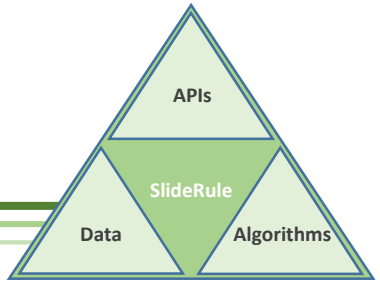
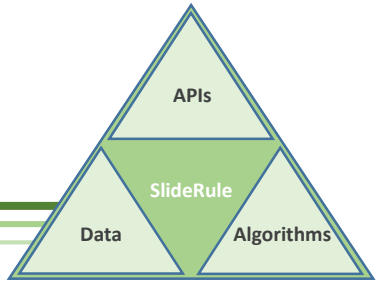


H5Coro: The Cloud-Optimized Read-Only Library

2021 ESIP Summer Meeting

JP Swinski/NASA/GSFC

July 23, 2021



Does HDF5 work well in the cloud?

Cloud

Data access is high latency and high throughput

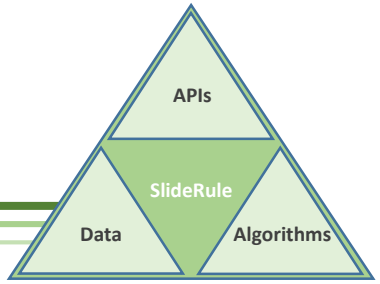
Compute is easily and dynamically scaled

HDF5

The *.h5 data format* works just fine in the cloud

The c library *hdf5* was not written for the cloud

What is H5Coro?



The **H**DF**5** **C**loud-**O**ptimized **R**ead-**O**nly (***H5Coro***) library is a new approach, undertaken by ICESat-2's SlideRule project, to write an HDF5 reader from scratch in C++ that is optimized for cloud environments

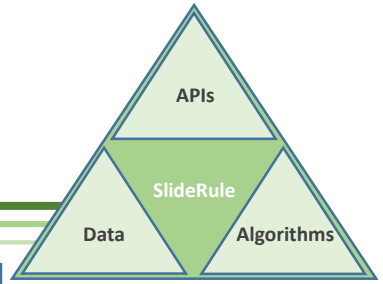
Data Model:

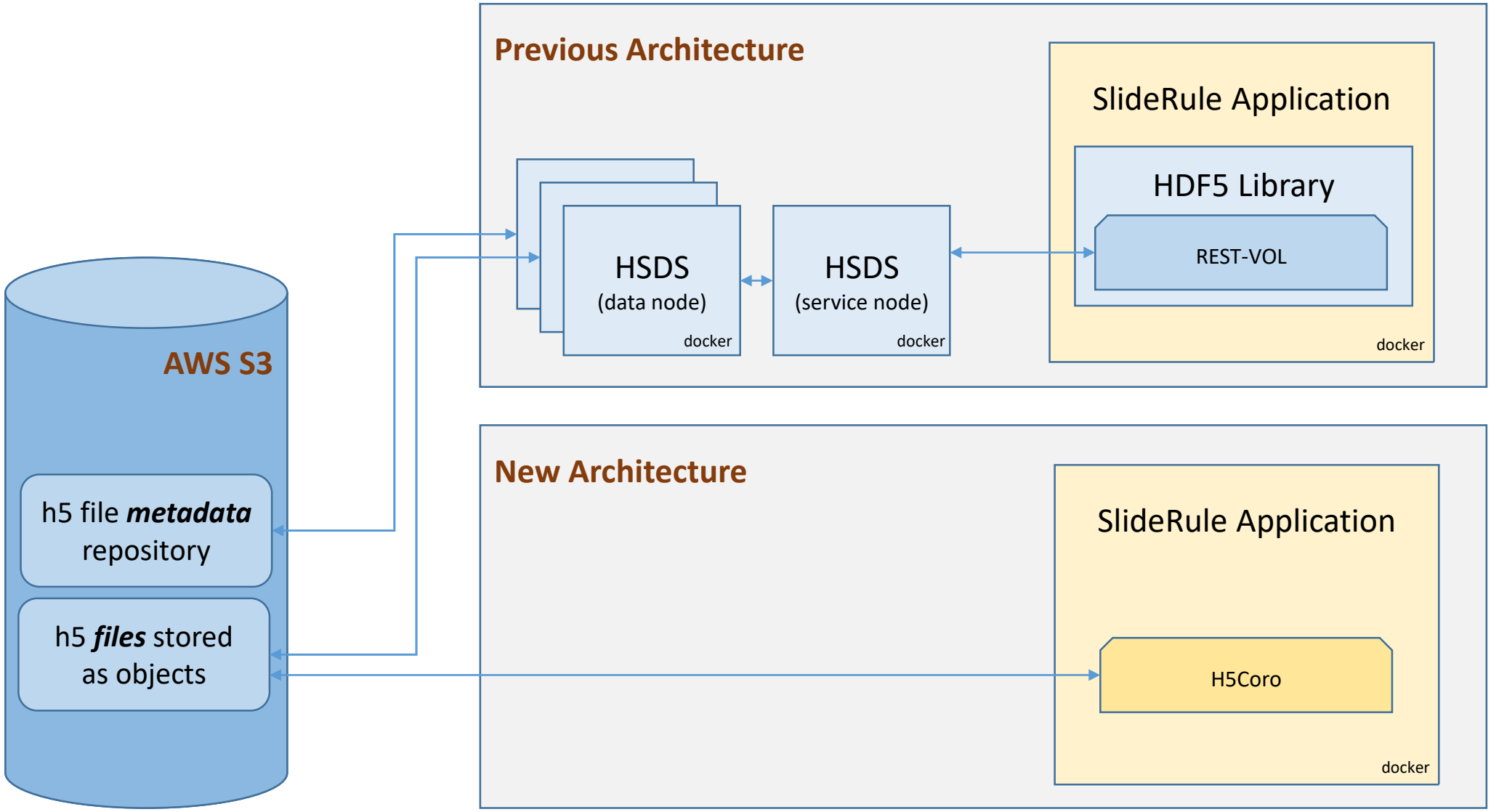
1. Data is static (write once, read many)
2. Data is time series, sequentially stored in memory
3. S3 is high latency and high throughput

Implementation:

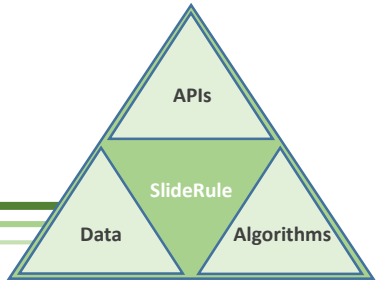
C++ library that strives to minimize the number of I/O operations through caching and Range GET heuristics.

Where H5Coro Fits In





Key Features



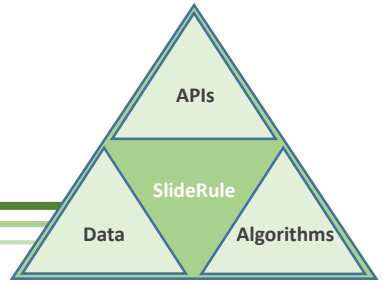
- **All reads are concurrent.** Multiple threads within the same application can issue read requests through H5Coro and those reads will get executed in parallel.

- **Intelligent range gets** are used to read as many dataset chunks as possible in each read operation. This drastically reduces the number of HTTP requests to S3 and means there is no longer a need to re-chunk the data (it actually works better on smaller chunk sizes due to the granularity of the request).

- **The system is serverless.** H5Coro is linked into the running application and scales naturally as the application scales. This reduces overall system complexity.

- **No metadata repository is needed.** Instead of caching the contents of the datasets which are large and may or may not be read again, the library focuses on caching the structure of the file so that successive reads to other datasets in the same file will not have to re-read and re-build the directory structure of the file.

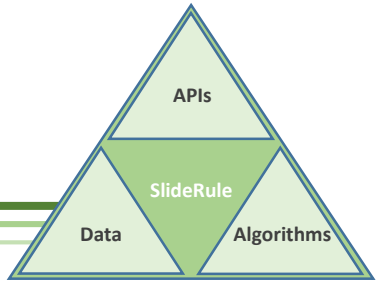
Performance Test



A series of side-by-side performance tests were run against SlideRule using HSDS and H5Coro.

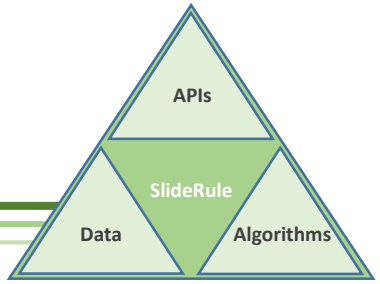
- The tests were run from a local, US east coast, home computer
- A Python script made four concurrent processing requests to a single SlideRule instance running in AWS us-west-2.
- The processing request was to calculate elevations for the Grand Mesa region and required a total of 66 granules to be read from S3.
- HSDS was deployed on the same EC2 instance as SlideRule and consisted of a single service node and eight data nodes.

Performance Comparisons



Library	File Storage	File Structure	Cached	Instance	Time (secs)
HDF5/REST-VOL	S3	Original	Yes	c5.2xlarge	9559 (~2 ½ hrs)
HDF5/REST-VOL	S3	Original	No	c5.2xlarge	9029
HDF5/REST-VOL	S3	Repacked	No	c5.2xlarge	3215
HDF5/REST-VOL	S3	Repacked	Yes	c5.2xlarge	3157
H5Coro	S3	Repacked	No	c5.xlarge	368
H5Coro	S3	Repacked	Yes	c5.xlarge	336
HDF5	Ext4	Original	No	desktop	154
H5Coro	S3	Original	No	c5.xlarge	116 (~2 mins)
H5Coro	S3	Original	Yes	c5.xlarge	72
H5Coro	Ext4/Buffered	Original	No	desktop	56

Performance Results

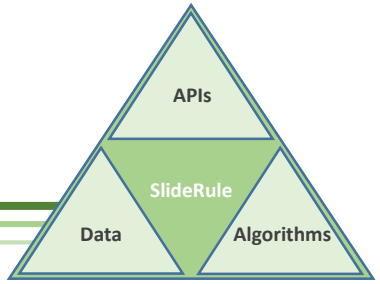


- For test runs where *no caching* was used, H5Coro performed 77x faster.

- For test runs where *caching* was used, H5Coro performed 132x faster.

- It is typical for us to now process regions that used to take 1 ½ hours, in 25 seconds.

H5Coro API

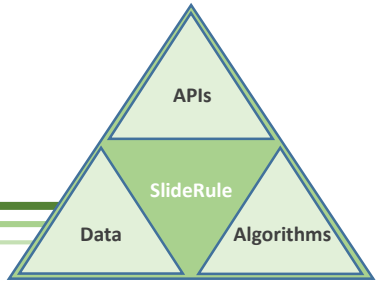



```
info_t    H5Coro::read  (const char* url,
                         const char* datasetname,
                         RecordObject::valType_t valtype,
                         long col,
                         long startrow,
                         long numrows,
                         context_t* context=NULL)
```

where

`url`	the fully qualified path to the H5 file (i.e. s3:///mybucket/folder/myfile.h5)
`datasetname`	the full path to the name of the dataset within the H5 file
`valtype`	the data type of the data to be returned; the recommended type is DYNAMIC, which tells the library to return the data in the type it is stored as in the file
`col`	the column to be read from multi-dimensional datasets; in order to read from multiple columns, multiple reads are needed
`startrow`	the starting row in the dataset to read from
`numrows`	the number of rows in the dataset to read
`context`	an opaque handle to a context structure which holds cached information about the file so that future read operations do not have to re-read metadata portions of the file
`info_t`	a structure holding the contents of the data read from the H5 file along with useful metadata like the data type

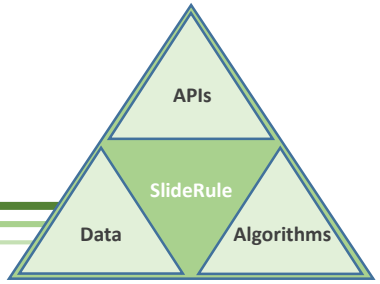
Caching



- A ***local file cache*** is used to store blocks of the H5 file being read. A minimal cache line size is configured at compile time and works on the assumption that the fields being read from the internal structures of the H5 file are often near each other. The *local file cache* is maintained in the context pointer optionally passed to the H5Coro::read call. This allows applications which read multiple datasets from a single file to re-use the *local file cache* between those reads. It is also important to note that dataset contents are not cached, this maximizes the available memory in the cache for file structure metadata.

- A ***global dataset cache*** is used to store the metadata information of a dataset. Given there are only a few pieces of information needed in order to know how to read a dataset, the H5Coro library maintains a large set of the most recently read datasets. If the dataset is requested again, the library does not need to re-traverse the H5 file structure in order to arrive at the necessary data object, but can skip directly to the step where the start and stop addresses of the data are guessed at and the data contents are read.

HDF5 Specification ***NOT*** Supported



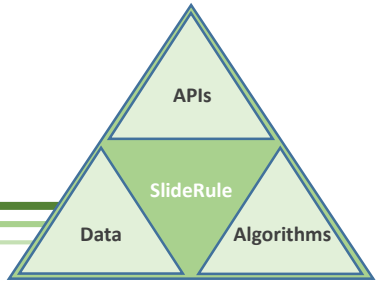
The following portions of the HDF5 format specification are intentionally not implemented:

- All write operations
- File free space management
- File driver information
- Virtual datasets

The following portions of the HDF5 format specification are intentionally constrained:

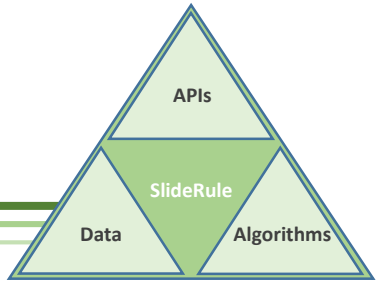
- Datasets with dimensions greater than 2 are flattened to 2 dimensions and left to the user to index.
- Only sequentially stored data can be read at one time, hyperslabs are not supported.
- Data type conversions are supported for fixed and floating point numbers only, but the intended use of the library is to return a raw memory block with the data values written sequentially into it, allowing the user to cast the memory to the correct array type.

Support for HDF5 File Structures



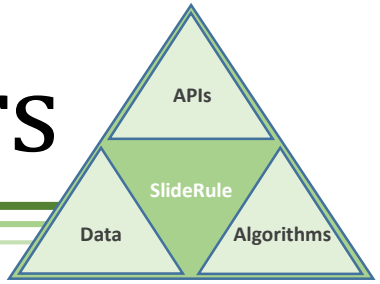
Format Element	Supported	Contains	Missing
Field Sizes	Yes	1, 2, 4, 8 bytes	
Superblock	Partial	Version 0	Version 1, 2, 3
B-Tree	Partial	Version 1	Version 2
Group Symbol Table	Yes	Version 1	
Local Heap	Yes	Version 0	
Global Heap	No		Version 1
Fractal Heap	Yes	Version 0	
Shared Object Header Message Table	No		Version 0
Data Object Headers	Yes	Version 1, 2	

Support for HDF5 Messages



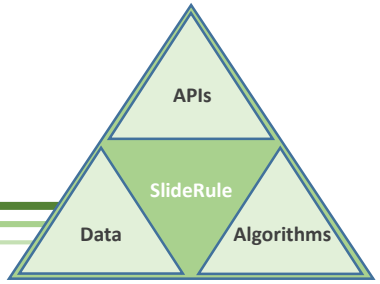
Format Element	Supported	Contains	Missing
Shared Message	No		Version 1
NIL Message	Yes	Unversioned	
Dataspace Message	Yes	Version 1	
Link Info Message	Yes	Version 0	
Datatype Message	Partial	Version 1	Version 0, 2, 3
Fill Value (Old) Message	No		Unversioned
Fill Value Message	Partial	Version 2	Version 1, 3
Link Message	Yes	Version 1	
External Data Files Message	No		Version 1
Data Layout Message	Partial	Version 3	Version 1, 2
Bogus Message	No		Unversioned
Group Info Message	No		Version 0
Filter Pipeline Message	Yes	Version 1	
Attribute Message	No		Version 1
Object Comment Message	No		Unversioned
Object Modification Time (Old) Message	No		Unversioned
Shared Message Table Message	No		Version 0
Object Header Continuation Message	Yes	Version 1, 2	
Symbol Table Message	Yes	Unversioned	
Object Modification Time Message	No		Version 1
B-Tree 'K' Value Message	No		Version 0
Driver Info Message	No		Version 0
Attribute Info Message	No		Version 0
Object Reference Count Message	No		Version 0

Support for HDF5 Storage, Types, Filters



Format Element	Supported	Contains	Missing
Compact Storage	Yes		
Continuous Storage	Yes		
Chunked Storage	Yes		
Fixed Point Type	Yes		
Floating Point Type	Yes		
Time Type	No		
String Type	No		
Bit Field Type	No		
Opaque Type	No		
Compound Type	No		
Reference Type	No		
Enumerated Type	No		
Variable Length Type	No		
Array Type	No		
Deflate Filter	Yes		
Shuffle Filter	Yes		
Fletcher32 Filter	No		
Szip Filter	No		
Nbit Filter	No		
Scale Offset Filter	No		

H5Coro Implementation Limitations

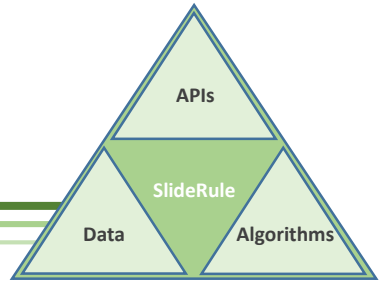


- The H5Coro library was written to optimize access to numerical time-series datasets which are chunked, compressed, and stored sequentially in memory. This layout is exploited in order to quickly determine the start and stop location of the data to be read.

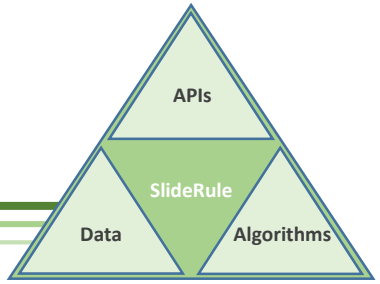
- The H5Coro library, as written, would not work well subsetting image data. Subsetting to a spatial region is likely to cause the requested data chunks to be located in non-sequential memory locations within the file. Such a layout is not anticipated by the heuristics used inside H5Coro for constructing read requests to S3, and would result in non-optimal performance.

- While the current implementation of H5Coro is poorly suited for such applications, the overall approach taken by the H5Coro still applies, and minor modifications to the internal heuristics used by H5Coro could produce drastic improvements in performance for image data.

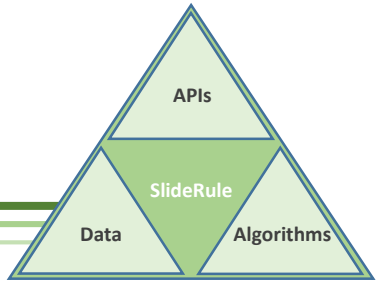
Acronyms



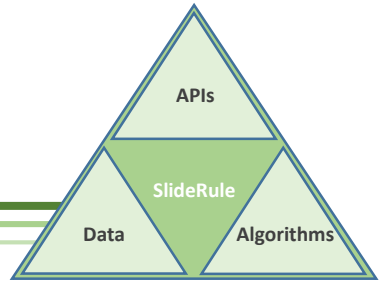
API	Application Program Interface
AWS	Amazon Web Services
EC2	Elastic Compute Cloud
GSFC	Goddard Space Flight Center
HDF5	Hierarchical Data Format version 5
HTTP	Hypertext Transfer Protocol
ICESat-2	Ice, Cloud, and land Elevation Satellite, 2nd generation
IO	Input / Output
IP	Internet Protocol
NASA	National Aeronautics and Space Administration
NSIDC	National Snow and Ice Data Center
REST	Representational State Transfer
S3	Simple Cloud Storage Service
TCP	Transmission Control Protocol



BACKUP



Why A New Approach?



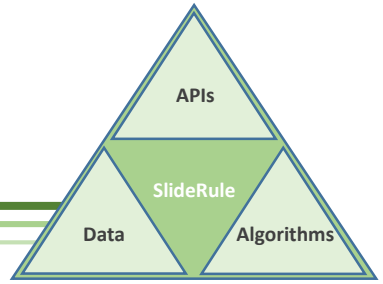
Consider…

Science product services running in AWS that access H5 data in S3

Python scripts written by researchers that read H5 files stored locally

Fortran programs run on supercomputer clusters that generate the official ICESat-2 H5 data products

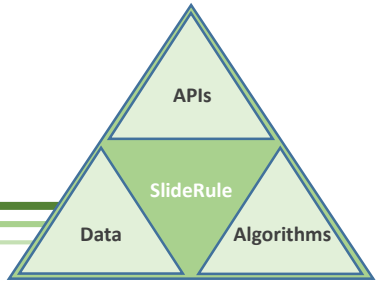
…all use the same underlying HDF5 library.



Historically, the application's use of the HDF5 library was tuned for each environment the application ran in.

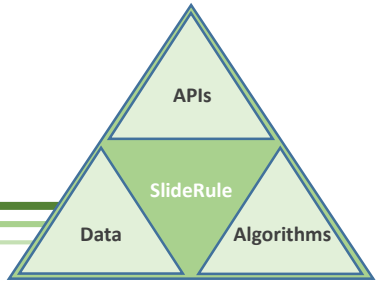
But that is not working well for cloud environments; to address this challenge, various efforts are looking to:

- Restructure the H5 data inside the file (repack)
- Overlay cloud optimized indexes on the H5 files
- Reformat the data into cloud native formats



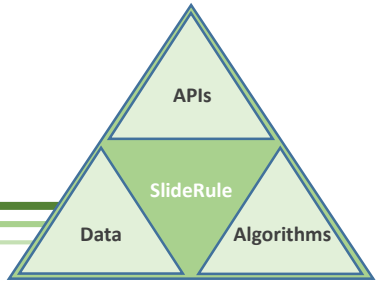
This presentation argues to add a fourth option to the mix:
instead of changing the data, change the library

Given that there is currently only a single implementation of the HDF5 format in use – the HDF5 library, it becomes the de facto standard; and the limitations of the library become the limitations of the format. If the library performs poorly in a cloud environment, it is said that the format is not suited to the cloud environment.



Overview of SlideRule

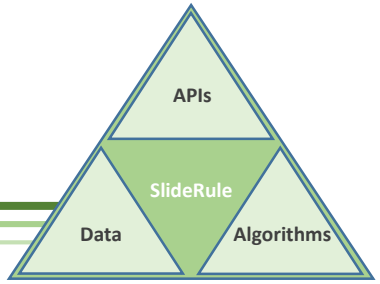
What is SlideRule



- SlideRule is a server-side framework implemented in **C++/Lua** that provides **REST APIs** for processing science data and returning results.

- The project is a collaboration between University of Washington and Goddard Space Flight Center, funded by the **ICESat-2** program. The initial target application is processing the lower-level ICESat-2 point-cloud and atmospheric datasets for seasonal **snow depth** mapping and **glacier** research.

ICESat-2 Program Information



Mission

- Ice, Cloud, and land Elevation Satellite (ICESat-2) launched on September 15, 2018, with a three-year minimal mission life
- The Advanced Topographical Laser Altimetry System (ATLAS) is the sole instrument; it fires a laser towards earth 10,000 times a second and measures the amount of time it takes individual photons to reflect of the earth's surface and return back to the spacecraft.
- The individual photon time measurements are used to calculate surface elevations to a cm-level resolution.

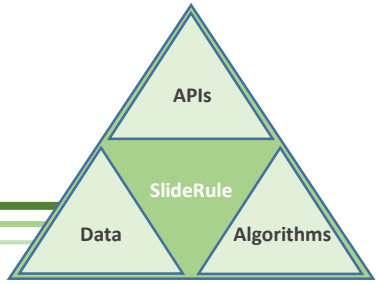
Data

- ICESat-2 produces about 150 TB/year of low-level data.
- At 100Mbps egress, it would take 4.5 months to retrieve one year of data.
- Continuous aggregated egress rate varies depending on the network infrastructure. Over Internet2, rates as high as 400Mbps can be achieved, which still puts the time needed to retrieve the data at about five weeks.

Algorithms

- ICESat-2 has two low-level data products – one used to study surface elevations, and one used to study atmospheric layers
- NASA provides seven Level-3A ICESat-2 data products covering a range of anticipated science applications
- NASA provides nine Level-3B ICESat-2 data products that target more specific science applications

SlideRule Project Objectives



Project Objective: Promote new scientific discovery by lowering the barrier of entry to using the ICESat-2 data.

Tie-In to NASA's Mission: To make NASA datasets which are publicly available, practically accessible.

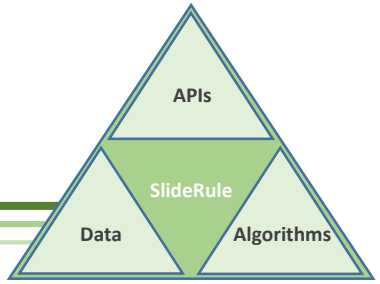
Problem Statement: The amount of data produced by ICESat-2 and the computational resources required by the algorithms that process the data, creates an often insurmountable barrier of entry to using the lower-level ICESat-2 data products. As a result, the typical use of ICESat-2 data is constrained to the pre-launch predicted science applications for which higher-level data products are generated.

Proposed Solution: Develop and deploy a publicly accessible ICESat-2 science data service that provides science data products generated on-demand using parameters supplied by researchers at the time of the request.

Key Benefits:

- There is a one-to-one mapping between resources spent producing data products and which data products are being used by the community.
- Unforeseen science applications are supported, with no additional cost, by the same system that supports the primary science objectives of the mission.
- The service-based architecture promotes integration with other agencies and organizations to improve the products and services they provide.
- Improvements to the algorithms that process the lower-level science data are immediately made available to the user communities (there is no longer a need to reprocess hundreds of terabytes of data, host the new version, and require users to re-download the data when a change is made in the processing algorithms).

SlideRule Design Goals



(1) Cost effective

- Near zero costs incurred when not in use
- Ability to scale in a cost-controlled way to handle processing demand

(2) Responsive Results

- For interactive sessions, the results for areas like Grand Mesa should be returned quickly enough that the user doesn't go off and do something else
- For integrated services (other software systems using our system as a service), the results for very small areas should be returned quickly enough that they can integrate it into their systems without losing the attention of their users.

(3) Simple, well documented API

- Public sliderule-python repository with packages available in PyPI (pip), and Conda
- Very small learning curve with behavior matching expectations

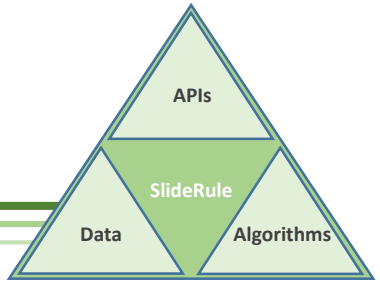
(4) Ability to easily integrate other large datasets

- GLAH12, Tan-DEM-X, Hi-MAT DEM

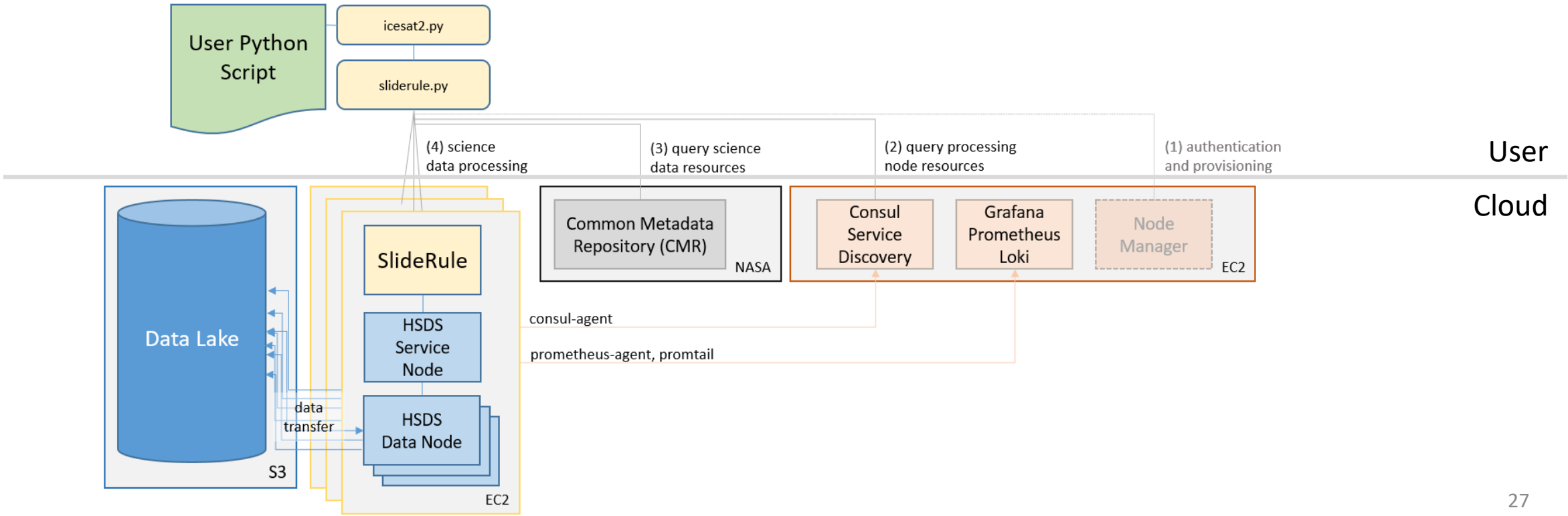
(5) Highly configurable processing engine

- Able to target a wide range of science use-cases without becoming overly complex

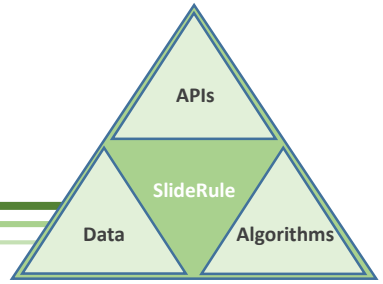
SlideRule Components w/ HSDS

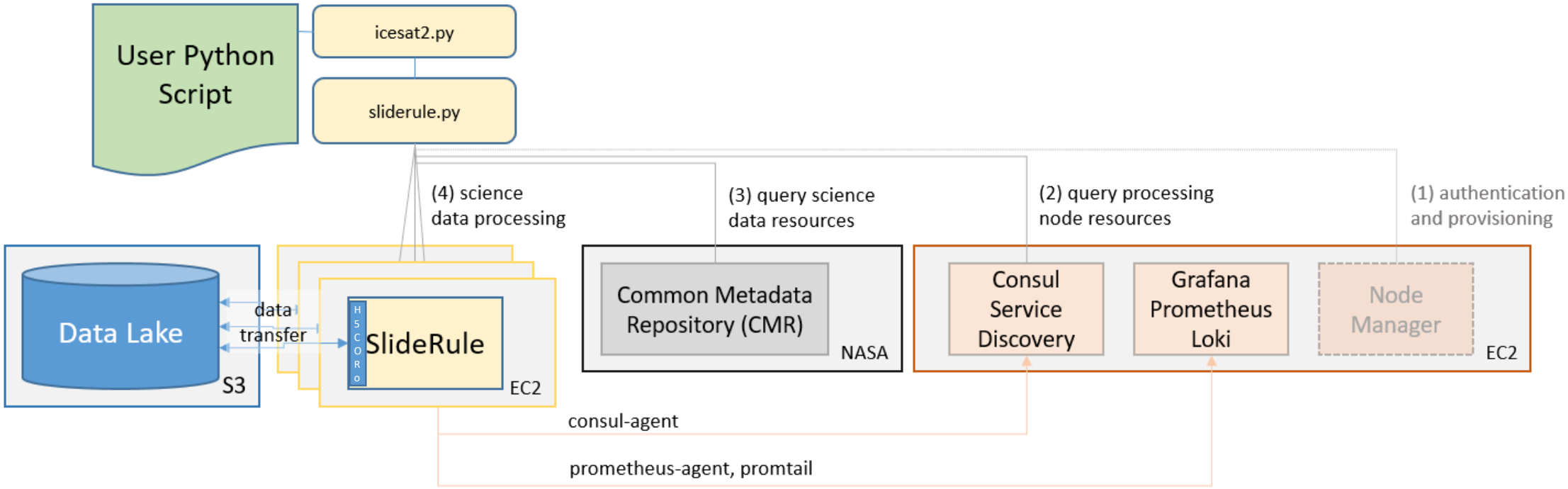


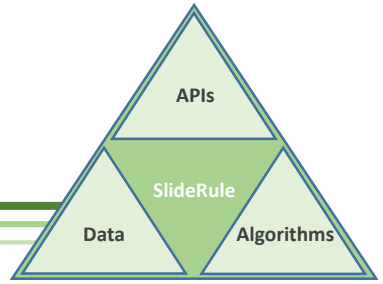
- Client side **Python** packages for easy interfacing
- Back-end data services provided by **HSDS** with data stored in **AWS S3**
- **CMR** used for data set queries



SlideRule Components w/ H5Coro

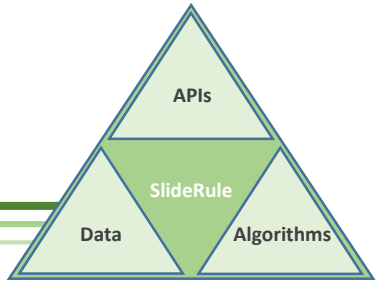






Limitations of Initial Design

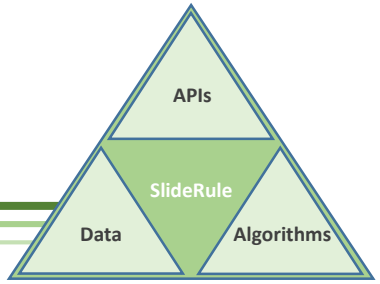
Limitations of Initial Design



Using the HDF5 library presented certain obstacles to us achieving our goals – namely reaching a balance between having a cost effective system and a responsive system.

- All read calls into the HDF5 library are serialized
- HSDS issues multiple HTTP requests per H5 dataset chunk
- HSDS requires a metadata repository

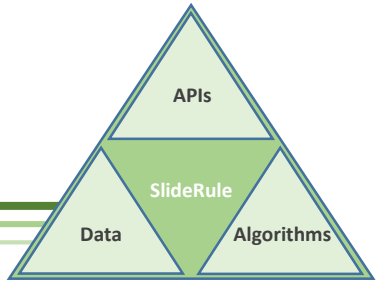
Obstacle 1: HDF5 Serializes Requests

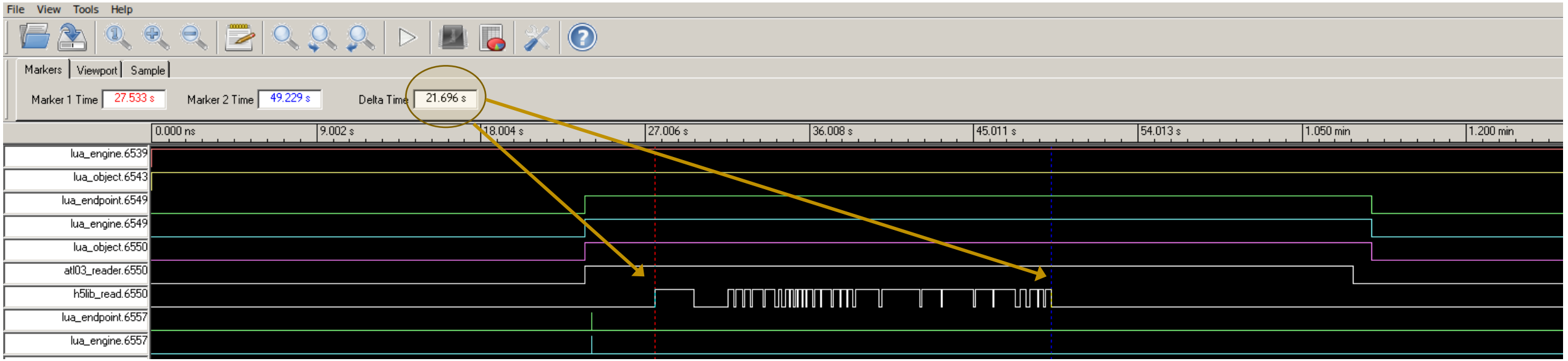


- All read calls into the HDF5 library are serialized inside the library due to a global API lock.

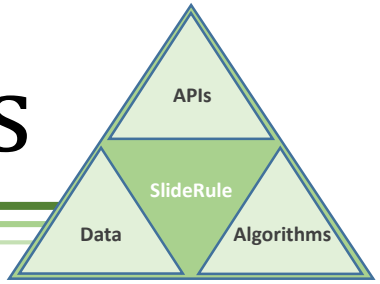
- Even though SlideRule issues many dataset read requests concurrently, and HSDS is capable of tremendous parallelism, the number of actual concurrent reads to S3 was limited to those associated with one dataset at a time due to those requests being serialized at inside the HDF5 library.

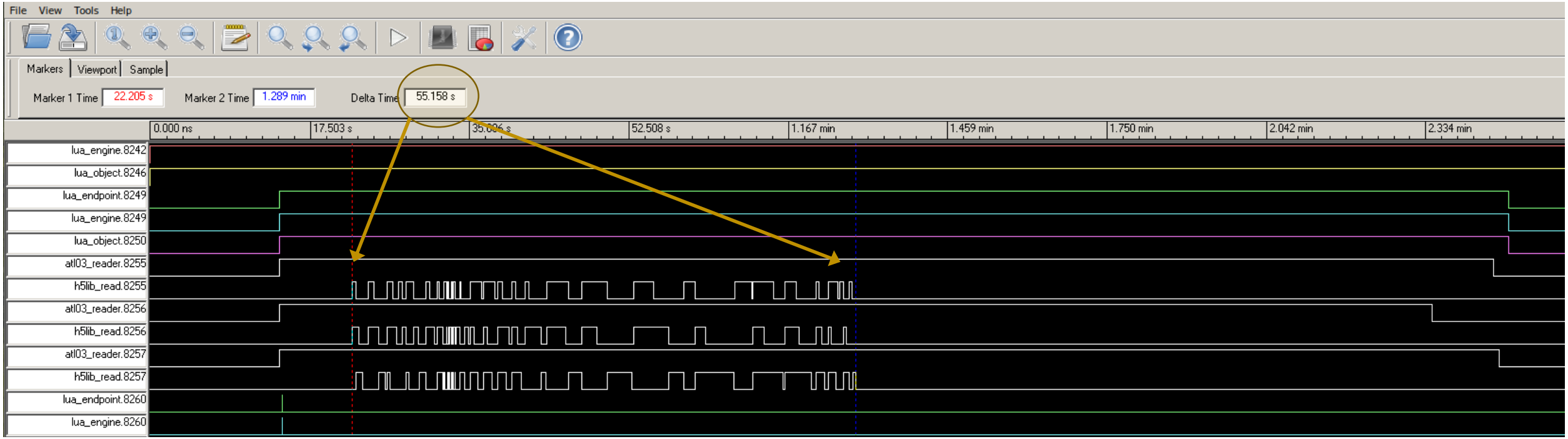
HDF5 Serialization of Reads – 1 Thread



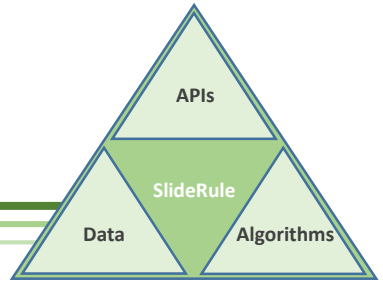


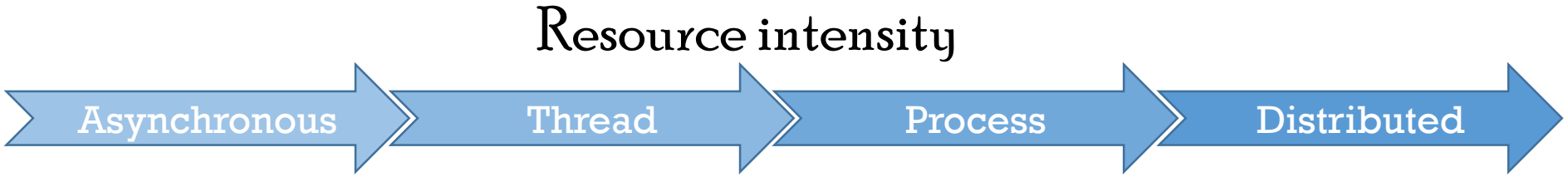
HSF5 Serialization of Reads – 3 Threads





The Cost of Parallelism



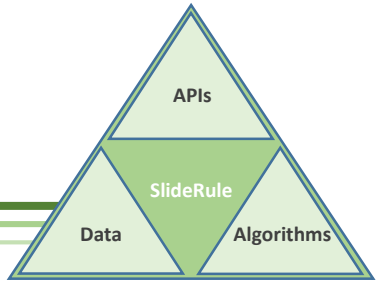


- The HDF5 library prevents an application from using an *asynchronous* or *multithreaded* model to achieve parallelism, and forces a *process* or *distributed* model.

- While large systems will still typically grow at the process and instance (distributed) level, ignoring gains in parallelism at the lower levels is inefficient and costly.

- We found it an order of magnitude more complex to replace call-backs and threads in our server code with Docker containers.

Obstacle 2: Chunk Size Dependency

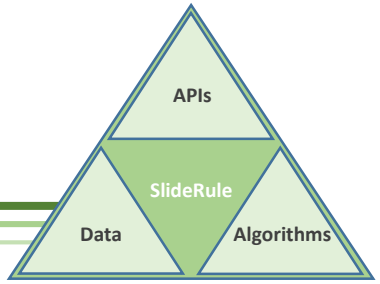


- HSDS issues multiple HTTP requests per H5 dataset chunk being read.

- The chunk size of the dataset is the single greatest factor in how performant the read is.

- Datasets that consist of many small chunks explode the number of TCP/IP socket connections that are needed and the per-read latencies dominate overall performance.

HSDS Performance Test Results

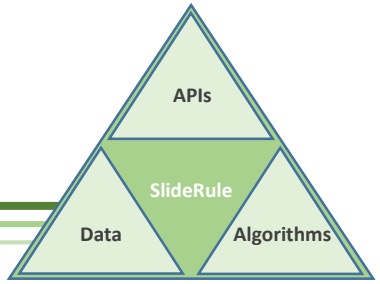


Format	Layout	Instance Type	Data Nodes	Http Compression	Duration (seconds)
Local File	Chunk:(80KB)	c5.xlarge	n/a	n/a	10
Native Ingest	n/a	c5.xlarge	4	Yes	60
Link Option	Chunk:(80KB)	c5.xlarge	4	Yes	990 to 1050
Link Option	Chunk:(80KB)	c5.xlarge	4	**No**	1150
Link Option	Chunk:(80KB)	c5.xlarge	**8**	Yes	990
Link Option	Chunk:(80KB)	**c5.4xlarge**	**16**	Yes	620
Link Option	**Chunk:(143KB)**	c5.xlarge	4	Yes	370
Link Option	**Chunk:(50MB)**	c5.xlarge	4	Yes	60
Link Option	**Continuous**	c5.xlarge	4	Yes	30 to 50

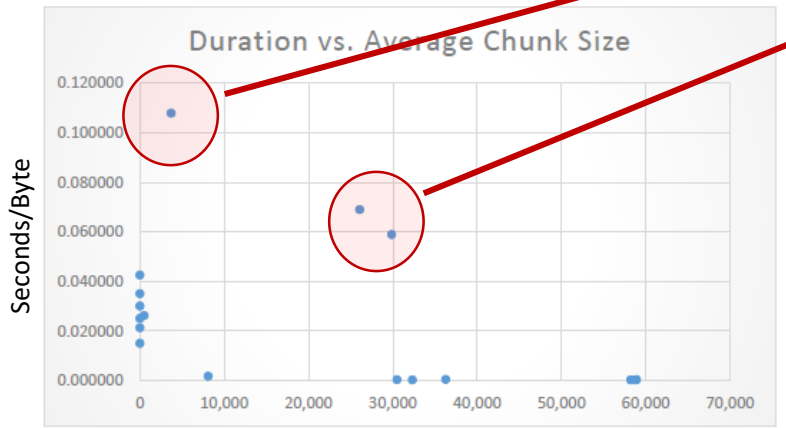
Notes:

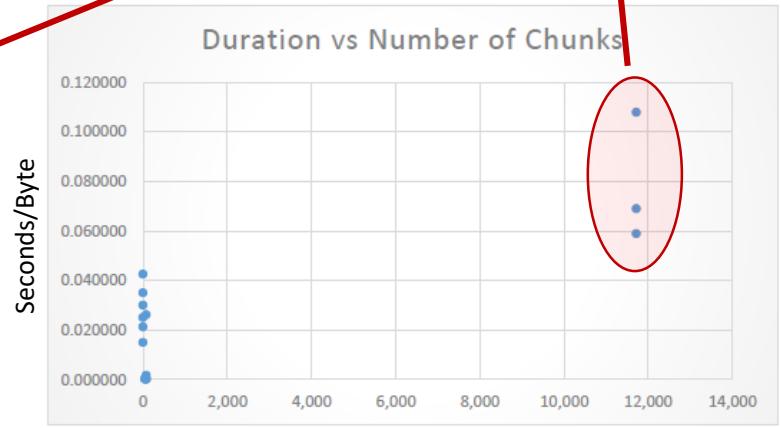
1. Read a total of 718MB of data out of 72 different datasets inside a large (~2GB) h5 file.
2. The c5.xlarge instance has 4 cores, 8GB of RAM, and up to 10Gbps of network connectivity.
3. All test runs used only one service node.
4. No caching was used, all data was fetched fresh from S3

HSDS Performance w/ Original Layout



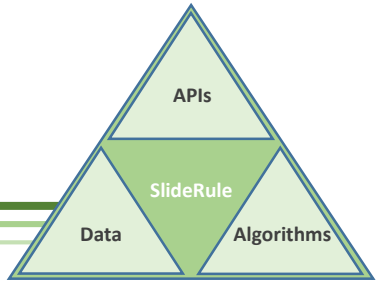
						Size	Elements	Chunks		Duration	
Dataset	Type	Storage Layout	Compression	Filters	Chunk Size	Total (bytes)	Total	Count	Average Size (bytes)	Average per Chunk (secs)	Average per Byte (secs)
/ancillary_data/atlas_sdp_gps_epoch	64-bit float	Compact	None	None	1	8	1	1	8	0.120	0.015000
/orbit_info/sc_orient	8-bit integer	Chunked	None	None	16	16	1	1	16	0.340	0.021250
/ancillary_data/start_rgt	32-bit integer	Compact	None	None	1	4	1	1	4	0.120	0.030000
/ancillary_data/end_rgt	32-bit integer	Compact	None	None	1	4	1	1	4	0.100	0.025000
/ancillary_data/start_cycle	32-bit integer	Compact	None	None	1	4	1	1	4	0.170	0.042500
/ancillary_data/end_cycle	32-bit integer	Compact	None	None	1	4	1	1	4	0.140	0.035000
<spot>/geolocation/delta_time	64-bit float	Chunked	GZIP: 6	None	10000	2,685,617	730,524	74	36,292	0.032	0.000396
<spot>/geolocation/segment_ph_cnt	32-bit integer	Chunked	GZIP: 6	Shuffle: 4	10000	600,951	730,524	74	8,121	0.032	0.001733
<spot>/geolocation/segment_id	32-bit integer	Chunked	GZIP: 6	Shuffle: 4	10000	38,586	730,524	74	521	0.031	0.026178
<spot>/geolocation/segment_dist_x	64-bit float	Chunked	GZIP: 6	None	10000	4,361,058	730,524	74	58,933	0.035	0.000267
<spot>/geolocation/reference_photon_lat	64-bit float	Chunked	GZIP: 6	None	10000	4,310,116	730,524	74	58,245	0.030	0.000230
<spot>/geolocation/reference_photon_lon	64-bit float	Chunked	GZIP: 6	None	10000	4,351,244	730,524	74	58,801	0.029	0.000222
<spot>/heights/dist_ph_along	32-bit integer	Chunked	GZIP: 6	None	10000	350,051,255	117,212,160	11,722	29,863	0.025	0.058842
<spot>/heights/h_ph	32-bit integer	Chunked	GZIP: 6	None	10000	305,855,771	117,212,160	11,722	26,092	0.026	0.068913
<spot>/heights/signal_conf_ph	8-bit integer	Chunked	GZIP: 6	Shuffle: 1	10000	43,666,117	117,212,160	11,722	3,725	0.006	0.107815
<spot>/bckgrd_atlas/delta_time	64-bit float	Chunked	GZIP: 6	None	10000	1,281,096	411,830	42	30,502	0.032	0.000261
<spot>/bckgrd_atlas/bckgrd_rate	32-bit integer	Chunked	GZIP: 6	None	10000	1,358,123	411,830	42	32,336	0.029	0.000226
						718,559,974	356,843,290	35,700	20,204	0.076	0.433833



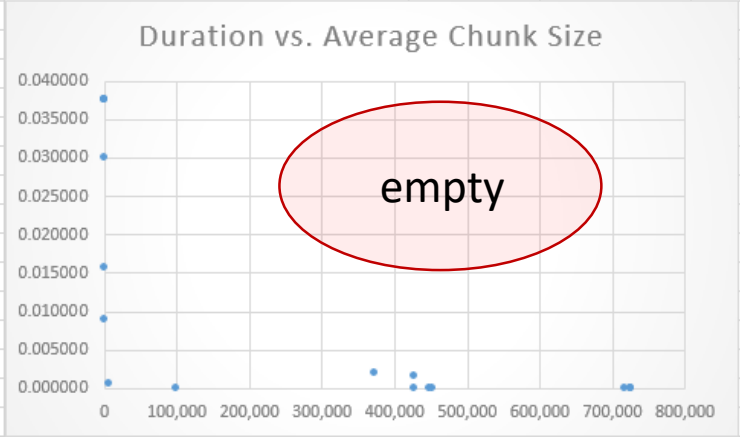


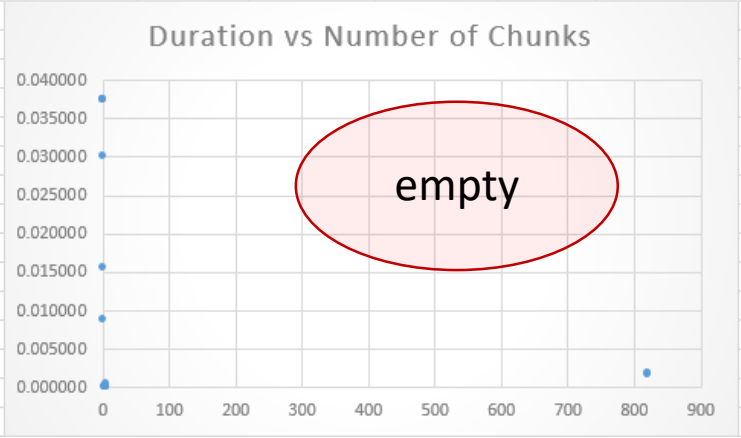
Performance is dominated by the number of chunks being read. This suggests that the per-chunk overhead drives performance.

HSDS Performance w/ 14x Chunk Size



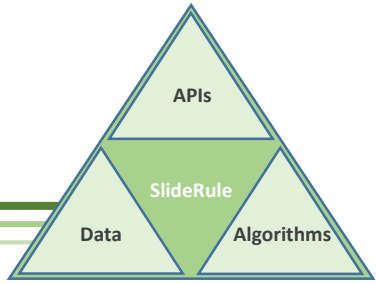
						Size	Elements	Chunks		Duration	
Dataset	Type	Storage Layout	Compression	Filters	Chunk Size	Total (bytes)	Total	Count	Average Size (bytes)	Average per Chunk (secs)	Average per Byte (secs)
/ancillary_data/atlas_sdp_gps_epoch	64-bit float	Compact	None	None	1	8	1	1	8	0.071	0.008875
/orbit_info/sc_orient	8-bit integer	Chunked	None	None	16	16	1	1	16	0.250	0.015625
/ancillary_data/start_rgt	32-bit integer	Compact	None	None	1	4	1	1	4	0.150	0.037500
/ancillary_data/end_rgt	32-bit integer	Compact	None	None	1	4	1	1	4	0.120	0.030000
/ancillary_data/start_cycle	32-bit integer	Compact	None	None	1	4	1	1	4	0.150	0.037500
/ancillary_data/end_cycle	32-bit integer	Compact	None	None	1	4	1	1	4	0.150	0.037500
<spot>/geolocation/delta_time	64-bit float	Chunked	GZIP: 6	None	143000	2,685,617	730,524	6	447,603	0.115	0.000009
<spot>/geolocation/segment_ph_cnt	32-bit integer	Chunked	GZIP: 6	Shuffle: 4	143000	600,951	730,524	6	100,159	0.127	0.000046
<spot>/geolocation/segment_id	32-bit integer	Chunked	GZIP: 6	Shuffle: 4	143000	38,586	730,524	6	6,431	0.098	0.000549
<spot>/geolocation/segment_dist_x	64-bit float	Chunked	GZIP: 6	None	143000	4,361,058	730,524	6	726,843	0.096	0.000005
<spot>/geolocation/reference_photon_lat	64-bit float	Chunked	GZIP: 6	None	143000	4,310,116	730,524	6	718,353	0.092	0.000005
<spot>/geolocation/reference_photon_lor	64-bit float	Chunked	GZIP: 6	None	143000	4,351,244	730,524	6	725,207	0.100	0.000005
<spot>/heights/dist_ph_along	32-bit integer	Chunked	GZIP: 6	None	143000	350,051,255	117,212,160	820	426,892	0.144	0.001662
<spot>/heights/h_ph	32-bit integer	Chunked	GZIP: 6	None	143000	305,855,771	117,212,160	820	372,995	0.146	0.001931
<spot>/bckgrd_atlas/delta_time	64-bit float	Chunked	GZIP: 6	None	143000	1,281,096	411,830	3	427,032	0.172	0.000007
<spot>/bckgrd_atlas/bckgrd_rate	32-bit integer	Chunked	GZIP: 6	None	143000	1,358,123	411,830	3	452,708	0.147	0.000006
						674,893,857	**239,631,130**	**1,688**	**275,266**	**0.133**	**0.171224**

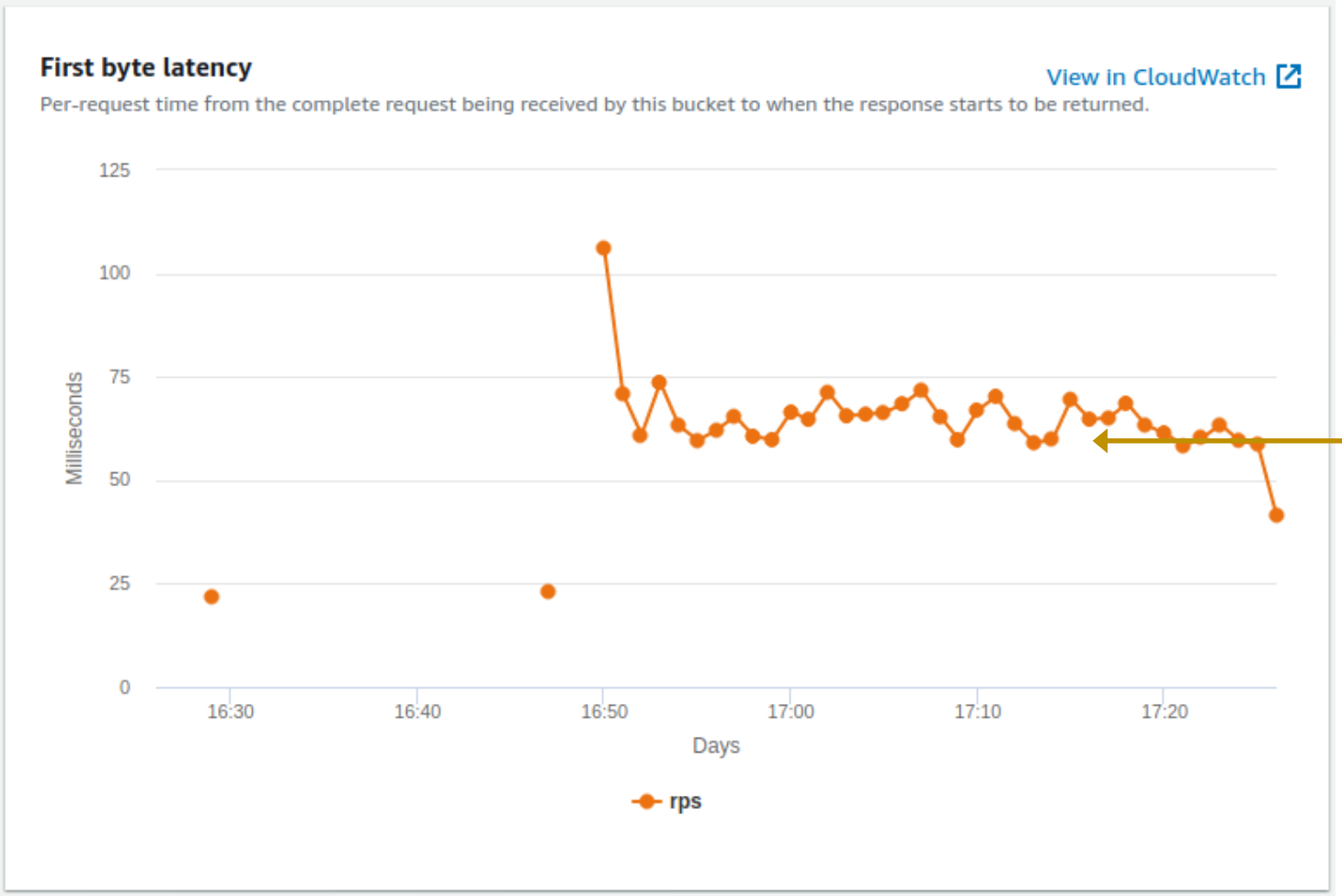




By reducing the number of chunks per dataset by 14x there are no longer any outliers driving the performance.

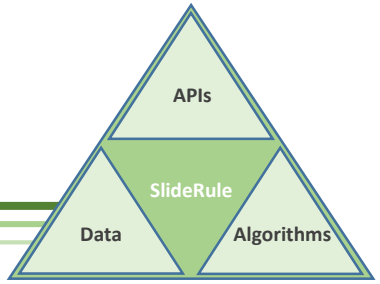
First Byte Latency Measurements





AWS is showing ~60ms of latency per request to S3; but internal HSDS logs show closer to ~50ms of latency.

Challenges in Chunk Size Selection

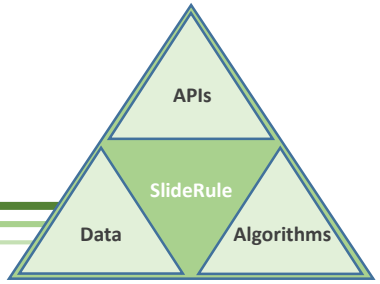


- **Optimizing for a given spatial extent is not possible** because the ICESat-2 data is dynamic – 10MB of photons in one part of the file may represent 1Km and in another part of the file may represent 10Km.

- **Optimizing for request size is not possible** because we are an on-demand data processing system. One user may be looking at a narrow spatial region that benefits from smaller chunk sizes, while another user may be sweeping large areas looking for features in the data.

- **We attempted to use the "break-even" size**, which is when the amount of time it takes to read the data matches the request latency penalty (e.g. if your data throughput is 1Gbps and latency is 50ms per request, the chunk size is the amount of data that can be read at 1Gbps over 50ms → 50Mbits). When a chunk size matches this size, then the worse case penalty paid for a request is 2x.
  - How do you determine what the expected latency and throughput are?
  - AWS S3 and lower-tier EC2 instance can experience large variations in network performance.

Obstacle 3: Metadata Repository

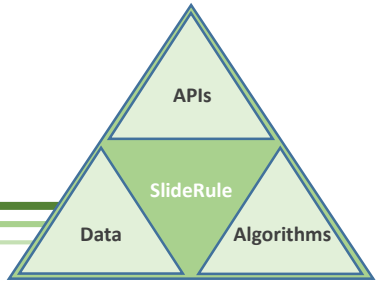


- A metadata repository is needed to hold pointers into the original H5 files which HSDS uses to know how to read the various datasets in the file.

- Before any H5 file in S3 can be read by our system, it must first be *loaded* through an HSDS pipeline to build and store the metadata for it. A typical region of interest consisting of 60 granules can take ~8 hours to load using a single c5.xlarge EC2 instance.

- The metadata repository must be maintained in order to use HSDS. A gap in funding which could result in the loss of the S3 bucket would required the entire repository to be rebuilt.

The Need for a Data Pipeline



- Building, maintaining, and running a data pipeline is expensive and time consuming:
  - read ICESat-2 data hosted by NASA
  - re-chunk it to optimize its chunk sizes for S3 access
  - upload it to our own S3 bucket
  - load it into HSDS to build and store the metadata HSDS needed

- With tooling we had at the time, we were able to fully load about 5GB of data per hour per EC2 instance.

- Assuming ICESat-2 produces 150TB of data each year, this would require us to fully automate, maintain, and continuously run a pipeline consisting of four EC2 instances 365 days a year, just to keep up with the new data ICESat-2 produces; that doesn't take into account historical data backlogs and re-releases due to version updates.