Open Access | https://doi.org/10.25080/HWGA5253

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23nd Python in Science Conference
ISSN: 2575-9752

Making Research Data Flow With Python

Josh Borrow[1], **Paul La Plante**[2,3], **James Aguirre**[1], and **Peter K. G. Williams**[4]

[1]Department of Physics and Astronomy, University of Pennsylvania, 209 South 33rd Street, Philadelphia, PA, USA 19104, [2]Department of Computer Science, University of Nevada, Las Vegas, NV 89154, [3]Nevada Center for Astrophysics, University of Nevada, Las Vegas, NV 89154, [4]Center for Astrophysics | Harvard & Smithsonian, 60 Garden St., Cambridge, MA 02138

Abstract

The increasing volume of research data in fields such as astronomy, biology, and engineering necessitates efficient distributed data management. Traditional commercial solutions are often unsuitable for the decentralized infrastructure typical of academic projects. This paper presents the Librarian, a custom framework designed for data transfer in large academic collaborations, designed for the Simons Observatory (SO) as a ground up re-architechture of a previous astronomical data management tool called the 'HERA Librarian' from which it takes its name. SO is a new-generation observatory designed for observing the Cosmic Microwave Background, and is located in the Atacama desert in Chile at over 5000 meters of elevation.

Existing tools like Globus Flows, iRODS, Rucio, and Datalad were evaluated but were found to be lacking in automation or simplicity. Librarian addresses these gaps by integrating with Globus for efficient data transfer and providing a RESTful API for easy interaction. It also supports transfers through the movement of physical media for environments with intermittent connectivity.

Using technologies like Python, FastAPI, and SQLAlchemy, the Librarian ensures robust, scalable, and user-friendly data management tailored to the needs of large-scale scientific projects. This solution demonstrates an effective method for managing the substantial data flows in modern 'big science' endeavors.

Keywords data, python, research

Published Jul 10, 2024

Correspondence to
Josh Borrow
josh@joshborrow.com

1. Introduction

Research data is ever-growing, with even small projects now producing terabytes of data. This has been matched by an increase in the typical size of workstation storage and compute, but as many fields like astronomy, biology, and engineering continue their march towards 'big science', distributed data analysis is becoming significantly more common. As such, there is a significant need for software that can seamlessly track and move data between sites without continuous human intervention. For the very largest of these projects (such as the Large Hadron Collider at CERN), massive development effort has been invested to create technologies for data pipelining, but these tools typically assume a high level of control over the software and hardware that is running for the project.

At the same time, data science and data analytics are becoming a larger part of industrial infrastructure, even in what appear to be non-technical fields like law, commerce, and media. This has led to a huge increase in data engineering and pipelining software, but this software is generally only available through private clouds like those provided by Amazon (Amazon Web Services), Microsoft (Azure), and Alphabet (Google Cloud). These services generally assume that there is an ingest point into one of their managed storage (e.g.

Amazon S3), using their compute and tools for analysis (e.g. Amazon Athena and Amazon EC2), and their tools for final data egress (e.g. Amazon API Gateway or Amazon QuickSight). These resources are mature, performant, and a result of hundreds of thousands or millions of developer-hours. However, they are not compatible with the current publicly-funded science model, or the way that large, distributed, academic collaborations work.

A monolithic stack like those provided by public clouds is almost antithetical to large, distributed, academic projects. In these projects, funding may come from a vast array of sources (e.g. faculty members funded through their institution, large funding agencies like the National Science Foundation, or more junior members funded through individual fellowships), and compute may be provided by many different providers at different scales (e.g. on-premises with the data source, small institutional Tier-2 or Tier-3 centers, or international-scale Tier-0 centers through an allocation). Pooling these funds and resources is an almost insurmountable challenge; aside from some $10 B+ projects, even 'massive' multi-hundred million dollar projects must design their staffing, software, and procedures for a level of flexibility (and, by corollary, simplicity) that would be incomprehensible in an industrial context.

This paper is about solving the base-level problem: how to get raw experiment data totaling around 1 PB a year from a remote site (in our case, the top of a mountain in Chile), to various science consumers distributed across the globe. Our data is comprised of folders (termed 'books'), each containing 10-20 GB of data, laid out in a pre-determined scheme in a POSIX filesystem.

2. Comparing Data Flows

In Figure 1, we show two example data flows: one for a commercial enterprise (on the top), and one for a typical academic project (bottom; this matches closely with the needs of the Simons Observatory).

A typical challenge for the commercial scenario is collating data from many different sources, and processing it in a latency-sensitive manner. Consider an e-commerce platform, which has sales data, market research, and user tracking data, which all need to be collected and normalised to make decisions. This data, though varied, is typically small (and all 'owned' in some sense by the company), meaning that it can all be placed inside one centralised data warehouse from which analysis can be performed, and data egress can take place (often through a shared service). Because these services must all be directly acquired by the business, they are all at a bare minimum highly configurable and scalable. It can hence make sense to either use a series of interconnecting services (e.g. those provided through the Apache Software Foundation if using on-premises hardware), or a managed service collection (e.g. using tools from AWS). The data flows, represented by arrows in this case, are then made much simpler, and specific business logic can be programmed around the assumption of a shared and well-known interface; this hence leads to our understanding of these interconnected services through the data producers, flows, and products.

For an academic project, there is usually one major data source, and analysis is usually compute-intensive. This could be an experiment or, in our case, an observatory. Connected to this experiment there is typically a small on-site data storage and analysis facility that repackages data and can perform simple analysis. Those running the experiment usually have complete control over only this platform; the rest are highly locked-down shared computing services like national facilities (e.g. NERSC) and university-managed clusters - funding agencies typically do not support the purchase of significant compute resources, as they maintain shared high performance computing clusters. An important corollary to this is that different facilities may have different resources available, for instance cold

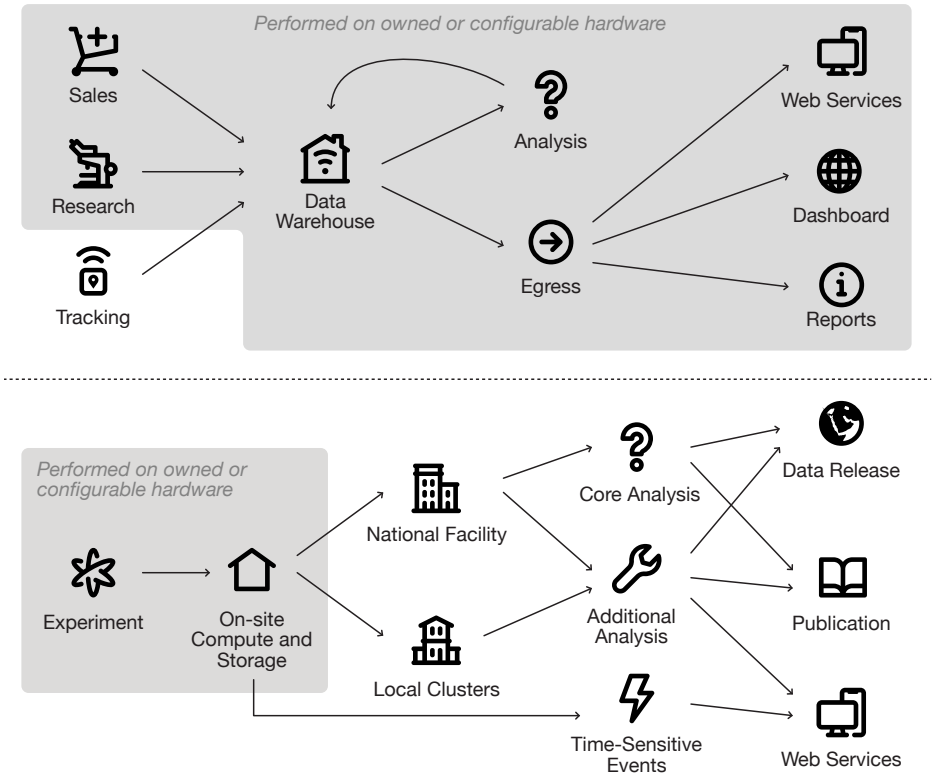

Figure 1. *Showing a typical layout of a commercial data engineering structure (top) versus an academic one (bottom). In a commercial context, it is typical that a significantly higher fraction of the resources are owned (or are directly configurable, e.g. cloud services) by the user. It is also much more common to have a centralised 'data warehouse' or 'data lake' which provides ingress and egress for all data products. In the academic context, only a very small fraction of the compute and storage are typically owned by stakeholders. National facilities, and even local clusters at universities, are typically rigid and cannot be re-configured to better suit the needs of an individual experiment. Further, due to limited compute and storage constraints, it is likely that multiple copies of experimental data must be kept at various sites, and tight control needs to be maintained on the specific sub-sets of data stored at each.*

storage for archival data, accelerators, or high-memory nodes. Data must be managed with the understanding that these resources may have finite lifetimes, may be withdrawn, or replaced at a completely different site. Further, as there is no centralised provider, data replication must be managed 'manually' by the collaboration through their network of shared machines, both for integrity and for availability at the sites that have different compute resources. Working on the data, specific archival products are produced: either those included in a data release (e.g. images, maps, etc.), and publications. Both regulation and community standards typically leads to these products being *immutable* and available *forever*, hosted by an external service. This reliance on external compute and storage leads to our understanding of this workflow through the flow of data into and out of the aforementioned providers.

Though they may have varied locations and resources, the computing centers are all usually organised in a broadly similar way, with dedicated data ingest (or transfer) nodes with a fast 'grid' internet connection (e.g. Internet2 in the U.S., JISC in the U.K., etc.) and access to shared disks. Compute is managed via a job scheduler (e.g. slurm), with file storage provided on GPFS or Lustre (i.e. POSIX-like, not object store) systems. As such, much of the community software has been built with the strong assumption that data is stored in files on a high-performance POSIX-compliant filesystem, with analysis performed through job submission

to compute nodes. Any data management system for such projects must hence be designed to be compatible with this workflow.

In the specific case of the Simons Observatory [1], we have a need for a piece of software to bring down data from our observatory at the rate of around 20 TB a week. This is accomplished using SneakerNet, where physical media is used to transport data, and a fiber internet connection. Our data is immutable, and in the form of 'books' (structured directories of binary data), within a pre-determined directory structure that must be replicated exactly on all of the downstream sites. At certain downstream sites, we have space for all of the data from the entire project, but at others only a recent, rolling, sub-set may be maintained.

2.1. Existing Software

Because of the significantly different constraints in the academic world versus the commercial world, there is a relatively limited set of tools that can be used to accomplish the goal of automated data management and transfer. In our search, we evaluated the following tools:

2.1.1. *Globus:*

Globus [2], [3] is a non-profit service from the University of Chicago, and specialises in the batch transfer of large data products. Globus works by having an 'endpoint server' running at each location, which can communicate over HTTP. Through the use of their web interface, API, or command-line client, it is possible to request asynchronous transfers of massive amounts of data, which are transferred at close to line speeds thanks to GridFTP being used for the underlying data movement. Globus is used extensively in the academic community, and is already available ubiquitously throughout the academic cluster ecosystem.

Globus, however, is not an automated tool; in general, one must tell Globus how, where, and when to move data. It is possible to set up recurring synchronisations of data, but this either requires complete copy at all sites (lest data that was just deleted be moved again) or that all sub-tasks be expressible as part of a 'Globus Flow', which may not always be possible. As such, Globus is more of a 'data transfer' tool, than a data management tool. It does not have significant cataloguing capability.

2.1.2. *Rucio and iRODS:*

Rucio [4] is a distributed data management system that was originally designed for the ATLAS high-energy physics experiment at the LHC, and as such is extremely scalable (i.e. exabytes or more). Rucio can be backed by different levels of storage, from fast (SSD) to slow (tape), and is declarative (meaning that one simply asks the system to follow a set of rules, like 'keep three copies of the data'), with its own user and permissions management system. Rucio is an exceptionally capable piece of software, but this comes with significant complexity. Further, data is managed externally to the underlying filesystem and permissions model of the host, potentially causing issues with the user agreements at shared facilities, and interacting with data generally requires interaction with the Rucio API. Though Rucio is certainly a fantastic tool, we found that it was too integrated for our needs of simply transferring data, and the declarative system is incompatible with our need to leverage SneakerNet.

A similar, albeit older and more mature, project to Rucio is iRODS (the integrated rule-oriented data system; R. Consortium [5]). iRODS, being more mature, integrates nicely with lots of software (providing a FUSE mount, FTP client integration, and more), and can likely be a perfect solution for someone looking for fully integrated data management. However, as it is built on a foundation of rule-based data management, it was discovered to be incompatible with our need to tightly control the specific transfers taking place.

2.1.3. *Datalad and git-annex:*

The git-annex [6] tool is an extension to Git, the version control system, to handle large quantities of binary data. Due to its extensive use by home and small business users, it explicitly supports SneakerNet transfers between sites, but is hampered by the fact that it relies on Git to track files (though not explicitly their contents; git-annex tracks checksums instead), which can become slow as repository size scales.

Datalad [7] is effectively a frontend for git-annex with a significant feature pool, and is used extensively in the bioscience community. For projects with small-to-medium size data needs, Datalad would be an exceptional candidate, but as we expect to need to manage multiple petabytes of data over a variety of storage needs, it was unfortunately not appropriate for the Simons Observatory.

3. The Librarian

After much consideration, the collaboration decided that building an orchestration and tracking framework on top of Globus was most appropriate. In addition, there was an existing piece of software used for the Hydrogen Epoch of Reionization Array (HERA) [8], a radio telescope designed to study the early Universe, called the 'HERA Librarian' [9], [10] that provided some of the abstractions that were required, though it was not suitable for production use for the much larger Simons Observatory. The HERA Librarian used a significant number of shell commands over SSH in its underlying architecture, and was designed to be fully synchronous. This lead to a complete re-write of the Librarian, primarily to eliminate these deficiencies, but also to migrate the application to a more modern web framework. There are a few key design concepts that were carried over from the HERA Librarian:

- There is a distinction drawn between 'Files' and 'Instances' of files in

the Librarian; a given server may know that 'File' exists but not actually have direct access to a copy of the underlying data.

- Files are immutable, enforced through checksumming, making synchronization

between Librarians uni-directional and much simpler.

- Instances of the Librarian server are free-standing and are loosely-coupled,

meaning that long outages or network downtime do not cause significant issues.

The Librarian[1] is made up of five major pieces:

1. A FastAPI[2] server that allows access through a REST HTTP API.
2. A database server (postgres[3] in production, SQLite[4] for

development) to track state and provide atomicity.

1. A background task[5] thread that performs data-intensive operations

like checksumming, local cloning, and scheduling of transfers.

1. A background transfer thread that manages the transfer queue and communicates

with Globus to query status.

1. A Python and associated command-line client for interacting with the API.

[1] https://github.com/simonsobs/librarian
[2] https://fastapi.tiangolo.com
[3] https://www.postgresql.org
[4] https://www.sqlite.org
[5] https://schedule.readthedocs.io/en/stable/index.html

Crucially, the background threads can directly access the database, without having to interact with the HTTP server. This reduces the level of 'CRUD' (create, read, update, delete) API code that is required for the project significantly, by allowing (e.g.) locking of database rows currently being transferred directly, rather than having this be handled through a series of API calls. These background tasks are scheduled automatically using the lightweight `schedule` library.

The Librarian is a specialized tool that excels in reproducing the data and layout of a POSIX-compatible filesystem on a system with downstream nodes in a loosely-coupled manner, all whilst maintaining tight control on *how* data is transferred. The Librarian is open source and available through GitHub[6], under the BSD-2-Clause license.

3.1. Technology Choices

For this project, it was crucial that we used the Python language, as this is the lingua franca of the collaboration; all members have at least some knowledge of Python. No other language comes close (by a significant margin), and all analysis tools use Python. In addition, Globus provide a complete source development kit for Python[7]. As our analysis tools will eventually communicate with the Librarian, and its REST API, it also made sense to employ pydantic[8] to develop the communication schema. By using pydantic, we could ensure that responses and requests were serialized and de-serialized by the exact same models, reducing both development time and errors significantly. With pydantic models as our base data structure, it was natural to choose FastAPI for the web server component.

In an ideal case, we would have used SQLModel to further reduce code duplication in the database layer. However, at the outset of the project, SQLModel[9] was still in very early development, and as such it made sense to use SQLAlchemy[10] as our object-relational mapping (ORM) to translate database operations to object manipulation.

3.2. Service Layout

The Librarian service is relatively simple; it allows for the ingestion of 'files' (which can themselves be directories containing many files) into a unified namespace, and this name-

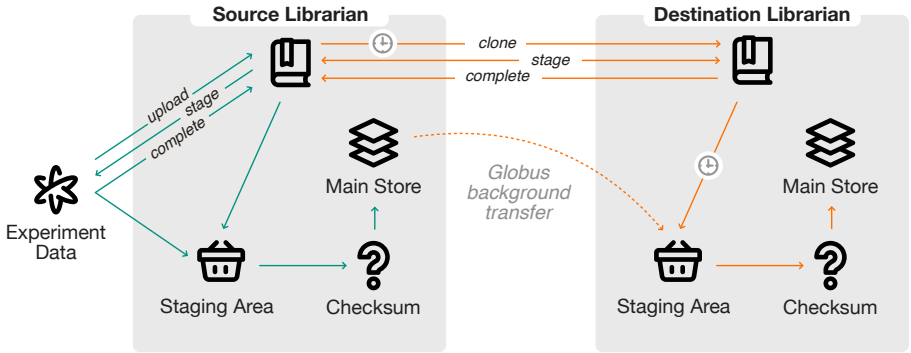

Figure 2. *Showing data ingest (green) and cloning between Librarian instances (orange). Data flows from the experiment, through a staging area, and to a main store, before being picked up by the background task (clocks) and copied by Globus in the background to the remote staging area. On the remote end, once the data is marked as being staged, the background task picks it up, checksums it, and it is marked as stored. More information on this process is available in the main text.*

[6] https://github.com/simonsobs/librarian
[7] https://globus-sdk-python.readthedocs.io/en/stable/index.html
[8] https://docs.pydantic.dev/latest/
[9] https://sqlmodel.tiangolo.com
[10] https://www.sqlalchemy.org

space can be synchronised with other sites using Globus. In this section we will give a brief overview of the key abstractions at place and the data flow within the application. This process is summarised in Figure 2.

In contrast to many other tools, and to the 'HERA Librarian' which preceeded the Librarian, files carry minimal science metadata along with them through the system. We found that metadata systems attached to data flow and management systems often lacked necessary features and caused significant complications (such as needing frequent database migrations) when coupled. As such, we chose to explicitly delegate metadata management to other subsystems within the project, focusing only on file transfers.

3.2.1. *User and Librarian Management:*

During the initial setup of the system, an administrator user is provisioned to facilitate further management tasks. This primary administrator has the ability to create additional user accounts through the `librarian` command-line tool.

Both 'users' (i.e. those interacting with the system to ingest data) and 'Librarians' (other copies of the application running on other systems) need accounts. These accounts can be configured with different levels of permissions to suit various needs. Specifically, accounts can be granted full administrator privileges, read and append only privileges, or callback-only privileges. Callback-only privalages are crucial for remote sites like telescopes, as they ensure that downstream Librarians only have extremely limited access to both the underlying data and its associated metadata. To ensure security, user passwords are salted and hashed in the database using the Argon2[11] algorithm. The API employs HTTP Basic Authentication for user verification and access control.

3.2.2. *Storage Management:*

In the Librarian system, storage is abstracted into entities known as 'stores'. These stores are provisioned during the setup process and can be migrated with a server restart. Each store comprises two components: a staging area for ingested files and a main store area, which has a global namespace for permanently storing data. While stores can have any underlying structure, they must provide methods for both ingestion and egress of data. In all current use cases, we employ a 'local store', which is a thin wrapper around a POSIX filesystem.

Stores provide two methods for data ingestion: transfer managers and async transfer managers. Transfer managers are synchronous, and for the 'local store', this involves a simple local file copy. Async transfer managers, on the other hand, are asynchronous and are used for inter-Librarian transfers, typically employing Globus for this purpose. Individual storage items are referred to as 'files', which can denote either individual files or directories.

Stores on the same device log all the 'instances' of 'files' they contain in the database. Additionally, all files can have 'remote instances', which are known instances of files located on another Librarian.

3.2.3. *Data Ingestion:*

Data ingestion follows a systematic process using accounts that have read and append privileges. Initially, a request to upload is made to the Librarian web server prompting it to create a temporary UUID-named directory in the staging area. Simultaneously, the client computes a checksum for the file. The server then provides this UUID and a set of transfer managers to the client.

[11] https://pypi.org/project/argon2-cffi/

Next, the client selects the most appropriate transfer manager to copy the file to the staging area. Once the file transfer is complete, the client informs the server of the completion. The server then verifies that the checksum matches the one provided in the upload request. If the checksums are consistent, the server ingests the file into the storage area.

From the client's perspective, the upload is extremely simple:

```
from hera_librarian import LibrarianClient
from hera_librarian.settings import client_settings
from pathlib import Path

client_info = client_settings.connections.get(
    "my_librarian_name"
)

client = LibrarianClient.from_info(client_info)

client.upload(
    Path("name_of_a_file_on_disk.txt"),
    Path("/hello/world/this/is/a/file.txt")
)
```

This then leads to a synchronous uploading of the local file to the global namespace, with `upload` returning once this is complete and, crucially, the upload is verified (via a checksum) by the server. For most applications, this fire-and-forget technique is all that is required; if `upload` returns successfully, the file is guaranteed to have been correctly uploaded to the server and is ready for use and export.

3.2.4. *Data Cloning:*

Data can be cloned in two main ways: locally and remotely. Local cloning involves making a copy to a different store on the same Librarian, which is often used for SneakerNet. This process is straightforward and handled by a single task that creates a copy on the second device and updates the database.

Remote cloning, on the other hand, is a more complex, multi-stage process. Initially, files without remote instances on the target Librarian are collated into a list. Outgoing transfers are then created in the database to track progress. These transfers are grouped into batches of 128 to 1024 files to be packaged in a single Globus transfer, due to the limitation that there can only be 100 listed Globus transfers.

A request is made to the destination Librarian to batch stage clones of these files, creating a UUID directory for each. The batch transfer is placed in the queue on the local Librarian. The transfer thread picks up this batch and sends it using Globus, periodically polling for status updates on the transfer. Once the transfer is complete, the source Librarian notifies the destination Librarian that the files are staged. The 'receive clone' background task on the destination Librarian picks up the staged files and verifies them against known checksums. Once verified, the files are placed in the main store, and a callback is generated to the source Librarian to complete the transfer and register a new remote instance.

3.2.5. *Data Access:*

Data can be accessed through the Librarian, by querying it for the location of individual instances of a file. However, because our stores generally just wrap the POSIX filesystem, users typically already know a-priori where the files that they need to access are, through the use of other science product databases. As such, data access is generally as simple as opening the file; it is where users expect it to be.

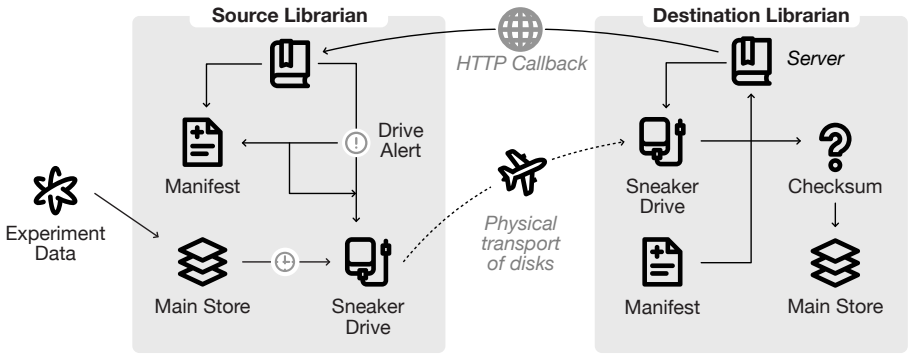

Figure 3. *Showing the SneakerNet flow between two Librarian instances. The sneaker drive is physically moved between computing sites, alongside a manifest of all files on the drive, to complete transfers in large batches.*

3.3. Data Down the Mountain

In the specific case of an observatory, there is an additional challenge here: lines representing data flows in Figure 2 no longer simply show bytes sent over HTTP connections to highly available services, but instead represent potentially interruptible inter-continental data movement. In the specific case of the Simons Observatory, the main data flow from the telescope (which is at 5000 meters above sea level in the Atacama Desert) to the main computing center (NERSC, a national facility on the west coast of the U.S.) is fraught with challenges.

The main internet connection from the telescope to the nearest town was, for a long time, a low-bandwidth radio link. This has now been (thankfully) replaced with a high throughput fiber connection, but given that this link may go down at any time due to the remote placement of the site (and no backup), another way to offload the data is required. The observatory produces around 20 TB of data a week, which is much more than the 50 MBit/s radio link can transfer; either way, that bandwidth is required for other uses. This necessitates a more primitive solution: so-called 'SneakerNet' transfers, where data is carried by hand on physical media.

SneakerNet transfers are supported natively by the Librarian, and occur through the following steps (as shown in Figure 3):

- A new store is provisioned that refers to the mounted drive that will be

carried in the process.
- A local clone background task catches up on missing files to this drive and,

as new files come in, copies them too.
- Once full, the server sends an alert (implemented using Slack notifications)

to administrators.
- Administrators use the command-line Librarian tools to generate a 'manifest'

of all files on the device: their checksums, names, etc.
- The drive is hand-carried to the destination, and ingested.
- The Librarian command-line tools are again used to ingest the manifest and

copied files.
- As the files are ingested, HTTP callbacks are made to the source Librarian to

convert the local 'instance' rows on the sneaker drive to 'remote instance' rows for the destination Librarian.

- The drive is re-formatted and carried back.

This is an unusual process, but comes in extremely handy in bandwidth constrained, remote, environments.

3.4. Testing

Testing complex data flows like those represented in the Librarian is a notoriously difficult task. For instance, testing the components that make up the SneakerNet workflow means being able to test two interacting web-servers, each backed by a database, with several additional threads running in the background. To aid with this goal, we make heavy use of dependency injection and integration testing, alongside more traditional unit testing.

For services that cannot be 'easily' replicated, like Globus, we make sure to always use dependency injection. Dependency injection is a technique where downstream functions and objects 'receive' the things that they require as arguments or parameters. One significant example where this is used in the Librarian is the transfer managers: by having the server tell the client how it can move data, you can return appropriate managers for testing. Instead of using a Globus transfer (which requires setting up an external server, and registering it with Globus), one can instead return a transfer manager that simply wraps `rsync` or a local copy instead, and test all the same code paths except the one line that actually moves the data. The component that performs the Globus transfer is then tested separately in a more appropriate environment, ensuring that it conforms to the exact same specification as the local copy manager. Another example is our database: thanks to SQLAlchemy, and appropriate design patterns, we can very easily swap out our production postgres database for SQLite during development, enabling much more simple testing patterns.

In an attempt to avoid 'dependency hell', we have developed a testing platform that heavily leverages `pytest-xprocess`[12] and pytest fixtures. Using this tool, one can create a real, live, running Librarian server in a separate process. This is used to test the interaction between client and server, and for end-to-end tests of the SneakerNet and inter-Librarian cloning process. Because this server is running locally, it makes it possible to directly query the database for testing, which we have found to be invaluable. The only drawback to this approach is that it is not possible to get test coverage for the lines that are performed inside the `xprocess` fork.

3.5. Deployment Details

As we have made a relatively straightforward selection of technologies, and the fact that the Librarian is a small application (less than 10'000 lines), deployment is a simple process. In addition, it makes it straightforward to explain the code paths that are taken by the Librarian.

Currently, we deploy Librarian using Docker. We require one container for the FastAPI server, and one for postgres, which are usually linked using helm charts (at NERSC) or docker compose (on other systems). We provide access to the database for Grafana dashboards, and have Slack integrations for both logging and notifications.

The focus on simplicity that we have made for the Librarian has made deployment simple; system administrators within the academic community are very familiar with Globus, and have been happy to assist with deployments of Librarian orchestration framework.

[12] https://pytest-xprocess.readthedocs.io/en/latest/

4. Conclusions

In this paper, we addressed the significant challenges faced by large academic collaborations in managing and transferring massive datasets across distributed sites. We evaluated existing data management tools such as Globus Flows, Rucio, iRODS, and Datalad, identifying their limitations in terms of automation, complexity, and compatibility with academic workflows.

Our solution, the Librarian was developed to meet these specific needs. By integrating with Globus, Librarian enables efficient, asynchronous data transfers. The system also supports SneakerNet, which is essential for environments with limited or intermittent connectivity like our observatory, facilitating physical data transfers through portable storage.

The use of widely adopted technologies such as Python, FastAPI, and SQLAlchemy ensures that Librarian is robust, scalable, and user-friendly. Its design aligns well with the decentralized and diverse infrastructure typical of academic projects, providing a practical and efficient method for handling the immense data flows inherent in modern 'big science' endeavors.

Librarian has demonstrated its effectiveness through its deployment at the Simons Observatory, highlighting its potential as a versatile and reliable data management solution for large-scale scientific collaborations. We provide Librarian as free, open source, software to the community.

Acknowledgements

JB acknowledges support from NSF grant AST-2153201. This material is based upon work supported by the National Science Foundation under Grant Nos. 1636646 and 1836019 and institutional support from the HERA collaboration partners. This research is funded in part by the Gordon and Betty Moore Foundation through Grant GBMF5212 to the Massachusetts Institute of Technology.

References

[1] P. Ade *et al.*, "The Simons Observatory: science goals and forecasts," |*jcap*, vol. 2019, no. 2, p. 56, 2019, doi: 10.1088/1475-7516/2019/02/056.

[2] I. Foster, "Globus Online: Accelerating and Democratizing Science through Cloud-Based Services," *IEEE Internet Computing*, vol. 15, no. 3, pp. 70–73, 2011, doi: 10.1109/MIC.2011.64.

[3] B. Allen *et al.*, "Software as a service for data scientists," *Commun. ACM*, vol. 55, no. 2, pp. 81–88, 2012, doi: 10.1145/2076450.2076468.

[4] M. Barisits *et al.*, "Rucio: Scientific Data Management," *Computing and Software for Big Science*, vol. 3, no. 1, p. 11, 2019, doi: 10.1007/s41781-019-0026-3.

[5] R. Consortium, "iRODS." [Online]. Available: https://irods.org/

[6] J. Hess, "git-annex." [Online]. Available: https://git-annex.branchable.com/

[7] Y. O. Halchenko *et al.*, "DataLad: distributed system for joint management of code, data, and their relationship," *Journal of Open Source Software*, vol. 6, no. 63, p. 3262, 2021, doi: 10.21105/joss.03262.

[8] D. R. DeBoer *et al.*, "Hydrogen Epoch of Reionization Array (HERA)," |*pasp*, vol. 129, no. 974, p. 45001, 2017, doi: 10.1088/1538-3873/129/974/045001.

[9] P. La Plante, P. K. G. Williams, and J. S. Dillon, "Developing a Real-Time Processing System for HERA," *URSI Radio Science Letters*, vol. 2, p. 41, 2020, doi: 10.46620/20-0041.

[10] P. La Plante *et al.*, "A Real Time Processing system for big data in astronomy: Applications to HERA," *Astronomy and Computing*, vol. 36, p. 100489, 2021, doi: 10.1016/j.ascom.2021.100489.