# Highly Scalable Data Service (HSDS): an open source implementation of HDF5 for the cloud

John Readey – The HDF Group

The HDF Group

# HDF for the Cloud



- Ideas
  - Web based - provide a RESTful API that is feature compatible with HDF5 Lib API
  - Utilize object storage – cost effective, scalable throughput, redundant
  - Elastic compute – scale throughput by autoscaling compute clusters
  - Compatibility - Provide client SDK so existing HDF applications can just work

# “The Cloud” defined



- “The cloud” doesn’t necessary mean AWS, Azure, Google Cloud, etc.
- Similar technologies can be utilized in on premise data centers as well
- Open source object storage systems such as OpenIO, Ceph, Rook.io
- Kubernetes to manage container-based workloads
  - Applications run as set of containers that read or write to object storage
  - Placement of containers managed by an orchestrator such as Kubernetes
- Advantages of the private cloud:
  - Can scale application beyond the capacity of a single server
  - More efficient utilization of hardware resources
  - Applications are robust in the event of server crashes, hardware failures
  - Architecture ports easily to public cloud vendors

# HSDS in one slide



- **RESTful interface to HDF5 using object storage**
- **Storage using AWS S3, Azure Blob Storage, OpenIO (portable to most other object storage systems)**
  - Built in redundancy
  - Cost effective
  - Scalable throughput
- **Runs as a cluster of Docker containers**
  - Elastically scale compute with usage
- **Feature compatible with HDF5 library**
- **Implemented in Python using asyncio**
  - Task oriented parallelism

# Why use object storage for HDF5 data?



- Lower cost/GB compared with NAS systems and esp. parallel file systems such as Lustre and GPFS
- Easier to scale up as storage needs increase
- Reduce bottlenecks due to POSIX limitations
- No limitation to the size of a “file”
- Enable ”cloud-native” architecture – container based, micro-services
- Support migration to public cloud in the future

# Object Storage Challenges for HDF



- Not POSIX!
- High latency (>0.1s) per request
- Not write/read consistent
- High throughput needs some tricks (use many async requests)
- Request charges can add up (public cloud)

*For HDF5, using the HDF5 library directly on an object storage system is a non-starter. Will need an alternative solution…*

# HDF Cloud Schema



How to store HDF5 content in S3?

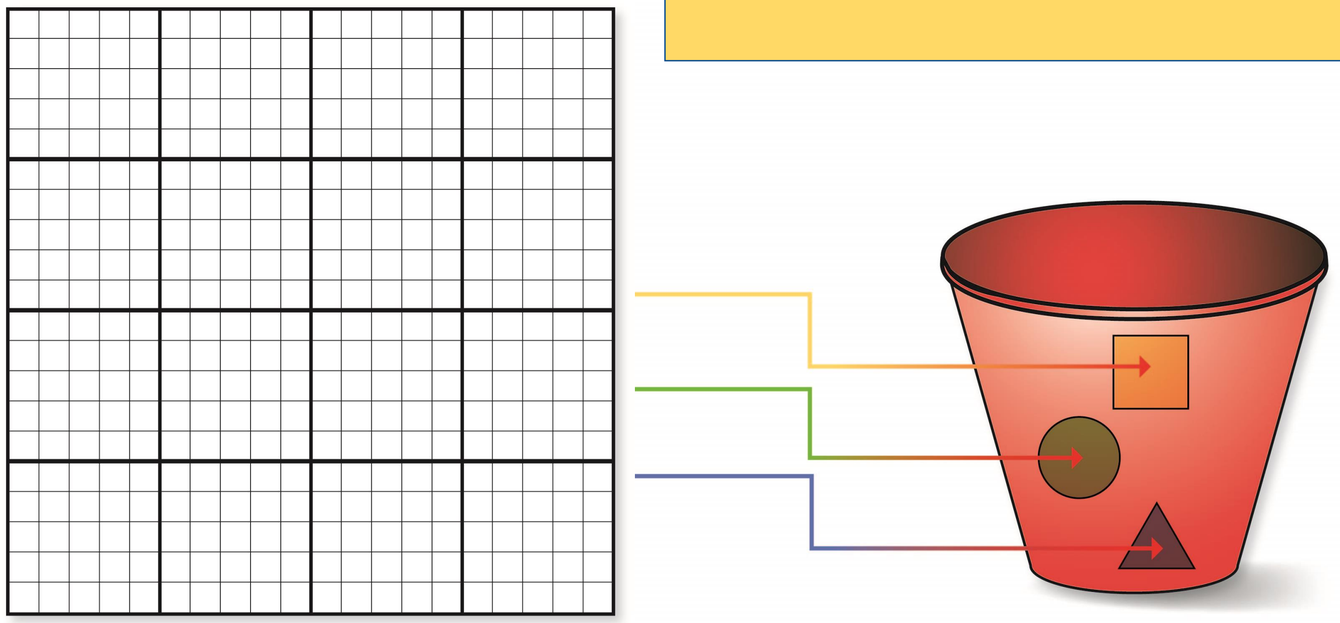
- Limit maximum storage object size
- Support parallelism for read/write
- Only data that is modified needs to be updated
- Multiple clients can be reading/updating the same “file”

**Big Idea: Map individual HDF5 objects (datasets, groups, chunks) as Object Storage Objects**

Each chunk (heavy outlines) get persisted as a separate object

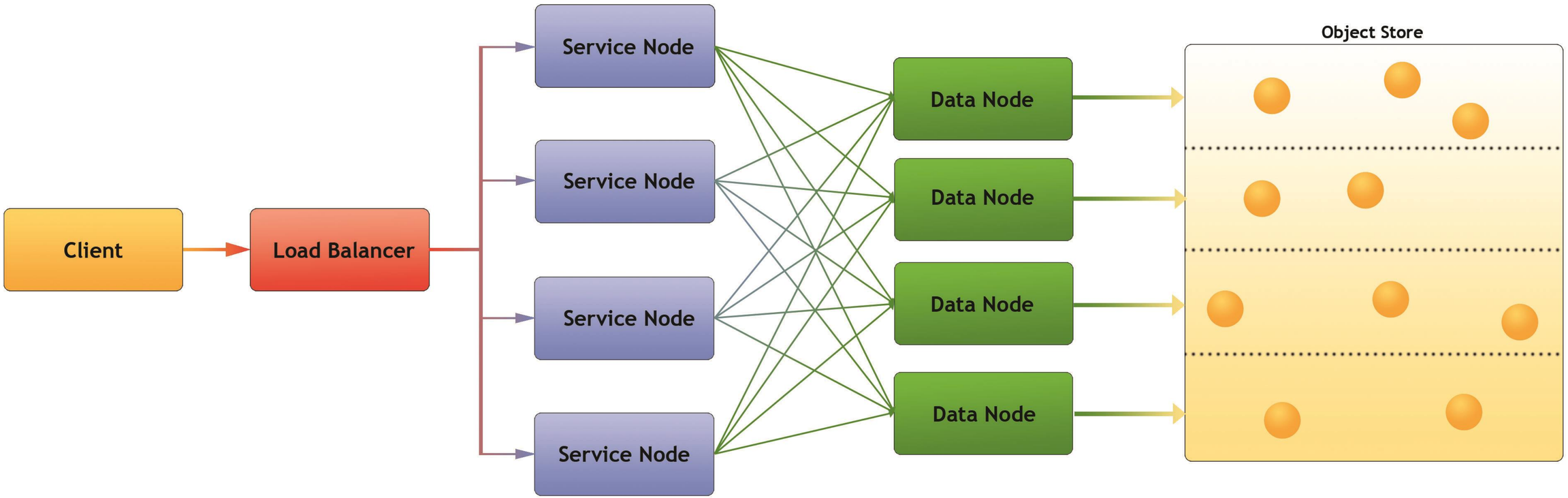
*Legend:*

- *Dataset is partitioned into chunks*
- *Each chunk stored as an S3 object*
- *Dataset meta data (type, shape, attributes, etc.) stored in a separate object (as JSON text)*



# Architecture






- Client: Any user of the service
- Load balancer – distributes requests to Service nodes
- Service Nodes – processes requests from clients (with help from Data Nodes)
- Data Nodes – responsible for partition of Object Store
- Object Store: Base storage service (e.g. AWS S3)

# HSDS Features



- **Simple + familiar API**
  - Clients can interact with service using REST API
  - SDKs provide language specific interface (e.g. h5pyd for Python)
  - Can read/write just the data they need (as opposed to transferring entire files)
  - Support for compression
  - Authentication via HTTP Basic Auth, Azure Active Directory or OpenID
  - Authorization via ACLs (Access Control List)
  - Audit Trail

- **Scalable performance**:
  - Can cache recently accessed data in RAM
  - Can parallelize requests across multiple nodes
  - More nodes ➔ better performance
  - Multiple clients can read/write to same data source
  - No limit to the amount of data that can be stored by the service

# HSDS Platforms



HSDS can be run on most container management systems:

docker

kubernetes

Amazon EKS

Azure Kubernetes Service (AKS)

DC/OS

Using different supported storage systems:

POSIX Filesystem

amazon web services S3

Microsoft Azure Blob Storage

OpenIO

ceph

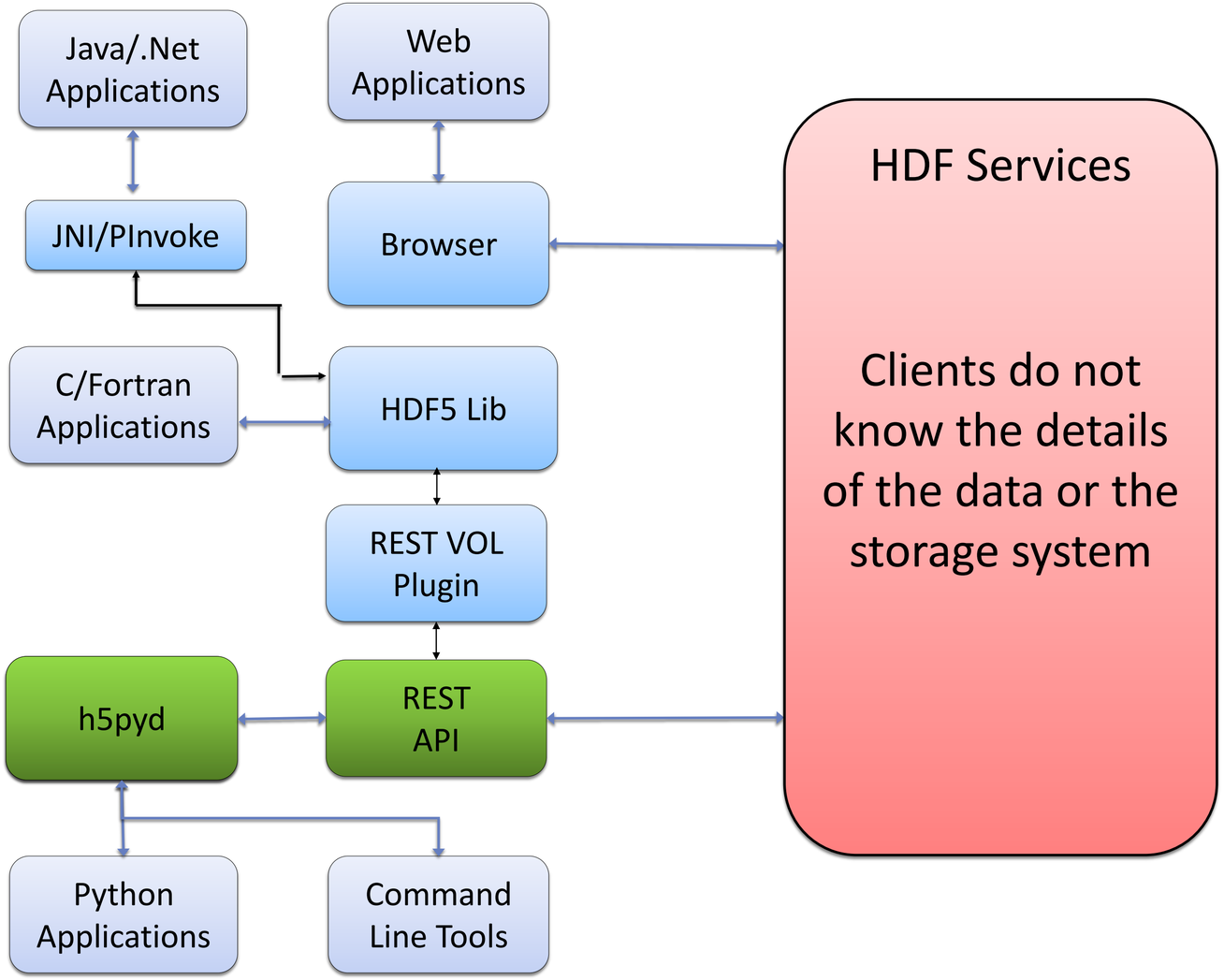
# Architecture



h5pyd for Python and REST VOL plugin are drop-in replacements for libraries used with local files.

No significant code change to access both local and cloud based data.

Data Access Options



# Client SDKs for Python and R



- For Python:
  - H5py is a popular Python package for HDF5
  - H5pyd (for h5py distributed) provides a h5py compatible api for accessing the server
  - Include several extensions to h5py:
    - List content in folders
    - Get/Set ACLs (access control list)
    - Pytables-like query interface
- For R:
    - Prototype SDK developed at Bioconductor:

      https://github.com/shwetagopaul92/rhdf5client

# REST VOL



- The HDF5 VOL architecture is a plugin layer for HDF5
- Public API stays the same, but different back ends can be implemented
- REST VOL substitutes REST API requests for file i/o action
- C/Fortran applications should be able to run as is
- C#/Java application can use library PInvoke/JNI interface with REST VOL

# Command Line Interface (CLI)



- Accessing HDF via a service means one can’t utilize usual shell commands: ls, rm, chmod, etc.

- Command line tools are a set of simple apps to use instead:
  - hsinfo: display server version, connect info
  - hsls: list content of folder or file
  - hstouch: create folder or file
  - hsdel: delete a file
  - hsload: upload an HDF5 file
  - hsget: download content from server to an HDF5 file
  - hsacl: create/list/update ACLs (Access Control Lists)

- Implemented in Python & uses h5pyd

# Supporting traditional HDF5 files



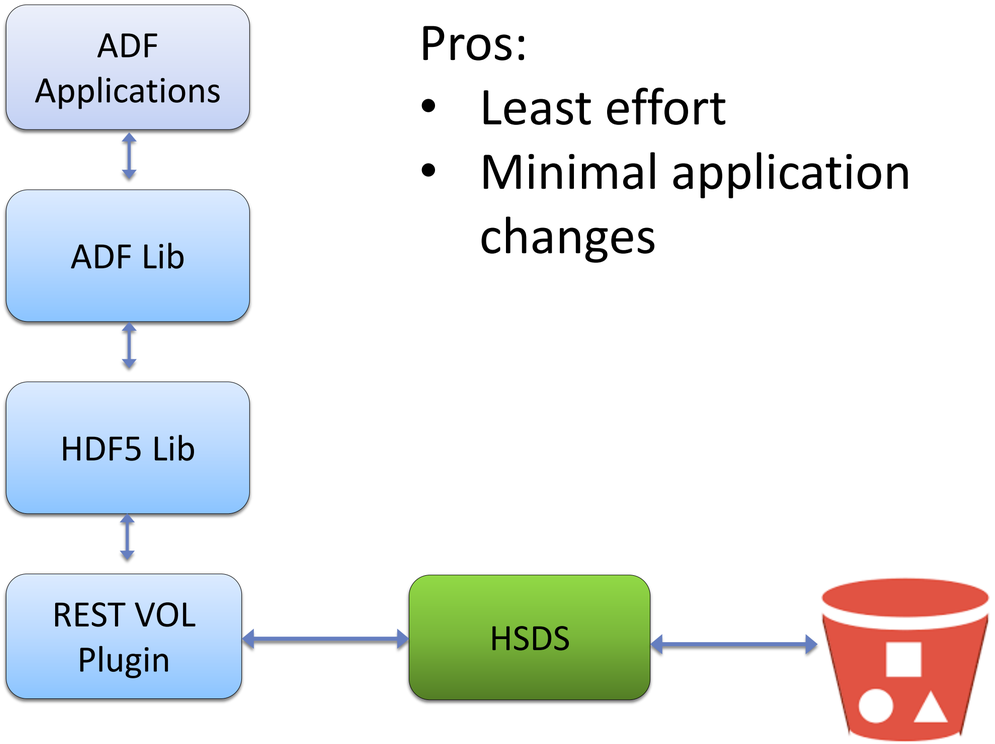
- Downside of the HDF S3 Schema is that data needs be transmogrified
- Since the bulk of the data is usually the chunk data it makes sense to leave the chunks in the file and access through range subsetting:
  - Convert just the metadata of the source HDF5 file to the S3 Schema
  - Store the source file as a S3 object
  - For data reads, metadata provides offset and length into the HDF5 file
  - S3 Range GET returns needed data
- Compared with access via HDF5 library, you reduce the number of S3 requests needed
- Modifications of the HDF5 file are not supported (sometimes a feature)

# ADF as a Service



Some preliminary speculations as ADF as a Service…

Approach #1: Use ADF library with REST VOL




Pros:

- Least effort
- Minimal application changes

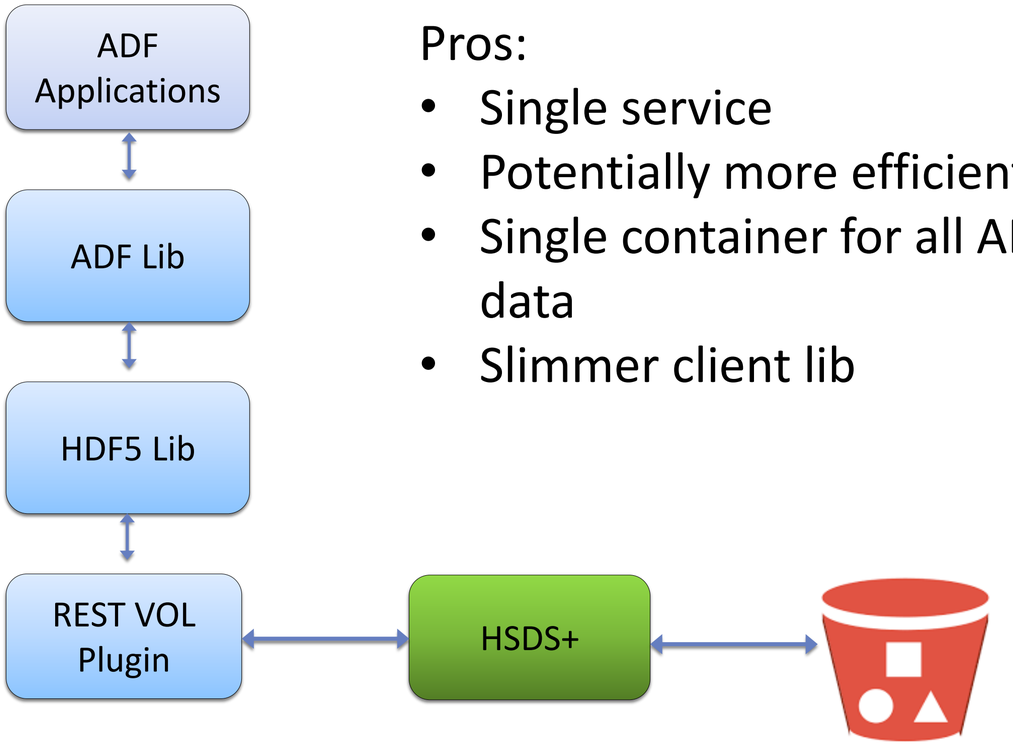
Cons:

- Not efficient for ontologies
- Doesn't provide R/Python interface
- Brittle call pipeline: Java -> C -> HTTP -> Python

# ADF as a Service



Approach #2: Enhance HSDS to support ADF (e.g. RDF API)




Pros:

- Single service
- Potentially more efficient
- Single container for all ADF data
- Slimmer client lib

Cons:

- Amount of effort unclear
- Doesn't provide R/Python interface

# ADF as a Service - cont



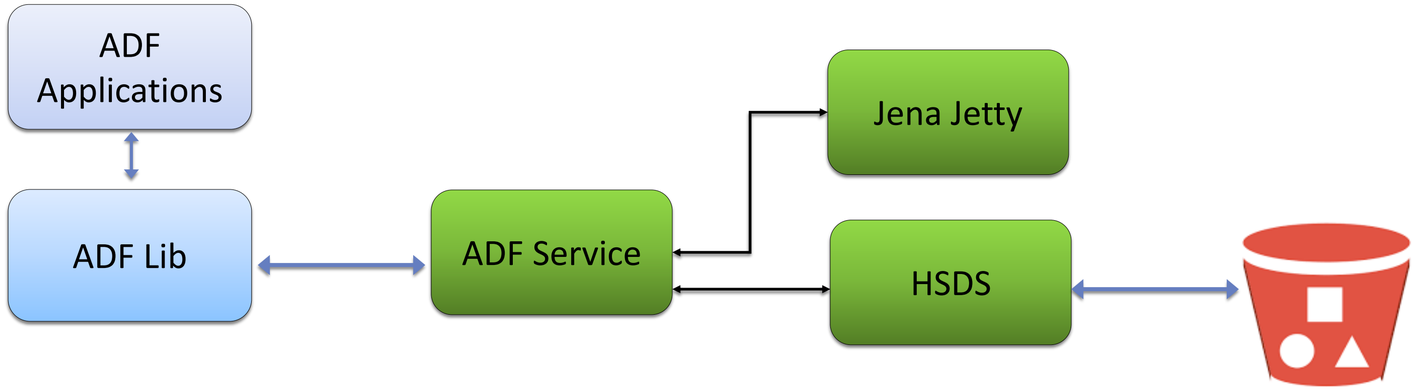
## Approach #3: Create ADF Service

Pros:

- REST API specific for ADF
- Simplest client library
- ADF service can delegate operations to specialized sub-service (e.g. HSDS for data cube, Jena for ontologies)

Cons:

- More effort
- Multiple services to manage
- No “drop down” for apps that want to work at the HDF level



# Questions?

# References



- HDF Schema: https://s3.amazonaws.com/hdfgroup/docs/obj_store_schema.pdf
- SciPy2017 talk: https://s3.amazonaws.com/hdfgroup/docs/hdf_data_services_scipy2017.pdf
- AWS Big Data Blog article: https://aws.amazon.com/blogs/big-data/power-from-wind-open-data-on-aws/
- AWS S3 Performance guidelines: https://docs.aws.amazon.com/AmazonS3/latest/dev/request-rate-perf-considerations.html