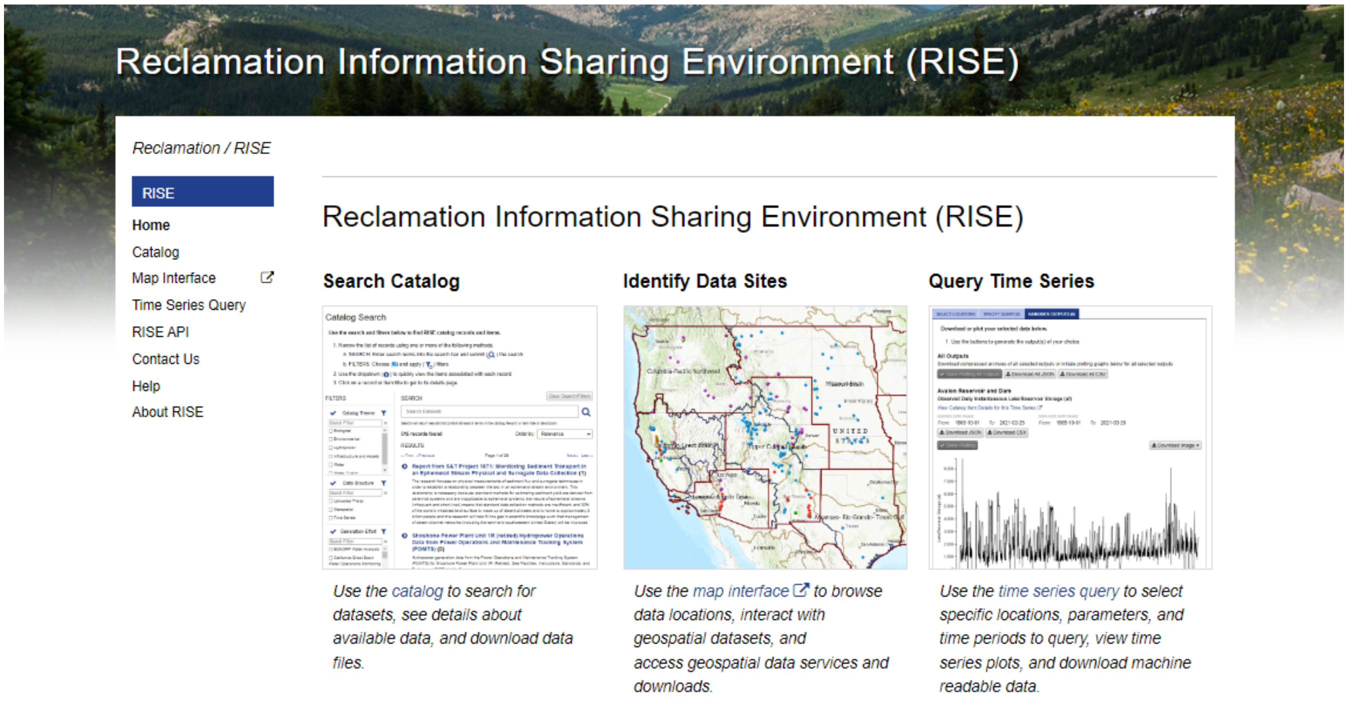
BUREAU OF RECLAMATION

Evaluation of Storage Formats for Archive and Transfer of Large Datasets in the RISE Platform

Science and Technology Program
Research and Development Office
Final Report No. ST-2022-22041-01



Reclamation Information Sharing Environment (RISE)

Search Catalog

Use the catalog to search for datasets, see details about available data, and download data files.

Identify Data Sites

Use the map interface to browse data locations, interact with geospatial datasets, and access geospatial data services and downloads.

Query Time Series

Use the time series query to select specific locations, parameters, and time periods to query, view time series plots, and download machine readable data.

U.S. Department of the Interior **December 2023**

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.			
1. REPORT DATE *(DD-MM-YYYY)* 31-12-2023	2. REPORT TYPE Research		3. DATES COVERED *(From - To)* 2022-2023
4. TITLE AND SUBTITLE Evaluation of Storage Formats for Archive and Transfer of Large Datasets in the RISE Platform			5a. CONTRACT NUMBER XXXR4524KS-RR4888FARD2201801
			5b. GRANT NUMBER
			5c. PROGRAM ELEMENT NUMBER 1541 (S&T)
6. AUTHOR(S) Drew Allan Loney, PhD PE; Allison Odell, PE			5d. PROJECT NUMBER Final Report No. ST-2022-22041-01
			5e. TASK NUMBER
			5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Applied Hydrology II, Technical Service Center, Bureau of Reclamation, Denver, CO 80225; Research and Development Office, Bureau of Reclamation, Denver, CO 80225			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Science and Technology Program, Research and Development Office, Bureau of Reclamation, U.S. Department of the Interior, Denver Federal Center, PO Box 25007, Denver, CO 80225-0007			10. SPONSOR/MONITOR'S ACRONYM(S) Reclamation
			11. SPONSOR/MONITOR'S REPORT NUMBER(S) Final Report No. ST-2022-22041-01
12. DISTRIBUTION/AVAILABILITY STATEMENT Final Report may be downloaded from https://www.usbr.gov/research/projects/index.html			
13. SUPPLEMENTARY NOTES			
14. ABSTRACT This report documents the evaluation and selection of large data file formats for the RISE platform. Additionally, a preliminary list of identified changes to the RISE platform needed to support the netCDF format is provided. The intent is to frame future RISE development by providing a roadmap to support large datasets within the platform.			
15. SUBJECT TERMS Open data, storage, modeling			
16. SECURITY CLASSIFICATION OF: a. REPORT U; b. ABSTRACT U; THIS PAGE U	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Drew Allan Loney, PhD PE; 19b. TELEPHONE NUMBER *(Include area code)* (303)445-2541

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Acknowledgements

The Science and Technology Program, Bureau of Reclamation, sponsored this research.

Cover Photo – The Reclamation Information Sharing Environment website (Reclamation, 2023)

Evaluation of Storage Formats for Archive and Transfer of Large Datasets in the RISE Platform

Final Report No. ST-2022-22041-01

Prepared by:

Bureau of Reclamation
Technical Service Center
Denver, Colorado

Bureau of Reclamation
Research and Development Office
Denver, Colorado

U.S. Department of the Interior **December 2023**

Evaluation of Storage Formats for Archive and Transfer of Large Datasets in the RISE Platform

DREW LONEY Digitally signed by DREW LONEY
Date: 2024.01.03 08:50:23 -07'00'

Prepared by: Drew Allan Loney, PhD PE

ALLISON ODELL Digitally signed by ALLISON ODELL
Date: 2024.01.02 14:59:20 -08'00'

Prepared by: Allison Odell, PE

Peer Review Certification

This section has been reviewed and is believed to be in accordance with the service agreement and standards of the profession.

LINDSAY BEARUP Digitally signed by LINDSAY BEARUP
Date: 2024.01.03 07:33:26 -07'00'

Peer reviewed by: Lindsay Bearup, PhD PE

Acronyms and Abbreviations

API	Application Program Interface
ASO	Airborne Snow Observatory
CMIP5	Coupled Model Intercomparison Project 5
ET	Evapotranspiration
GIS	Geographic Information System
GUI	Graphical User Interface
HDF5	Hierarchical Data Format 5
IT	Information Technology
lidar	Light Detection and Ranging
NASA	National Aeronautics and Space Administration
netCDF	network Common Data Form
NVMe	Nonvolatile Memory express
PRISM	Parameter-elevation Regressions On Independent Slopes Model
RISE	Reclamation Information Sharing Environment
SPT	Standard Penetration Test
UCAR	University Corporation for Atmospheric Research
WRF	Weather, Research, and Forecasting Model

Contents

Figures

Appendices

Executive Summary

The size of engineering and scientific datasets continues to grow as observation coverage improves and models increase in complexity. Neglecting these trends, the size of existing datasets will naturally continue to grow with the period of record. The number, size, and complexity of Reclamation datasets is a challenge for processing, archiving, and serving data. These operations are necessary for Reclamation to utilize existing information as well as to comply with government open data requirements (Foundations for Evidence-Based Policymaking Act of 2018, 2019). Reclamation has developed the Reclamation Information Sharing Environment (RISE) platform to publish its mission-related datasets for access by internal and external users (Bureau of Reclamation, 2023). For RISE to successfully serve large datasets, data must be stored in formats that minimize processing, storage, and bandwidth requirements. Moreover, file formats must be selected that are compatible with the observations and models that produce the datasets. This has led to a need to evaluate file formats to better incorporate large data file formats in the RISE platform.

The Reclamation Research and Development Office funded an evaluation of file formats for large datasets to use in RISE through the Science & Technology Program. A team of Reclamation scientific and information technology (IT) subject matter experts evaluated multiple file formats commonly utilized for scientific data through literature review and independent benchmarks. The network Common Data Form (netCDF) and Zarr formats were identified as open-source options that could meet a variety of Reclamation use cases. The formats allow for metadata, data compression, subsetting, and appending in a single file using an efficient binary format. Additionally, the Zarr format is optimized for cloud storage applications. While support of both formats would provide the most flexibility, the maturity of the netCDF format led to its prioritization as the preferred RISE file format for large datasets.

This report documents the evaluation and selection of large data file formats for the RISE platform. Additionally, a preliminary list of identified changes to the RISE platform needed to support the netCDF format is provided. The intent is to frame future RISE development by providing a roadmap to support large datasets within the platform.

1.0 Introduction

The size of scientific and engineering datasets continues to grow as observation coverage improves and models increase in complexity. Neglecting these trends, the size of existing datasets will naturally continue to grow with the period of record. The number, size, and complexity of Reclamation datasets is a challenge for processing, archiving, and serving data. These operations are necessary for Reclamation to utilize existing information as well as to comply with government open data requirements (Foundations for Evidence-Based Policymaking Act of 2018, 2019). Reclamation has developed the Reclamation Information Sharing Environment (RISE) platform to publish its mission-related datasets for access by internal and external users (Bureau of Reclamation, 2023). For RISE to successfully serve large datasets, data must be stored in formats that minimize processing, storage, and bandwidth requirements. Moreover, file formats must be selected that are compatible with the observations and models that produce the datasets.

Here, large datasets refer to single, large files containing a group of related values that can potentially grow into the hundreds of terabytes. Reclamation naturally produces large datasets from its measurements and model output. Measurements can be large from both temporal and spatial coverage or in the number of available variables. Examples include lidar and high frequency sampling evapotranspiration (ET) data. Reclamation also generates large datasets from model output, such as the Weather, Research, and Forecasting (WRF) model (National Center for Atmospheric Research, 2023). While these datasets may not be tied to a specific use case, oftentimes a considerable number of related large datasets may be created for a specific project. As RISE exists to support data publication and these datasets may fall under open data requirements, it follows to expand the capability of the RISE platform to support these datasets.

Large datasets come with many challenges. For the person who creates the dataset, there is the initial challenge of adding the data into RISE. Large datasets may overwhelm a network connection or take unfeasibly long to transfer given the data size and network speed. Once the dataset is in RISE, it needs to be stored in a format that minimizes the storage requirement while maintaining data integrity and redundancy. The final challenge is transferring the dataset to a data consumer from RISE when requested. This may require preprocessing within RISE to extract and prepare the data. It also mirrors the bandwidth and transfer challenges of adding data into RISE. The complexity of the workflow creates tradeoffs between storage, compute time, and bandwidth that must be investigated to determine what approach best meets the needs of Reclamation.

The purpose of this document is to describe the selection of netCDF (and secondarily Zarr) as the file format to incorporate large dataset support into the RISE platform. Additionally, possible workflows and changes to the RISE system to accommodate large datasets are described. The intent is to highlight how data producers and consumers will interact with the system to provide and request data in the format and provide a roadmap for implementation of netCDF support within the RISE platform.

2.0 Format Evaluation

The Reclamation team began with a literature review of commonly available scientific file formats and studies that characterize their performance. While there are numerous ad hoc studies from the open-source community, few provided a level a rigor that the team desired for decision making. These more rigorous studies included: Ambatipudi & Byna, 2023; Jamison & Sommerville, 2023; Lulla et al., 2022; Moore et al., 2021; and National Aeronautics and Space Administration, 2023b. The most meticulous was a National Aeronautics and Space Administration (NASA) funded study conducted under contract by Raytheon (Durbin et al., 2020). The study highlighted the GeoTIFF, netCDF, and Zarr formats as particular standouts for being open source as well as having multi-dimensional data support, reasonable performance, and a large user base. However, the study ultimately did not make a single recommendation on a particular format, noting that the preferred format would be determined by how a user applied weights to evaluation metrics and would change based on the use case. The study did note that at the time of the analysis the Zarr format had not been accepted by a commonly accepted standards body. Since that time, the Open Geospatial Consortium has accepted a Zarr standard (*Zarr Storage Specification 2.0 Community Standard*, 2022). Based on the NASA recommendation, the GeoTiff, netCDF, and Zarr formats were selected for further investigation.

GeoTIFF is a raster file format commonly used for satellite and aerial imagery (NASA, 2021a). The original standard dates to 1995. Since then, it has gone through several revisions, including development of a version that is cloud optimized. Because of its early development, the format is broadly supported and is a stalwart within the geographic information systems (GIS) and remote sensing communities. However, the format has not been broadly accepted by the modeling community as the format struggles with complex multi-dimensional data structures. Storing multiple related variables of different shapes in the same file can also be challenging.

NetCDF is a broadly accepted machine/architecture independent file format utilized across the scientific community (NASA, 2021b). The netCDF-4 standard is also compatible with the Hierarchical Data Format Version 5 (HDF5) engine, another storage format that has additional features relative to netCDF such as compression, parallel operations, and chunking. The netCDF format facilitates access to the more advanced HDF5 functionality with a more user-friendly application program interface (API). The netCDF format standard is maintained by the University Corporation for Atmospheric Research (UCAR) (University Corporation for Atmospheric Research, 2023). NetCDF is broadly flexible to handle a variety of data structures within the same file. A challenge of the format can be to ensure the necessary software libraries to work with the data are installed on the local machine, an issue made more complex if non-standard libraries such as HDF5 are utilized.

Zarr is a recent standard intended to extend the benefits of the netCDF format for cloud computing (NASA, 2023a; NumFOCUS, 2023). While the data model is similar to netCDF albeit with alterations to the data groups, it is a distinct format maintained separately from the netCDF standard. Cloud support is implemented by allowing users to access and download groups within the file independently, reducing bandwidth and storage by eliminating the need to download the full file. Zarr also supports more compression formats than netCDF. However, not all programming languages and models have developed support for the format; the Python language is the primary implementation and leads the development cycle. Additionally, where Zarr is supported, it can experience more issues as support is less mature than netCDF or GeoTiff.

2.1 Benchmarks

The evaluation team concurred with the NASA finding that the GeoTIFF, netCDF, and Zarr formats were best suited to Reclamation's requirements focused on scientific and engineering datasets. To better understand tradeoffs among the formats, Reclamation conducted benchmarks on datasets relevant to its use cases. Benchmarks were written in Python 3.11 using the xarray package, which implements all three formats in a standardized framework (Hoyer & Joseph, 2017). In addition to ensuring that all performance differences were related to the file formats alone, xarray also supports the most advanced features for each format. Code for the benchmarks is given in Appendix A.

Two benchmark datasets were utilized to compare format performance with different sized datasets. Climate projection data from the Coupled Model Intercomparison Project 5 (CMIP5) available on the Green Data Oasis site was utilized with approximately 2 GB of data (Lawrence Livermore National Laboratory, 2022). Input CMIP5 data consisted of ten years of consecutive annual precipitation files chosen at random. Each file provided coverage for the continental United States at 1/16th degree resolution. The second benchmark utilized precipitation data from the Parameter-elevation Regressions On Independent Slopes Model (PRISM) and was approximately 20 GB in size (*PRISM Climate Group, Oregon State University*, 2020). Input PRISM data consisted of five years of precipitation data from the 800m resolution product for the continental United States. The difference in dataset size was intended to capture how performance would scale as data size increased. Complexity of variables in the file was not explicitly examined as xarray supports only banded GeoTIFFs of constant shape rather than the arbitrarily complex structures of both netCDF and Zarr.

For both datasets, the source data format was converted into GeoTIFF, netCDF, and Zarr prior to the benchmarks using the default xarray settings for each format, if the data source had not already given the data in the desired format. Each benchmark was completed 100 times with median and range presented within the boxplots. Multiple iterations of the benchmark were conducted to describe variability not directly related to the benchmarks, such as system scheduling and read/write queues, that are not controllable by the benchmark but may affect the reported times. Benchmarks were completed on a Dell Precision 7560 laptop with an Intel i9-11950H processor with 32GB of memory on a NVMe based hard drive. While absolute performance is important, more critical is the relative performance among the formats.

2.1.1 CMIP5 Benchmark

Figure 1 gives the number of files and mean file size needed to represent the CMIP5 dataset. It is evident that there is a tradeoff between the complexity of the file structure and the number of files needed to store the data. Because the GeoTIFF format does not include a time dimension, each day in the CMIP5 dataset needs to be stored in a separate file. While it is possible to include days as raster bands, annotating the time dimension for the bands would not be as straightforward as compared to the netCDF and Zarr formats. Overall, the storage required for the GeoTIFF format was approximately three times that of the other formats. The netCDF and Zarr formats compared similarly to each other. The number of output files for the netCDF format was larger than the single output file for the Zarr format as the source data came as annual netCDF files and were directly utilized without preprocessing into a single netCDF file. The difference in the number of files is not anticipated to meaningfully affect the benchmark as xarray implements lazy read from disk for both formats. Total storage for the netCDF was less than that of the Zarr.

Figures 2 through 6 give the time and storage benchmarks for each format. The time required to load, access a year of data, and write that year to a new file were each tested. The storage required for that year of data was recorded for the netCDF and Zarr files. GeoTIFF was not further analyzed given its poor performance storing the initial dataset. Additionally, the time to subset each file to the Folsom Reservoir catchment was also calculated for the uncompressed and compressed data. For implementation of the commands used in the benchmarking analysis, see the python script in Appendix A.

No meaningful performance difference between the netCDF and Zarr formats is discernable. This reinforces the similarity between the data types in most circumstances. It is interesting to note that the storage required after the subset is larger than the full dataset for both formats. This is counterintuitive as less data is being stored in the subset. The reason for this is unknown. Subsetting employs the rasterio package that may alter the structure of the data such that it can be stored less efficiently. It is possible that this could be resolved with future tuning. The consistency between the formats implies this is caused by the Python library used to perform the subset or a result of a change to the data structure during the subset operation.

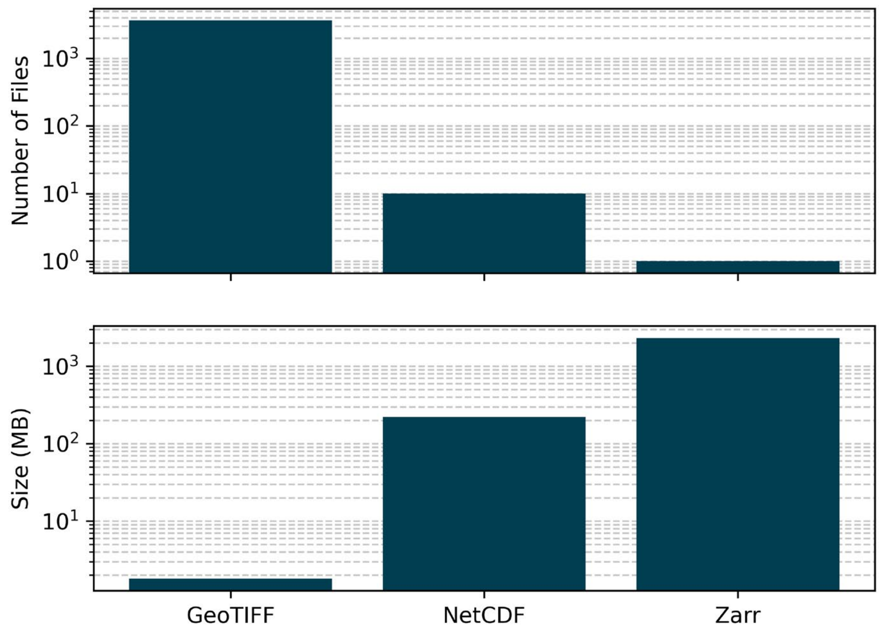


Figure 1: Number of files and mean file size required to represent the CMIP5 data with each format

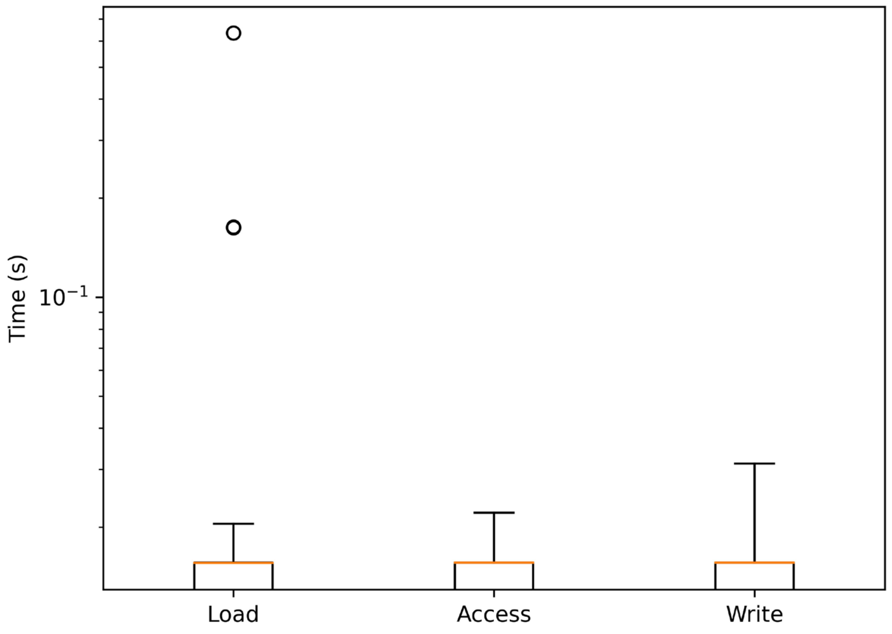


Figure 2: Time benchmarks for the CMIP5 dataset using the GeoTIFF format

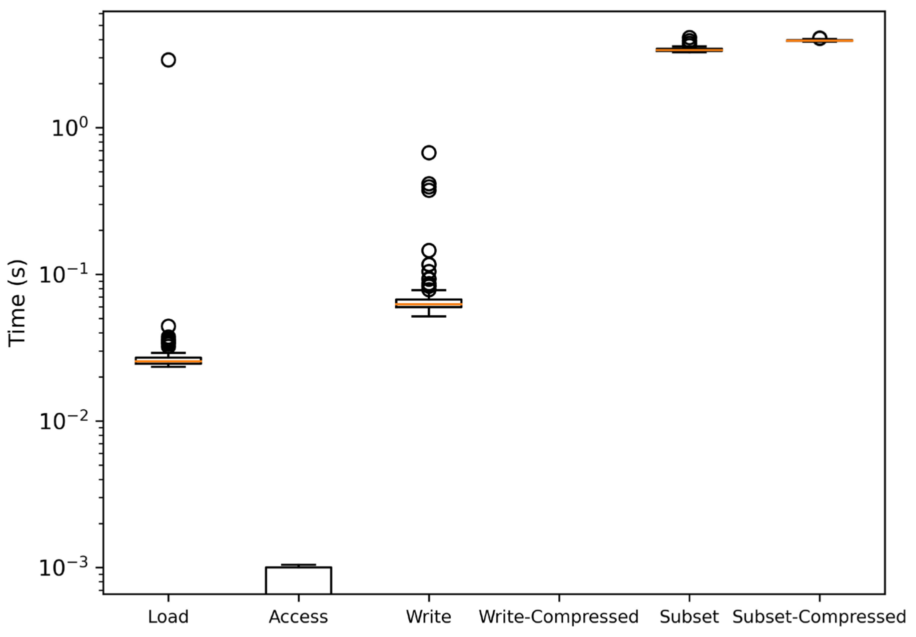


Figure 3: Time benchmarks for the CMIP5 dataset using the netCDF format

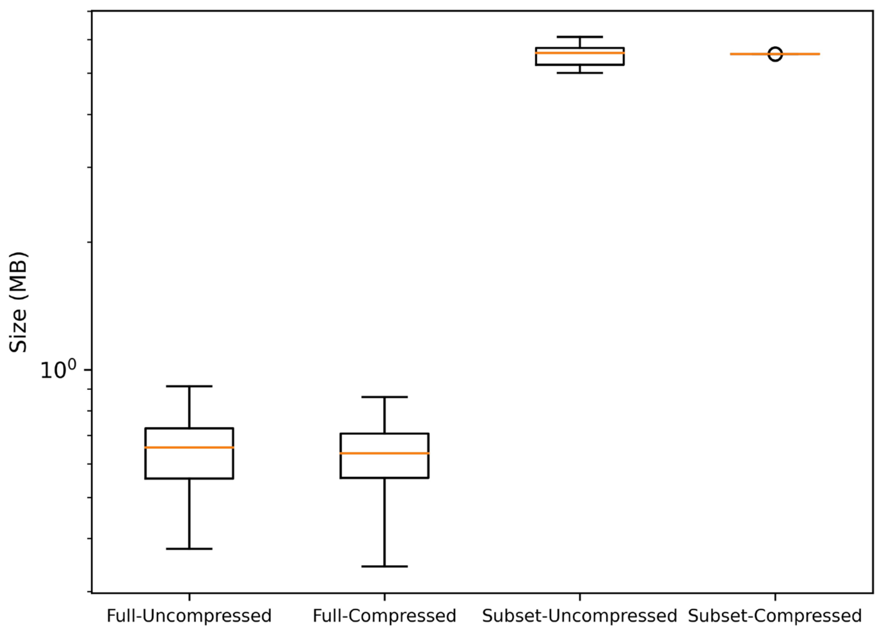


Figure 4: Storage benchmarks for the CMIP5 dataset using the netCDF format

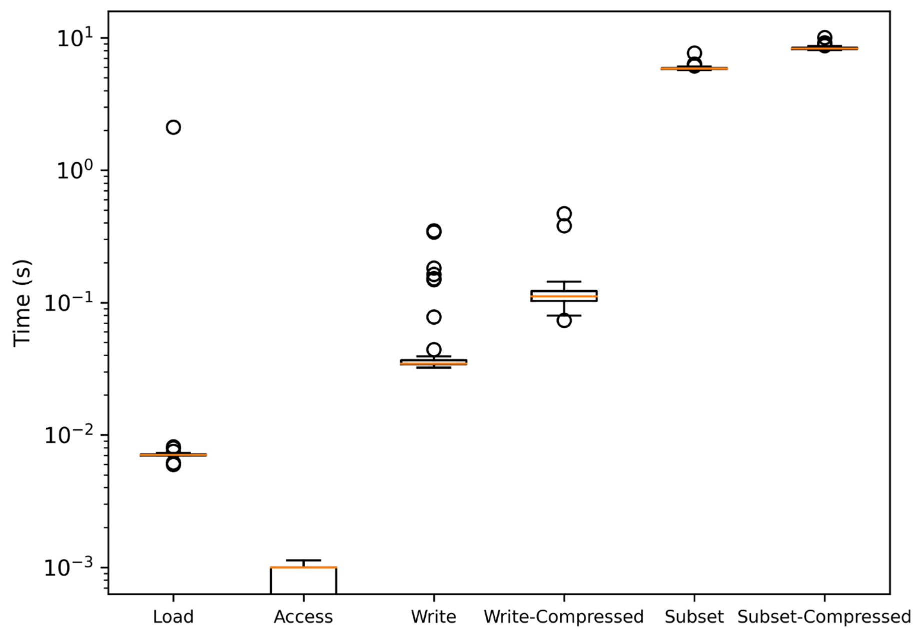


Figure 5: Time benchmarks for the CMIP5 dataset using the Zarr format

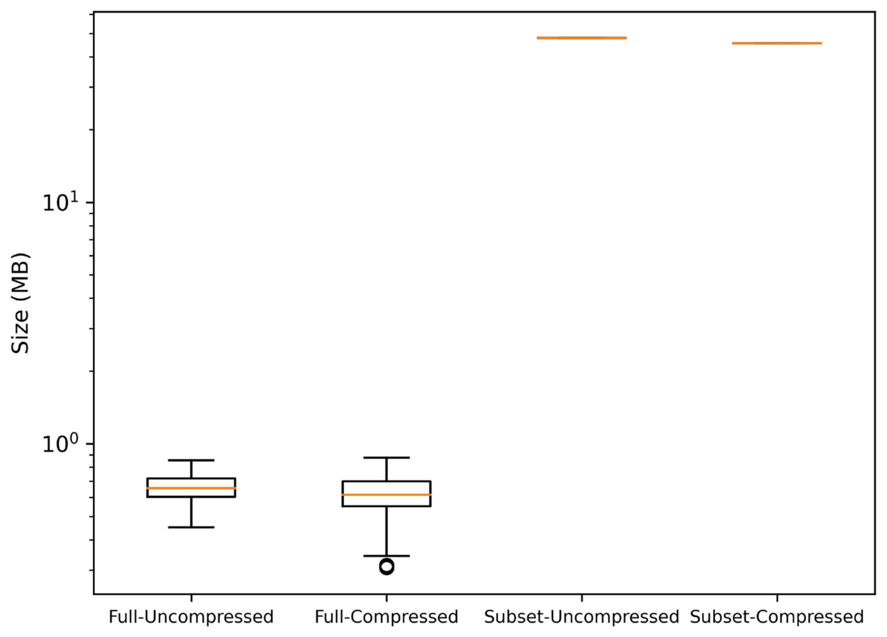


Figure 6: Storage benchmarks for the CMIP5 dataset using the Zarr format

2.1.2 PRISM Benchmark

Figure 7 gives the number of files and mean file size needed to represent the PRISM dataset. As with CMIP5 dataset, GeoTIFF requires a much greater number of files and significantly larger storage (by approximately a factor of seven and a half) for the data.

Figures 8 through 12 give the time and storage benchmarks for each format. NetCDF and Zarr again are similar throughout most of the benchmarks, with Zarr performing slightly better on writing non-compressed data. However, while the subset to the Folsom Reservoir domain completed successfully with netCDF, the Zarr format would not complete consistently across all 100 sample runs despite multiple attempts. This stability issue reinforces the findings from the literature review that the Zarr format is still maturing.

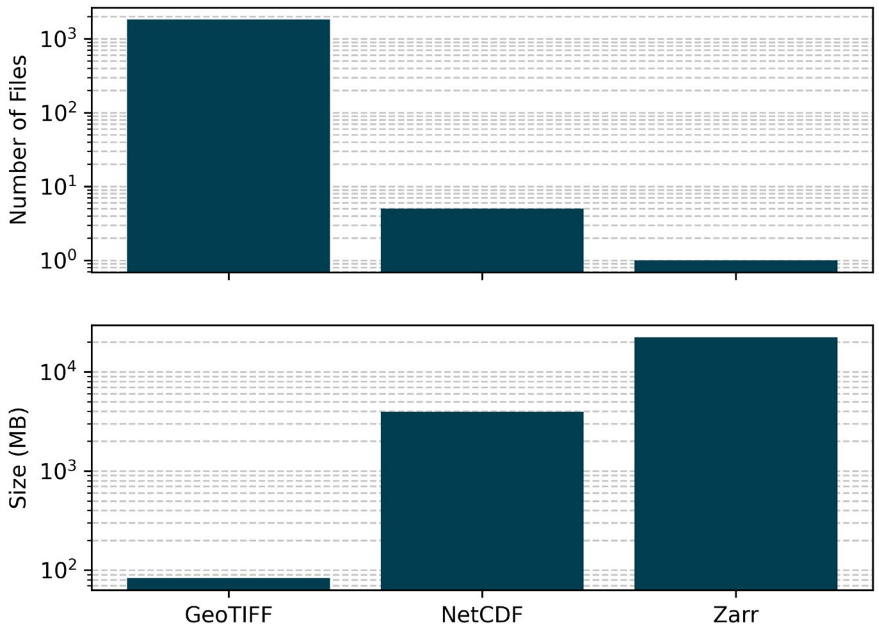


Figure 7: Number of files and mean file size required to represent the PRISM data with each format

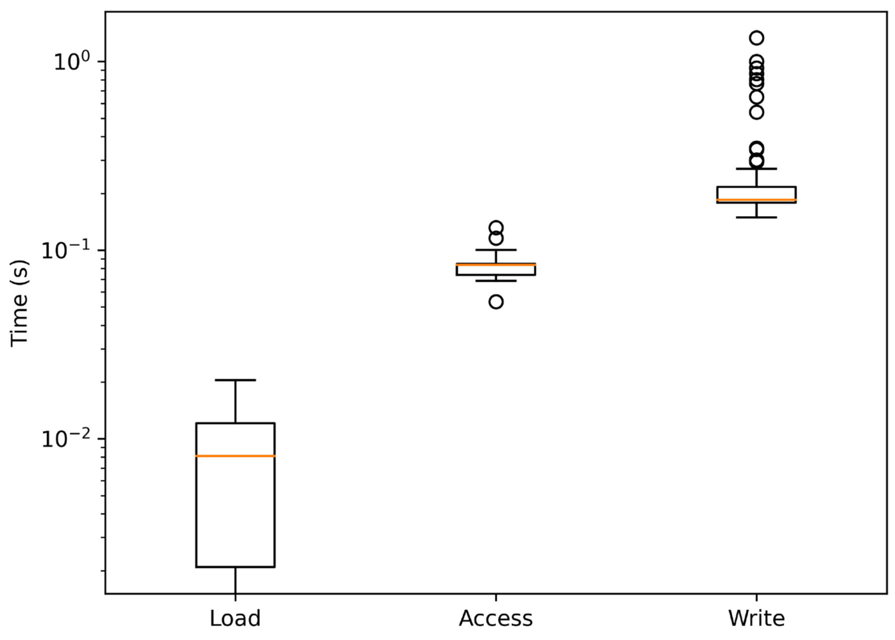


Figure 8: Time benchmarks for the PRISM dataset using the GeoTIFF format

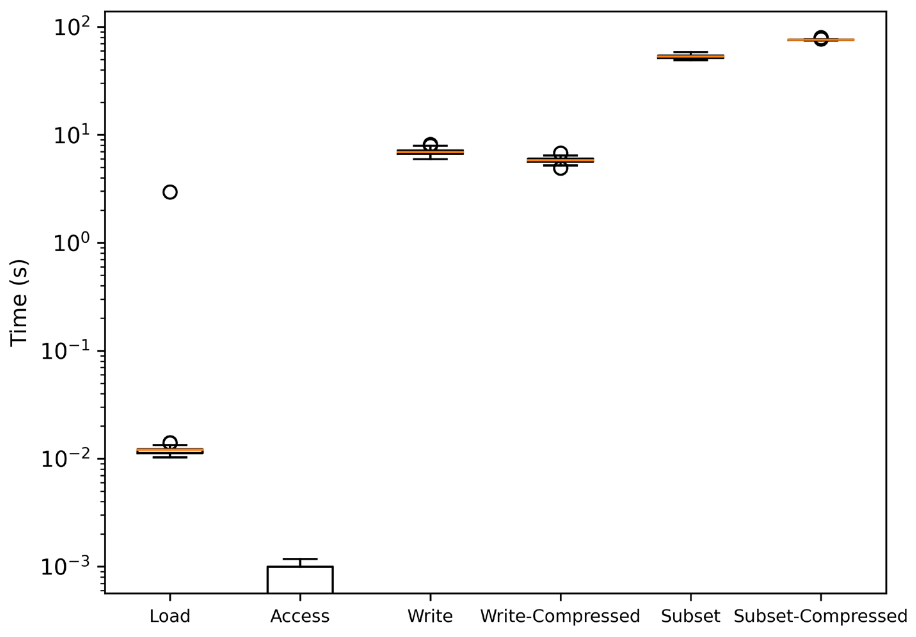


Figure 9: Time benchmarks for the PRISM dataset using the netCDF format

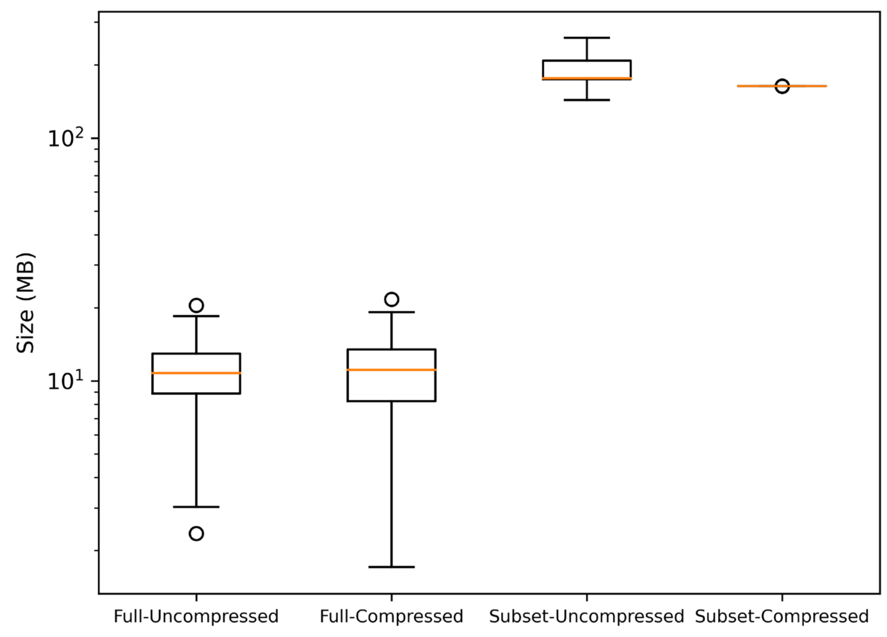


Figure 10: Storage benchmarks for the PRISM dataset using the netCDF format

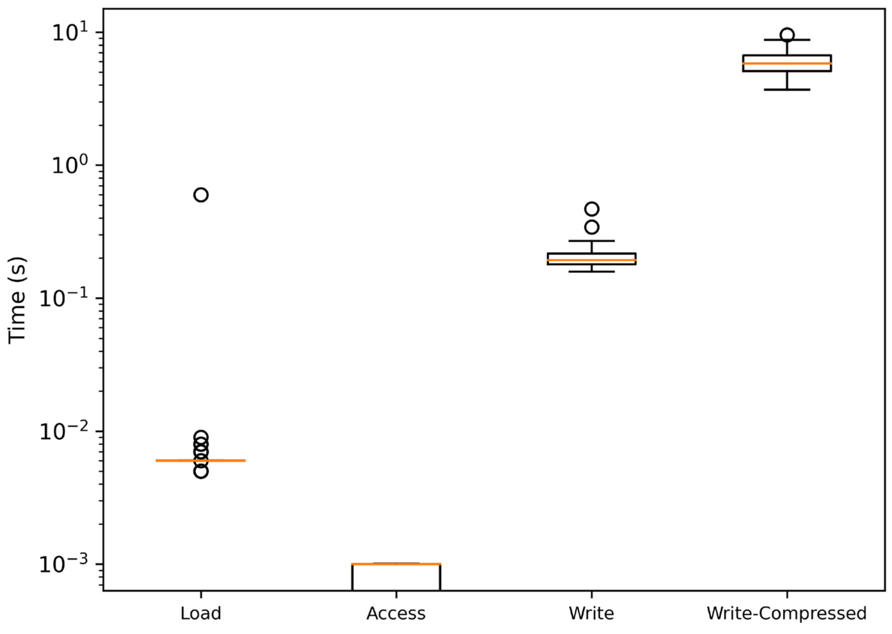


Figure 11: Time benchmarks for the PRISM dataset using the Zarr format

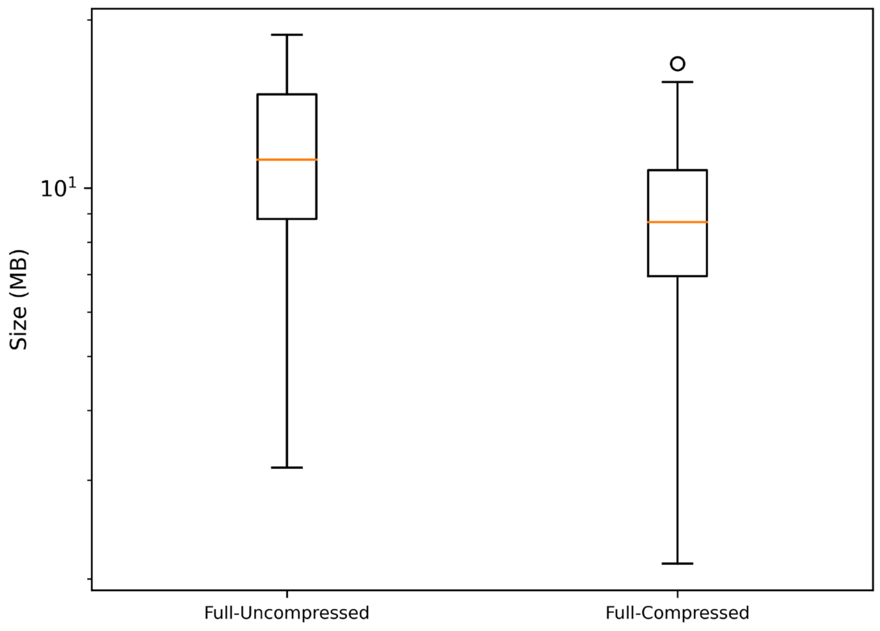


Figure 12: Storage benchmarks for the PRISM dataset using the Zarr format

2.1.3 Format Selection

The benchmarks, in combination with the literature review, indicate clear advantages for the netCDF format across the selected Reclamation use cases. While GeoTIFF could support the necessary operations, the number of files and increased storage requirements with default settings combined with a simple internal file structure would require significant additional effort and resources to support as compared to the other formats. The Zarr format, although structurally similar to netCDF, did not successfully complete the subset operations. While the format has potential benefits for cloud storage, Reclamation does not currently provide direct external access to RISE data negating any potential advantage of the Zarr format. Based on all available information, netCDF should be the preferred primary format for RISE with secondary preference for Zarr, particularly as the format continues to mature.

3.0 Data Workflows

Suggested data producer and data consumer workflows are outlined below. As detailed technical requirements are developed for RISE, the workflows may be adjusted. The workflows illustrate how RISE users – both data producers and consumers – will interact with large datasets after the proposed changes are implemented. Focus is given to how the data producers must prepare, transmit, and record the data for RISE to create records and items.

3.1 Data Producer Workflow

The data producer workflow focuses on moving data from where it is generated into RISE through an upload workflow. Management of large datasets is nontrivial, requiring specialized workflows to construct, transfer, and store the data. This challenge grows with the number of variables and the size of the dataset. Additionally, as the data moves from the producer into RISE through the upload workflow, it becomes more challenging for issues to be addressed and those resolving the issues have less familiarity with the data. For RISE to reasonably support large datasets, the data producer must play a central role in the upload workflow. It is a reasonable expectation for a data producer to be a somewhat more advanced user given the complexity of creating large datasets.

The data producer workflow for data upload is conceptualized as follows:

1) *The data is generated at the source by the data producer*
   The user will have some data production technique, be it experimental measurements or a numerical model. The data producer will need to collect and manage the source data in preparation for subsequent operations. Additionally, the data producer will need to collect the metadata that will also be required for subsequent operations.

2) *The data producer determines wat data will be published in RISE and performs dataset screening*
   RISE requires that all datasets go through a comprehensive screening process to ensure that no sensitive information is published, that data has undergone appropriate quality control processes and/or quality is clearly defined in the metadata, that data complies with applicable requirements related to accessibility and open formats, and that publication is coordinated with appropriate stakeholders, if needed. If any sensitive data is determined to be present in the data, the data producer may opt to redact, generalize, or remediate the sensitive in some other way prior to publication. The data producer is best suited to perform these tasks.

3) *The data producer converts the data from the source format to the netCDF format*
   Different data will require different methods to convert from the source format to the netCDF format. The data producer is best suited for this operation as they understand the data as the source data expert. Unless the source outputs into a RISE-compatible netCDF format, the user will need to write a converter from the source data format to the RISE netCDF format. This can be facilitated by example scripts posted on the RISE site but will ultimately be the responsibility of the data producer to package the source data into the netCDF.

   Having the data producer perform the conversion has tradeoffs but ultimately benefits the upload workflow. Data producers will require local resources to create a second copy of the datasets in the netCDF format, potentially introducing limitations in both storage and compute. However, this limitation can be mitigated on a case-by-case basis and is preferable to uploading source data files piecemeal to a centralized location and subsequently processing at the centralized location. Processing at a centralized location could create a significant constriction in transferring the data that may require multiple iterations and opens the potential for incomplete transfers of source files introducing artifacts in final dataset.

   Asking the data producer to create the netCDF performs the conversion at the same level as the data source expert. This individual best understands the metadata describing the source data, the format of the source data, and what data is required for RISE publication. Additionally, it allows the data source expert to examine the netCDF files to confirm that the conversion resulted in accurate output. While other individuals could verify that a netCDF file contains data in subsequent steps, only the data source expert can verify that the information is correct for their application. Following the premise that producers of large data are likely to be more advanced users and given support documentation/scripts available to document the process, the benefits to data accuracy and streamlining of the workflow outweigh the additional effort asked from the user.

4) *The netCDF structure is verified by the data producer*
   The subsequent data upload step can consume a significant amount of time, bandwidth, and storage. It is therefore beneficial to ensure that there are no readily addressable issues with the dataset prior to the upload to minimize the number of times the upload of a netCDF file is necessary.

   The RISE team will ask the data producer to run a verification script on the netCDF file produced after the conversion. This will check for both common issues within a netCDF data structure, for example the dimension of the data arrays match the dimension of their coordinate dimension arrays, as well as RISE specific information, such as metadata. Enforcing consistency on the netCDF data structure is intended to serve as a high-level quality check on the data source conversion to mitigate common errors. The RISE specific checks can be developed because RISE will read the netCDF data during record/item creation to maintain consistency between the data and the record. It is therefore possible to enforce the inclusion and desired format of metadata in the file in addition to any other attributes Reclamation requires to establish provenance. The RISE team will develop, maintain, and make available the verification script to ensure it conforms to changing requirements. Given its ability to perform out-of-memory calculations and flexibility across multiple computing architectures, the current preferred format for the verification script is Python.

If the verification script completes successfully, the data producer will know that the converted netCDF file is compliant with RISE requirements. If the verification script fails, it will flag the non-compliant attributes for the user who will then be able to return to the conversion step to make any changes. Because the verification will complete quickly, the data producer can iterate toward a RISE compliant netCDF file.

5) *The data producer uploads the data into RISE*
The data producer will then upload the data into the RISE staging area. The uploading interface may take several paths based on future development decisions and the type/size of data being uploaded; many of the potential methods are interchangeable from the viewpoint of the data producer. For example, two potential methods of uploading data include using the existing web interface or through a RISE specific network shared drive available to mount in Windows. It is likely that multiple methods may be required to facilitate netCDF data of different file sizes.

When the data are available in the RISE staging area, the verification script is again run to confirm that the data structure is valid and contains all the necessary attributes after upload. Additionally, if the file is not compressed, the data and coordinate arrays will be compressed if the verification is successful. This reduces RISE long term storage requirements at the expense of greater initial compute. Compression requirements can be minimized by building the compression operation into the example scripts provided to convert the source data to the netCDF format. Building compression into the data conversion step, to the extent possible, will help to minimize the upload bandwidth as well. When both operations are complete, the netCDF file will be tagged for review by RISE staff.

As an expansion to the current forms for adding data into RISE, the data producer will be able to indicate information about the netCDF file for data variables and coordinates. This will facilitate review by RISE staff who will, in general, not be familiar with the source data.

6) *RISE administrator enters the netCDF dataset into the database*
Given the amount of resources required by large netCDF datasets, review should be a requirement before a record is created. This both ensures that the data is necessary for public access as well as provides an additional level of review beyond the automated verification scripts. While the default Reclamation posture should be to data sharing, manual review of large datasets ensures that the data being shared is meaningful and necessary.

When manual review is complete, the data will be entered into RISE as an item within a record. Information to populate the database fields will be read directly from the netCDF file to ensure that the file and the record and item are consistent. The file may be entered with all data variables contained in the same item or separated out into different items. The former may require an additional structure definition within RISE.

When the data is entered as a record, the data producer will be notified that the dataset is available for review after other existing RISE workflow items are also addressed, such as data screening and release approval. The data producer should then review that record to ensure that all the information has been entered correctly. If the manual review raises issues with the data, the data producer will be notified about the issue(s). They will need to then return to source data conversion to correct the issue and reupload the file. In general, no modifications will be made to the netCDF file (besides compression) once the file is received by RISE to maintain data integrity.

7) *RISE administrator activates data*
   The final step in RISE publication is submission of the release approval documentation and data publication agreement. Once all approvals and agreements are in place for the items, the RISE administrator will activate the data to make it accessible to data consumers.

The proposed workflow balances actions between the data producer and RISE staff to maintain data integrity, minimize overall resource requirements, and limit responsibilities of RISE staff. Given the data complexity, the variety of different sources, and computational challenges of large datasets, having a workflow that has the data producer heavily involved in publishing the data is necessary to provide this balance. The intent is for RISE to provide documentation and support in templates and initial scripts, with the data producers adapting these products to their own cases. If the data producers share those conversion scripts back with the RISE community, these can over time serve as a starting point for other users with similar data sources.

3.2 Data Consumer Workflow

The data consumer download workflow focuses on taking data that is in RISE and serving it to a user. The challenges within the download workflow arise from asking the user what information from RISE they need, extracting that information from the RISE database, and transferring that information to the user in a computationally efficient manner. Determining the needed data from the user is relatively invariant with the dataset scale. However, accessing the information within RISE and transferring it to the user can become a significant challenge as the number of requests increase. The data consumer download workflow is intended to balance system responsiveness with the computational requirements needed to serve large datasets.

The data consumer workflow is conceptualized as follows:

1) *The data consumer selects the desired products using the graphical user interface (GUI)*
   The existing RISE interface allows a data consumer to interactively search for products using a combination of text-based data catalog and/or a map-based catalog. This would

continue as implemented with additional options available for items containing netCDF information.

When an item contains netCDF information, an additional option for this type of request will become available. The data consumer will be able to select from among three different options:

- All netCDF in Record – Downloads the all netCDF files for the record as they are available in RISE without any modification. If the record contains a single netCDF file with multiple variables, the complete file is downloaded.
- Item – Downloads the full netCDF file associated with the item as it is available on RISE without any modification. If items exist as separeate netCDF files rather than a single file for the record, the entire item netCDF file would be downloaded directly.
- Subset – Downloads a portion of the item data as it exists on RISE. This may be a combination of variables, spatial extents, and time durations.

If the 'Full' or 'Item' options are selected by the data consumer, they will be taken directly to the data transfer step (step three below). Because subsetting requires additional information to complete the request, the user will be taken first to the subsetting step (step two below), then to the data transfer step (step three).

2) *The data consumer refines the subset request*
To extract information from the netCDF files, additional information is required to determine the spatial and temporal extents as well as the variables. The data consumer would be taken to an intermediate screen that provides information about the record/item. Here, they would select the variables they wish to download as well as the spatial and temporal extents. The list of variables and extents would be an input into step three.

The process of subsetting can rapidly become complex. The intent within RISE should be to provide simple functionality that will cover the majority of data consumers. This will be selecting by variable and time within a rectangular spatial bounding box. If there is a need for more advanced workflows, such as clipping an item to a shapefile, that should be done by a user who downloads the full item and performs additional processing. This balances the competing objectives of providing a helpful user experience, minimizing computational requirements, and managing the maintainability of the RISE system.

The format of the netCDF file facilitates the subset request. Time and spatial coordinates are stored as coordinate vectors in the data structure. A subset can query those file attributes and display them for the data consumer. Specification of the subset could be implemented numerous ways. For variables, a single or multi-level variable selection list would be appropriate. For temporal extent, various preset date ranges along with a user-specified date field would likely work well. For spatial extents, a map selection tool would be preferrable.

3) *The data is transferred to the data consumer*
   The data will then be parsed, prepared for transfer, and then transferred to the consumer. For a full or item-based selection, RISE will open the item netCDF file, read the variable, and generate a temporary file. If the whole file is requested, this step is not required. For the subset request, the record/item file will be opened, the temporal/spatial filter applied, and a temporary file created. Out-of-memory tools, such as xarray in Python, can make the subsetting operation efficient.

   There may be a lag between the user request and when the file can be sent for download, depending on the subset request and the total number of requests being made. To provide a responsive environment, the data request should go into a queue system (delayed result backend) with the data consumer receiving an email with a link to download the data when the request is complete. The user can click the link in the email to download the products of the request. The data associated with the link will stay available for a limited period of time after the request is processed (~7 days), after which it will be deleted to recover storage space. However, if the user downloads the data, the request may be deleted before 7 days if the RISE has a high number of requests and needs to recover storage. An analogous system is the Green Data Oasis website that provides climate projections (Lawrence Livermore National Laboratory, 2022). The data consumer would receive the data in the netCDF format without conversion.

While the introduction of subsetting adds complexity into the RISE website, it can reduce the amount of data transferred significantly in some circumstances. It is important to note that subsetting is not necessary for RISE to recognize a significant portion of the benefit associated with the netCDF nor is it part of the minimum functionality required for the system to support the netCDF format. It is anticipated that the full and item downloading capabilities will be first deployed. Subsetting could be a subsequent feature were it to be developed at all.

In addition to the GUI interface, the application program interface (API) will require revision to support the netCDF format. This would mirror the changes to the GUI by including additional keywords in the API request. The keywords would need to indicate the type of the request against a record that contained a netCDF file. If it is a subset request, the API request will require keywords for the time and spatial extent. The API request will take an email as well such that the request can be entered into the delayed result backend. From this point, the data consumer workflow follows that of the GUI process. A request submitted via the API would have an email sent when the data was ready to be accesses, and the user could download that via the provided link.

4.0 RISE Modifications

To accommodate the netCDF format, enhancements will be required to the RISE platform across the front end and backends. The front end is the portion of the site that data producers and consumers interact with to submit and request data. The backend is the compute infrastructure which stores and serves data using input from the front end. As part of future efforts, more detailed requirements and feature analysis should be conducted. However, the following sections provide both an overview summary and a more detailed analysis of the anticipated modifications needed to RISE to aid in scoping future work.

4.1 Modification Overview

The following broad items are necessary to expand RISE to support the large datasets using the netCDF format:

1) *Accept uploads into RISE larger than 5GB through the web interface, preferably 100 GB for flexibility*
Justification: Many scientific and engineering models generate output much larger than the current upload limit. The standardization of broadband connections makes transfer of large files more feasible. Additionally, a larger upload limit will minimize the number of special requests supported by the RISE team to input data.

2) *Allow overall maximum file size in RISE object storage of 1 TB*
Justification: Numerous scientific, engineering, and data workflows create very large datasets, either as a direct output or from post processing. Output of these workflows is increasingly required to be shared with the public. A 1 TB size limit reasonably ensures that files of any size can be accepted. Above 1 TB, it is reasonable to assume that the data can be partitioned across multiple files, based on either time or variables, such that the resultant files are each below 1 TB.

3) *Develop the RISE netCDF specification*
Justification: The same information can be stored multiple ways while conforming to the base netCDF specification. RISE will need to interface with multiple various files containing disparate data in an automated fashion. Creation of a RISE netCDF standard allows RISE to better understand the contents of the stored netCDF files. Additionally, a RISE specific netCDF allows data producers a means to evaluate if their netCDF files are ready for RISE publication.

4) *Modify the RISE query logic to parse netCDF metadata*
Justification: The netCDF file format provides for storage of data, metadata, variable names, units, and other documentation needed to understand the file contents. Multiple variables may be included in a single file if the data are somehow related. It is best

practice to embed metadata information with data in a single netCDF file for traceability. RISE should be able to read the metadata information to enforce data quality and populate its own fields to preserve traceability from the netCDF file.

5) *Modify the RISE record/item structure to interact with netCDF file internal datasets*
   Justification: Large datasets often have structure to the data in time, space, or variable convention. It is often more efficient, in both storage and compute requirements, to use the structure in the data to aid in data operations and storage. Additionally, using delayed and out-of-memory operations enabled by netCDF libraries will further reduce storage and compute requirements.

6) *Modify the RISE user interface to allow subsetting of netCDF file datasets*
   Justification: Given the ability of a netCDF file to store multiple variables across space and time, users may want only a portion of the data. Providing a file with the minimal data saves RISE bandwidth costs by transferring only the requested portion of the file as well as reduces storage requirements and costs for the data consumer. RISE should accept user input to select a portion of the netCDF file, extract the data from the stored file into a subset netCDF file, and transfer the subset netCDF file to the user.

7) *Create a RISE Delayed Result backend*
   Justification: NetCDF operations are more computationally complex than existing operations, particularly for subset operations, such that they cannot be done in real time within the RISE interface. Handling multiple operations simultaneously can also be difficult with limited computational resources. To manage these requests, a new backend is required that manages requests through a queue in order of priority, notifies the user when the request is available, and removes files when the request is complete.

8) *Create preprocessing scripts to check data format prior to transfer to RISE*
   Justification: Metadata and data must be ensured as accurate through quality checks before creating a RISE record/item. Large datasets are time consuming to create and transfer, implying the error should be identified prior to data transmittal by the data producer. Having a quality control script that checks a netCDF file prior to transmittal for RISE compliance will reduce quality issues and allow data producers to correctly format their files.

4.2 Detailed Modifications

Although small netCDF files can be supported under the current configuration as a file upload, this approach lacks the integration of the internal data and metadata within RISE filter and query architecture. The intention of this section is to detail changes across RISE core functions needed to support netCDF files as a primary data type. This should serve as a preliminary assessment that gets expanded as part of future scoping and implementation activities.

4.2.1 RISE netCDF Backend

The RISE netCDF backend is a new component that is able to read, write, query, and filter netCDF files. As other RISE components need to interact with netCDF files, the RISE netCDF backend would provide the functionality needed to do so. This is a wholly new component that may be added to RISE.

4.2.2 RISE Delayed Result Backend

The RISE Delayed Result backend is a new component that manages jobs from the time the user submits them through the completion operations and the user is notified. Additionally, the Delayed Result backend will manage cleaning up old jobs and managing real time resources. Other RISE systems will utilize the Delayed Result backend when operations are too computationally heavy or the storage requirements too large to complete in real time. This is a wholly new component that will be added to RISE.

4.2.3 RISE Database

The RISE database stores metadata that describes records and items as well as the data that is associated with them. To support the addition of netCDF files, the following changes have been identified:

1) Add additional metadata fields needed to describe netCDF files
2) Create scripting to populate RISE database fields with metadata from the netCDF file
3) Increase the maximum attached file size to 1 TB
4) Connect with the RISE netCDF backend to handle file operations

4.2.4 RISE Data Administration User Interface (UI)

The RISE Data Administration UI allows for file upload and management. To support the addition of netCDF files, the following changes have been identified:

1) Accept uploads greater than 5 GB
2) Run the netCDF verification script when upload is complete
3) Connect with the RISE netCDF backend to handle file operations
4) Prompt RISE staff upon automated verification completion to manually approve the data

As this is the stage of the current workflow where data upload occurs, these additional modifications are not directly related to the administrative UI but to uploads more broadly:

1) Provide alternative instructions for uploading large datasets
2) Provide links to example netCDF conversion and verification scripts
3) Add connection for network mapped drives as an alternative upload source

4.2.5 RISE Catalog

The RISE catalog presents information to the data consumer as to what information is available. To support the addition of netCDF files, the following changes have been identified:

1) Add a new structure attribute that allows for netCDF files
2) Add interface to allow download of full items if stored as single netCDF files, using the RISE netCDF backend through the RISE Delayed Result backend
3) Include connection to the RISE netCDF query interface to allow subsetting
4) Connect with the RISE netCDF backend to handle file operations

4.2.6 RISE Time Series Query

The RISE time series query enables data consumers to filter, query, and download time series data. To support the addition of netCDF files, the following changes have been identified:

1) Incorporate information from the time attributes of netCDF files, through either a database query or direct read of the coordinate from the netCDF file
2) Include link to the RISE netCDF Subsetting Interface to allow spatial in addition to temporal subsetting
3) Disable plotting functionality for netCDF derived data
4) Connect with the RISE netCDF backend to handle file operations through the RISE Delayed Result backend

4.2.7 RISE NetCDF Subsetting Interface

The RISE netCDF subsetting interface is a new component that would support selection by the data consumer of items as well as temporal and spatial extents from a netCDF file. While this provides useful functionality to the user and minimizes compute/storage requirements, the subsetting feature is a lower priority than the catalog and timeseries query functionality as the users can obtain the full dataset from those other mechanisms. The subsetting interface should be capable of the following operations:

1) Receive as input either a catalog or item netCDF file
2) Interface with the RISE netCDF backend to obtain item, time, and spatial information
3) Present the item and time attributes as a list picker object
4) Present the spatial information as a map that allows the user to select a portion of the dataset
5) Use the RISE netCDF backed to handle file operations through the RISE Delayed Result backend

4.2.8 RISE API

The RISE API is an automated mechanism through which data can be added to and requested from RISE programmatically. To support the addition of netCDF files, the following changes have been identified:

1) Introduce new API keywords to handle the request of a netCDF, in addition to the item, temporal, and spatial subsetting
2) Use the RISE netCDF backed to handle file operations through the RISE Delayed Result backend
3) Adapt API to return an informative response to the user if the RISE Delayed Result backend is invoked

4.2.9 Administrative Features

Besides the implementation of new technical features, administrate features will be required to support the netCDF file type and large datasets generally. The following changes have been identified:

1) Development and documentation of example data conversion scripts for data producers
2) Development and documentation of a RISE netCDF file format specification, giving the required metadata and data structure
3) Development and documentation of verification scripts to confirm a netCDF file conforms to the required metadata and data structure
4) Documentation of the data producer standard operating procedure for generating a netCDF file for RISE
5) Documentation of the data producer standard operating procedure for uploading a large dataset to RISE

5.0 Conclusion

Reclamation evaluated multiple file formats commonly utilized for scientific and engineering datasets. The netCDF format supports a variety of data and is further enhanced by its integration with the HDF5 libraries. The format is platform independent and supported by open-source libraries for a variety of common programming languages. The file structure allows for metadata, data compression, subsetting, and appending in an efficient binary format. Closely related to netCDF, the open-source Zarr format maintains a similar feature set with additional optimizations for access and retrieval using cloud storage. While support of both formats is desirable to maximize RISE flexibility, it was determined that netCDF support should be prioritized as the format is more mature than Zarr. If Reclamation were to utilize cloud-based storage in the future, netCDF can be readily converted to the Zarr format were it required. It was therefore determined that netCDF should be the preferred storage format for handling large datasets in the RISE platform.

It is anticipated that the file formats and workflow recommended from this effort will be adopted through use of the RISE platform. The majority of RISE users request data from the platform. RISE will return large data to them in netCDF by default requiring no additional effort for users to adopt the format. Reclamation data producers may need to create additional operations to transition their information into the new RISE format. These can be created as part of future RISE modifications.

This work served as a preliminary analysis to identify a file format which balanced scientific, engineering, and information technology requirements. It is recommended that future work begin by expanding the high-level change assessment conducted within this scoping work.

6.0 References

Ambatipudi, S., & Byna, S. (2023). *A Comparison of HDF5, Zarr, and netCDF4 in Performing Common I/O Operations* (arXiv:2207.09503). arXiv. http://arxiv.org/abs/2207.09503

Bureau of Reclamation. (2023, July 21). *Reclamation Information Sharing Environment (RISE)*. https://data.usbr.gov/

Chris Durbin, Patrick Quinn, & Dana Shum. (2020). *Cloud Optimized Format Study* (Technical Paper EED2-TP-125). Raytheon. https://ntrs.nasa.gov/api/citations/20200001178/downloads/20200001178.pdf

Hoyer, S., & Joseph, H. (2017). xarray: N-D labeled Arrays and Datasets in Python. *Journal of Open Research Software*, *5*(1). https://doi.org/10.5334/jors.148

Jamison, T., & Sommerville, G. (2023, May 1). *Decrease geospatial query latency from minutes to seconds using Zarr on Amazon S3*. Amazon Web Services. https://aws.amazon.com/blogs/publicsector/decrease-geospatial-query-latency-minutes-seconds-using-zarr-amazon-s3/

Lawrence Livermore National Laboratory. (2022, September). *Downscaled CMIP3 and CMIP5 Climate and Hydrology Projections*. https://gdo-dcp.ucllnl.org/downscaled_cmip_projections/#Welcome

Lulla, V., Fishgold, L., & Tuhinanshu, T. (2022, September 22). *Benchmarking Zarr and Parquet Data Retreval using the National Water Model (NWM) in a Cloud-native environment*. Azavea. https://www.azavea.com/blog/2022/09/22/benchmarking-zarr-and-parquet-data-retrieval-using-the-national-water-model-nwm-in-a-cloud-native-environment/

Moore, J., Allan, C., Besson, S., Burel, J.-M., Diel, E., Gault, D., Kozlowski, K., Lindner, D., Linkert, M., Manz, T., Moore, W., Pape, C., Tischer, C., & Swedlow, J. R. (2021). OME-NGFF: a next-generation file format for expanding bioimaging data-access strategies. *Nature Methods*, *18*(12), 1496–1498. https://doi.org/10.1038/s41592-021-01326-w

National Aeronautics and Space Administration. (2021a, March 1). *ESDS Program: GeoTIFF*. EarthData. https://www.earthdata.nasa.gov/esdis/esco/standards-and-practices/geotiff

National Aeronautics and Space Administration. (2021b, May 20). *ESDS Program: NetCDF-4/HDF5 File Format*. EarthData. https://www.earthdata.nasa.gov/esdis/esco/standards-and-practices/netcdf-4hdf5-file-format

National Aeronautics and Space Administration. (2023a, March 14). *Zarr*. EarthData. https://wiki.earthdata.nasa.gov/display/ESO/Zarr+Format

National Aeronautics and Space Administration. (2023b, July 19). *ESDS Program: Standards and Practices*. EarthData. https://www.earthdata.nasa.gov/esdis/esco/standards-and-practices

National Center for Atmospheric Research. (2023). *Weather Research and Forecasting Model (WRF)*. Mesoscale & Microscale Meteorology Laboratory. https://www.mmm.ucar.edu/models/wrf

NumFOCUS. (2023). *Zarr*. Zarr. https://zarr.dev/

PRISM Climate Group, Oregon State University. (2020). http://www.prism.oregonstate.edu/

Foundations for Evidence-Based Policymaking Act of 2018, Pub. L. No. 115–435 (2019). https://www.congress.gov/bill/115th-congress/house-bill/4174

University Corporation for Atmospheric Research. (2023, May 22). *Network Common Data Form (NetCDF)*. Unidata - Data Services and Tools for Geoscience. https://www.unidata.ucar.edu/software/netcdf/

Zarr Storage Specification 2.0 Community Standard (21-050r1; Version 2). (2022).

7.0 Acknowledgments

The authors would like the thank the Reclamation Research and Development office for providing funding to this project through a Science and Technology Program scoping proposal.

Additionally, the authors would like to recognize the evaluation team from across Reclamation that provided feedback over the course of the project:

- Kimberly Atwood
- Gregory Gault
- Kathleen Holman
- Vanessa King
- Jason Kraft
- Michael Rosenberger
- Douglas Woolridge

These individuals created an interdisciplinary team spanning the GIS, RISE, engineering, and IT stakeholder groups. The broad perspective helped to ensure a robust finding that considered Bureau-wide needs.

Appendix A

Python Processing Scripts

A.1 CMIP5 Initial Reprocessing

Source data for the CMIP5 benchmarks are available on the Green Data Oasis website (Lawrence Livermore National Laboratory, 2022). Data can be regenerated for the benchmarks by downloading precipitation information from the site.

A.1.1 GeoTiff Reprocessing

```
import numpy as np
import rioxarray, os
from multiprocessing import Pool
from itertools import repeat


def write_raster(o_dataset, s_variable, s_output_path, i_timestep):
    """
    Converts input raster data into a GeoTiff format

    Parameters
    ----------
    o_dataset: object
        Contains the data to be written as GeoTiff
    s_variable: str
        Name of the variable within that file that is to be written
    s_output_path: str
        Stem of the output path for the file
    i_timestep: int
        Index in the file that should be output

    Returns
    -------
    None. File is written to disk.
    """

    o_dataset = o_dataset[s_variable][i_timestep, :, :]

    # Write to a geotiff
        o_dataset.rio.to_raster(os.path.join(s_output_path, 'gdo_' + o_dataset['time'].time.values.flatten()[0].strftime('%Y-%m-%d') +
                                        '.tiff'))


if __name__ == "__main__":

    ### Define the input information ###
    s_path = 'D:/scratch/rise/data/gdo/netcdf'
    ia_years = np.arange(1950, 1960, 1)

    s_variable = "pr"
    s_output_path = 'gdo'

    ### Get the input files ###
    sl_files = os.listdir(s_path)
    sl_files = [os.path.join(s_path, x) for x in sl_files]

    ### Make the output directory if it doesn't exist ###
    if not os.path.isdir(s_output_path):
        os.makedirs(s_output_path)


    ### Open the compute pool ###
    o_pool = Pool(processes=3)

    ### Use xarray to convert and preserve the metadata ###
    for i_entry_file in range(0, len(sl_files), 1):
        # Open the file and chunck along the time dimension
        o_dataset = rioxarray.open_rasterio(sl_files[i_entry_file], variable=s_variable, cache=False)

        # Parallel execution
        o_pool.starmap(write_raster, zip(repeat(o_dataset, o_dataset['time'].shape[0]),
                                         repeat(s_variable, o_dataset['time'].shape[0]),
                                         repeat(s_output_path, o_dataset['time'].shape[0]),
                                         range(0, o_dataset['time'].shape[0], 1)))

        # Close the dataset
        o_dataset.close()

    ### Close the compute pool ###
    o_pool.close()
```

A.1.2 NetCDF Reprocessing

NetCDF conversion scripts are not provided as the initial data source was given in NetCDF format. No reprocessing was done to transform the initial data.

A.1.3 Zarr Reprocessing

```
import netCDF4
import numpy as np
import zarr, os
import xarray as xr


if __name__ == "__main__":

    ### Define the input information ###
    s_path = 'D:/scratch/rise/data/gdo/netcdf'
    ia_years = np.arange(1950, 1960, 1)

    s_variable = "pr"
    sl_coordinates = ["lat", "lon", "time"]

    ### Get the input files ###
    sl_files = os.listdir(s_path)
    sl_files = [os.path.join(s_path, x) for x in sl_files]


    ### Use xarray to convert and preserve the metadata ###
    # Open the file and chunck along the time dimension
    o_dataset = xr.open_mfdataset(sl_files, chunks={'time': 1}, parallel=True)
```

```
# Extract the desired variable
o_dataset = o_dataset[s_variable]

# Convert the data array to a dataset to allow export
o_dataset = o_dataset.to_dataset()

# Rechunk for efficiency
o_dataset.chunk({'time': 1, 'lon': o_dataset['lon'].shape[0], 'lat': o_dataset['lat'].shape[0]})

# Write to zrr
o_dataset.to_zarr('gdo.zarr', mode='w')
```

A.2 PRISM Initial Reprocessing

Source data for the PRISM benchmarks are available on the Oregon State University website (*PRISM Climate Group, Oregon State University*, 2020). While the benchmarks used the paid 800 meter resolution product, the 4 kilometer product is available at no cost. Data can be regenerated for the benchmarks by downloading precipitation information from the site which will produce similar results.

A.2.1 GeoTiff Reprocessing

```
import numpy as np
import rioxarray, os
from multiprocessing import Pool
from itertools import repeat


def write_raster(o_dataset, s_variable, s_output_path, i_timestep):
    """
    Converts input raster data into a GeoTiff format

    Parameters
    ----------
    o_dataset: object
        Contains the data to be written as GeoTiff
    s_variable: str
        Name of the variable within that file that is to be written
    s_output_path: str
        Stem of the output path for the file
    i_timestep: int
        Index in the file that should be output

    Returns
    -------
    None. File is written to disk.
    """

    o_dataset = o_dataset[i_timestep, :, :]

    # Write to a geotiff
    o_dataset.rio.to_raster(os.path.join(s_output_path, 'prism_' + o_dataset['time'].time.values.flatten()[0].strftime('%Y-%m-%d') +
                                '.tiff'))


if __name__ == "__main__":

    ### Define the input information ###
    s_path = 'D:/scratch/rise/data/prism/netcdf'
    ia_years = np.arange(1981, 1986, 1)

    s_variable = "ppt"
    s_output_path = 'prism'

    ### Get the input files ###
    sl_files = os.listdir(s_path)
    sl_files = [os.path.join(s_path, x) for x in sl_files]

    ### Make the output directory if it doesn't exist ###
    if not os.path.isdir(s_output_path):
        os.makedirs(s_output_path)

    ### Open the compute pool ###
    o_pool = Pool(processes=7)

    ### Use xarray to convert and preserve the metadata ###
    for i_entry_file in range(0, len(sl_files), 1):
        # Open the file and chunk along the time dimension
        o_dataset = rioxarray.open_rasterio(sl_files[i_entry_file], variable=s_variable, cache=False)

        # Parallel execution
        o_pool.starmap(write_raster, zip(repeat(o_dataset, o_dataset['time'].shape[0]),
                                         repeat(s_variable, o_dataset['time'].shape[0]),
                                         repeat(s_output_path, o_dataset['time'].shape[0]),
                                         range(0, o_dataset['time'].shape[0], 1)))

        # Close the dataset
        o_dataset.close()

    ### Close the compute pool ###
    o_pool.close()
```

A.2.2 NetCDF Reprocessing

NetCDF conversion scripts are not provided as the initial data source was given in NetCDF format. No reprocessing was done to transform the initial data.

A.2.3 Zarr Reprocessing

```
import netCDF4
import numpy as np
import zarr, os
import xarray as xr


if __name__ == "__main__":

    ### Define the input information ###
    s_path = 'D:/scratch/rise/data/prism/netcdf'
    ia_years = np.arange(1981, 1986, 1)

    s_variable = "ppt"
    sl_coordinates = ["x", "y", "time"]

    ### Get the input files ###
    sl_files = os.listdir(s_path)
    sl_files = [os.path.join(s_path, x) for x in sl_files]
```

```
### Use xarray to convert and preserve the metadata ###
# Open the file and chunck along the time dimension
o_dataset = xr.open_mfdataset(sl_files, chunks={'time': 1}, parallel=True)

# Extract the desired variable
o_dataset = o_dataset[s_variable]

# Convert the data array to a dataset to allow export
o_dataset = o_dataset.to_dataset()

# Rechunk for efficiency
o_dataset.chunk({'time': 1, 'x': o_dataset['x'].shape[0], 'y': o_dataset['y'].shape[0]})

# Write to zrr
o_dataset.to_zarr('prism.zarr', mode='w')
```

A.3 CMIP5 GeoTiff Benchmark

```
import rioxarray
import random, os, time
import numpy as np
import matplotlib.pyplot as plt

if __name__ == "__main__":

    ### Get the names of the file in the directory ###
    # Set the path
    s_target_directory = "D:/scratch/rise/data/gdo/geotiff"

    # Get the list of files
    sl_files = os.listdir(s_target_directory)
    sl_files = [os.path.join(s_target_directory, x) for x in sl_files]


    ### Load each file into memory in succession at least 100 times ###
    da_load_times = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Start the timeer
        d_start_time = time.time()

        # Open the file
        o_file = rioxarray.open_rasterio(sl_files[i_target_index])

        # Close the file
        o_file.close()

        # Stop the timeer
        d_stop_time = time.time()

        # Log the amount of time required
        da_load_times[i_entry_sample] = d_stop_time - d_start_time


    ### Access time ###
    da_access_times = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Open the file
        o_file = rioxarray.open_rasterio(sl_files[i_target_index])

        # Start the timeer
        d_start_time = time.time()

        # Open the index
        dm_data = o_file.values
        # Stop the timeer
        d_stop_time = time.time()

        # Close the file
        o_file.close()

        # Log the amount of time required
        da_access_times[i_entry_sample] = d_stop_time - d_start_time


    ### Write time ###
    da_write_times_no_compress = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Open the file
        o_file = rioxarray.open_rasterio(sl_files[i_target_index])

        # Start the timeer
        d_start_time = time.time()

        # Create new file
        o_file.rio.to_raster(str(i_entry_sample) + '.tiff')

        # Stop the timeer
        d_stop_time = time.time()

        # Close the file
        o_file.close()

        # Log the amount of time required
        da_write_times_no_compress[i_entry_sample] = d_stop_time - d_start_time

    ### Plot the times ###
    plt.boxplot([da_load_times, da_access_times, da_write_times_no_compress])
    plt.xticks(np.arange(1, 4), ['Load', 'Access', 'Write'])
    plt.ylabel('Time (s)')

    ax = plt.gca()
    ax.set_yscale('log')

    plt.savefig('geotiff_gdo_times.png', dpi=600)
    plt.close()
```

A.4 CMIP5 NetCDF Benchmark

```python
import netCDF4
import random, os, time
import numpy as np
import matplotlib.pyplot as plt
import xarray as xr
import geopandas as gpd
from shapely.geometry import mapping


def subset(s_scratch_path, o_shapefile, s_type, s_local_name, s_output_name, i_compression):
    """
    Worker process to clip the downloaded files against the information in the shape file

    Parameters
    ----------
    o_shapefile: object
        Shapefile object
    s_type: str
        Variable name
    s_local_name: str
        Path to file to subset
    s_output_name: str
        Path to output file
    i_compression: int
        Compression level

    Returns
    -------
    None. Output is written to the disk.

    """

    # Process the output
    with xr.open_dataset(s_local_name, chunks='auto', engine='h5netcdf') as o_cmip_raster:
        o_cmip_raster = o_cmip_raster[s_type]
        o_cmip_raster = o_cmip_raster.assign_coords(lon=(((o_cmip_raster.lon + 180) % 360) - 180))

        o_cmip_raster.rio.set_crs("epsg:4326", inplace=True)
        o_cmip_raster.rio.write_crs("epsg:4326", inplace=True)

        o_cmip_raster = o_cmip_raster.sortby(o_cmip_raster.lon)
        o_cmip_raster.rio.set_spatial_dims('lon', 'lat')

        with o_cmip_raster.rio.clip(o_shapefile.geometry.apply(mapping), o_shapefile.crs, all_touched=True) as dm_cmip_raster_clipped:
            dm_cmip_raster_clipped.rio.write_crs("epsg:4326", inplace=True)

            # Write the data
            dm_cmip_raster_clipped.to_netcdf(os.path.join(s_scratch_path, s_output_name + "_clipped.nc"), engine='h5netcdf',
                                             encoding={s_type: {"zlib": True, "complevel": i_compression},
                                                       'lat': {"zlib": True, "complevel": i_compression},
                                                       'lon': {"zlib": True, "complevel": i_compression},
                                                       'time': {"zlib": True, "complevel": i_compression}})

        # Delete the source file
        o_cmip_raster.close()
        dm_cmip_raster_clipped.close()


if __name__ == "__main__":

    ### Get the names of the file in the directory ###
    # Set the path
    s_target_directory = "C:/Users/dloney/OneDrive - DOI/projects/rise/dataset_benchmark/data/gdo/netcdf"
    s_scratch_path = "D:/rise_scratch"

    # Get the list of files
    sl_files = os.listdir(s_target_directory)
    sl_files = [os.path.join(s_target_directory, x) for x in sl_files]

    ### Create the output directory ###
    if not os.path.exists(s_scratch_path):
        os.makedirs(s_scratch_path)

    ### Load each file into memory in succession at least 100 times ###
    da_load_times = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Start the timeer
        d_start_time = time.time()

        # Open the file
        #o_file = netCDF4.Dataset(sl_files[i_target_index])
        o_file = xr.open_dataset(sl_files[i_target_index], engine='h5netcdf')

        # Close the file
        o_file.close()

        # Stop the timeer
        d_stop_time = time.time()

        # Log the amount of time required
        da_load_times[i_entry_sample] = d_stop_time - d_start_time


    ### Access time ###
    da_access_times = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Open the file
        #o_file = netCDF4.Dataset(sl_files[i_target_index])
        o_file = xr.open_dataset(sl_files[i_target_index], engine='h5netcdf')

        # Start the timeer
        d_start_time = time.time()

        # Get the dimensions of the data
        o_times = o_file['time']
        # Get a random index to open
        i_target_index = random.randint(0, len(o_times) - 1)

        # Open the index
        dm_data = o_file['pr'][i_target_index, :, :]

        # Stop the timeer
        d_stop_time = time.time()

        # Close the file
        o_file.close()

        # Log the amount of time required
        da_access_times[i_entry_sample] = d_stop_time - d_start_time


    ### Write time ###
    da_write_times_no_compress = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Open the file
        o_file = xr.open_dataset(sl_files[i_target_index], engine='h5netcdf')

        # Get the dimensions of the data
        o_times = o_file['time']

        # Get a random index to open
        i_target_index = random.randint(0, len(o_times) - 1)

        # Open the index
        dm_data = o_file['pr'][i_target_index, :, :]

        # Start the timeer
        d_start_time = time.time()

        # Create new file
        dm_data.to_netcdf(os.path.join(s_scratch_path, str(i_entry_sample) + '.nc'), engine='h5netcdf')

        # Stop the timeer
        d_stop_time = time.time()

        # Close the file
        o_file.close()

        # Log the amount of time required
        da_write_times_no_compress[i_entry_sample] = d_stop_time - d_start_time

    # Get the files size
    sl_file_sizes_no_compress = [os.path.getsize(os.path.join(s_scratch_path, str(x) + '.nc')) / (1024.0 ** 2.0) for x in range(0, 100, 1)]
    for i_entry_sample in range(0, 100, 1):
        os.remove(os.path.join(s_scratch_path, str(i_entry_sample) + '.nc'))

    ### Write time ###
    da_write_times_compress = np.zeros(100)
```

```
for i_entry_sample in range(0, da_load_times.shape[0], 1):
    # Get a random index to open
    i_target_index = random.randint(0, len(sl_files) - 1)

    # Open the file
    #o_file = netCDF4.Dataset(sl_files[i_target_index])
    o_file = xr.open_dataset(sl_files[i_target_index], engine='h5netcdf')

    # Get the dimensions of the data
    o_times = o_file['time']

    # Get a random index to open
    i_target_index = random.randint(0, len(o_times) - 1)

    # Open the index
    dm_data = o_file['pr'][i_target_index, :, :]

    # Start the timeer
    d_start_time = time.time()

    # Create new file
    dm_data.to_netcdf(os.path.join(s_scratch_path, str(i_entry_sample) + '.nc'), engine='h5netcdf',
                      encoding={'pr': {"zlib": True, "complevel": 9},
                                'lat': {"zlib": True, "complevel": 9},
                                'lon': {"zlib": True, "complevel": 9},
                                'time': {"zlib": True, "complevel": 9}})

    # Log the amount of time required
    da_write_times_compress[i_entry_sample] = d_stop_time - d_start_time

sl_file_sizes_compress = [os.path.getsize(os.path.join(s_scratch_path, str(x) + '.nc')) / (1024.0 ** 2.0) for x in range(0, 100, 1)]
for i_entry_sample in range(0, 100, 1):
    os.remove(os.path.join(s_scratch_path, str(i_entry_sample) + '.nc'))

### Subset ###
# Open the shapefile #
s_shapefile_path = "C:/Users/dloney/OneDrive - DOI/projects/rise/dataset_benchmark/data/basin/WBDHU2.shp"
o_shapefile = gpd.read_file(s_shapefile_path)

# Process each of the files
da_subset_times_uncompressed = np.zeros(100)
for i_entry_sample in range(0, da_subset_times_uncompressed.shape[0], 1):

    # Get a random index to open
    i_target_index = random.randint(0, len(sl_files) - 1)

    # Start the timeer
    d_start_time = time.time()

    # Create the subset
    subset(s_scratch_path, o_shapefile, 'pr', sl_files[i_target_index], str(i_entry_sample), 1)

    # Stop the timer
    d_stop_time = time.time()

    # Store the time
    da_subset_times_uncompressed[i_entry_sample] = d_stop_time - d_start_time
```

```
sl_subset_sizes_uncompress = [os.path.getsize(os.path.join(s_scratch_path, str(x) + '_clipped.nc')) / (1024.0 ** 2.0)
                              for x in range(0, 100, 1)]
for i_entry_sample in range(0, 100, 1):
    os.remove(os.path.join(s_scratch_path, str(i_entry_sample) + '_clipped.nc'))

# Process each of the files
da_subset_times_compressed = np.zeros(100)
for i_entry_sample in range(0, da_subset_times_uncompressed.shape[0], 1):
    # Start the timeer
    d_start_time = time.time()

    # Create the subset
    subset(s_scratch_path, o_shapefile, 'pr', sl_files[i_target_index], str(i_entry_sample), 9)

    # Stop the timer
    d_stop_time = time.time()

    # Store the time
    da_subset_times_compressed[i_entry_sample] = d_stop_time - d_start_time

sl_subset_sizes_compress = [os.path.getsize(os.path.join(s_scratch_path, str(x) + '_clipped.nc')) / (1024.0 ** 2.0)
                            for x in range(0, 100, 1)]
for i_entry_sample in range(0, 100, 1):
    os.remove(os.path.join(s_scratch_path, str(i_entry_sample) + '_clipped.nc'))

### Plot the times ###
plt.boxplot([da_load_times, da_access_times, da_write_times_no_compress, da_write_times_compress,
             da_subset_times_uncompressed, da_subset_times_compressed])
plt.xticks(np.arange(1, 7), ['Load', 'Access', 'Write', 'Write-Compressed', 'Subset', 'Subset-Compressed'], fontsize=8)
plt.ylabel('Time (s)')

o_ax = plt.gca()
o_ax.set_yscale('log')

plt.savefig('netcdf_gdo_times.png', dpi=600)
plt.close()

### Plot the sizes ###
plt.boxplot([sl_file_sizes_no_compress, sl_file_sizes_compress, sl_subset_sizes_uncompress, sl_subset_sizes_compress])
plt.xticks(np.arange(1, 5), ['Full-Uncompressed', 'Full-Compressed', 'Subset-Uncompressed', 'Subset-Compressed'], fontsize=8)
plt.ylabel('Size (MB)')

o_ax = plt.gca()
o_ax.set_yscale('log')

plt.savefig('netcdf_gdo_storage.png', dpi=600)
plt.close()
```

A.5 CMIP5 Zarr Benchmark

```
import xarray as xr
import random, os, time, zarr, shutil
import numpy as np
import matplotlib.pyplot as plt
import geopandas as gpd
from shapely.geometry import mapping


def subset(s_scratch_path, o_shapefile, s_type, s_local_name, s_output_name, i_compression):
    """
    Worker process to clip the downloaded files against the information in the shape file

    Parameters
    ----------
    o_shapefile: object
        Shapefile object
    s_type: str
        Variable name
    s_local_name: str
        Path to file to subset
    s_output_name: str
        Path to output file
    i_compression: int
        Compression level

    Returns
    -------
    None. Output is written to the disk.

    """

    # Process the output
    with xr.open_dataset(s_local_name, engine='zarr', chunks='auto') as o_cmip_raster:
        o_cmip_raster = o_cmip_raster[s_type]
        o_cmip_raster = o_cmip_raster.assign_coords(lon=(((o_cmip_raster.lon + 180) % 360) - 180))

        o_cmip_raster.rio.set_crs("epsg:4326", inplace=True)
        o_cmip_raster.rio.write_crs("epsg:4326", inplace=True)

        o_cmip_raster = o_cmip_raster.sortby(o_cmip_raster.lon)
        o_cmip_raster.rio.set_spatial_dims('lon', 'lat')

        with o_cmip_raster.rio.clip(o_shapefile.geometry.apply(mapping), o_shapefile.crs, all_touched=True) as dm_cmip_raster_clipped:
            dm_cmip_raster_clipped.rio.write_crs("epsg:4326", inplace=True)
            dm_cmip_raster_clipped = dm_cmip_raster_clipped.to_dataset()

            # Write the data
            compressor = zarr.lzma.LZMA(preset=i_compression)
            dm_cmip_raster_clipped.to_zarr(os.path.join(s_scratch_path, s_output_name + "_clipped.zarr"),
                                           encoding={s_type: {'compressor': compressor},
                                                     'lat': {'compressor': compressor},
                                                     'lon': {'compressor': compressor},
                                                     'time': {'compressor': compressor}})

        # Delete the source file
        o_cmip_raster.close()
        dm_cmip_raster_clipped.close()


def get_size(s_start_path):
    """
    Recursively gets the size of all files in the path

    Parameters
    ----------
    s_start_path: str
        Path to walk recursively

    Returns
    -------
    d_total_size: float
        Size of the directory in bytes

    """

    d_total_size = 0
    for s_dir_path, l_dir_names, l_file_names in os.walk(s_start_path):
        # Loop on each file in the folder
        for s_file in l_file_names:
            # Construct the path to the file
            s_file_path = os.path.join(s_dir_path, s_file)

            # Skip if it is symbolic link
            if not os.path.islink(s_file_path):
                # Count the file size
                d_total_size += os.path.getsize(s_file_path)

    return d_total_size


if __name__ == "__main__":

    ### Get the names of the file in the directory ###
    # Set the path
    s_target_directory = "C:/Users/dloney/OneDrive - DOI/projects/rise/dataset_benchmark/data/gdo/zarr"
    s_scratch_path = "D:/rise_scratch"

    # Get the list of files
    sl_files = os.listdir(s_target_directory)
    sl_files = [os.path.join(s_target_directory, x) for x in sl_files]

    ### Create the output directory ###
    if not os.path.exists(s_scratch_path):
        os.makedirs(s_scratch_path)


    ### Load each file into memory in succession at least 100 times ###
    da_load_times = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)
        # Start the timeer
        d_start_time = time.time()

        # Open the file
        o_file = xr.open_zarr(sl_files[i_target_index])

        # Close the file
        o_file.close()

        # Stop the timeer
        d_stop_time = time.time()

        # Log the amount of time required
        da_load_times[i_entry_sample] = d_stop_time - d_start_time


    ### Access time ###
    da_access_times = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Open the file
        o_file = xr.open_zarr(sl_files[i_target_index])

        # Start the timeer
        d_start_time = time.time()

        # Get the dimensions of the data
        o_times = o_file['time']

        # Get a random index to open
        i_target_index = random.randint(0, len(o_times) - 1)

        # Open the index
        dm_data = o_file['pr'][i_target_index, :, :]

        # Stop the timeer
        d_stop_time = time.time()

        # Close the file
        o_file.close()

        # Log the amount of time required
        da_access_times[i_entry_sample] = d_stop_time - d_start_time


    ### Write time ###
    da_write_times_no_compress = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Open the file
        o_file = xr.open_zarr(sl_files[i_target_index])
        # Get the dimensions of the data
        o_times = o_file['time']

        # Get a random index to open
        i_target_index = random.randint(0, len(o_times) - 1)

        # Open the index
        dm_data = o_file['pr'][i_target_index, :, :]

        # Start the timeer
        d_start_time = time.time()

        # Create new file
        dm_data = dm_data.to_dataset()
        dm_data.to_zarr(os.path.join(s_scratch_path, str(i_entry_sample) + '.zarr'), mode='w')

        # Stop the timeer
        d_stop_time = time.time()

        # Close the file
        o_file.close()

        # Log the amount of time required
        da_write_times_no_compress[i_entry_sample] = d_stop_time - d_start_time

    # Get the files size
    sl_file_sizes_no_compress = [get_size(os.path.join(s_scratch_path, str(x) + '.zarr')) / (1024.0 ** 2.0) for x in range(0, 100, 1)]
    for i_entry_sample in range(0, 100, 1):
        shutil.rmtree(os.path.join(s_scratch_path, str(i_entry_sample) + '.zarr'))

    ## Write time ###
    da_write_times_compress = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Open the file
        o_file = xr.open_zarr(sl_files[i_target_index])

        # Get the dimensions of the data
        o_times = o_file['time']

        # Get a random index to open
        i_target_index = random.randint(0, len(o_times) - 1)

        # Open the index
        dm_data = o_file['pr'][i_target_index, :, :]

        # Start the timeer
        d_start_time = time.time()

        # Create new file
        dm_data = dm_data.to_dataset()

        compressor = zarr.lzma.LZMA(preset=0)
        dm_data.to_zarr(os.path.join(s_scratch_path, str(i_entry_sample) + '.zarr'),
                                                  encoding={'pr': {'compressor': compressor},
                                              'lat': {'compressor': compressor},
                                              'lon': {'compressor': compressor},
                                              'time': {'compressor': compressor}})

            # Stop the timer
            d_stop_time = time.time()

            # Close the file
            o_file.close()

            # Log the amount of time required
            da_write_times_compress[i_entry_sample] = d_stop_time - d_start_time

        sl_file_sizes_compress = [get_size(os.path.join(s_scratch_path, str(x) + '.zarr')) / (1024.0 ** 2.0) for x in range(0, 100, 1)]
        for i_entry_sample in range(0, 100, 1):
            shutil.rmtree(os.path.join(s_scratch_path, str(i_entry_sample) + '.zarr'))

        ### Subset ###
        # Open the shapefile #
        s_shapefile_path = "C:/Users/dloney/OneDrive - DOI/projects/rise/dataset_benchmark/data/basin/WBDHU2.shp"
        o_shapefile = gpd.read_file(s_shapefile_path)

        # Process each of the files
        da_subset_times_uncompressed = np.zeros(100)
        for i_entry_sample in range(0, da_subset_times_uncompressed.shape[0], 1):
            # Get a random index to open
            i_target_index = random.randint(0, len(sl_files) - 1)

            # Start the timeer
            d_start_time = time.time()

            # Create the subset
            subset(s_scratch_path, o_shapefile, 'pr', sl_files[i_target_index], str(i_entry_sample), 1)

            # Stop the timer
            d_stop_time = time.time()

            # Store the time
            da_subset_times_uncompressed[i_entry_sample] = d_stop_time - d_start_time

        sl_subset_sizes_uncompress = [get_size(os.path.join(s_scratch_path, str(x) + '_clipped.zarr')) / (1024.0 ** 2.0)
                                      for x in range(0, 100, 1)]
        for i_entry_sample in range(0, 100, 1):
            shutil.rmtree(os.path.join(s_scratch_path, str(i_entry_sample) + '_clipped.zarr'))

        # Process each of the files
        da_subset_times_compressed = np.zeros(100)
        for i_entry_sample in range(0, da_subset_times_uncompressed.shape[0], 1):
            # Start the timeer
            d_start_time = time.time()

            # Create the subset
            subset(s_scratch_path, o_shapefile, 'pr', sl_files[i_target_index], str(i_entry_sample), 9)

            # Stop the timer
            d_stop_time = time.time()

            # Store the time
        da_subset_times_compressed[i_entry_sample] = d_stop_time - d_start_time

    sl_subset_sizes_compress = [get_size(os.path.join(s_scratch_path, str(x) + '_clipped.zarr')) / (1024.0 ** 2.0) for x in range(0, 100, 1)]
    for i_entry_sample in range(0, 100, 1):
        shutil.rmtree(os.path.join(s_scratch_path, str(i_entry_sample) + '_clipped.zarr'))

    ### Plot the times ###
    plt.boxplot([da_load_times, da_access_times, da_write_times_no_compress, da_write_times_compress,
                 da_subset_times_uncompressed, da_subset_times_compressed])
    plt.xticks(np.arange(1, 7), ['Load', 'Access', 'Write', 'Write-Compressed', 'Subset', 'Subset-Compressed'],   fontsize=8)
    plt.ylabel('Time (s)')

    ax = plt.gca()
    ax.set_yscale('log')

    plt.savefig('zarr_gdo_times.png', dpi=600)
    plt.close()

    ### Plot the sizes ###
    plt.boxplot([sl_file_sizes_no_compress, sl_file_sizes_compress, sl_subset_sizes_uncompress, sl_subset_sizes_compress])
    plt.xticks(np.arange(1, 5), ['Full-Uncompressed', 'Full-Compressed', 'Subset-Uncompressed', 'Subset-Compressed'],  fontsize=8)
    plt.ylabel('Size (MB)')

    ax = plt.gca()
    ax.set_yscale('log')

    plt.savefig('zarr_gdo_storage.png', dpi=600)
    plt.close()
```

A.6 PRISM GeoTiff Benchmark

```
import rioxarray
import random, os, time
import numpy as np
import matplotlib.pyplot as plt

if __name__ == "__main__":

    ### Get the names of the file in the directory ###
    # Set the path
    s_target_directory = "D:/scratch/rise/data/prism/geotiff"

    # Get the list of files
    sl_files = os.listdir(s_target_directory)
    sl_files = [os.path.join(s_target_directory, x) for x in sl_files]


    ### Load each file into memory in succession at least 100 times ###
    da_load_times = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Start the timeer
        d_start_time = time.time()

        # Open the file
        o_file = rioxarray.open_rasterio(sl_files[i_target_index])

        # Close the file
        o_file.close()

        # Stop the timeer
        d_stop_time = time.time()

        # Log the amount of time required
        da_load_times[i_entry_sample] = d_stop_time - d_start_time


    ### Access time ###
    da_access_times = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Open the file
        o_file = rioxarray.open_rasterio(sl_files[i_target_index])

        # Start the timeer
        d_start_time = time.time()

        # Open the index
        dm_data = o_file.values
    # Stop the timeer
    d_stop_time = time.time()

    # Close the file
    o_file.close()

    # Log the amount of time required
    da_access_times[i_entry_sample] = d_stop_time - d_start_time


### Write time ###
da_write_times_no_compress = np.zeros(100)

for i_entry_sample in range(0, da_load_times.shape[0], 1):
    # Get a random index to open
    i_target_index = random.randint(0, len(sl_files) - 1)

    # Open the file
    o_file = rioxarray.open_rasterio(sl_files[i_target_index])

    # Start the timeer
    d_start_time = time.time()

    # Create new file
    o_file.rio.to_raster(str(i_entry_sample) + '.tiff')

    # Stop the timeer
    d_stop_time = time.time()

    # Close the file
    o_file.close()

    # Log the amount of time required
    da_write_times_no_compress[i_entry_sample] = d_stop_time - d_start_time

### Plot the times ###
plt.boxplot([da_load_times, da_access_times, da_write_times_no_compress])
plt.xticks(np.arange(1, 4), ['Load', 'Access', 'Write'])
plt.ylabel('Time (s)')

ax = plt.gca()
ax.set_yscale('log')

plt.savefig('geotiff_prism_times.png', dpi=600)
plt.close()
```

A.7 PRISM NetCDF Benchmark

```
import xarray as xr
import netCDF4
import random, os, time
import numpy as np
import matplotlib.pyplot as plt
import geopandas as gpd
from shapely.geometry import mapping


def subset(s_scratch_path, o_shapefile, s_type, s_local_name, s_output_name, i_compression):
    """
    Worker process to clip the downloaded files against the information in the shape file

    Parameters
    ----------
    o_shapefile: object
        Shapefile object
    s_type: str
        Variable name
    s_local_name: str
        Path to file to subset
    s_output_name: str
        Path to output file
    i_compression: int
        Compression level

    Returns
    -------
    None. Output is written to the disk.

    """

    # Process the output
    with xr.open_dataset(s_local_name, chunks='auto', engine='netcdf4') as o_cmip_raster:
        o_cmip_raster = o_cmip_raster[s_type]
        o_cmip_raster = o_cmip_raster.assign_coords(lon=(((o_cmip_raster.x + 180) % 360) - 180))

        o_cmip_raster.rio.set_crs("epsg:4326", inplace=True)
        o_cmip_raster.rio.write_crs("epsg:4326", inplace=True)

        o_cmip_raster = o_cmip_raster.sortby(o_cmip_raster.x)
        o_cmip_raster.rio.set_spatial_dims('x', 'y')

        with o_cmip_raster.rio.clip(o_shapefile.geometry.apply(mapping), o_shapefile.crs, all_touched=True) as dm_cmip_raster_clipped:
            dm_cmip_raster_clipped.rio.write_crs("epsg:4326", inplace=True)

            # Write the data
            dm_cmip_raster_clipped.to_netcdf(os.path.join(s_scratch_path, s_output_name + "_clipped.nc"), engine='h5netcdf',
                                             encoding={s_type: {"zlib": True, "complevel": i_compression},
                                                       'x': {"zlib": True, "complevel": i_compression},
                                                       'y': {"zlib": True, "complevel": i_compression},
                                                       'time': {"zlib": True, "complevel": i_compression}})

    # Delete the source file
    o_cmip_raster.close()
    dm_cmip_raster_clipped.close()


if __name__ == "__main__":

    ### Get the names of the file in the directory ###
    # Set the path
    s_target_directory = "C:/Users/dloney/OneDrive - DOI/projects/rise/dataset_benchmark/data/prism/netcdf"
    s_scratch_path = "D:/rise_scratch"

    # Get the list of files
    sl_files = os.listdir(s_target_directory)
    sl_files = [os.path.join(s_target_directory, x) for x in sl_files]

    ### Create the output directory ###
    if not os.path.exists(s_scratch_path):
        os.makedirs(s_scratch_path)

    ### Load each file into memory in succession at least 100 times ###
    da_load_times = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Start the timeer
        d_start_time = time.time()

        # Open the file
        o_file = xr.open_dataset(sl_files[i_target_index], engine='h5netcdf')

        # Close the file
        o_file.close()

        # Stop the timeer
        d_stop_time = time.time()

        # Log the amount of time required
        da_load_times[i_entry_sample] = d_stop_time - d_start_time


    ### Access time ###
    da_access_times = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Open the file
        o_file = xr.open_dataset(sl_files[i_target_index], engine='h5netcdf')

        # Start the timeer
        d_start_time = time.time()

        # Get the dimensions of the data
        o_times = o_file['time']

        # Get a random index to open
        i_target_index = random.randint(0, len(o_times) - 1)

        # Open the index
        dm_data = o_file['ppt'][i_target_index, :, :]

        # Stop the timeer
        d_stop_time = time.time()

        # Close the file
        o_file.close()

        # Log the amount of time required
        da_access_times[i_entry_sample] = d_stop_time - d_start_time


    ### Write time ###
    da_write_times_no_compress = np.zeros(100)

    for i_entry_sample in range(0, da_load_times.shape[0], 1):
        # Get a random index to open
        i_target_index = random.randint(0, len(sl_files) - 1)

        # Open the file
        o_file = xr.open_dataset(sl_files[i_target_index], engine='h5netcdf')

        # Get the dimensions of the data
        o_times = o_file['time']

        # Get a random index to open
        i_target_index_2 = random.randint(0, len(o_times) - 1)

        # Open the index
        dm_data = o_file['ppt'][i_target_index_2, :, :]

        # Start the timeer
        d_start_time = time.time()

        # Create new file
        dm_data.to_netcdf(os.path.join(s_scratch_path, str(i_entry_sample) + '.nc'), engine='h5netcdf')

        # Stop the timeer
        d_stop_time = time.time()

        # Close the file
        o_file.close()

        # Log the amount of time required
        da_write_times_no_compress[i_entry_sample] = d_stop_time - d_start_time

    # Get the files size
    sl_file_sizes_no_compress = [os.path.getsize(os.path.join(s_scratch_path, str(x) + '.nc')) / (1024.0 ** 2.0) for x in range(0, 100, 1)]
    for i_entry_sample in range(0, 100, 1):
        os.remove(os.path.join(s_scratch_path, str(i_entry_sample) + '.nc'))

    ### Write time ###
    da_write_times_compress = np.zeros(100)

    for i_entry_sample in range(0, da_write_times_compress.shape[0], 1):
```

```
    # Get a random index to open
    i_target_index = random.randint(0, len(sl_files) - 1)

    # Open the file
    o_file = xr.open_dataset(sl_files[i_target_index], engine='h5netcdf')

    # Get the dimensions of the data
    o_times = o_file['time']

    # Get a random index to open
    i_target_index_2 = random.randint(0, len(o_times) - 1)

    # Open the index
    dm_data = o_file['ppt'][i_target_index_2, :, :]

    # Start the timeer
    d_start_time = time.time()

    # Create new file
    dm_data.to_netcdf(os.path.join(s_scratch_path, str(i_entry_sample) + '.nc'), engine='h5netcdf',
                      encoding={'ppt': {"zlib": True, "complevel": 9},
                                'x': {"zlib": True, "complevel": 9},
                                'y': {"zlib": True, "complevel": 9},
                                'time': {"zlib": True, "complevel": 9}})

    # Stop the timeer
    d_stop_time = time.time()

    # Close the file
    o_file.close()

    # Log the amount of time required
    da_write_times_compress[i_entry_sample] = d_stop_time - d_start_time

sl_file_sizes_compress = [os.path.getsize(os.path.join(s_scratch_path, str(x) + '.nc')) / (1024.0 ** 2.0) for x in range(0, 100, 1)]
for i_entry_sample in range(0, 100, 1):
    os.remove(os.path.join(s_scratch_path, str(i_entry_sample) + '.nc'))

### Subset ###
# Open the shapefile #
s_shapefile_path = "C:/Users/dloney/OneDrive - DOI/projects/rise/dataset_benchmark/data/basin/WBDHU2.shp"
o_shapefile = gpd.read_file(s_shapefile_path)

# Process each of the files
da_subset_times_uncompressed = np.zeros(100)
print('Uncompressed subset')
for i_entry_sample in range(0, da_subset_times_uncompressed.shape[0], 1):

    # Get a random index to open
    i_target_index = random.randint(0, len(sl_files) - 1)

    # Start the timeer
    d_start_time = time.time()

    # Create the subset
    subset(s_scratch_path, o_shapefile, 'ppt', sl_files[i_target_index], str(i_entry_sample), 1)
    # Stop the timer
    d_stop_time = time.time()

    # Store the time
    da_subset_times_uncompressed[i_entry_sample] = d_stop_time - d_start_time

sl_subset_sizes_uncompress = [os.path.getsize(os.path.join(s_scratch_path, str(x) + '_clipped.nc')) / (1024.0 ** 2.0)
                              for x in range(0, 100, 1)]
for i_entry_sample in range(0, 100, 1):
    os.remove(os.path.join(s_scratch_path, str(i_entry_sample) + '_clipped.nc'))

# Process each of the files
da_subset_times_compressed = np.zeros(100)
for i_entry_sample in range(0, da_subset_times_uncompressed.shape[0], 1):
    # Start the timeer
    d_start_time = time.time()

    # Create the subset
    subset(s_scratch_path, o_shapefile, 'ppt', sl_files[i_target_index], str(i_entry_sample), 9)

    # Stop the timer
    d_stop_time = time.time()

    # Store the time
    da_subset_times_compressed[i_entry_sample] = d_stop_time - d_start_time

sl_subset_sizes_compress = [os.path.getsize(os.path.join(s_scratch_path, str(x) + '_clipped.nc')) / (1024.0 ** 2.0)
                            for x in range(0, 100, 1)]
for i_entry_sample in range(0, 100, 1):
    os.remove(os.path.join(s_scratch_path, str(i_entry_sample) + '_clipped.nc'))

### Plot the times ###
plt.boxplot([da_load_times, da_access_times, da_write_times_no_compress, da_write_times_compress,
             da_subset_times_uncompressed, da_subset_times_compressed])
plt.xticks(np.arange(1, 7), ['Load', 'Access', 'Write', 'Write-Compressed', 'Subset', 'Subset-Compressed'],  fontsize=8)
plt.ylabel('Time (s)')

o_ax = plt.gca()
o_ax.set_yscale('log')

plt.savefig('netcdf_prism_times.png', dpi=600)
plt.close()

### Plot the sizes ###
plt.boxplot([sl_file_sizes_no_compress, sl_file_sizes_compress, sl_subset_sizes_uncompress, sl_subset_sizes_compress])
plt.xticks(np.arange(1, 5), ['Full-Uncompressed', 'Full-Compressed', 'Subset-Uncompressed', 'Subset-Compressed'], fontsize=8)
plt.ylabel('Size (MB)')

o_ax = plt.gca()
o_ax.set_yscale('log')

plt.savefig('netcdf_prism_storage.png', dpi=600)
plt.close()
```

A.8 PRISM Zarr Benchmark

```
import xarray as xr
import random, os, time, shutil
import numpy as np
import matplotlib.pyplot as plt
import geopandas as gpd
from shapely.geometry import mapping
import zarr


def get_size(s_start_path):
    """
    Recursively gets the size of all files in the path

    Parameters
    ----------
    s_start_path: str
        Path to walk recursively

    Returns
    -------
    d_total_size: float
        Size of the directory in bytes

    """

    d_total_size = 0
    for s_dir_path, l_dir_names, l_file_names in os.walk(s_start_path):
        # Loop on each file in the folder
        for s_file in l_file_names:
            # Construct the path to the file
            s_file_path = os.path.join(s_dir_path, s_file)

            # Skip if it is symbolic link
            if not os.path.islink(s_file_path):
                # Count the file size
                d_total_size += os.path.getsize(s_file_path)

    return d_total_size


if __name__ == "__main__":

    ### Get the names of the file in the directory ###
    # Set the path
    s_target_directory = "C:/Users/dloney/OneDrive - DOI/projects/rise/dataset_benchmark/data/prism/zarr"
    s_scratch_path = "D:/rise_scratch"

    # Get the list of files
    sl_files = os.listdir(s_target_directory)
    sl_files = [os.path.join(s_target_directory, x) for x in sl_files]

    ### Create the output directory ###
    if not os.path.exists(s_scratch_path):
        os.makedirs(s_scratch_path)
```

```
### Load each file into memory in succession at least 100 times ###
da_load_times = np.zeros(100)

for i_entry_sample in range(0, da_load_times.shape[0], 1):
    # Get a random index to open
    i_target_index = random.randint(0, len(sl_files) - 1)

    # Start the timeer
    d_start_time = time.time()

    # Open the file
    o_file = xr.open_zarr(sl_files[i_target_index])

    # Close the file
    o_file.close()

    # Stop the timeer
    d_stop_time = time.time()

    # Log the amount of time required
    da_load_times[i_entry_sample] = d_stop_time - d_start_time

### Access time ###
da_access_times = np.zeros(100)

for i_entry_sample in range(0, da_load_times.shape[0], 1):
    # Get a random index to open
    i_target_index = random.randint(0, len(sl_files) - 1)

    # Open the file
    o_file = xr.open_zarr(sl_files[i_target_index])

    # Start the timeer
    d_start_time = time.time()

    # Get the dimensions of the data
    o_times = o_file['time']

    # Get a random index to open
    i_target_index = random.randint(0, len(o_times) - 1)

    # Open the index
    dm_data = o_file['ppt'][i_target_index, :, :]

    # Stop the timeer
    d_stop_time = time.time()

    # Close the file
    o_file.close()

    # Log the amount of time required
    da_access_times[i_entry_sample] = d_stop_time - d_start_time

    ### Write time ###
    da_write_times_no_compress = np.zeros(100)

for i_entry_sample in range(0, da_load_times.shape[0], 1):
    # Get a random index to open
    i_target_index = random.randint(0, len(sl_files) - 1)

    # Open the file
    o_file = xr.open_zarr(sl_files[i_target_index])

    # Get the dimensions of the data
    o_times = o_file['time']

    # Get a random index to open
    i_target_index = random.randint(0, len(o_times) - 1)

    # Open the index
    dm_data = o_file['ppt'][i_target_index, :, :]

    # Start the timeer
    d_start_time = time.time()

    # Create new file
    dm_data = dm_data.to_dataset().compute()
    dm_data.to_zarr(os.path.join(s_scratch_path, str(i_entry_sample) + '.zarr'), mode='w')

    # Stop the timeer
    d_stop_time = time.time()

    # Close the file
    o_file.close()

    # Log the amount of time required
    da_write_times_no_compress[i_entry_sample] = d_stop_time - d_start_time

# Get the files size
sl_file_sizes_no_compress = [get_size(os.path.join(s_scratch_path, str(x) + '.zarr')) / (1024.0 ** 2.0) for x in range(0, 100, 1)]
for i_entry_sample in range(0, 100, 1):
    shutil.rmtree(os.path.join(s_scratch_path, str(i_entry_sample) + '.zarr'))

## Write time ###
da_write_times_compress = np.zeros(100)

for i_entry_sample in range(0, da_load_times.shape[0], 1):
    # Get a random index to open
    i_target_index = random.randint(0, len(sl_files) - 1)

    # Open the file
    o_file = xr.open_zarr(sl_files[i_target_index])

    # Get the dimensions of the data
    o_times = o_file['time']

    # Get a random index to open
    i_target_index = random.randint(0, len(o_times) - 1)

    # Open the index
    dm_data = o_file['ppt'][i_target_index, :, :]

    # Start the timeer
    d_start_time = time.time()
```

```
        # Create new file
        dm_data = dm_data.to_dataset().compute()

        compressor = zarr.lzma.LZMA(preset=9)
        dm_data.to_zarr(os.path.join(s_scratch_path, str(i_entry_sample) + '.zarr'),
                        encoding={'ppt': {'compressor': compressor},
                                  'x': {'compressor': compressor},
                                  'y': {'compressor': compressor},
                                  'time': {'compressor': compressor}})

        # Stop the timeer
        d_stop_time = time.time()

        # Close the file
        o_file.close()

        # Log the amount of time required
        da_write_times_compress[i_entry_sample] = d_stop_time - d_start_time

    sl_file_sizes_compress = [get_size(os.path.join(s_scratch_path, str(x) + '.zarr')) / (1024.0 ** 2.0) for x in range(0, 100, 1)]
    for i_entry_sample in range(0, 100, 1):
        shutil.rmtree(os.path.join(s_scratch_path, str(i_entry_sample) + '.zarr'))

    ### Subset ###
    # Open the shapefile #
    s_shapefile_path = "C:/Users/dloney/OneDrive - DOI/projects/rise/dataset_benchmark/data/basin/WBDHU2.shp"
    o_shapefile = gpd.read_file(s_shapefile_path)

    ### Plot the times ###
    plt.boxplot([da_load_times, da_access_times, da_write_times_no_compress, da_write_times_compress])
    plt.xticks(np.arange(1, 5), ['Load', 'Access', 'Write', 'Write-Compressed'], fontsize=8)
    plt.ylabel('Time (s)')

    ax = plt.gca()
    ax.set_yscale('log')

    plt.savefig('zarr_prism_times.png', dpi=600)
    plt.close()

    ### Plot the sizes ###
    plt.boxplot([sl_file_sizes_no_compress, sl_file_sizes_compress])
    plt.xticks(np.arange(1, 3), ['Full-Uncompressed', 'Full-Compressed'], fontsize=8)
    plt.ylabel('Size (MB)')

    ax = plt.gca()
    ax.set_yscale('log')

    plt.savefig('zarr_prism_storage.png', dpi=600)
    plt.close()
```