AWS Proof of Concept Project Report

Hsin-Fang Chiang, Dino Bektesevic,

2019-12-12

1 Background

In April 2019, LSST DM began a proof of concept (PoC) project with the Amazon Web Services (AWS) and HTCondor teams to explore whether a cloud deployment of the Data Release Production (DRP) is feasible. The execution plan is in DMTN-114. For this project AWS granted us credits to use their platform. The team met biweekly to discuss the progress and plans. In this document we report on the work carried out, lessons learned, as well as the results of scaling and pricing tests.

2 Technology stack

2.1 Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (EC2) service provides secure, resizable compute capacity in the cloud based on virtualization technologies. The unit of compute resources is an instance, and users can scale up and down the number of instances. EC2 instances come with multiple instance types for a configuration of memory, CPU, and storage. An Amazon Machine Image (AMI) bundles the Operating System (OS) with required data and configurations into a convenient image that can be launched on any EC2 instance type. Amazon Elastic File System (Amazon EFS) provides a POSIX-compliant, scalable, elastic NFS file system designed to provide parallel shared access to thousands of AWS compute instances. We have used On-demand and Spot instances in the AWS PoC. On demand instances cost more but they can not be deallocated by AWS. Spot instances are significantly cheaper, but at any time AWS, given a 2 mintue warning, can request the compute resources back and deallocate them.

2.2 Amazon Simple Storage Service (S3)

Simple Storage Service (S3) is an object storage that allows massive amounts of unstructured data where each object typically includes the data, related metadata and is identified by a globally unique identifier, to be stored and accessed from EC2 instances or elsewhere in a durable and highly scalable way. Buckets, organizational units in which related objects are generally stored, enable easier access and priviledges administration. Access, read, write, delete and other atomical units of action on the objects themselves can be allowed or forbidden at the account, bucket or individual object level. Logging is available for all actions on the Bucket level and/or at the individual object granularity. It is also possible to define and issue complex alert conditions on Bucket or object actions which can execute arbitrary actions or workflows, but S3 itself is not POSIX compliant.

2.3 PostgreSQL

PostgreSQL is one of the most popular open source relational database systems available. The choice to go with PostgreSQL was based on the fact that it's a very popular and well supported open source software that suffers from no additional licensing fees usually associated with proprietary software. Relational Database Service (RDS) is the AWS cloud service that launches and configures databases with ease. The RDS databases can be backed up into snapshots as well as exported to downloadable files on S3.

2.4 HTCondor

HTCondor(Thain et al., 2005) provides distributed job parallelization and is a powerful batch system for high throughput computing (HTC). A HTCondor pool is a collection of compute resources, and HTCondor matches job requests with available resources following their job requirements. The `condor_annex` module allows HTCondor deployment on cloud resources. Instances allocated by `condor_annex` are added to a shared resource pool on which jobs added in the job queue are scheduled on. Unused compute resources are automatically deallocated after some set time spent idling. HTCondor has been succesfully used by HEP Cloud project, scaling up to 60,000 cores in multi-region deployement on AWS, and by IceCube experiment, where they scaled up to 51,000 GPUs across multiple regions and multiple cloud providers (AWS, Azure, and Google Cloud).

2.5 Pegasus

Pegasus(Deelman et al., 2015) is a workflow management system built on top of HTCondor. Popular in large-scale scientific computing, it provides command line and API interfaces for scientists to write abstract workflow independent of the underlying computing infrastructure. Pegasus workflows are expressed as Directed Acyclic Graphs (DAGs) where jobs are the nodes and job dependencies are represented by the edges. Pegasus utilizes HTCondor DAGMan as its execution engine, but also supports various execution environments and data management strategies.

2.6 LSST Software Stack

The LSST Science Pipelines Software Stack is a collection of data processing codes for optical and near-infrared astronomy, and is used for DRP (Bosch et al., 2018). Besides image processing and astronomical catalog analysis, it includes a middleware layer for data access, task definition and execution. The LSST Data Butler is a framework that abstracts the underlying data persistence and implementations from Science Pipelines algorithms, and a data repository is a set of datasets under control of the Data Butler (Jenness et al., 2018). The Data Butler concepts can be found in LDM-463. Recently, the middleware layer is ungergoing a major refactoring, with the new middleware known as the Generation 3 Middleware ("Gen3" Middleware; DMTN-056). This AWS PoC added new features in the Generation 3 Data Butler; see Sect 5.

3 Architecture design

The system at the end of the PoC is shown in Figure 3. All components are hosted on the AWS platform.

AMIs containing LSST Software Stack, Pegasus, and configured HTCondor are pre-built in two flavors: master and worker. These contain the complete environment required to run DRP workflows and scale them out with HTCondor. The user launches an on-demand EC2 instance using the master AMI to be the submit host and the central manager of HTCondor pool. The user logs in to this master to control the workflow execution.

The workflow of the LSST DRP in Generation 3 Middleware is represented as a Directed Acyclic Graph (DAG) known as Quantum Graph. In the Generation 3 Middleware, input and output datasets are defined in the Pipeline Tasks and each unit is known as a Quantum. The Quantum Graph is aware of the Quantum dependency and determines the execution workflow. For simplicity we map each Quantum to a standalone job, despite this may not be optimal for

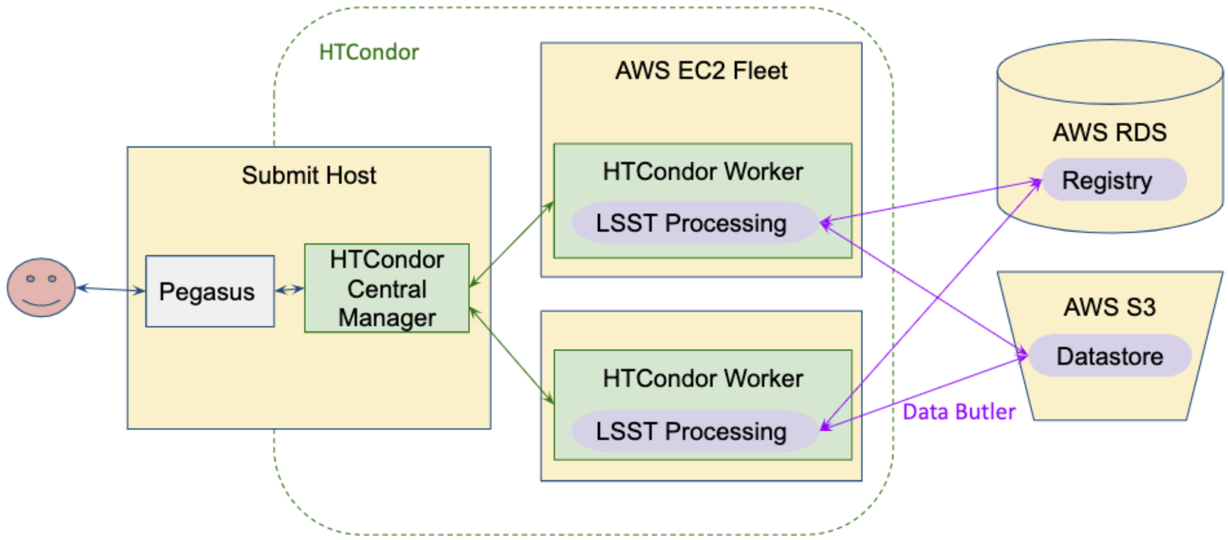

FIGURE 1: Diagram of the AWS PoC system architecture.

the performance. Pegasus is used to submit the workflow to HTCondor as well as monitor the workflow execution. Each LSST Pipeline Task Quantum is resource-independent. Pegasus adds other necessary jobs, such as data transfer, to the executable workflow. HTCondor DAGMan is the workflow execution engine behind Pegasus and controls the processes.

Compute resources are procured from the master node via `condor_annex` by targeting the worker AMI. Annex is a resource acquisition and external compute resource separation management unit. Each resource pool can have multiple annexes. Each annex manages its own lifecycle. It is possible to request on-demand or spot instance fleets.

LSST Pipeline jobs are executed on the worker instances; they use Data Butler for data access. New Data Butler backends were implemented during the PoC, see Sect 5.

The Butler datastore is located in an S3 Bucket and follows the same hierarchical structure that POSIX datastore does. Consumed and produced datasets are read and written directly from S3 as bytes, whenever possible, and only downloaded to temporary files for objects that are not serializable. Since the directory structure is preserved by the S3 datastore the entire data repository is trivially transferable between the cloud and a local filesystem.

The Butler registry is a RDS PostgreSQL database that keeps track of all LSST science files. Files that are not managed by Data Butler are managed by the Condor File IO via Pegasus. These include the Data Butler configuration file, pipeline definition (Quantum) files, and the log files. The master instance serves as the staging site for these files. Various network protocols transfer the files between the master and the workers; instances do not share a filesystem.

3.1 Alternative architecture designs

We discussed different architecture designs but did not pursue all of them due to time constraints.

One prominent idea was to use HTCondor utilities to transfer files with S3, rather than relying on Data Butler to communicate directly with S3. In this design HTCondor controls all file transfers, and all datasets are declared in the execution workflow. The proposed implementation was partially external to the LSST code base, and such a data transfer plug-in was added to the HTCondor code base recently. In each job there is a POSIX data repository for local data access, and a mechanism is needed to trigger and perform data ingestion at a central S3 data repository. This requires additional Generation 3 Middleware utilities for posix-S3 datastore copying and URL generation. These issues disfavored this architecture. A favorable benefit of this approach is the potential of bulk data transfer management optimizations.

With respect to the database, we discussed the possibility of having local registries within individual jobs. Each job has a fast local SQLite registry. After each job finishes, the indiviual registry is then collated into the central database. Besides simplifying failure recovery, another benefit is to avoid potential bottlenecks related to large scale parallel SQL transactions by bundling multiple transactions together and handling database connections in a centralized manner. This envisions a fully shared-nothing architecture, but implementing this requires more thorough design work in conjunction with the Generation 3 Middleware development. Also, the current implementations of the Butler registry depend on conflict resolution syntax to resolve insertions of complexs datasets, which may not work in this scenario.

4 Approaches and strategies

AWS PoC's goal is to demonstrate that running Data Release Production (DRP) on AWS is possible. The following approaches were taken.

4.1 Progress in phases

DMTN-114 proposed development phases that broadly outlined how progress can be made and when performance would be monitored. The outlined plan was to move the execution to AWS without any backend modifications first and then gradually switch out each component to AWS.

Before this AWS PoC, the DRP execution relied a shared POSIX-compliant filesystem. In the first phase such filesystem was replaced by Amazon EFS. The data repository, including a POSIX datastore and a SQLite registry, was entirely located on EFS that was mouted to each of the EC2 compute instances.

Later, the S3 datastore replaced the need of EFS, and the PostgreSQL RDS replaced the SQLite registry. Also, a Personal HTCondor Pool on an single EC2 instance was used before we tested the capability to launch new EC2 instances as the HTCondor workers and execute jobs. This approach allowed us to debug and adapt more smoothly and always had a fallback option in integrating and testing new features and components.

4.2 Test minimally before scaling up

Small tasks are tested before workflows of jobs, and small workflows are tested before large workflows. The canonical ci_hsc dataset and the accompanying CI workflow provide a minimal HSC test dataset and a representative DRP workflow and algorithms. This test workflow is an important step in integrating new changes. For larger scale tests we used the HSC-RC2 dataset.

4.3 Use Generation 3 Middleware

One of the mandates in the AWS PoC was to use the Generation 3 Middleware (DMTN-056), which has been designed to ease the DRP execution and automation compared to the previous Generation 2 Middleware. Generation 3 Middleware design is more pliable to alternative backends, compared to Generation 2 Middleware. As it was still relatively early in its implementation, findings in the AWS PoC were feedbacked to the Generation 3 Middleware team and influenced its development.

On the other hand, as during the AWS PoC the Generation 3 Middleware has been under active development, backwards incompatible changes occurred often, and the APIs were unstable. Furthermore, there were no integration tests when the AWS PoC started and the test coverage has been suboptimal. Hence, changes often broke functionality. Lack of a good integration test is still a major concern going forward. New integration tests may not be accepted due to the much longer time required to run the test suite.

Keeping this in mind, We decided not to always follow the bleeding edge version and updated only when unnecessary, such as important bug fixes. The stack versions should not have strong impacts in the AWS PoC conclusions in terms of the cloud feasibility.

4.4 Focus on end-to-end execution

Potential optimizations and further investigations were identified throughout the PoC project but were not carried out. Some ideas are described in Sec 7.

5 Building AWS support into the Data Butler

The Data Butler is the overarching data IO abstraction through which all LSST data access is mediated DMTN-056. Datasets are referred to by their unique IDs, or a set of identifying references, which are then resolved through a Registry that matches the dataset IDs, or references, to the location, file format and the Python object type of the dataset. The system that persists, reads and potentially modifies the datasets is called the Datastore. The Registry is almost always backed by a SQL database and the Datastore is usually backed by a shared filesystem. A major focus of the AWS PoC was to implement, and investigate issues related to, an S3 backed Datastore and a PostgreSQL backed Registry.

At the time the AWS PoC project began, Generation 3 Data Butler implemented `PosixDatastore`, a local or shared filesystem datastore, and a SqliteRegistry. OracleRegistry followed soon after the AWS PoC work began. Initially the focus was on implementing an S3 backed datastore called S3Datastore. The interface between AWS services and LSST Stack would be based on the official AWS SDK called boto3. In March 2019, Dino Bektesevic visited Tim Jenness for this work and implemented the early versions of a new module in the `daf_butler` called `s3utils`, an `S3Datastore` class, the `PosixDatastore` equivalent, and a set of appropriate unit tests that demonstrated its functionality and correctness. The unit tests for the datastore utilize the `moto` library which mocks requests and responses sent to AWS services, so that no additional external infrastructure is required to use it. `PostgreSqlRegistry` class was implemented partially during the visit and completed shortly after the visit. The initial implementation showcasing the required changes to the code was submitted as a Draft Pull Request PR-147 in `daf_butler`.

The tentative implementation revealed issues with how the Data Butler treated Uniform Resource Identifiers, or URIs, which were, at the time, not being handled correctly, as per standards defined in FC-3986, by the `Location` class. In May 2019 DM-19916 resolved the issues with the `ButlerURI` class. Major efforts were then invested into refining the newly added code to the level of production quality as well as updating the remaining Generation 3 Butler to use the updated `ButlerURI` code instead. Every call to OS functionality had to be generalized to take a URI and from it determine the appropriate operation - a call to OS functionality, a AWS operation or something else. This led changes in `Butler`, `Config`, `ButlerConfig` and YAML Loader classes. These changes made the whole of Data Butler more general and pliable to future changes, such as adding support for other cloud providers.

Further integration of the S3 backend required a change to Formatter classes to enable data serialization and deserialization to and from bytes. Formatters present interfaces for reading and writing of Python objects to and from files. They are the mechanism underlying how Data Butler is capable of presenting data as science products in the form of Python objects, abstracting away the underlying file types. Modifications were made to `JsonFormatter`, `PickeFormatter`, `YamlFormatter`, `PexConfigFormatter` and the generic abstract class `Formatter`. This concluded the last of changes required for `S3Datastore` integration and the code was merged in PR-179 in `daf_butler` (DM-13361). It became apparent that similarities were shared between `PosixDatastore` and `S3Datastore`, and would be shared by other future datastore implentations. This prompted DM-21009 to reduce code duplication in the general datastore code.

`PostgreSqlRegistry` was added afterwards. The initial implementation from March 2019 was based on `OracleRegistry`. Similarities between the two implementations led to a re-implementation of the generic `SqlRegistry` class in July. Problems were caused, for both Oracle and PostgreSQL, by the table naming conventions and additionally, for PostgreSQL, the table views did not conform to the assumptions made. In July the `PostgreSqlRegistry` was re-implemented in terms of the more general `SqlRegistry` and a new SQLAlchemy expressions compiler was written, so that table views could be generated correctly. The policy for additional registry implementations was not to accept associated unit tests, as they are dependent on existing outside architecture, meant that checking whether it worked or not had to be based on manually executing actual data workflows. In July and Auguest Dino Bektesevic migrated existing SQLite registries to PostgreSQL and with Hsin-Fang Chiang started testing using the ci_hsc workflow on AWS. The code was merged into the master branch of `daf_butler` in August with PR-161. A major issue was then discovered when issuing rollback statements during error recovery stemming from assumptions made when implementing how all of the current SQL registries handle errors during transactions. A stopgap solution, that works for all currently implemented registries, was implemented in DM-21210 and a more complete solution was implemented later in DM-21201.

Outstanding issues are presented in terms of security and authorization when dealing with both `S3Datastore` and `PostgreSqlRegisty`, with `PostgreSqlRegisty` being especially sensitive to these issues. Security has received the outmost attention by the LSST AWS PoC group. Significant attention was paid to preserving the flexibility of the authentication in order to be able to incorporate external authenticators such as Oracle Wallets and AWS IAM Roles and Policies. There were several different iterations and improvements made to the authentication implementation (DM-20992, DM-21146, DM-21222) that resulted with the current implementation. DM-20992 re-implemented the `DbAuth` module that was previously C++ wrapped in the Generation 2 Butler `daf_persistence` to pupre Python in the Generation 3 Butler, so that the module would support basic file based authentication in absence of external authentication methods. Additional layers of security are achieved through EC2/S3/RDS interfaces by IP white/blacklisting , IAM, Policies etc. These policies can be very granular, affecting individually selected objects, Bucket-wide to placing all instances on the same, externally innaccessible, Virtual Private Network (VPN).

Adding the support for AWS into the Butler exercised a significant fraction of the Generation 3 Data Butler. During the process many faults and unpredictable behaviors were discovered and solved. Many problems touched, and continue to exercise, the general Generation 3 Data Butler implementation, as well as assumptions made during their implementation. Recounting the wide list of major improvements to the codebase, hopefully, reveals how productive this exercise has been in helping generalizing and strengthening the whole Generation 3 Data Butler codebase.

5.1 PostgreSQL performance

The following described performance issue applies strictly to w_2019_38 builds and earlier. The QuantumGraph is created by issuing a very large SQL statement that, effectively, creates a cartesian product between many, of the 15-ish or so, tables in the registry. The results of the large query are then parsed in Python and a slew of many different, small, follow-up queries are issued. This diverse workflow presents a difficult challenge in DB optimization. The Generation 3 Middleware team reports that on average it takes 0.5 to 1.5h to create QuantumGraphs for tract-size DRP workflows with Oracle on NCSA infrastructure. In our tests we were unable to create even the simplest QuantumGraphs, even after doubling of the RDS instance resources, and were forced to abort multiple times after 30+h of execution. The lack of performance did not seem related to hardware limitations, since none of the performance metrics availible showed heavy hardware loads nor didthe problem go away even after significant increase of RDS resources.

PostgreSQL, and Oracle, regularly collect statistics on database objects used in queries. These statistics are then used to generate execution plans. These execution plans are then oftentimes cached. This in general results in performance gains. But because Butler generates all SQL dynamically there are no guarantees that a cached, or even pre-prepared execution plan can be executed. In PostgreSQL many of these optimizations then fail.

Specifically what seems to happen with PostgreSQL is that all the views materialize completely on disk. Even when the views are part of larger statements, in which the outter statements have strict constraints on them, the outter constraints do not seem to penetrate to the view statement. It is unclear, from within AWS PoC, whether Oracle DBMS does not suffer from this issue because it has a better SQL statement optimization engine or if this is something more directly tied to how PostgreSQL implements views.

The performance penalty of the behaviour is debilitating. Bellow is an abreviated query plan for the simplest case of `select * from visit_detector_patch_join limit 4;` that suffers from the described issue.

```
Limit .... (actual time=12732.766..12732.774 rows=4 loops=1)
->Unique .... (actual time=12732.765..12732.770 rows=4 loops=1)
  ->Sort .... (actual time=12732.764..12732.767 rows=6 loops=1)
    Sort Key: ....
    Sort Method: external merge  Disk: 209008kB
    ->Hash Join .... (actual time=763.846..1524.504 rows=4642107 loops=1)
      Hash Cond: ....
      ->Seq Scan on .... (actual time=0.008..16.289 rows=206960 loops=1)
      ->Hash .... (actual time=763.723..763.724 rows=3259107 loops=1)
```

```
        Buckets: 65536  Batches: 64  Memory Usage: 3635kB
        ->  Seq Scan on patch_skypix_join .... (actual time=0.006..240.128 rows=3259107 loops=1)
Planning time: 0.323 ms
Execution time: 12759.391 ms
```

Note that the total materialized size of the view on disk is 200MB and that it took 12 seconds to retrieve only 4 results! There is no clear solution to this problem from within the Middleware codebase.

The sollution is however possible by manually redefining the views as materialized views instead, adding triggers that recreate the views on any insert statement to the underlying tables that make the view, and then adding indexing onto the materialized views. This reduces the time it takes to create a QuantumGraph to the point where it's comparable to that reported by the DM team. The query plan for the same SQL select statement as described above now looks like:

```
Limit .... (actual time=0.009..0.010 rows=4 loops=1)
->Seq Scan on visit_detector_patch_join .... (actual time=0.008..0.008 rows=4 loops=1)
Planning time: 0.053 ms
Execution time: 0.021 ms
```

Recently DM-17023 reworked and completely re-implemented the SQL schema and data model for the registries. We have not yet tested the performance of the new registry schema in PostgreSQL. The issues described here well optimize the set of specific queries, at the cost of a performance hit for all inserts into visits, patches..., but an important take away lesson here should be that because all of the SQL is generated completely dynamically guaranteeing execution plan stability and performance is dificult. Not all DBMSs optimize SQL statements euqally well. Tying the CI and testing to only one database doesn't suficiently test the code performance and because the SQL is generated completely dynamically to fix potential poor performance requires schema modifications to be made external to the Middleware code. Because some of the performance loss can be debilitating, it is possible that certain DBMSs are, in this implementation, tied into implementing unoptimal solutions just to be able to run DRP.

6 Results of the tract-sized runs

After successful execution with the `ci_hsc` dataset, we scaled up the run to one full tract of the HSC-RC2 dataset, as defined in DM-11345. The full HSC-RC2 input repository contains 108108 objects and totals ~1.5TB, including 432 raw visits in 3 tracts and ~0.7TB of calibration data.

Task	Count
Initialization	1
IsrTasks	6787
CharacterizeImageTasks	6787
CalibrateTasks	6787
MakeWarpTasks	4580
CompareWarpAssembleCoaddTasks	395
DetectCoaddSourcesTasks	395
MergeDetectionsTasks	79
DeblendCoaddSourcesSingleTasks	395
MeasureMergedCoaddSourcesTasks	395
MergeMeasurementsTasks	79
ForcedPhotCoaddTasks	395
Total	27075

TABLE 1: Task breakdown of the HSC-RC2 tract=9615 workflow

In this project, we targeted tract=9615 which was executed with the Oracle backend on the NCSA cluster in July 2019 as the S2019 milestone of the Generation 3 Middleware team; see DM-19915. In terms of raw inputs, tract=9615 contribute around 26%, or ∼0.2 TB, of the raw data in the HSC-RC2 dataset. We ignored patch 28 and 72 due to a coaddition pipeline issue as reported in DM-20695. A Butler repo was first made on NCSA's GPFS with a sqlite registry, and then transferred to the S3 bucket and the RDS instance. All tract-sized runs reported in this DMTN used LSST Software Stack release `w_2019_38`. The version of HTCondor was 8.9.3.

The workflow contains 1 initialization job and 27074 regular PipelineTask jobs. Science configurations from `https://github.com/lsst-dm/gen3-hsc-rc2` were used to generate a Quantum Graph. We transformed the Quanta into jobs in the Pegasus format with one-to-one mapping. The breakdown of the tasks in the workflow is in Table 1.

Generally speaking, there are two types of jobs: small-memory and large-memory jobs. Small memory jobs take less than 4GB per jobs, and large memory jobs can take up to ∼30GB per job. For simplicity, we consider all jobs of MakeWarpTask, CompareWarpAssembleCoaddTask, DeblendCoaddSourcesSingleTask, and MeasureMergedCoaddSourcesTask as large-memory jobs and require ∼30GB of memory in their job requirements. HTCondor only matches jobs to machines with sufficient memory. AWS instances come with different flavors and the `r` family provides memory optimized instances with ∼8GB per core. The worker instances are configured to be HTCondor partitionable slots which dynamically splits resources and creates new slots to suit the jobs.

Run ID	Workflow wall time	Cumulative job wall time	Pipetask Min (sec)	Pipetask Max (sec)	Pipetask Mean (sec)	Pipetask Total (sec)
20191026T041828	28.4 hrs	61 days, 10 hrs	17.025	5936.038	195.465	5292217.997
20191121T015100	11.7 hrs	65 days, 8 hrs	18.272	5852.514	207.691	5623244.556
20191127T192022	8.7 hrs	62 days, 16 hrs	17.861	6243.819	199.636	5405141.464
20191127T192345	10.0 hrs	62 days, 23 hrs	19.297	6300.657	200.601	5431273.315

TABLE 2: Run summary

The submit host is an AWS on-demand instance, typically `m5.large` or larger. Spot fleets are requested after the Pegasus workflow start. Typically `m4` or `m5` instances are used for the single frame processing or other small-memory jobs, and `r4` instances are used for large-memory jobs. After the workflow finishes, remaining running Spot instances may be terminated on the AWS console. Besides the 27075 pipetask invocations, Pegasus added 2712 data transfer jobs and one directory creation job. The total output size from the tract=9615 workflow is ~4.1 TB with 74360 objects.

6.1 Notes from the successful runs

Details of all runs are summarized in DM-21817, and in the following we summarize the successful runs only. Table 2 lists the runtime as reported by Pegasus tools.

In the first successful run `20191026T041828+0000`, a fleet of 40 `m5.xlarge` instances were used for single frame processing and then a fleet 50 `r4.2xlarge` memory optimized instances for the rest. A `m5.large` on-demand instance served as the master. The single frame processing part finished in 4 hours; coadd and beyond took 16 hours. In this run, the memory requirement of the large-memory jobs was slightly higher than half of a `r4.2xlarge`, resulting in instance resources not fully used. This run spanned two billing days.

In the repeated run `20191121T015100+0000`, the master was also a `m5.large` on-demand Instance. Fleets containing 75 `m4.xlarge` instances and 50 `r4.2xlarge` were launched. The memory requirement of the large-memory jobs was adjusted so that two such jobs can run on a `r4.2xlarge` simultaneously. Due to the larger fleet, the whole workflow finished within 12 hours in one bill day.

We then ran two of the same workflow graphs `20191127T192022+0000` and `20191127T192345+0000` simultaneously, simulating a larger input size. The master was a `m5.2xlarge` on-demand instance. A Spot fleet containing 150 `m5.xlarge` instances ran the single frame processing for the first three hours, and a fleet of 150 `r4.2xlarge` instances ran the rest of the workflow. 600 jobs ran simultaneously during single frame processing. Larger fleets were used to help finishing the workflows in a shorter wallclock time. This run spanned two billing days.

6.2 Cost analysis

The main components of the charges come from (a) EC2 Spot instances, (b) EC2 on-demand instances, (c) S3 storage, (d) RDS, and (e) others. We extract and analyze information using the AWS Cost Explorer tools.

1. **EC2 Spot instances**. EC2 Spot instances are the workers that execute the processing jobs, so this essentially is the cost of the compute power and scales with the compute resources needed to accomplish the processing campaign. The exact pricing for Spot instances varies based on supply and demand of the overall EC2 capacity. For instances used in our test runs, Spot instances cost around 20-25% of the on-demand instance price. Charges continue as long as the instances are up and running, even if no jobs are assigned to the instances. One hour of Spot instance typically costs $0.045 for `m5.xlarge` and $0.08 for `r4.2xlarge`. Different mix of instance types could affect the performance as well as the total cost. Considering the idle time as necessary overheads, we paid ~$0.035 per active CPU hour in average. We could pay less with better workload control and instance lifecycle control.
2. **On-demand EC2 instance**. We use an on-demand EC2 instance to serve as the submit host and the central workflow manager because we do not want it terminated by AWS. The current price of m5.xlarge on-demand instances is $0.192 per hour.
3. **RDS**. Throughout our test runs we used a `db.m5.xlarge` instance, which has 4 vCPU and 16 GB of memory, to host the Butler Registry of the HSC-RC2 repository. We are aware this DB instance is more powerful than we usually need but we keep it running. The charge is therefore proportional to the span of time. For simplicity we count all RDS charge during the runs towards the cost of the runs, which is an overestimate because we also host other small database instances for testing purposes. For example, a `db.t2.micro` has been running alongside to host a Butler Registry for the `ci_hsc` repo which costs $1.3 per day.
4. **S3**. The charge of S3 is dominated by the data storage cost, which is $0.023 per GB

per month for the first 50TB. This means it costs ∼$1.2 per day storing the input repo (∼1.5TB) alone, and ∼$3.1 per day storing one set of the tract=9615 workflow outputs (∼4.1TB). Note that outbound data transfer is not free at AWS. At the rate of $0.09 per GB, transferring one tract=9615 output dataset out of AWS would cost ∼$370. We also incur charges per quantity of requests, which cost ∼$4.3 for each run of the tract=9615 workflow.

5. **Other charges**. The majority of other charges come from the Elastic Block Store (EBS) that provides storage for use with EC2. This includes SSD-backed volumes and snapshots, both of which are priced per size and time. Other charges are relatively small, such as a Business Support plan to get help from AWS engineers, and CloudWatch for additional monitoring information. In our accounting here this also includes charges from instances used in work indepedent of the workflow execution, so this should be seen as an upper bound.

Table 3 and Figure 2 provide an overview of the cost breakdown for each run. This includes all charges incurred on the billing days during which the runs were done, so this can be seen as an upper bound. The price dropped significantly from the first run to the rest, mostly due to operator knowledge and optimization for this workflow execution. We have not optimized the overall usage; more discussions are in Sect 7.

	Runs		
Category	20191026T041828	20191121T015100	20191127T192022 20191127T192345
EC2 on-demand instances for master	2.88	1.54	2.05
EC2 Spot instances	94.69	58.55	52.94
RDS	26.38	13.82	13.82
S3 (including other storage cost)	17.29	13.48	21.48
Others/mostly EC2	18.38	7.12	6.37
Total	159.62	94.51	96.66

TABLE 3: Cost breakdown for each run

6.3 Cost projection

The raw input exposures in the HSC-RC2 tract=9615 workflow contains ∼0.2TB from 112 visits. Compared to one night of LSST data in full operations, this is only ∼11% in the number of visits. In one of our tests, we doubled the input size to explore how it scales up. It scales roughly linearly in cost and total compute resources. In processing wallclock time, it does not take

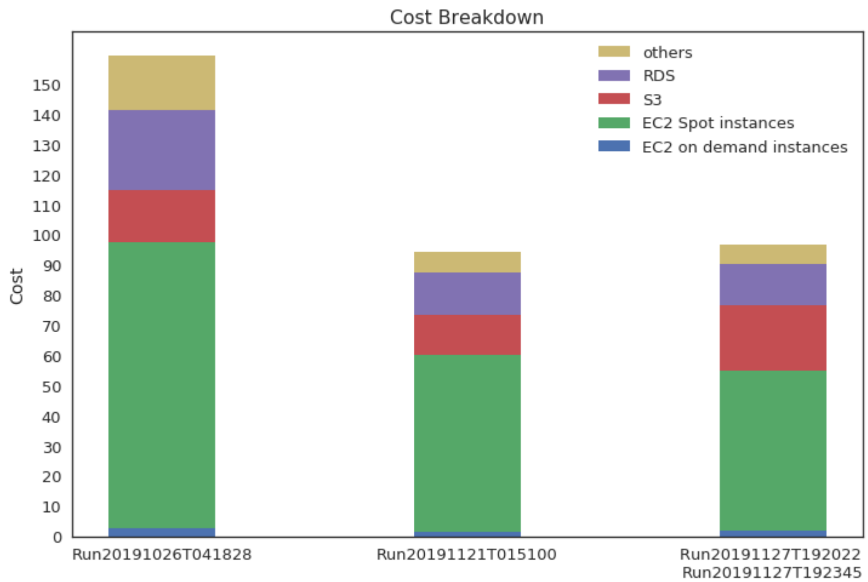

FIGURE 2: Cost breakdown for each run

longer because more EC2 instances can be deployed. For one tract of DRP test workflow the cost was around $95. If such scaling relationship holds for larger amount of data, 1000 input HSC visits will cost around $850 to process.

7 Potential Improvements and more lessons learned

In this session we describe issues we have encountered during the execution and ideas on how to improve the code or the instance and workflow management. We discuss both intermittent failures that we understand and expect to occasionally encounter even in production, as well as higher level design or tooling improvements.

Failures can occur due to non-pipeline issues such as underlying infrastructure. The fault rate may be small but as we scale up we start to encounter some. Some examples are listed below; most seem transient.

1. Database connection timeout. Attempting to connect to the RDS instance failed.

   ```
   sqlalchemy.exc.OperationalError: (psycopg2.OperationalError) could not connect to server: Connection timed out
   ```

2. After a file was added to a S3 bucket and during ingestion in the Butler registry, S3 re-

ported a file does not exist. This will be fixed in DM-22201.

```
FileNotFoundError: File at 's3://hsc-rc2-test1-hfc/repo-w38/hfc30/srcMatchFull/1316/srcMatchFull_1316_24_HSC_17.fits' does
    ↪ not exist; note that paths to ingest are assumed to be relative to self.root unless they are absolute.
```

3. S3 read timeout before science processing started in a job.

```
botocore.exceptions.ReadTimeoutError: Read timeout on endpoint URL: "None"
```

4. Worker out-of-memory while running jobs. For the same pipeline and input data, this is reproducible. But we may not always have accurate memory usage prediction for any input data before running the jobs. We can configure HTCondor to increase the memory requirement in the retry. However sometimes OOM crashed the instances and appeared as a network issue which is not desirable behaviour.
5. Launching HTCondor Annex workers failed with connectivity check collector issues.

```
Connectivity check found wrong collector (f5fc15573ffb9c93 vs a006066e73c412da).
```

6. Dataset ConflictingDefinitionError. In rare occassions we observed dataset conflict errors from the registry without obvious reasons such as duplicate collection names or retries. DM-21201 has refactored the code to robustify such transactions. It could also be related to Spot instances getting terminated; see next section on failure recovery.
7. Database out of shared memory.

```
psycopg2.OperationalError: out of shared memory
```

It was usggested that increasing `max_locks_per_transaction` may help.

More generally, improvements in system design and tooling rise to prominence. We discuss some ideas below.

1. **Better job failure recovery strategies**. Our jobs write directly to the S3 bucket and the RDS instance via the Data Butler. If a job fails in a state that partial outputs are written but the job does not fully finish, recovery is not trivial. Such failed partial writes are handled on the registry side the DBMS but on the datastore side they are handled by wrapping failable transactions in a transaction context manager. The context manager takes a callable that then proceedes with the cleanup. For S3 datastore, at the time of writing, the utility functions preforming the cleanup have not yet been writte (in progress). The larger issue with job failures is the lack of DRP workflow failure handling. There is no way yet to reliably pause the DAG execution, overwrite files from a failed job, or other ways to handle such scenarios.

2. **Container based software stack**. We have found it tricky to handle the LSST stack installation, dependencies, and environments to be used together with other software. One possible way to avoid the headaches is to use docker based stack releases. This may also ensure consistency of software on the master and the workers more easily, as well as improve shareability.

3. **Better cluster management tooling**. Our current operational approach requires manually deploy suitable types and sizes of fleets based on our understanding of the overall workload. Strategies on the instance choices and timing of requests therefore affect the cost, and manual adjustments are usually needed to reduce cost. Annex can remove idle instances out of the condor pool but may not terminate the instances until the lease expires. Also once instances drop out of the pool they can't be added back easily. Tooling become essential for the operations. For example we may use scripts to auto-scale the Annex condor pool based on the current demand for resources, or based on predictng resource demands from the Quantum Graph. The execution overheads from idle instances may be mitigated by utilizing some of other AWS services, such as Simple Workflow service (SWF) or AWS Batch service. This may bind the solutions strictly to AWS, a disadvantage of such adoption.

4. **End-to-end CI**. This should include all operational components to do an end-to-end run. This includes Butler repo generation, registry generation, Quantum Graph generation, job composition, workflow translation, job execution, and so on. Many of the components were in the development phase and workarounds were used during the PoC. For example a native Generation 3 ingestion was not available so a Gen2-to-Gen3 conversion was needed, and in fact many of the registries were migrated to RDS partially manually by executing bespoke scripts made specifically for such purpose that are not generally portable cross backwards-incompatible changes to the Middleware code. As we put together the pieces, the absence of to do so in an automatic fashion became a key burden. In retrospect we probably should have invested more time to automate the end-to-end workflow, even with workarounds, but the efforts were plagued by the lack of a canonical ingest tool. Additionally, the DM team has concerns in the long execution time from overly-inclusive CI tests. Adding S3 datastore CI tests to the default CI suite would double the total CI execution time.

5. **Credential handling**. Security concerns received special attention in the AWS PoC. Natively none of the Middleware code requires the credentials to exist on the instance in plain text, even though they support that option. Authentication can be performed via Identity and Access Management Roles to both S3 and RDS services. IAM Role authen-

tication is more secure than other authentication methods, but require understanding of how AWS security and permissions work and sometimes require tedious setup. For example for RDS IAM access an account admin needs to create the IAM Role then DB admin needs to log in to the DB interactively and create a role with rds_iam permissions and then also grant usage priviledges on the DB schema, tables etc. While we have tried an IAM setup in the AWS PoC we often stored credentials onto the master and worker instances in `~/.lsst/db-auth.yaml`, `~/.aws/credentials` or in environmental variables which is not the best practice of handling the access. Utilities and tooling that help speed up such setup would be highly desireable.

6. **Robustify and give better error messages**. It has been observed that sometimes the error messages could be misleading.
7. **Other AWS services**. Services such as AWS Batch and AWS Glue could potentially offer powerful tools to our use cases. AWS Batch provides EC2 lifecycle management and workflow control, which may reduce the cost of our EC2 usage. AWS Glue provides extract, transform, and load (ETL) services with respect to our S3 storage usage, which may improve our data warehousing experience and form Data Lake.

8 Summary

In this AWS PoC project we have demonstrated the feasibility of LSST DRP data processing on the cloud. We implemented AWS backends in the LSST Generation 3 Middleware, allowing processing entirely on the AWS platform using AWS S3 object store, PostgreSQL database, and HTCondor software. We analyzed cost usage in our test execution, and estimated cost for larger processing campaigns. We showcased our progress in a live demonstration in the LSST Project Community Workshop in Aug 2019, as well as a hands-on tutorial in the Petabytes to Science Workshop in Nov 2019. Ideas of improvements necessary for larger-scale production are identified and discussed.

A References

References

[LDM-463], Becla, J., Pease, N., 2017, *Data Access Design*, LDM-463, URL `https://ls.st/LDM-463`

Bosch, J., Armstrong, R., Bickerton, S., et al., 2018, PASJ, 70, S5 (arXiv:1705.06766), doi:10.1093/pasj/psx080, ADS Link

Deelman, E., Vahi, K., Juve, G., et al., 2015, Future Generation Computer Systems, 46, 17, URL http://pegasus.isi.edu/publications/2014/2014-fgcs-deelman.pdf,
Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575, doi:10.1016/j.future.2014.10.008

Jenness, T., Bosch, J.F., Schellart, P., et al., 2018, arXiv e-prints, arXiv:1812.08085 (arXiv:1812.08085), ADS Link

[DMTN-114], Lim, K.T., Guy, L., Chiang, H.F., 2019, *LSST + Amazon Web Services Proof of Concept*, DMTN-114, URL http://dmtn-114.lsst.io

[DMTN-056], Pim Schellart, J.B., 2018, *Butler Redesign Strawman*, DMTN-056, URL http://DMTN-056.lsst.io

Thain, D., Tannenbaum, T., Livny, M., 2005, Concurrency - Practice and Experience, 17, 323

B Acronyms