray

This file describes the composition of an array including the dimensions or shape, the chunking strategy, and the compression used on each chunk.

```
{'shape': [8, 6, 6],
 'chunks': [4, 3],
 'compressor': {'blocksize': 0,
                'clevel': 5,
                'cname': 'lz4',
                'id': 'blosc',
                'shuffle': 1},
 'dtype': '<f8',
 'fill_value': 0.0,
 'filters': None,
 'order': 'C',
 'zarr_format': 2}
```

2.1.2 .zattrs

This file describes attributes associated with an array. For example, the CF standard name of the measurement described and its units.

```
{'standard_name': 'air_temperature',
'units': 'degC'}
```

2.1.3 .zmetadata

In order to optimize performance for network distributed stores containing multiple datasets/groups, Zarr allows you to consolidate metadata associated with multiple arrays (.zarray, .zattrs) in a single top level artifact (.zmetadata). This means that the entire structure of the zarr array can be determined with a single request.

2.2 Data

Each array contains one or more chunks. Each chunk is a binary, compressed object/file. That chunk will contain a segment of the array defined by the.zarray file. These chunks are named according to their location in the array.

3 API

Zarr provides library support in numerous languages to read and write Zarr to a number of persistence stores [12]. Examples of API functionality are shown in Python for simplicity. It should be noted that file access is generally achieved by other libraries that use Zarr as a data store. Those libraries are detailed in the 'relevant libraries and extensions' section of this document.

3.1 Implementations

All that is required to implement a Zarr reader is the ability to parse JSON, to navigate a file system directory structure, and to inflate compressed blobs according to the indicated compression algorithm. Below are listed examples of libraries that implement Zarr[4]

- Python: https://github.com/zarr-developers/zarr-python
- Julia: https://github.com/JuliaIO/Zarr.jl
- C: https://github.com/Unidata/netcdf-c
- C++: https://github.com/constantinpape/z5
- Scala: https://github.com/lasersonlab/ndarray.scala
- Java: https://github.com/bcdev/jzarr
- Javascript: https://github.com/gzuidhof/zarr.js
- R: https://github.com/grimbough/Rarr
- Rust: https://github.com/aschampion/rust-n5

[4] For an exhaustive list see https://zarr.dev/implementations/

3.1.1 Creating and chunking a Zarr array

The following, pythonic code, is a trivial example of a Zarr array creation to demonstrate that chunking characteristics can be assigned on creation.

```
>import zarr
>z=zarr.zeros((10000,10000),chunks=(1000,1000), dtype='i4')
>z
<zarr.core.Array (10000, 10000) int32>
```

3.1.2 Compression

Each chunk in a Zarr array can be compressed. This means that data users can take advantage of the lower latency compression affords but still limit the amount of data they access. Data curators can choose from a variety of compression algorithms to suit their needs.

```
>from numcodecs import Blosc
>compressor=Blosc(cname='zstd',clevel=3,shuffle=Blosc.BITSHUFFLE)
>data=np.arange(100000000,dtype='i4').reshape(10000,10000)
>z=zarr.array(data,chunks=(1000,1000),compressor=compressor)
>z.compressor
Blosc(cname='zstd',clevel=3,shuffle=BITSHUFFLE,blocksize=0)
```

3.1.3 Reading from and writing to a Zarr array

In python, writing to and reading from a Zarr array can be achieved in the same way one would do it using NumPy.

```
>z[0:0] = 42
>z[0, 0]
42
>z[0,1]
0
```

3.1.4 Storage options

Python Zarr can use any object that implements the MutableMapping interface from the collections module in the Python standard library as the store for a group or an array.

3.1.4.1 Local storage

The simplest local storage is a file system. Zarr refers to this as a directory store.

```
> store = zarr.DirectoryStore('data/example.zarr')
```

But other local stores can be used, such as Zip Files or relational databases.

```
> store = zarr.ZipStore('data/example.zip', mode='r')
> store = zarr.SQLiteStore('data/example.sqldb')
```

3.1.4.2 Distributed storage

Zarr provides interfaces to different distributed storage mechanisms, a key factor in our recommendation.

```
>import s3fs
>import zarr
>s3=s3fs.S3FileSystem(anon=True,client_kwargs=dict(region_name
='eu-west-2'))
>store=s3fs.S3Map(root='zarr-demo/store', s3=s3, check=False)
>root = zarr.group(store=store)
>z = root['foo/bar/baz']

>import azure.storage.blob
>container_client = azure.storage.blob.ContainerClient(...)
>store=zarr.ABSStore(client=container_client,prefix='zarr-
testing')
>root=zarr.group(store=store, overwrite=True)
```

3.1.5 Parallel computing

Zarr arrays have been designed for both concurrent read and write operations in parallel computations. Concurrent write operations may occur, if each writer is updating a different chunk.

Both multi-threaded and multi-process parallelism are possible. If each worker in a parallel computation is writing to a separate chunk, then no synchronization is required.

If concurrent writes are required across chunks then Zarr can be configured to support that but performance will suffer as a consequence.

```
>z=zarr.zeros((10000,10000),chunks=(1000,1000),  dtype='i4',
synchronizer=zarr.ThreadSynchronizer())

>synchronizer=zarr.ProcessSynchronizer('data/example.sync')

>z=zarr.open_array('data/example',mode='w',shape=(10000,
10000),chunks=(1000,1000), dtype='i4',synchronizer=synchronizer)
```

3.2 Relevant libraries and extensions

The relative simplicity of the Zarr format dramatically facilitates the implementations of Zarr drivers in different programming languages.

3.2.1 GeoZarr specification

GeoZarr[13] is a geospatial specification for Zarr. It leverages CF conventions to provide a means of georeferencing multidimensional arrays of geospatial observations. As of August 2023, the charter for a GeoZarr working group at OGC has been submitted for review [14].

3.2.2 Xarray library

The popular N-Dimensional Array library, Xarray, supports reading from and writing to Zarr arrays.

A user can leverage the xarray.open_zarr method on an existing Zarr array and utilize the suite of XArray functionality to access and analyze the data. Xarray will leverage the Zarr specification [2] and API to be performant when interfacing with distributed network storage, essentially abstracting away the underlying storage mechanism from the user.

3.2.3 Dask library

Zarr can be used with Dask to provide multi-threaded or multi-process parallelization utilizing lazy evaluation of arrays (ie. only accessing the parts of the arrays you need).

3.2.4 Intake library

Intake is a lightweight interface for loading and sharing data in data science projects. The intake-xarray library supports the Zarr data format.

3.2.5 Kerchunk library

Kerchunk allows you to access a variety of legacy scientific formats (including NetCDF and HDF5) as if they were a Zarr array, with the benefits Zarr provides for distributed network storage such as cloud object stores. This technique is attractive in that it does not require the reformatting and copying of the original archive to a Zarr array. Kerchunk works with the original format to provide to the user the functionality and performance of a Zarr array.

3.2.6 GDAL library

The GDAL [15] translation library provides a common API for working with a variety of raster data formats as well as tools for conversion between different raster formats. Zarr is a supported format through both GDAL's classic (2-dimensional) raster API and, since GDAL 3.1, through GDAL's multidimensional data API, which allows both direct analysis of Zarr archives using certain GDAL utilities and transformation between Zarr and other formats (like NetCDF or GeoTIFF). Support exists for version 2 of the specification with support for version 3 in an experimental phase.

3.2.7 NetCDF library

Zarr archives that follow specific metadata specifications — specifically, either the Xarray-Zarr specification or the NCZarr specification — can be opened and analyzed using the NetCDF C library.

4 Interoperability and Applicability Considerations

Zarr arrays are most suitable for gridded data that are accessible via a distributed network. Non-gridded data, swath data for example, are less well-suited for replication or representation as a Zarr array.

EOSDIS' migration to the cloud makes a strong case for the provision of data formats that are designed for cloud usage in addition to our traditional formats. Zarr is a strong candidate for multi-dimensional data which constitutes the bulk of the EOSDIS data archive, along with Kerchunk's ability to provide a Zarr interface for legacy formats, and HDF5's cloud optimized additions[5]. Cloud Optimized GeoTIFF (COG) does have applicability to EOSDIS data, but is out of scope for this document.

While we expect EOSDIS to produce Zarr data stores, we also expect the community at large to produce them, deriving them from EOSDIS archival data. One example of this is the Pangeo Forge effort mentioned elsewhere in this document. A further driver for a more community-driven approach is that the considerations taken to produce a Zarr array depend on the user consuming the data, specifically the chunking strategy. This means that one archived data set could have multiple Zarr arrays tailored for different uses. This plays, somewhat, into the Pangeo-Forge philosophy that the generation of cloud-optimized data is not the sole domain of the data curator but also of the data scientist.

Zarr is a work in progress, particularly in the realm of metadata standards. Attempts are in progress to mitigate this risk with proposals such as GeoZarr which will attempt to standardize geospatial Zarr arrays. Even so, there is a ground swell of adoption both within EOSDIS (see GES DISC work for Giovanni) and the earth science data community (see Pangeo Forge).

There are areas in the geospatial tools domain where Zarr is not well supported. QGIS does not currently support Zarr as a data format (although there is a pending feature request). ArcGIS Pro 3.2, however, does support Zarr as a multidimensional raster data format.

The Zarr format has some issues in the area of updating existing Zarr arrays with new data when that array is concurrently being accessed by users. When a user reads the metadata associated with a Zarr array in order to obtain the relevant parts they assume that the metadata is static. Updating a Zarr array can alter this metadata, introducing risk to the access of the data based on the old metadata. The relative size of a Zarr array increases this risk, compared to other data formats. The enhancement proposal ZEP 2 has provisions to mitigate this risk. This is outlined in section 5.1

5 Future versions of the specification

The current version of Zarr is version 2 [2] and that is the version endorsed by this recommended standard document. Version 3 of Zarr is currently under development at time of writing. Version 3 focuses on the following,

- Feature parity and full interoperability across all major programming languages.

[5] https://www.youtube.com/watch?v=bDH59YTXpkc

- Support for novel encoding technologies, storage technologies and features by a broader community.
- Reasonable performance characteristics of all Zarr implementations across a variety of different underlying storage technologies, including storage with high latency per operation.
- Improve performance for data with a very large number of chunks and/or with a variety of common access patterns.

Additionally there are a number of Zarr Enhancement Proposals (ZEP) that are in flight including the sharding codec detailed below.

5.1 ZEP 2 Sharding codec

The storage of very large (tera and ultimately peta-scale), chunked arrays, is inefficient or even impossible due to the number of files/objects required. As the number of chunks increases in a Zarr array the number of files required in a traditional file system can reach block size and inode limits. In object store systems such as S3 and GCS, large numbers of small objects cannot be handled efficiently.

To reduce the number of entities (files or objects) in a Zarr array without increasing the chunk size (which would be problematic for streaming data in browser-based visualization software) ZEP 2 allows combining multiple chunks into single storage keys. This technique is referred to as sharding.

Users can either read an entire shard (outer chunk) or, read the metadata of the shard to determine the byte-range of one or more chunks within that shard and then obtain only the (inner) chunk(s) they are interested in.

Sharding has the potential to mitigate some risks outside of very large arrays. For example, the differing chunk size needs for visualization and analytics could be catered for by shard size and chunk size considerations. It also has the provision to allow in-place update of compressed chunks without altering the metadata (a risk outlined in section 4) by leveraging unused space within a chunk.

6 References

6.1 Normative references

[1] Zarr homepage: https://zarr.dev

[2] Zarr specification: https://github.com/zarr-developers/zarr-specs

6.2 Informative references

[3] S3: https://aws.amazon.com/s3/

[4] Dask: https://www.dask.org/

[5] Xarray: https://docs.xarray.dev/en/stable/

[6] Pangeo Forge: https://pangeo-forge.org/

[8] Giovanni: https://earth.gsfc.nasa.gov/ocean/data/giovanni

[8] AWS open data registry: https://registry.opendata.aws

[9] MUR dataset on the AWS open data registry: https://registry.opendata.aws/mur/

[10] GIBS: https://wiki.earthdata.nasa.gov/display/GIBS/

[11] VEDA: https://www.earthdata.nasa.gov/esds/veda

[12] Zarr documentation: https://zarr.readthedocs.io/en/stable/

[13] GeoZarr: https://github.com/zarr-developers/geozarr-spec

[14] OGC GeoZarr Standards Working Group Charter: https://portal.ogc.org/files/105667 and https://eosdis.slack.com/archives/C04D3084GAZ/p1693497131582929

[15] GDAL: https://gdal.org/drivers/raster/zarr.html

7 RFC Authors' Addresses

Newman, Douglas J. douglas.j.newman@nasa.gov

ESDIS Standards Office (ESCO)

Email: esco-staff@lists.nasa.gov

Web: https://earthdata.nasa.gov/esdis/esdis-standards-office-esco

Appendix A

Acronym	Description
API	Application Programming Interface
ARCO	Analysis-ready cloud optimized data
AWS	Amazon Web Services
CF	Climate and Forecast Metadata conventions
CMR	Common Metadata Repository
DOE	Department of Energy
EOSDIS	Earth Observing System Data and Information System
GES DISC	Goddard Earth Sciences Data and Information Services Center
GDAL	Geospatial Data Abstraction Library
GIBS	Global Imagery Browse Services
Giovanni	Geospatial Interactive Online Visualization ANd aNalysis Infrastructure
IMPACT	Interagency Implementation and Advanced Concepts Team
HDF	Hierarchical Data Format

MUR	Multi-scale Ultra-high Resolution
NetCDF	Network Common Data Form
NOAA	National Oceanic and Atmospheric Administration
OGC	Open Geospatial Consortium
PO.DAAC	Physical Oceanography Distributed Active Archive Center
S3	Simple Storage Service
SST	Sea Surface Temperature
VEDA	Visualization, Exploration, and Data Analysis
WMS	OGC Web Mapping Service
WMTS	OGC Web Map Tiling Service