powered by aws

VPBank Technology Hackathon 2024

General Brief

Please fill up this table and use this document as a template to write your proposal.

Challenge Statement	Customer 360
Team Name	Cerebro

Team Members

Full Name	**Role**	**Email Address**	**School Name (if applicable)**	**Faculty / Area of Study**	**LinkedIn Profile URL**
Phạm Đình Gia Dũng	Cloud Engineer	dungpham.020901@gmail.com	Hanoi University of Science and Technology	Data Science and Artificial Intelligence	https://www.linkedin.com/in/giadungpham/
Hà Thị Phương Hoa	Data Analyst	htphoa95@gmail.com	National Economics University	Corporate Finance	https://www.linkedin.com/in/hannah-ha-a44984188/
Đỗ Tuấn Anh	Data Analyst	tuananhdoo2505@gmail.com	Hanoi School of Business and Management	Management of Enterprise and Technology	https://www.linkedin.com/in/aarondo2505/
Phạm Bá Hiếu	Full Stack Developer	phamhieutb.dev@gmail.com	Academy of Cryptography Techniques	Information technology	https://www.linkedin.com/in/phamhieus

Content Outline

Detailed Solution

1. **Sourcing simulation:** Leveraging AWS products and third-party technologies to simulate realistic source data, our data will be stored in 4 formats: CSV, XML, Relational Table, JSON, and 3 sources: S3 buckets, REST API, External Database

    a. **Bank Customer:** A customer table in a retail bank's database stores essential information about the bank's individual clients. It acts as a central hub for customer data, often linked to other tables containing details on accounts, transactions, and contacts. This data are generated randomly and stored as XML files in S3 buckets in the following schema:

    - customer_id (text): Unique ID including both character and numerical value
    - identity_id (text): Sequence of 12 digits according to country convention
    - last_name (text): Customer’s last name
    - first_name (text): Customer’s first name
    - date_of_birth (date): Customer’s birthday
    - age (text): Customer’s age
    - gender (text): Customer’s gender
    - address (text): Customer’s address
    - status (text): Customer’s status
    - hometown (text): Customer’s hometown
    - phone (text): Customer’s mobile phone
    - email (text): Customer’s personal email
    - register_date (date): Date of recording registration for opening a bank account
    - customer_type (text): Customers are divided into personal or business categories

    b. **Current Account (Bank):** The bank account table in a retail bank's database focuses on the details of individual accounts held by customers. It serves as a central repository for account information and links to other transactional tables for a comprehensive view. Data are generated randomly and stored as XML files in S3 buckets in the following schema

    - account_id (text): Unique ID of current bank account
    - customer_id (text): Unique ID of a customer defined by bank system

- account_number (text): Unique serial number of each customer's bank account
- account_type (text): Bank account type
- pin (text): Unique personal identification number of each customer
- cvv (text): The 3-digit verification code proves that the person making the transaction owns the physical card
- date_opened (date): Bank account opening date
- date_closed (text): Bank account ending date (if applicable)
- account_status (text): Current account activity status
- branch (text): Bank branch that opens customer's current account

c. **Savings Account:** The savings account table in a retail bank's database zooms in on the specific details of customer savings accounts. It inherits information from the general bank customer data and adds details relevant to savings accounts. Data are generated randomly and stored as XML files in S3 buckets in the following schema:

- savings_account_id (text): Unique ID of customer's savings account
- product_id (text): Unique ID of a banking product
- customer_id (text): Unique ID including both character and numerical value
- account_number (text): Unique serial number of each customer's bank account
- deposit_amount (text): Current deposit value in bank account
- opening_date (date): Savings account opening date
- maturity_date (date): The date on which the savings deposit reaches its maturity date
- status (text): Account status if active or matured
- created_at (timestamp): Exact timestamp that customer opens a savings account
- updated_at (timestamp): Exact timestamp that customer updates a savings account
- seller_id (text): Unique ID of the seller of the savings account

d. **Current Transaction**: Acts as a detailed record of all financial activities occurring between a customer's bank account and other bank accounts. It captures every money transfer, and other monetary movement, providing a historical trail for each account.

Data are generated randomly and stored as a table in an external database in the following schema:

- transaction_id (text): Unique ID of a current transaction made by customer
- sender_account_no (text): Unique account number of the sender involved in a transaction
- sender_bank (text): Identifier of the bank from which a transaction originates (in this case the sender's bank is VPBank)
- transaction_date (date): The specific date when a transaction occurred
- transaction_time (timestamp): The exact time when a financial transaction occurred
- transaction_type (text): Type of transactions
- status (text): Transaction status is successful or not
- amount (text): Value of the transaction amount
- fee (double precision): Transaction fees (in case of money transfer outside the territory of Vietnam)
- currency (text): Currency unit
- description (text): Content of a banking transaction
- rate (double precision): Currency conversion rate

e. **Loan:** The loan table in a retail bank's database serves as a central hub for information on all loans issued to customers. It provides a comprehensive view of each loan, its status, and the borrower associated with it. Data are generated randomly and stored as a table in an external database in the following schema:

- customer_id (text): Unique ID including both character and numerical value
- loan_id (text): Unique ID for each individual loan
- loan_service (text): Category of a loan service
- loan_type (text): The type of loan a customer has taken out from the bank
- loan_amount (text): The monetary value of the loan
- interest_rate (double precision): Interest arising from the loan
- term (text): The duration from the disbursement of the loan to the maturity date
- created_date (date): Loan proposal date
- start_date (date): Loan starting date

- end_date (date): Loan ending date
- status (text): The current state or condition of a loan
- due_principal (double precision): The amount of principal that is due or outstanding on a loan at a specific point in time
- due_interest (double precision): The amount of interest due on a loan at a specific point in time
- paid_principal (double precision): The amount of the original loan principal that has been repaid by the borrower (customer) in a specific transaction
- paid_interest (double precision): The amount of interest paid out on a loan
- undue_principal (double precision): The outstanding principal amount of a loan that is not yet due for repayment
- undue_interest (double precision): The amount of interest deemed excessive or not in accordance with the agreed terms of a loan
- overdue_principal (double precision): The amount of the principal on a loan that is past due or has not been paid by the agreed-upon date
- overdue_interest (double precision): The amount of interest that has accumulated on a loan that is past its due date
- to_collect_principal (double precision): The amount of the principal on a loan needed to be collected
- to_collect_interest (double precision): The amount of the interest on a loan needed to be collected
- interest_on_overdue_loan (double precision): The amount of interest accrued on a loan that is past its due date
- unpaid_principal (double precision): The remaining amount of the principal (original loan amount) that has not yet been paid back by the borrower (customer)
- unpaid_interest (double precision): The amount of interest on a loan that has not yet been paid by the customer
- description (text): A detailed description of a specific loan
- created_time (date): Loan origination date
- release_time (text): Loan disbursement date
- overdue_date (date): Loan payment due date

- disbursement_date (text): The date when the loan is released
- modified_time (text): The time when a loan record was last modified or updated in the system

f. **Savings Account Transaction**: Acts as a detailed record of monetary movement (Deposit, Withdrawal) of a customer's savings account. Data are generated randomly and stored as a table in an external database in the following schema:

- transaction_id (text): Unique ID of a transaction in a savings account
- savings_account_id (text): Unique ID of a savings account
- transaction_date (date): The specific date when a transaction occurred
- transaction_type (text): Type of transactions
- transaction_amount (double precision): Value of the transaction amount
- balance_after (double precision): Balance after transaction
- created_at (timestamp): The exact timestamp that the customer opens a savings account
- updated_at (timestamp): The exact timestamp that the customer updates a savings account
- description (text): A detailed description of a savings account
- seller_id (text): Unique ID of the seller of the savings account
- channel (text): Channel where savings account transactions occur

g. **Loan Payment**: Track a borrower's payment history and identify any delinquencies. Apply payments to the principal and interest portions of the loan. Generate statements reflecting loan payments made and the remaining balance. Data are generated randomly and stored as a table in an external database in the following schema:

- scheduled_payment_date (date): The predetermined date on which a specific payment is expected to be processed
- payment_amount (double precision): The value of a payment made
- principal_amount (double precision): The original amount of money borrowed
- interest_amount (double precision): Interest value
- paid_amount (double precision): Loan payment that has been paid
- paid_date (date): Payment day

- loan_payment_id (serial): Unique loan payment ID

h. **Saving Products**: Retail banks offer a variety of saving products to suit different customer needs and financial goals, with characteristics like interest rates, fees, accessibility, and liquidity. Data are collected from VPBank's product page and stored in our own built REST API in the following schema:
   - product_id (text): Unique ID of a savings product
   - product_name (text): Savings product name
   - min_term (bigint): The minimum term required for a savings product offered by the bank
   - max_term (bigint): The maximum term required for a savings product offered by the bank
   - interest_rate (double precision): Interest arising from the savings
   - min_deposit (bigint): The minimum initial deposit amount required to open or maintain a specific product.
   - deposit_method (text): Method of deposit
   - interest_payment (text): The value of the interest on the loan
   - promotion (text):
   - created_at (timestamp): The exact timestamp that the customer opens a savings account
   - updated_at (timestamp): The exact timestamp that the customer updates a savings account

i. **ATM**: Stores information about the bank's Automated Teller Machines (ATMs). It serves as a central hub for managing and monitoring information, specifications, and location of ATMs. Data are collected from VPBank's official source and stored in our own built REST API in the following schema:
   - address (text): Exact location of an ATM
   - atm_247_label (text): A particular transaction was performed at an ATM available 24 hours a day
   - atm_id (text): Unique ID of ATM
   - is_atm (boolean): A particular transaction was conducted using an ATM

- is_atm_247 (boolean): A particular transaction was conducted using an ATM 24/7
- is_branch (boolean): Check if the ATM belongs to any branches
- is_cdm (boolean):
- is_household (boolean):
- is_sme (boolean):
- latitude (text): Latitude of the ATM
- longitude (text): Longitude of the ATM
- name (text): ATM name
- phone_number (text): ATM phone number

j. **Bank Branch**: Acts as a digital record of all the bank's physical locations. It provides essential information for managing branch operations, customer service, and resource allocation. Data are collected from VPBank's official source and stored as CSV files in S3 buckets in the following schema:

- region (text): Brand region
- province (text): Brand province
- branch_name (text): Brand name
- branch_id (text): Unique ID of a bank branch
- address (text): Detailed address

2. **Data Ingestion and Integration pipeline for Relational and Graph Database**

a. Technologies: AWS Glue, AWS Lambda

b. **Data Extraction, Transformation, and Loading Process (ETL)**

i. Create Glue Connections, Catalog Databases, and Crawler

First, we create Glue connections for External Databases, these connections serve as both source and output of data. Next, we create Catalog Databases to store metadata of data sources. Finally, we create Crawlers to parse data sources, read their metadata, and populate Catalog Databases in the form of catalog tables. These catalog tables not only serve as a representation of source data but can also be used inside Glue script to quickly reference files and tables without specifying data connection and schema. Catalog tables can also serve as a metadata layer that AWS Athena uses to grant the ability to query data straight from files.

ii. **Create Glue Script**

Since we already have a relational database serving as the Data Warehouse and a dedicated dbt project for further transformation and analytics purposes, Glue will mostly serve as a pipeline to integrate data from source to Databases (Postgres and Neo4j) with basic transformation, almost 1:1 replication of source data. Most of the script will be run in Spark environments. All Glue scripts used in this solution will be stored in the submitted folder.

- Read catalog data: Data stored in files (CSV, XML) and external databases that we have previously created catalog tables can be quickly referenced by specifying the catalog database and table, read data will stored in Glue's Dynamic Frame.
- Read from REST API: We use *requests* library to make API calls and returned JSONs are read into Spark Dataframe, these Spark Dataframe are then converted to Glue's Dynamic Frame.
- Write to Relational database (Postgres): Postgres databases are supported sources for Crawler, so we can insert to tables using catalog information.
- Write to Graph Database (Neo4j): We use the *neomodel* library to establish connections to the Neo4j instance and build functions to construct Cypher queries based on Dataframe to populate Neo4j with nodes and relationships.

iii. **Create Workflow**

We create a Workflow to populate the entirety of Relational and Graph databases, starting with a scheduled trigger that triggers the workflow every day at midnight, then concurrently running jobs to integrate data from source to databases, jobs that are dependent on one or multiple upstreams will be placed as watchers for conditional triggers to make sure that upstreams jobs finished before running.

3. **Setup Postgres on RDS**
    a. **Specification:** Considering the scale of datasets, transformation, analytical workloads, and the $200 budget, we choose Postgres hosted on an m5.large instance. The instance is deployed on a single AZ for cost-saving measures (We recommend multi-AZ for production deployment). A comparison between this approach and other alternatives will be discussed in the upcoming section.
    b. **Network configuration**: Considering the fact that our team works remotely most of the time, during the development process the instance are open for public access. This can be easily changed by restricting access to certain IPs in subnets in which our database is hosted.

4. **Setup Neo4j on EC2**
    a. **Specification:** Considering the scale of datasets, transformation, analytical workloads, and the $200 budget, we choose to host an instance of Neo4j on an m5.large EC2 machine. A comparison between this approach and other alternatives will be discussed in the upcoming section.
    b. **Setup**: We configure the EC2 machine to run on the Ubuntu platform. The process of installing Neo4j is identical to installing it on a local Ubuntu machine with an additional script to start the database on machine startup. For further cost savings, we set up a Lambda function to turn off the host machine during non-working hours.
    c. **Network configuration**: Considering the fact that our team works remotely most of the time, during the development process the instances are open for public access. This can be easily changed by restricting access to certain IPs in subnets in which our database is hosted.
    d. **Transformation**: All transformations are handled by Glue, data inside Neo4j are ready to query from the start.

5. **Kubernetes Cluster on K8s**
    a. **Specification:** Considering the scale of datasets, transformation, analytical workloads, and the $200 budget, we decided to deploy a single k8s cluster with a node group consisting of 3 m5.large machines.

b. **Cluster setup and configuration**: We use *eksctl* commands to quickly configure and set up the cluster with desired node group specification and required dependencies. For the scope of this solution, the EBS CSI add-on and additional AWS roles for AWS Application Load Balancing are required and also installed through eksctl commands.

c. **Custom Docker images for deployment**: First we set up Docker image repositories on ECR to store custom images. All application images (dbt docs, API server) and custom images (Airflow) are developed on GitHub / GitLab projects, we also set up GitHub actions / GitLab pipelines to quickly build images and push them to repositories based on changes on certain branches.

d. **Deploy applications and services**: All applications and services are deployed via Helm charts. For deployments containing UI or API that need to be exposed for internet access, we deploy AWS Application Load Balancer for them (For an easy development process, all services are publicly accessible). All helm charts are stored in the submitted folder. Here are the applications and services deployed on EKS and their helm chart configurations:

   ○ dbt docs (dbt project documentation interface): This one is just an nginx service to host an HTML web page, not many complex configurations are needed.

   ○ Airflow: We use a custom Airflow image with pre-installed dependencies like git and several Python libraries so that it can pull dbt projects from git for model execution. We configured gitsync to quickly sync production DAGs with local development without the need to update deployment. Finally, we use an external Postgres instance for metadata instead of deploying a database alongside Airflow. We also deploy a balancer to grant internet access to Airflow UI.

   ○ Superset: We deploy a balancer to grant internet access to Superset UI

   ○ API server: We deploy a balancer to grant internet access to API calls

6. **Data Transformation**

   a. **Environments:**

   - **Data Import**

     - Internal data: Stores data that the dev team creates internally according to the schema:

        - vpb_user: Summary of tables belonging to the data stream about users logged by the dev team.
- External data: stores data controlled by the outside world, not created by devs, but usually pulled in by API/crawl, or created by teams themselves, stored in schemas:
    - ff (from file): saves all data sources coming from importing Google Sheets - data in Google is unstable, so the data team may not be the one controlling the data quality of this type of data

b. Data transformation

- **dwh_stg**
    - **dwh_stg.stg_ (staging):** saves the first transform tables of the imported data for the purpose of cleaning or restructuring the data for ease of use. If the imported data generates a table in the source schema, all transform code. Others do not use the original import data but call it from the source table to use
    - **dwh_stg.int_ (intermediate):** where to store intermediate calculation tables as fields calculated from the stg table.
- **dwh (data mart):** a place to store tables designed according to dimensional modeling, oriented for end-user use. Besides, to make it convenient for users to easily search tables by functions and tasks, dwh is organized as a sub-schema from dwh as follows:
    - dwh_savings: saves data tables
    - dwh_loan: saves data tables for margin lending products
- **Sensitive data:**
    - [Schema_name]_pii: separate real and encrypted data of PII
- **Virtual DB on BI Tools** (e.g: Superset, PowerBI, ...): gives access (row-level if necessary) to users to explore data; Data comes from a single source, the data schema (data mart) and is decentralized to row-level by group

c. Rules for building data transform tables

- **dwh (mart):** a place to store tables designed according to dimensional modeling, oriented for end-user use. Besides, to make it convenient for users to easily search tables by functions and tasks, dwh is organized as a sub-schema from dwh as follows:

- Dim table: Is a table that stores information about the attributes (characteristics/properties) of an object.
- Fact table: Fact tables usually include two types of fields: dimension and measure, of which there is always a date dimension (time in general, which can be a year, month, quarter, hour, minute, second, not just the date format).

- **A fact table can lack measures - this is called a factless table.** The fact table does not have a PK but accepts FK fields as unique to the table. Distinguish three main types of fact tables:
    - fact_[entity_name] (Transaction fact table) stores each event/event_transaction that occurs -> is considered an atomic table, the foundation for building other fact tables,
        - Each event record stores data only once. It includes a "created_at" field indicating the event time. Typically, the number of rows in the fact transaction table matches the source table.,
        - It can be considered the source table but in a simpler form and absolutely does not store data in an editable way. If there is a change in the value of a transaction at a time after the transaction is created, it needs to be created. The new record table to save this edited value does not contain aggregate measures, but measure only has meaning as a parameter of the event
    - fact_period_[entity_name]_daily/weekly (period snapshot fact table) is a summary table from the transaction fact table. From this table, you can select an entity according to your needs to summarize with a period (day, week, month, 15 days,...), which can be saved. Many types of periods are in one table or separate.

d. Monitor and Scheduling

We use Airflow to orchestrate and trigger dbt runs for the relational database. All dbt models are developed and stored within a GitHub/GitLab repository, which Airflow can pull into its local directory and trigger them using the same dbt command on a local machine. Models are executed using dbt commands, which are placed inside Airflow Bash Operator tasks and can be scheduled and run based on configuration. Since all of the computation will be handled by our relational database, Airflow can run on relatively little resources.

7. **API:**

   a. **Overview**

   The API, built on the Django framework and following a three-tier architecture, is designed to handle requests from third-party clients, including those with filtering criteria.

   b. **Key features**

- Upon receiving a request, the API processes two main sections: query and order by.
- The query section of the API comprises three elements:
  + Field name
  + Expression (such as greater than, equal to, less than, in, etc.)
  + Value for comparison
- The order by section includes two elements:
  + Columns
  + Sort order (ascending or descending)
- For the query section, the API accepts all fields, each with its associated expression and corresponding value, processing them to create a conditional query.
- For the order by section, the API accepts the columns and the sort order (ascending or descending).
- Finally, the API constructs the complete query for the Redshift database.
- This API will be containerized using Docker and published to the Elastic Container Registry (ECR) and deployed on EKS.

8. **Web application**

   a. **Overview**

   The web application, developed using Django for the backend and ReactJS for the front end, is designed to efficiently manage and analyze customer data. It offers a comprehensive suite of features to support businesses in viewing existing customers, identifying potential customers, examining detailed customer profiles, and recommending products tailored to customer needs.

b. Key features

- Data Filtering and API Builder: Our web application serves as a means to preview data, with the ability to create filters and construct API calls for said table with API arguments change corresponding to filters used.
- Customer Management: Users can view a complete list of current customers, with options to filter and sort based on various criteria such as location, and transaction history, …
- Customer Profiles: Detailed profiles for each customer are available, displaying critical information such as contact details, account balances, transaction history, loan status, and feedback. This feature helps bank employees gain deeper insights into individual customers and personalize their interactions.
- Product Recommendations: Leveraging the data from customer profiles, the application provides personalized recommendations for banking products and services. These suggestions are based on the customer's financial activities, account status, and expressed preferences, aiming to enhance customer satisfaction and increase the uptake of bank offerings.
- Interactive Interface: The combination of Django and ReactJS ensures a seamless, interactive user experience. ReactJS handles the dynamic, responsive front-end interactions, while Django manages the robust backend processes, including database interactions and user authentication.
- Scalability: Built with scalability in mind, the application can handle growing amounts of data and user activity, making it suitable for banks of varying sizes.

Cost Estimation

1. **AWS Glue:** Considering that VPBank has around 50 procedures/subproducts spanning 1000 tables with about 1TB of new data every day, considering that we only integrate the database and use basic transformation. This will translate to about 100 jobs using 5 DPUs with each of them running for 10 minutes, and about 50 crawlers using 5 DPUs with each of them running for 5 minutes, all with a monthly cost of
   a. Jobs: 100 Jobs x 5 DPUs x 10 minutes x 30 days x 0.44 USD per DPU-Hour = 1,100 USD

  b. Crawlers: 50 Crawlers x 5 DPUs x 5 minutes x 30 days x 0.44 USD per DPU-Hour = 225 USD
  c. Overall: 1,375 USD / Month

2. **Postgres RDS**: Considering that VPBank has around 50 procedures/subproducts spanning 1000 tables with about 1TB of new data every day, and currently you guys only store data that are less than 90 days old. Of course, it's difficult to estimate the average table amount of columns and rows, but we can estimate the maximum size of the data by joining all data into 1 giant table, this table will have roughly 100 text columns (using 256 bytes for 1 cell), 900 numeric columns (using 8 bytes for 1 cell) and 150.000.000 records. This amount of data will require at least db.m5d.4xlarge (vCPU: 16, Memory: 64GiB) instances to be able to sufficiently transform and query data, for production we should also expect the database to be multi-AZ and have at least 2 read replicas for faster query time. So overall we can say that the cost will be massive
    a. Data size: 8 * 900 * 150,000,000 + 256 * 100 * 150,000,000 = 4.47 TB (* 1.5 = 6860 GB for future proof)
    b. Data cost: 6,860 GB per month x 0.23 USD x 2 instances = 3,155.60 USD
    c. Instances: 2 instance(s) x 3.352 USD hourly x (50 / 100 Utilized/Month) x 730 hours in a month = 2446.9600 USD
    d. Overall: 5,600 USD

3. **Neo4j on EC2**: Considering the same data from the above estimation, only 10% of them (based on product) will be converted to Graph data, this translates to roughly 25 million nodes and 125 million relationships. For this amount of data, a c6g.4xlarge (vCPU: 16, Memory: 32GiB) will be sufficient. We will need at least 2 instances of the database for backup measures.
    a. Data size: 150,000,000 * (8 * 9 + 256 * 1) = 45.82 GB (*1,5 = 70 GB for future proof)
    b. Data cost: $75 (EBS storage and Data transfer)
    c. Instances: 2 instances x 0.544 USD On Demand x 365 hours in a month = 400 USD
    d. Overall: 475 USD

4. **EKS Cluster:** Out of the bat we need 73 USD just to keep 1 cluster managed. Since we host a bunch of services on the cluster (Including an expected-to-be-resource-intensive API server), a worker group consisting of 3 to 5 c6g.4xlarge (vCPU: 16, Memory: 32GiB) will be sufficient.
    a. Managed cluster: 1 cluster * 73 = 73 USD

   b. Workers: 3 instances x 0.544 USD On Demand hourly cost x 365 hours in a month = 595.680000 USD
   c. Overall: 670 USD
5. **Total cost**: 1,375 + 5,600 + 475 + 670 = 8,120 USD per month

Future Improvements

1. **ETL pipelines**
   a. So what are the drawbacks?: The use of relational database. When VPBank's user base grows and our data increases in size, soon traditional relational databases won't be able to comfortably transform data by batch within an appropriate time frame, to accommodate the amount of data, and both computation instance and storage cost will increase (to an extend that is not viable for business). The same case will happen for columnar database solutions like Redshift (Maybe the computational capability is better but the operating cost will still be a deal-breaker).
   b. Is there any solution?: Yes, instead of placing the heavy parts of transformation on relational databases, we can leverage Spark for faster, serverless computation. Instead of using SSD storage for databases, we can use S3 as storage. The new approach should look like this:
      - Use Glue and Spark to extract, and transform data from sources, but instead of a database, load them into S3 buckets.
      - Further data transformation and analytics will also be handled by Glue, staging, intermediate, and marts tables will also be stored in S3.
      - Use a Table format (For example: Apache Iceberg) to add metadata to S3 folders, files, and partitions, which allows us to see those files as tables.
      - Use a query service (For example AWS Athena) to query data straight from S3, this service serves as the Relational Database in the eyes of Data Analysts

      Or if you dare to step out of the AWS ecosystem, Databricks can do all of these within a single platform.
   c. What are the benefits of the new approach?

- Spark is faster, Glue is serverless, and we can dynamically assign resources for large jobs, handling concurrent computation is more cost-efficient than a relational database.
- S3 for storage is vastly cheaper than Database, and mostly charged for I/O means it is perfect for storing more historical data for backup purposes.
- Our data are truly gathered in the same place, both structured and unstructured data.
- Combine the ETL process and Transformation/Analytics process into a single coherent pipeline, with the use of Spark SQL, Data Analysts/Business Intelligence and Data Engineers can largely communicate through the same language of SQL.