A Serverless Data Lake with Amazon Web Services (AWS)

6/23/2020

Ralf Brockhaus
Smilefish
4590 MacArthur Blvd #500
Newport Beach, CA 92660

smilefish
Machine Learning and Data Science

Overview

The following case study describes a data lake solution that we developed and continued to maintain for one of our digital marketing clients.

This is our 3rd iteration implementing data lakes in AWS. This time, we are taking the challenge to commit to a serverless approach. The lake's cost at rest should be minimal.

Every chain is only as strong as its weakest link. So we also take the opportunity to formally review our solution based on *Amazon's well-architected framework* and optimize it to evenly meet Amazon's 5 pillars of excellence for *business operations, security, reliability, performance,* and *cost*.

Goals

1. Host a full US consumer header with 300 million datasets of 400 data points with monthly updates.
2. Ability to manage 1 billion business leads and associated data records in structured, semi-structured, and unstructured formats.
3. Architect a solution optimized for business agility, monetization, compliance, and user experience.
4. Validate the implementation based on Amazon's 5 pillars of excellence: business operations, security, reliability, performance, and cost.
5. Completely separate data storage from data processing and shift towards a low-maintenance serverless implementation where possible.
6. Prepare the client for advanced data analytics and applications of true machine learning.

Technology

We selected **AWS S3** for storage, secured by **IAM**, monitored via **Cloud Watch**. The automated ingestion process utilizes the AWS **Glue Crawler** and **Data Catalog.** We maintain the metadata in **DynamoDB**. Automated **ETL** is executed via **Lambda** managed under the AWS **API Gateway.** It uses **Athena CTAS** and **ETL** queries. **Athena** is now also the main analytical query engine. Our admin dashboard is implemented using **ReactJS** and **Highcharts** for data visualization. The dashboard is connected to our **API Gateway**.
Our operational *Consumer Header* services are still partially implemented in NodeJS and Python hosted on **EC2** with **RDS / Postgres** for quick relational transactions.

Table of contents:

Data Lake

A data lake is a centralized repository that allows a business to store all structured, semi-structured, and unstructured data at an unlimited scale. This data can be stored as-is, be processed for different types of analytics and machine learning. The result allows for better business decisions and higher monetization.

We are covering the very helpful (yet more formal and dry) part of the well-architected framework further down in the document and focus on the task at hand first:

Characteristics

- Our data lake needs to provide the ability to store all data, regardless of source, structure, and amount in its original/pristine format to create a *Single Source of Truth* for all further processing and analytics.
- The implementation needs to decouple storage from computation to enable the selection of best in class choices for processing and analytics.
- Data providers should only be pointed to a drop-off location and have no restrictions on format, quantity, delivery speed, or time of day.
- The data lake should support quick ingestion and automated processing at appropriate speed with minimal maintenance cost.
- The data lake needs to be designed for low-cost storage with cost tiers that also allow keeping historic data that is less frequently accessed as its business value declines.
- The lake needs to provide comprehensive protection, security, and tracking as data flows through the tiers within the lake.
- The case study for consumer data needs to meet privacy compliance requirements (PII / Non-PII). We also work with medical applications. The selected technology and access policies need to meet HIPAA compliance and other mandates for managing medical data.

Reference Architecture

By selecting AWS S3 as our core storage solution, we can build a data lake with virtually unlimited storage at a low cost. Building the architecture around S3 also enables us to completely decouple storage from computation, which allows us to selectively combine other AWS services to meet all business requirements. We found this helpful to future proof our solutions in which we can conveniently experiment and replace individual components as requirements change or skills and technology evolve.

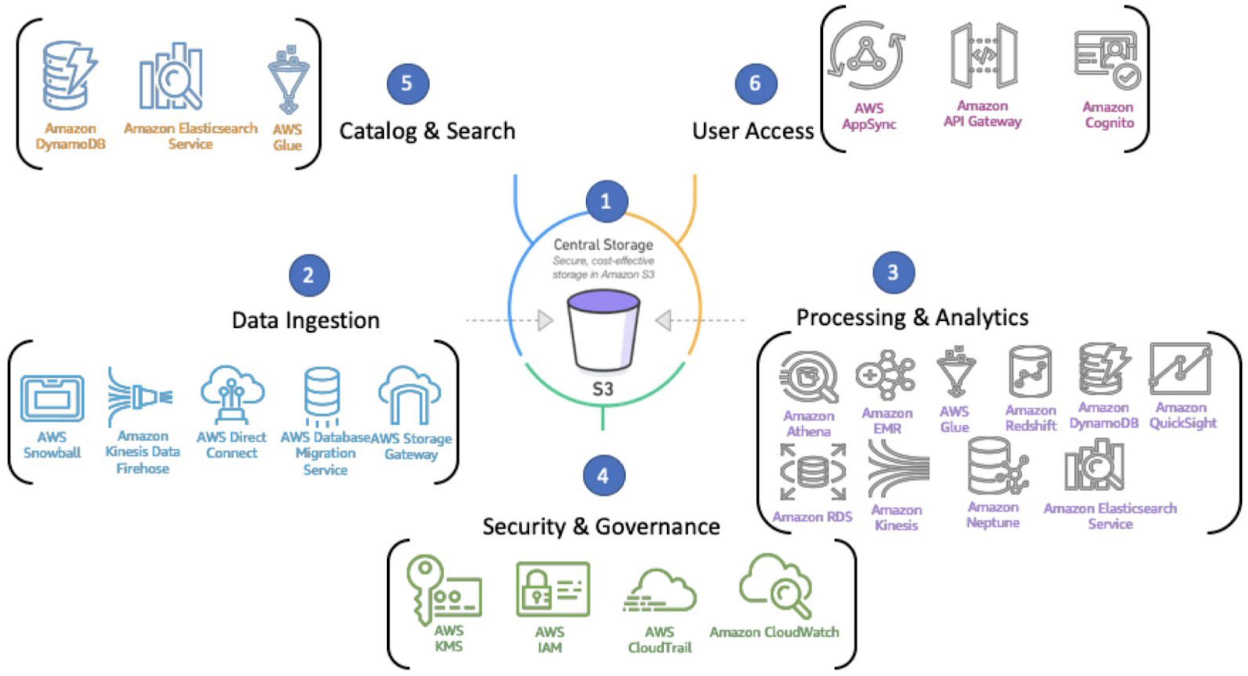

Diagram 1: Reference Architecture for *AWS Well-Architected Framework*

(1) Central Storage

We configured dedicated S3 buckets in logical data tier layers to store all pristine original data as well as all datasets created by our ETL and analytics environments:

Tier 1: buckets for pristine raw data from ingestion services

Tier 2: data resulting from ETL jobs. We store this data mostly in columnar Parquet format to be optimized for analytics, cost, and speed (see benchmark further down).

Tier 3: business-data generated from tier 2 data catered to convenient access in specific business scenarios. For faster operational needs, this data can also be loaded into RDS, Redshift, or DynamoDB databases. Equally, data from these databases can be synchronized or offloaded into the s3 lake for consistent analytics.

(2) Data Ingestion

Our Tier 1 consumer data is mostly uploaded via FTP or data streaming to dedicated buckets. Tier 1 also includes data from Salesforce, lead-generating websites, external advertising, and analytics (Google) a well as crawlers.

(3) Processing and Analytics

Every new Tier 1 data file triggers an event to catalog the metadata in RDS or DynamoDB and automatically starts the ETL for Tier 2 with AWS Glue, Lambda, and (in the past) dedicated EC2 routines. The data is then ready for analytics in Athena and results can be visualized in AWS Quicksight, Tableau, Microsoft Power BI or served by business applications that rely on RDS, Redshift, or DynamoDB (automated process).

(4) Protect and Secure

To fully cover authentication, authorization, encryption, compliance, and auditability, we need to review all data at rest and in transit. We then need to consistently enforce these concepts vertically throughout all storage and processing services.

Implementing our data lake with focus one technology provider allowed us to store keys in KSM, configure roles and policies in IAM, and audit all processing via Cloud Trail.

We strongly recommend implementing configuration in the form of scripts and treating these scripts as software and therefore fully tested and under version control. (CI-CD)

Next, we defined specific policies for "3rd party" integrations with Salesforce, Google, and Facebook.

(5) Data Catalog and Search

An effective data catalog service is essential in harvesting business value from the data lake. It ensures that the necessary metadata about every item in the lake is captured, governed, maintained, indexed, and searchable. We automate these services via AWS Glue and Lambda functions that are triggered when files arrive, get modified or deleted. (see *Avoid the Data Swamp*).

(6) Access and User Interface

Business value is driven by our ability to monetize the data in the lake. We implemented secure serverless APIs with Lambda functions under the Amazon API Gateway.

- Ingestion of datasets and their metadata. For our Consumer Header application, this also includes synchronization with Salesforce and payment gateways.
- API endpoint to configure ETL to increase data value from Tier 1 to Tier2.
- API endpoints to load data into DynamoDB, RDS, and Redshift business applications and automated back-synchronization into Tier2 for analytics.
- This access also allows for visualization in analytics dashboards which we implemented with ReactJS and Highcharts talking to DynamoDB, RDS, or directly to Athena.

Avoid Data Swamp!

With power comes responsibility. Many organizations fall into 'data-greed' and face challenges given the opportunity to collect unlimited amounts of raw data and store it at low cost without oversight and governance. Avoiding a 'data-swamp' requires mechanisms that guarantee semantic consistency, governance, and access control.

We used various tools to establish governance and manage our data lake's metadata in **DynamoDB** (and **RDS** if needed**)**. For this case study, we continue our journey into serverless architecture and see high value in **AWS Glue Crawlers** and **Glue Data Catalog** which helped our migration to Athena as it makes services interchangeable.

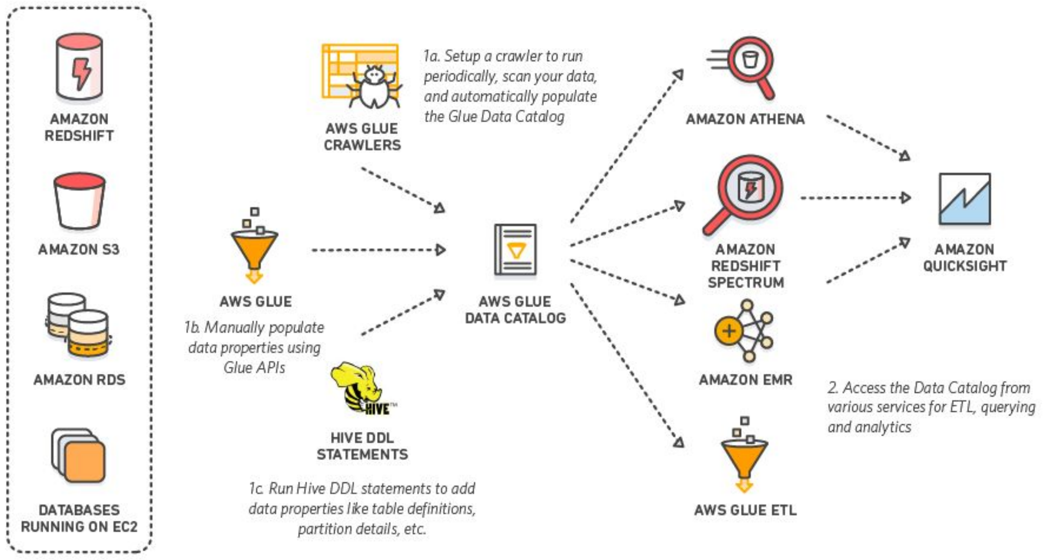
AMAZON REDSHIFT
AMAZON S3
AMAZON RDS
DATABASES RUNNING ON EC2
AWS GLUE CRAWLERS
1a. Setup a crawler to run periodically, scan your data, and automatically populate the Glue Data Catalog
AWS GLUE
1b. Manually populate data properties using Glue APIs
HIVE DDL STATEMENTS
1c. Run Hive DDL statements to add data properties like table definitions, partition details, etc.
AWS GLUE DATA CATALOG
AMAZON ATHENA
AMAZON REDSHIFT SPECTRUM
AMAZON EMR
AWS GLUE ETL
AMAZON QUICKSIGHT
2. Access the Data Catalog from various services for ETL, querying and analytics

Diagram 2: AWS Glue Data Catalog

We desire flexibility in selecting tools across the AWS ecosystem. **Glue** provides flexibility due to support not only for our data lake in **S3**, but also the **Redshift** data warehouse and our operational **RDS Postgres** databases. At the same time, we can utilize the resulting data catalogs in **Athena**, **Redshift Spectrum**, and **AWS Glue ETL**.

For this case study, we utilize **Glue** to crawl the consumer header, create data tables which we use in **Athena**, and we also use it to quickly attach visualization in **Quicksight**. For now, we continue to maintain all our metadata in DynamoDB and RDS, but we are considering a deeper integration with Glue in the future.

Case Study Part 1

In the following bench mar, we are setting up a simplified data lake and use the example of a consumer header to illustrate the configuration with S3, IAM, Glue and execute a benchmark with Athena CTAS, ELT, and analytical queries to show drastic performance gains and cost reduction via serverless queries executed directly on low-cost s3 storage.

Data Ingestion

First, we are setting up the storage for the pristine consumer header data, monthly updates, and all data formats the result from ETL:

- We configure separate **S3 buckets** for data upload and processing. This allows us to create user accounts and policies in **IAM** to specify locations for our partners to upload their data restricted to specifically secured buckets. All the partner needs to know are the locations and credentials.
    - The monthly consumer header has 400+ attributes and roughly 300 million rows. The partner uploads the data in multiple compressed files of roughly 7.5 GB each.
    - Note: Glue and Athena can work directly on uncompressed files as long as they are consistent and stored in the assigned S3 “folder”.
- The upload triggers scripts via **Cloud Watch Rules** to copy and process the data to “internal” **S3** buckets for which we defined a separate set of policies in **IAM**. Some of this work we initially tested with the **AWS UI console**, but we then automate staging and production via shell scripts and use **AWS command-line** tools.
- To copy the data we use **AWS APIs** and the **AWS command-line tool**:
    - AWS s3 cp s3://consumer/intake-$data s3://consumer/$date-pristine
    - The same concept works for almost all AWS services and APIs.

We automatically catalog metadata for every file:

- The file detection triggers the copy to “internal” buckets as well as **Lambda** functions to create metadata we manage in **DynamoDB.**
- Last but not least we utilize **AWS Glue** to crawl the file(s), verify the consumer header’s column set, and generate data table definitions that line up with our metadata.
    - Glue is very helpful in creating tables, but please verify the types and names. We stumbled i.e. over little gotchas with long names in capital letters.
    - You might want to agree with a vendor on a header sample and use it for verification, and you can also load the sample’s header into the **Glue command** via **Glue classifiers**.

- Via the same **Lambda** functions, we can also integrate virus checkers, notifications via AWS **SES** (simple email service) or text messages.

Processing

In other projects, we use **Redshift** and **Redshift Spectrum** and execute bigger ETL jobs on **EC2**. However, for this *Consumer Header Project*, we fully committed to the serverless approach. Using **Lambda** functions and moving most of the ETL pipeline into **Athena** allowed us to drastically reduce cost and maintenance.

Amazon charges $5 per terabyte of data scanned by **Athena** and there are a lot of tricks to reduce the amount of data to be scanned.

Athea seamlessly utilizes the table definition we created via **Glue** and we can directly read information from the compressed (.gz) files in **S3**:

```
SELECT count(HouseholdID) FROM
"sf-digital-transformation"."consumer_raw";
```

- Run time: 5 minutes 9 seconds, Data scanned: 30.85 GB
- Results: 309.062.253 rows
- A query directly executed on zipped CSV is very slow and relatively expensive.
- We could still optimize the data by breaking it into much smaller partitions. Instead, move directly to Apache Parquet files.

Apache Parquet is a tabular data format optimized for analytical access. The processing cost is drastically lower and speed is faster as queries only traverse the columns of interest. The binary format and compression further reduce the storage cost at rest.

We create parquet via **Athea CTAP** queries and can execute ETL at the same time. Here, we just focus on ingesting the data:

```
CREATE TABLE
"sf-digital-transformation"."consumer_2020_03_all_cols_parquet"
WITH ( format='PARQUET',
external_location='s3://consumer/2020-03/consumer-all-cols-parquet',parqu
et_compression = 'SNAPPY' ) AS
SELECT HouseholdID,PersonID,

      ... (here we deleted 400 columns to keep this document short ) ...

WirelessPhone,ZipCode FROM "sf-digital-transformation"."consumer_raw"
```

- Run time: 80 minutes 46 seconds, Data scanned: 30.85 GB, Cost $ 0.15
- Results: 309.062.253 rows
- The data files are compressed with SNAPPY
- This query reads and writes a lot of data. For long queries, you might need to expand Athena's default timeout (for your region!) from 30 to 120 minutes.

Querying the newly created parquet allows you to avoid many ‘coffee breaks’:

```
SELECT count(addressid) FROM
"sf-digital-transformation"."consumer_2020_03_all_parquet";
```

- Run time: 4.67 seconds, Data scanned: 771.12 MB, Cost $ 0.008 (vs 5min/30GB/$ 0.15)
- Results: 309.062.253 rows

Optimization

We can further optimize the parquet queries by partitioning and bucketing data towards common query parameters like state, zip-code, or household income. Parquet in combination with Athena’s CTAS queries allows us to execute complex ETL queries and the low-cost S3 storage allows us to prepare and store multiple of these data sets to have them ready for fast and cost-effective access.

Let’s run a query for all consumer data in California:

```
SELECT count(addressid) FROM
"sf-digital-transformation"."consumer_2020_03_all_parquet" where
state='CA';
```

- Run time: 12.55 seconds, Data scanned: 501.53 MB
- Results: 7.178.997 rows

For comparison, we create a consumer header dataset bucketed by state:

```
CREATE TABLE
"sf-digital-transformation"."consumer_2020_03_parquet_state_buckets" WITH
( format='PARQUET',
external_location='s3://consumer/2020-03/consumer-parquet-state-buckets',
parquet_compression = 'SNAPPY', bucketed_by = ARRAY['state'],
bucket_count = 60 ) AS
SELECT HouseholdID,PersonID,

      ... (here we deleted 400 columns to keep this document short ) ...

WirelessPhone,ZipCode FROM
"sf-digital-transformation"."consumer_2020_03_all_cols_parquet"
```

- Run time: 103 minutes 50 seconds, Data scanned: 30.33 GB
- Results: 309.062.253 rows
- It took even longer than creating the unbucketed data as it had to be placed in 60 dedicated files. Again, this is likely a small price to pay.

Now we can run all queries that require state-specific data optimized for cost:

```
SELECT count(addressid) FROM
"sf-digital-transformation"."consumer_2020_03_parquet_state_buckets"
where state='CA';
```

- ○ Run time: 6.18 seconds, data scanned: 49.5 MB
- ○ Compared to 12.55 seconds and 501.53 MB to query the un-bucketed data set.
- ○ Results: 7.178.997 rows

Benchmark and Summary

The above comparison based on one column is not entirely fair. For a more realistic benchmark, we are executing queries with 8 columns:

```
SELECT firstname, lastname, housenumber,
predirection,streetname,streetsuffix,state,zipcode from
"sf-digital-transformation"."xxx" where state='CA';
```

Approach	Results	Time	Data scanned	Cost
Query gz csv files directly	7.178.997	337.36 seconds,	30.85 GB	$ 0.15063
Query parquet	7.178.997	28.32 seconds,	1.75 GB	$ 0.00854
Query parquet with state buckets	7.178.997	25.28 seconds	0.18 GB	$ 0.00088

Considering the query cost, maintenance, and scalability, we will favor the serverless model for new development and also consider migrating projects.

We started to review projects to identify peaks in our processing. We selectively moved any functions causing those peaks into Lambda, instead of adding additional EC2 servers and RDS nodes. A documented cost analysis might follow in a separate case study.

References:

- https://d1.awsstatic.com/whitepapers/architecture/wellarchitected-Analytics-Lens.pdf
- https://d1.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf