Rubin Observatory

Report of Google Cloud Proof of Concept 2020

Hsin-Fang Chiang and Kian-Tat Lim

2020-08-27

1 Introduction

In spring 2020, we started the second Proof of Concept (PoC) engagement with the Google Cloud team; see DMTN-150 for the plans. Section 2 reports on the batch Data Release Production (DRP) processing on Google Cloud Platform (GCP) and its cost. Section 3 reports on data transfer testing between Chile and Google Cloud Storage (GCS).

2 DRP Processing

2.1 System Architecture and Technology Stack

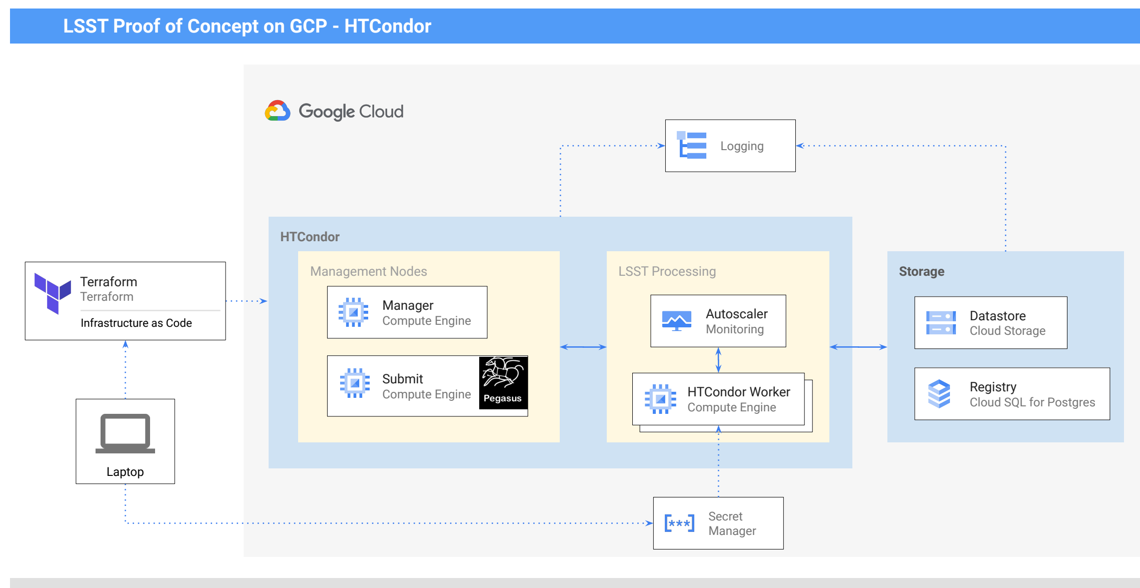

FIGURE 1: Architecture diagram of the overall system

The overall system architecture is shown in Figure 1. All components are hosted on GCP

except Terraform, which is infrastructure as code running from a laptop. The LSST Science Pipelines Software Stack contains the applications for image processing and analysis. A machine image based on a recent stack release together with other necessary software is built using Packer scripts. The LSST Generation 3 Middleware, including Data Butler and PipelineTask framework, is used for this PoC. The Butler datastore is a S3 compliant storage on GCS; objects are accessed using the `boto3` library. The Butler registry is a Google Cloud SQL PostgreSQL database. The usage of Data Butler is similar to the AWS PoC [DMTN-137], but the stack version is new. Credentials of the GCS bucket and the PostgreSQL instance are stored in Google Secret Manager. In the startup process of each machine, the credentials are retrieved and stored in its local environments, which are used to access the GCS bucket and the PostgreSQL instance.

Google Compute Engine (GCE) deployed via Terraform provides the compute resources. HTCondor is the underlying workload management system for batch jobs. Pegasus is used on top of HTCondor for workflow submission and monitoring. The HTCondor pool is automatically scaled with the load via a Managed Instance Group. The HTCondor execute machines, or workers, can be either on-demand VMs or Preemptible VMs. HTCondor Annex is an alternative to acquire cloud compute resources, but we did not use HTCondor Annex yet in this PoC due to time constraints (Section 2.4). The stackdriver-based Cloud Monitoring and the fluentd-based Cloud Logging are turned on in the HTCondor cluster. Pipeline logs are also sent to Cloud Logging, so we can search and examine logs in the Logs Viewer for troubleshooting.

2.2 Execution Results

2.2.1 Test Runs

Similar to DMTN-137, we first executed the `ci_hsc` workflow, scaled up to process one tract HSC-RC2 dataset (DM-11345), and then the full HSC-RC2 dataset. So far, execution involved some manual intervention steps and has not been fully automated yet. The overall steps were: Butler repo setup, QuantumGraph generation, workflow generation, and job execution. To create a Butler repo on GCP after creating the GCS bucket and the PostgresSQL database, the bootstrap script at `https://github.com/lsst-dm/gen3-hsc-rc2` was run from the Verification Cluster located at NCSA to transfer the data to the GCS bucket and populate the Butler repo. We did not optimize this step and it took around 40 hours for the full HSC-RC2 repo. It can

Task	Count	Mean Runtime (sec)
Init	1	27.0
IsrTask	6765	58.6
CharacterizeImageTask	6765	169.9
CalibrateTask	6765	85.7
MakeWarpTask	4206	68.6
CompareWarpAssembleCoaddTask	405	424.9
DetectCoaddSourcesTask	405	88.6
MergeDetectionsTask	81	152.6
DeblendCoaddSourcesSingleTask	405	276.1
MeasureMergedCoaddSourcesTask	405	2541.4
MergeMeasurementsTask	81	40.0
ForcedPhotCoaddTask	405	3780.1
Workflow wall time		8 hrs 52 mins
Cumulative job wall time		61 days 5 hrs

TABLE 1: Task breakdown of the HSC-RC2 tract=9615 workflow for the 20200729T200516 run. There were 26688 pipeline jobs in total. The mean runtime was the time spent on the resource as seen by Condor DAGMan. The minimum runtime of the Pipetasks was 7.5 sec and the maximum was 6784.9 sec. The software stack was `w_2020_30`.

be improved by copying the data to the GCS first and creating the Butler repo from GCE. We may also parallelize the ingestion process in the future.

In the Generation 3 Middleware, each executable unit is represented by a Quantum, and the DRP workflow is represented as a Quantum Graph with Quanta interdependency. For the execution workflow, we added one initialization job and translated Quanta into jobs in the Pegasus format with one-to-one mapping. For simplicity we considered all jobs of `MakeWarpTask`, `CompareWarpAssembleCoaddTask`, `DeblendCoaddSourcesSingleTask`, and `MeasureMergedCoaddSourcesTask` as large-memory jobs and required 30GB of memory; all other jobs required 2 GB. Pegasus also added other necessary jobs to the execution workflow, such as data transfer of Quantum files and log files. We used the submit node as the staging site, in the same manner as the AWS PoC [DMTN-137].

With the LSST software stack version `w_2020_30`, the tract `tract=9615` contains 26688 PipelineTask Quanta. Table 1 shows an example run; this is comparable to Table 1 in DMTN-137 and most differences were likely resulted from the LSST software stack version differences.

The full HSC-RC2 dataset contains 3 tracts and around 1.5TB of input data, including 767 GB of raw images. Currently we ignore narrow bands, so there are 404 visits in total. One full

Draft

Task	Total Count	Count1	Count2	Mean Runtime (sec)
Init		1	1	80.0
IsrTask	30278	30278	1	55.0
CharacterizeImageTask	30278	30277	1	129.2
CalibrateTask	30278	30277	1	74.7
MakeWarpTask	20182	20180	2	74.0
CompareWarpAssembleCoaddTask	1180	1178	2	700.3
DetectCoaddSourcesTask	1180	1178	2	105.2
MergeDetectionsTask	236	234	2	143.5
DeblendCoaddSourcesSingleTask	1180	1170	10	651.3
MeasureMergedCoaddSourcesTask	1180	1170	10	4426.7
MergeMeasurementsTask	236	234	2	40.7
ForcedPhotCoaddTask	1180	1170	10	6327.9

TABLE 2: Example task breakdown of the full HSC-RC2 workflow. There were 117388 pipeline Quanta in total. The mean runtime is the time spent on the resource as seen by Condor DAGMan in the 20200804T005441 + 20200806T041934 run using on-demand `n1-standard-8` machines. Count 1 is the count from the first submission 20200804T005441; Count 2 is the count from the second, rescue submission 20200806T041934. One IsrTask failed in the first submission and was retried in the second submission. The software stack was `w_2020_30`.

HSC-RC2 workflow generates around 10TB of output data. Using the stack version `w_2020_30`, the QuantumGraph generation took around 13 hours for the full HSC-RC2 dataset, resulted in 117388 Quanta in total in our test workflow. We excluded patches that were not covered in all filters; this was a workaround as `CompareWarpAssembleCoaddTask` did not write out empty images which were expected in the QuantumGraph workflow. Scripts and pipeline configurations can be found in `https://github.com/lsst-dm/google-poc-2020/tree/master/drp`. When the workflow did not finish fully in one submission, we made a rescue workflow to run in another submission. The rescue workflow only contained failed jobs or their dependents. Making the rescue workflow required a manual step using a temporary patch in the `pipe_base` package (DM-25809), and took around 12 hours on one CPU. Table 2 shows an example for the full HSC-RC2 workflow.

For the computes, we tested with GCP's first generation general-purpose machine types (N1) as well as the second generation general-purpose machine types (N2). N2 provides better price-performanance and is used in the IDF quotes. More specifically, the N1 machine `n1-standard-8` with 8 vCPUs and 30 GB of memory, and the N2 machine `n2-standard-8` with 8 vCPUs and 32 GB of memory were used.

The same HSC-RC2 workflow was run with multiple setups:

Run	Setup	Max CPUs	Workflow walltime	Cumulative job time	Compute cost
20200731T011149+20200802T164424	on-demand N1 workers	800	55 hrs	308 days	$1592
20200804T005441+20200806T041934	preemptible N1 workers	1600	36 hrs	275 days, 22 hrs	$459
20200806T215620+20200808T170615	preemptible N2 workers	1600	25 hrs	213 days, 10 hrs	$390
20200811T172329	on-demand N2 workers	800	31 hrs	300 days, 2 hrs	$1191
20200814T002816+20200816T060623	preemptible N2 workers	1600	29 hrs	208 days, 22 hrs	$388

TABLE 3: Run summary of the HSC-RC2 workflow with different setups. The workflow wall time only includes Pegasus records, and may be dominated by random job failures and the rescue graph. The costs are overestimates. The Postgres instance size was increased between the 20200802T164424 run and the 20200804T005441 run; it stayed the same afterwards.

1. Using on-demand N1 VMs as workers. Maximum 800 CPUs (100 VMs) simultaneously.
2. Using on-demand N2 VMs as workers. Maximum 800 CPUs (100 VMs) simultaneously.
3. Using preemptible N1 VMs as workers. Maximum 1600 CPUs (200 VMs) simultaneously.
4. Using preemptible N2 VMs as workers. Maximum 1600 CPUs (200 VMs) simultaneously.

Table 3 summarizes the compute time of the runs. Non-preemptible instances were used as HTCondor master and submit nodes.

2.2.2 Experienced Errors

1. **GCS 403 HTTP error**. Since we started large scale testing in early July, random 403 errors were received when Butler made HeadObject calls:

   ```
   botocore.exceptions.ClientError: An error occurred (403) when calling the HeadObject operation: Forbidden
   ```

   The stack interpreted it as

   ```
   PermissionError: Forbidden HEAD operation error occurred. Verify s3:ListBucket and s3:GetObject permissions are granted
       ↪ for your IAM user.
   ```

   However, the permission setup was the same for all jobs but only a very small fraction of jobs encountered this. Butler did a HeadObject to check if an object existed before attempting to upload. If the object did not exist it expected a 404 error. At the time we used the `v20` stack and did not retry failed HTTP requests. (A request-level retry using the `backoff` library was added by Dom Zippilli afterwards.) A Google Cloud Support Ticket was filed to investigate the root cause, and concluded that there were transient issues at the IAM server during the same timeframe. The instability of the IAM server was fixed on July 22. We have not seen this error since.

2. **Butler IntegrityError** Conflicts of datasets were found occasionally when pipelines attempted to write outputs, for example:

Draft

```
  File "/opt/lsst/software/stack/conda/miniconda3-py37_4.8.2/envs/lsst-scipipe-1a1d771/lib/python3.7/site-packages/
      ↪ sqlalchemy/engine/default.py", line 590, in do_execute
    cursor.execute(statement, parameters)
psycopg2.errors.UniqueViolation: duplicate key value violates unique constraint "
      ↪ dataset_collection_24cc_unq_dataset_type_id_collection_c55de2f4"
DETAIL:  Key (dataset_type_id, collection_id, instrument, detector, visit)=(119, 60, HSC, 88, 1308) already exists.

  File "/opt/lsst/software/stack/conda/miniconda3-py37_4.8.2/envs/lsst-scipipe-1a1d771/lib/python3.7/site-packages/
      ↪ sqlalchemy/engine/default.py", line 590, in do_execute
    cursor.execute(statement, parameters)
sqlalchemy.exc.IntegrityError: (psycopg2.errors.UniqueViolation) duplicate key value violates unique constraint "
      ↪ dataset_collection_24cc_unq_dataset_type_id_collection_c55de2f4"
DETAIL:  Key (dataset_type_id, collection_id, instrument, detector, visit)=(119, 60, HSC, 88, 1308) already exists.


lsst.daf.butler.registry._exceptions.ConflictingDefinitionError: A database constraint failure was triggered by inserting
      ↪ one or more datasets of type DatasetType(src, {abstract_filter, instrument, detector, physical_filter,
      ↪ visit_system, visit}, SourceCatalog) into collection 'hfc18'. This probably means a dataset with the same data ID
      ↪ and dataset type already exists, but it may also mean a dimension row is missing.
```

In the scenario that a pipeline job was killed after partial outputs were written due to a machine shutdown, the follow-up process would redo the job and attempt to write all outputs. Butler did not allow duplicate datasets nor overwriting existing datasets, hence a conflict error occurred. This was expected and understood on preemptible machines; DM-26131 discusses how to handle preemption notice with a possible shutdown script and complete cleanup actions before the instance stops. However, machine reboot can happen to regular machines too and we encountered the same error occasionally. Closer investigation revealed that shutdown of regular instances due to maintenance events on the host machines can be mitigated with the live migrate option. Since we started using the live migrate option we have not seen the issue. Furthermore, DM-25818 removed the file existence check and allowed overwrite the stored file; the ticket was merged after the `w_2020_30` release.

3. **Database size** When we scaled up the number of workers without a sufficiently large Postgres instance, we encountered database operational errors such as:

```
sqlalchemy.exc.OperationalError: (psycopg2.OperationalError) FATAL:  remaining connection slots are reserved for non-
      ↪ replication superuser connections
```

and

```
Failed to build graph: (psycopg2.errors.ConfigurationLimitExceeded) temporary file size exceeds temp_file_limit (1025563kB
      ↪ )
```

As the number of simultaneous connections increased, it was expected that the Postgres instance needed to scale up. These errors disappeared once the machine CPUs, memory, or the database flags such as `max_connections` or `temp_file_limit` of the database instance were increased.

4. **Other database issues** There were more kinds of database operational errors, such as:

```
  File "/opt/lsst/software/stack/conda/miniconda3-py37_4.8.2/envs/lsst-scipipe-1a1d771/lib/python3.7/site-packages/
      ↪ sqlalchemy/engine/default.py", line 590, in do_execute
    cursor.execute(statement, parameters)
psycopg2.OperationalError: server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
```

```
sqlalchemy.exc.OperationalError: (psycopg2.OperationalError) could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/var/run/cloud-sql-proxy/light-team-275220:us-central1:drp-rc2-w30/.s.PGSQL
            ↪ .5432"?
```

These errors appeared to reduce with a larger database size per maximum simultaneous processes, but did not disappear completely. The source of the error remains unclear. In the current stack, Butler expects a long-lived database connection during pipeline execution and the connection pooling is disabled which might lead to unanticipated connection churns (DM-26302). We also see errors from the rate limit of the Cloud SQL Admin API calls. When worker instances are added to the HTCondor cluster by the instance group autoscaling, there can be a surge of such Cloud SQL Admin API calls as the Cloud SQL Proxy starts in each instance. It could be a quota limit or an issue in the code or configuration of the Google Cloud SQL Proxy, or something not yet considered. We intend to investigate the source of the issue further.

5. **Other rare transient errors** For example, we occasionally saw

```
urllib3.exceptions.ProtocolError: ('Connection aborted.', OSError(0, 'Error'))
botocore.exceptions.ConnectionClosedError: Connection was closed before we received a valid response
```

It was very rare and working as expected.

2.3 Cost Analysis

On GCP, the daily billing data can be exported to a BigQuery dataset and analysis can be done through SQL queries.

2.3.1 GCS

The GCS charge is dominated by the data storage cost, which is $0.02 per GB per month for standard storage. Storing 10 TB of the HSC-RC2 outputs costs $200 per month. As we operate

completely within the GCP in the same region, there is no egress charge. If we transferred 10 TB out of Google Cloud, it would cost around $1110 in the normal rate; but there is a special offer to Internet2 Higher Education members and the National Research and Education Network (GÉANT NRENs) members, and data egress fees may be waived up to a maximum discount of 15% of the total monthly Google Cloud fees. Besides storage, for each run of HSC-RC2 workflow, the operations usage was typically less than $4.

2.3.2 Cloud SQL

A PostgreSQL instance with 10 vCPUs, 65 GB of memory, and 15 GB of SSD storage was used and cost around $23 per day while we ran the tests. Storage capacity can be incrementally increased as needed.

2.3.3 Compute

Table 3 summarizes the compute cost of running the HSC-RC2 workflow in different setups. The workflow wall time was likely dominated by non-deterministic factors such as which jobs happened to fail and the size of the rescue workflow, therefore not very meaningful currently. Instances were also underutilized during to the manual part of the workflow. For example, the HTCondor cluster was left idle while the rescue workflow was made. We have not optimized the overall usage. Based on our limited tests, using N2 gave some performance boost with lower cost. Each run using preemptible N2 workers cost about $390.

2.3.4 Cost Comparison

We use the full run of the HSC-RC2 dataset with preemptible N2 workers to extrapolate the cost. For each processing run through the DRP workflow, the total cost was roughly $390 (GCP) + $46 (Cloud SQL for 2 days) + $17 (GCS) = $453. Note that it included all 3 tracts of the HSC-RC2 dataset, which was 4.5 times larger than the single-tract run used for cost analysis in DMTN-137 with regard to the input raw CCD images. With the dataset size in consideration, the run on GCP was roughly 6% more expensive than the estimated cost in DMTN-137. However, factors such as the database size, machine setups, manual intervention, and the rescue workflow can easily cause more than 10% of differences in cost. For example, in the AWS PoC we did not have a rescue workflow and only used the runs without failures. Less clock time implied less cost from the database. If we only considered 1 day of Cloud SQL usage, the cost

would be about the same as the estimated cost on AWS. In other words, the cost differences on GCP and on AWS were not significant in our test runs if we only look at list pricing.

2.4 Future Work

As the HSC-RC2 dataset is much smaller than the original target size, we will continue to scale up the data volume.

Other possible improvements include:

1. Use HTCondor Annex.
2. Use high-memory machine types.
3. Add native GCS Butler datastore rather than using its S3 interface.
4. Move the staging site. In this PoC we used the submit node as the staging site. Bektesevic et al. (2020) found that moving the staging to S3 can lead to better performance and reduce cost by 40-60 percent.
5. Improve failure recovery and eventually use retry features from the workflow manager. Newer version of the stack has some overwriting features.
6. Supply a shutdown script to handle the machine preemption (DM-26131). This might mean a pipetask feature to clean up the unfinished work.
7. Improve efficiency of the scripts and automate the manual parts of the execution.
8. Fully integrate pipeline logs into Google Logging.

2.5 Conclusions

Using a similar system architecture from the AWS PoC [DMTN-137], we were able to conduct DRP test runs with a recent DM stack on GCP. We utilized HTCondor and multiple Google Cloud products, including Compute Engine, Cloud Storage, Cloud SQL, and Operations Suite. The number of simultaneous jobs peaked at 1600, the largest we have tested in the cloud environments so far. Based on our test runs, the estimated cost on GCP is about the same as the estimate on AWS. Note that this report is not intended to cover commercial discounts and incentives which might change the pricing conclusion significantly.

3 Data Transfer

The goal of this part of the proof of concept was to demonstrate operational-level transfers of pixel data from hosts in Chile to Google Cloud. In particular, we wanted to show that high bandwidths could be achieved with parallel transfers across multiple machines as well as low latencies.

Previously [DMTN-125], images had been transferred similarly via a shared 10 Gbit/sec connection to the public Internet using the `gsutil` tool. A net transfer bandwidth of 1.5 Gbit/sec was achieved, but long startup times of over 5 sec made it infeasible to use this same mechanism for production.

This proof of concept attempted to do better in two ways: first, by using the Google Cloud Storage API directly to better control the upload process and reuse existing TCP connections and, second, by configuring a dedicated private interconnection between the Rubin Observatory private long-haul network and the Google Cloud network, avoiding the public Internet.

3.1 Experimental Setup

3.1.1 Software

A test harness script was written in Python. This script does the following:

- Parses command line arguments
- Uses `fork` to spawn one or more child processes, each of which independently execute the steps below
- Configures an uploader instance to connect to a storage destination
- At configured intervals after a given start time:
  - Copies a representative FITS file from disk to `/tmp`
  - Compresses the FITS file in `/tmp` using `fpack`
  - Uploads the resulting `.fz` file to the storage destination
- Waits for all child processes to terminate

Draft

- Enters an infinite loop awaiting manual termination

This software is available from `https://github.com/lsst-dm/google-poc-2020/tree/master/apxfr`.

A comparison was done between various compression options. Uncompressed, the data to be transferred is 75,582,720 bytes. `gzip -1` takes 1.45 sec (mean of 5 measurements) to compress the data to 22,622,249 bytes. Copying the data and running `fpack` using the default tiled Rice compression takes 0.35 sec (mean of 5 measurements) to compress the data to 15,137,280 bytes. `fpack` compression was thus used for all transfers.

3.1.2 Network Configuration

As prior tests had found that using a shared outbound network from the La Serena data center as well as traversing the public Internet to reach Google Cloud services led to significant variation in data transfer times, for this proof of concept a private interconnect was established between the Rubin Observatory long-haul network and the Google Cloud network. The two networks shared a common point of presence at the Equinix colocation facility in Miami, so it was relatively easy to arrange and configure a 10 Gbit/sec connection there. We gratefully acknowledge significant assistance from Google personnel in encouraging Equinix to create the interconnect in a short timeframe and giving clear directions for configuration.

The network links used for the tests therefore comprised:

- Local area network switches in the Base data center (10 Gbit/sec per machine)
- Main router in the Base data center
- Rubin long-haul network segments from La Serena to Miami (100 Gbit/sec)
- Dedicated interconnect in Miami (10 Gbit/sec)
- Google Cloud internal network from Miami to us-central1 region in Iowa

A Cloud Router and VLAN were configured in the Google Cloud to advertise the Google VPC network for the proof of concept project over the interconnect via BGP. A range of private IP addresses for the Google Cloud Storage API endpoint was configured into the Cloud Router as

well so that routing to those addresses was guaranteed to traverse the desired links. Those private IP addresses were configured into container or machine `/etc/hosts` files so that the endpoint domain name (`storage.googleapis.com`) did not need to be changed in the client software.

Note that the maximum transmission unit (MTU) size for the dedicated interconnect link is limited to 1440 bytes by the Google router. Jumbo frames were not available on this link.

The measured (via `ping`) time to the Miami exit router is 158 milliseconds. The measured (via `ping`) time to a virtual machine on the Cloud VPC network was 194 milliseconds. The measured (via Google's `perfdiag` tool) connection time to Google's storage server for this network is 200 milliseconds.

3.1.3 TCP Configuration

The machines used for testing were configured to maximize TCP throughput given the very high bandwidth-delay product (10 Gbit/sec $\times$ 200 millisec $=$ 250 MB) for the network. In particular, the following `sysctl` variables were set:

- `net.core.rmem_max = 536870912`
- `net.core.wmem_max = 536870912`
- `net.ipv4.no_pmtu_disc = 0`
- `net.ipv4.tcp_adv_win_scale = 1`
- `net.ipv4.tcp_congestion_control = htcp`
- `net.ipv4.tcp_rmem = 4096 87380 536870912`
- `net.ipv4.tcp_slow_start_after_idle = 0`
- `net.ipv4.tcp_wmem = 4096 87380 536870912`

Path MTU discovery is required because the machines and LAN are otherwise configured for jumbo frames.

Window scaling is necessary to use large windows to avoid waiting for acknowledgements; the increased `rmem` and `wmem` values set the maximum window size to be very large. An attempt to find an appropriate window size was attempted using the `iperf3` tool, but all specified window sizes seemed to perform worse than the default, so no explicit tuning was done here.

H-TCP is an optimized congestion control algorithm for high speed networks with high latency.

Disabling TCP slow start after idle allows a single connection to maintain maximum throughput even for bursty transmissions. An attempt was made to use TCP keepalive options at the application level to maximize performance over a reused connection, but setting the `HTTPConnection.default_socket_options` did not seem to have any useful effect, so this was abandoned in favor of setting the `net.ipv4.tcp_slow_start_after_idle = 0` parameter mentioned above. This reduced transfer times from about 2.5 seconds to about 1 second.

3.1.4 Kubernetes and Bare Metal

The transfer software harness was intended to be run from a Docker container deployed under the management of Kubernetes. Scaling the deployment would be simple, with a variety of options for constraining node placement and avoiding disruption to other services. Logs could be collected from the cluster for analysis.

When this was found to cause excessive overhead, testing reverted to running on the ”bare metal” nodes directly from a local shell. (Note that these were the same nodes in the Kubernetes cluster, just accessed differently.)

3.2 Procedure

The harness script was run with options as follows:

- `--destination gsapi://bucketname/path`
- `--numexp 5 or 10` (for statistics and to see if performance improved with the number of transfers)
- `--interval 17` (to simulate the interval between exposures for ComCam and LSSTCam)
- `--compress` (to use `fpack` as explained above)

Draft

- `--private` (when running inside a container; not needed on bare metal)
- `--ccds varying`
- `--starttime varying`

A start time a few minutes in the future was selected for each run, whether on a single node, multiple nodes, or in Kubernetes, to ensure that all harness processes had started and were synchronized for the first transfer.

The log messages from the script were captured, either via redirection of `stderr` or by a `fluentd`/ElasticSearch log capture mechanism. These were searched for the string "`End transfer`" and the length in seconds of the transfer was extracted from those lines. Note that the (constant) compression time is not included in these measurements.

Since latency to the delivery of the final pixel is the key performance metric in terms of the delivery of Alerts, the maximum transfer time is highlighted, along with a simple conversion to average bandwidth in bytes/sec by dividing this into the total bytes transferred. Minimum and mean transfer times are also reported to give a sense of the distribution.

It was determined that the first of a sequence of transfers took substantially longer than the subsequent ones. Initially, this was attributed to authentication overhead. Downloading a file during initialization, before the first transfer, to "prime" the authentication did speed things up a bit, but initial connections were still substantially slower (approximately 3 seconds). It is likely that this can be ameliorated by doing an *upload* before the first measured transfer, but all analysis instead simply ignored the first transfer.

Attempts were made to run tests while the networks and nodes were otherwise unused and quiescent, but it was not possible to rule out interference from local network traffic in the La Serena switches and router. As a result, it is not necessarily possible to compare timings from one set of tests to another, but it is believed that comparisons within a set of tests are meaningful.

In addition, there have been reports of sporadic packet loss on the Rubin long-haul network (IT-2362). These could be the cause of unusually long transfers, so occasional outliers were removed from timings as described below.

3.3 Results

3.3.1 Interconnect

It was determined that the private interconnect performed substantially better than the public Internet, as expected. Not only were timings better, the variation in timings was also dramatically lessened.

The following measurements were for a single CCD transferred from a single node, reusing the TCP connection without slow start:

Connection	Private	Public
Minimum	0.96	4.07
Mean	0.98	4.46
Maximum	1.01	5.02
Bandwidth (Mbit/sec)	120	24

Draft

3.3.2 API

A comparison between the `gsutil` utility (version 4.53) and the Python Google Cloud Storage API (version 1.28.1) was performed. Invoking a separate `gsutil` process for each transfer, having to perform authentication each time, and being unable to reuse the connection proved to be much slower, as expected. Using the `gsutil cp -I` option to specify a list of URLs on `stdin` would allow connection reuse, but it forces the pathnames to be the same in the source and destination, which is inconvenient for working with the Data Butler. Since the same performance could be obtained with the Python API, along with more flexibility, use of `gsutil` was abandoned, and the Python API was used for all tests to Google Cloud Storage.

Tests were also done using the standard `curl` utility. Upload using `curl` to issue an HTTP POST request took about the same amount of time (3.18 sec) as initial connections via the API. Note that the authentication header was preloaded with a known token in the `curl` case. While `curl` cannot easily be used to simulate a held-open connection with multiple files sent at intervals, it was tested in a mode where two copies of the (compressed) file were sent back-to-back. This took 4.04 sec, implying 0.86 sec for the second transfer. It is not unexpected that this might be one round-trip-time faster than the standard case with a long interval between transfers.

The 1.36 second maximum time for copying, compressing, and sending a CCD directly to Google Cloud Storage via the Python API is well within the budget of 5 seconds allocated for transfer from the DAQ to distributors at the US Data Facility. As a result, no further investigations of custom tooling for the transfer were needed.

3.3.3 Scaling to Multiple CCDs

A sequence of runs were done on a single node using multiple processes to transfer one CCD per process.

CCDs	4	8	16	20	24
Minimum	0.78	0.77	0.76	0.78	0.77
Mean	0.96	0.93	0.95	1.00	1.02
Maximum	1.12	1.21	1.53	1.54	1.93
Bandwidth (Mbit/sec)	432	799	1270	1570	1500

Since the maximum transfer time started to increase and the aggregate bandwidth started to decrease, no values above 24 were tested in this set.

Outliers of 3.44 and 7.80 seconds, respectively, were removed from the 8 and 16 CCD tests.

3.3.4 Scaling to Multiple Nodes

Nodes	1	1	1	1	2	3
CCDs/Node	1	10	20	30	10	10
Minimum	0.92	0.77	0.78	0.75	1.01	0.83
Mean	0.99	1.04	1.27	0.98	2.48	3.71
Maximum	1.13	1.97	1.76	1.37	5.20	15.94
Bandwidth (Mbit/sec)	107	430	1000	2660	335	228

Outliers of 2.82, 2.41, and 7.22 seconds, respectively were removed from the 1/10, 1/20, and 2/10 tests.

It is clear that the net bandwidth drops off drastically as more nodes are added. One possible explanation is that congestion is occurring in the local network. It is also clear that even the single-node timings, while broadly consistent within sets, have unexplained anomalies, but

Draft

nevertheless they are well within the budgeted time window.

3.4 Future Work

The problem with using multiple transmitting nodes needs to be resolved. Potential congestion and/or packet loss in the switches and routers will be investigated.

Comparisons with transmission to an object store at NCSA, the terminus of the Rubin long-haul network, will be performed.

The Google Cloud destination VPC network and the storage bucket are located in Iowa, which is approximately 30 milliseconds further in network delay than South Carolina, the closest data center to Miami. Testing with that location may show modest speed improvements.

3.5 Conclusions

The dedicated network and private interconnect have definitely improved the speed and consistency of transfers. The timings from one node, even for as many as 20-30 CCDs, are well within the necessary budget. But the inability to use multiple hosts to transfer data limits the scalability of this solution. This issue would need to be resolved prior to production.

A References

Bektesevic, D., Connolly, A., Chiang, H.F., et al., 2020, In: Proceedings of Gateways 2020

[DMTN-137], Chiang, H.F., Bektesevic, D., the AWS-PoC team, 2020, *AWS Proof of Concept Project Report*, DMTN-137, URL http://DMTN-137.lsst.io

[DMTN-125], Lim, K.T., 2019, *Google Cloud Engagement Results*, DMTN-125, URL http://dmtn-125.lsst.io

[DMTN-150], Lim, K.T., 2020, *LSST + Google Cloud Proof of Concept*, DMTN-150, URL http://DMTN-150.lsst.io

B Acronyms

Acronym	Description
API	Application Programming Interface
AWS	Amazon Web Services
BGP	Border Gateway Protocol
CCD	Charge-Coupled Device
CPU	Central Processing Unit
ComCam	The commissioning camera is a single-raft, 9-CCD camera that will be installed in LSST during commissioning, before the final camera is ready.
DAQ	Data Acquisition System
DM	Data Management
DMTN	DM Technical Note
DRP	Data Release Production
FITS	Flexible Image Transport System
GB	Gigabyte
GCE	Google Cloud Engine
GCP	Google Cloud Platform
GCS	Generic Control System
HSC	Hyper Suprime-Cam
HTTP	HyperText Transfer Protocol
IAM	Identity Access Management
IDF	Interim Data Facility
IP	Internet Protocol
IT	Information Technology
LAN	Local Area Network
LSST	Legacy Survey of Space and Time (formerly Large Synoptic Survey Telescope)
MB	MegaByte
MTU	Maximum Transmission Unit
NCSA	National Center for Supercomputing Applications
S3	(Amazon) Simple Storage Service
SQL	Structured Query Language
SSD	Solid-State Disk

Draft

TB	TeraByte
TCP	Transmission Control Protocol
US	United States
VLAN	Virtual Local Area Network
VPC	Virtual Private Cloud
stdin	standard input

Draft