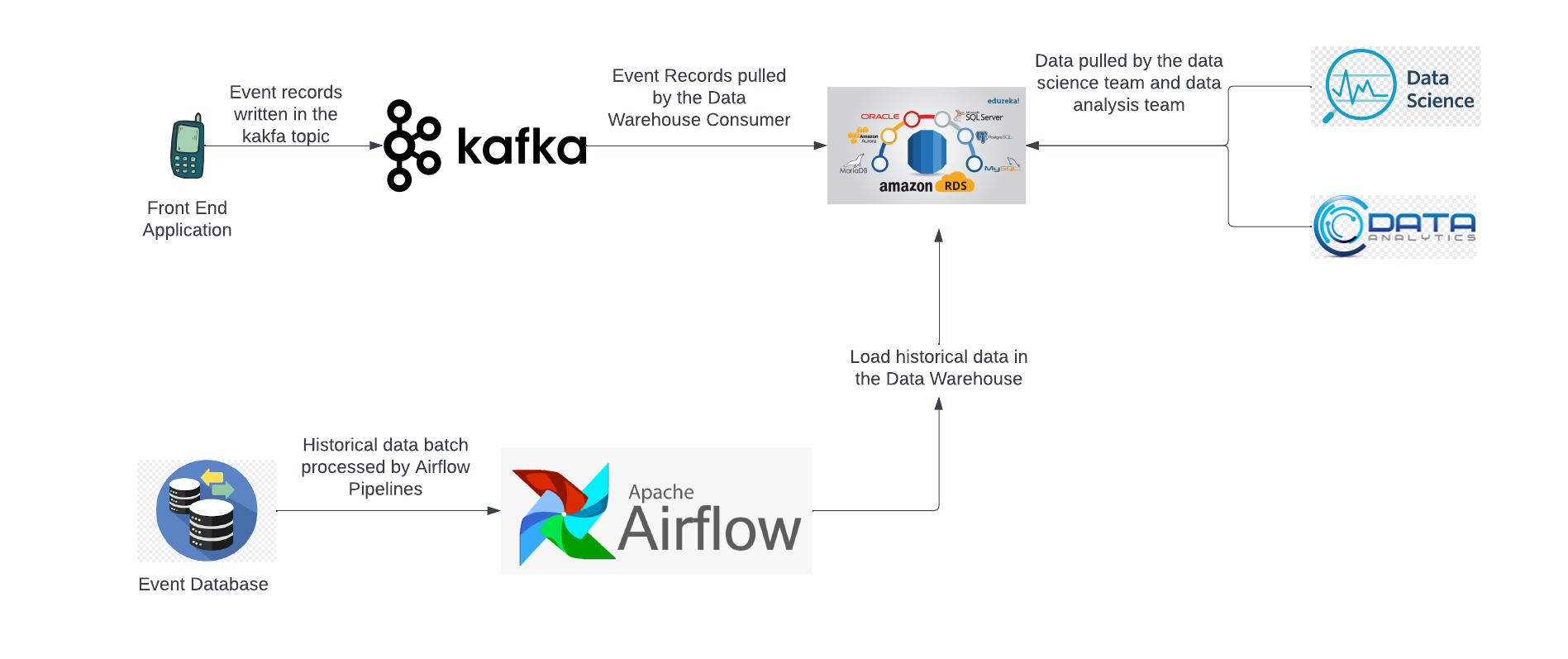
**System Design:** Following is the high level system design to stream and batch process the events data.



We have the following 2 types of data that needs to be populated in the warehouse for Data Science and Data analytics team to perform their respective actions:

1. Real time new events
2. Historical Data (for Backfilling)

The 2 types of data possibilities leads to usage of 2 different set of pipelines for ETL’ing the events data in the Data Warehouse(referred as RDS in the rest of the document)

1. **Real time Streaming Pipeline:**

This pipeline consists of the following components:

1. Front End application (Producer) : The application that generates the event based data and publishes
2. Kafka Cluster: The 2-3 node kafka cluster will act as a broker between the producer (front end in this case) and the RDS
3. Data Warehouse: Data Store where all of the data is being housed. AWS RDS in this case.

* **Workings:**

1. The producer at the front end generates and tracks events and publishes the event data in corresponding Kafka Topics.
2. The data is then read by the RDS consumer and pushed in the RDS.
3. **Kafka Cluster Architecture**: Depending upon the frequency of the data we can have an n-node cluster. Judging by the event volume given in the assessment, 2 node Kafka clusters would suffice the purpose.
4. **Batch Processing:**

To backfill and get the historical data we can create batch pipelines to transfer the data from the source DB to RDS.

**Partition Strategy:** Depending upon the volume of the data we can create a partition strategy to efficiently retrieve all of the DBs. For example, the given table describes the feasibility of extraction, based upon the batch size. Creating a batch of millions of records might lead to the failure of source or the pipeline. Hence, depending upon the size of the historical data, we can create appropriate partition strategy for batch processing

| **Partition** | **Number of records** | **Batch extraction feasibility** |
| --- | --- | --- |
| yearly | 10 millions | NO |
| 6 months | 5 million | NO |
| 3 months | 2.5 million | NO |
| 1 month | 1 million | YES |

After a partition strategy is decided, the data can be then extracted using the pipelines orchestrated in airflow. We can have multiple daily/monthly/weekly/yearly pipelines depending on the batch size for each merchant. For example, there will be 10 different pipelines running for 10 different merchants, each pipeline processing custom batch partitions.

**Technical Aspects:**

1. **Technology selections:** Following are the technology sections and their reason is defined in the section following this one:

a. Apache Kafka for data streaming

b. AWS RDS for Data Warehouse. It has the following benefits:

### Backup & Recovery

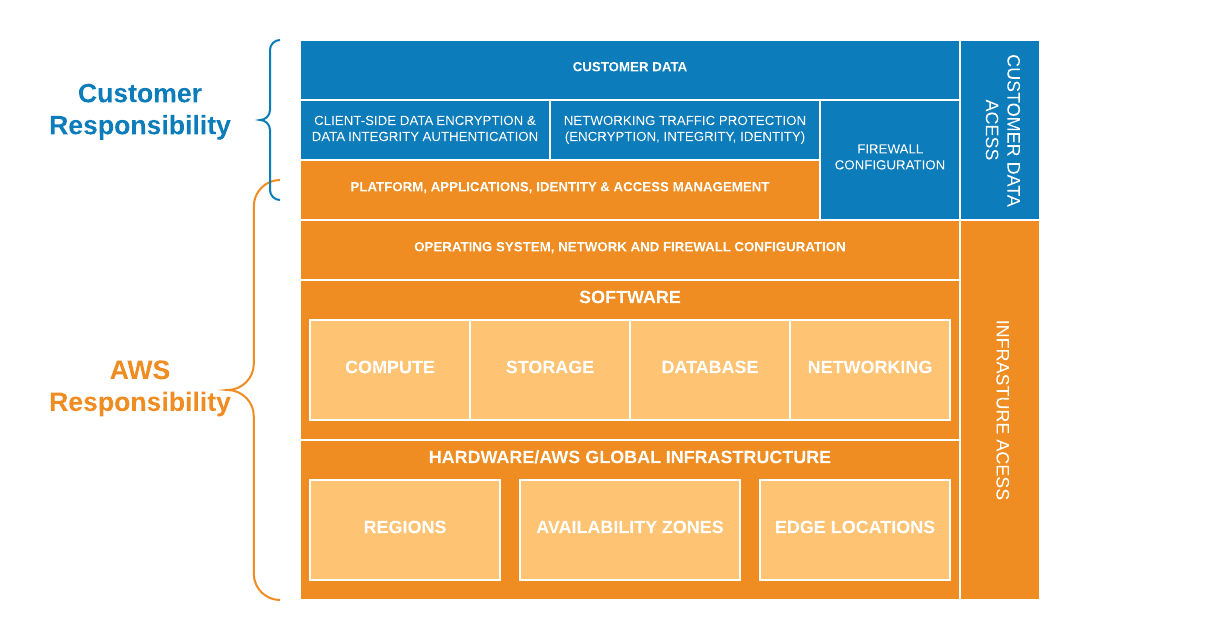
### High Availability

### Monitoring/Metrics

### Easy Deployment

### Fast Storage Options

### Security



c. Apache Airflow for pipeline orchestration

1. **Latency:** Kafka in itself provides low latency of the range of milliseconds.
2. **Scalability:** Using Apache Airflow for batch processing helps in spawning multiple pipelines for multiple merchants and on top of that using some managed service provider of Airflow like AWS Managed Apache Airflow will reduce the overhead of scaling and maintaining Airflow.

The intelligent auto partitioning logic easily helps in scaling down or up the pipeline nodes based upon the size of the batch/partition i.e. if the size of historical data is huge, the partitioning logic will easily divide it into considerable smaller chunks for the pipeline to run effectively

1. **Failure Modes:** Since Kafka is itself very robust and failure resilient as well having 2-3 kafka nodes in a cluster will help a long way in ensuring the cluster never goes entirely and the data gets replicated in multiple nodes for backup. In case of a failure of the main node in the cluster, the backup node can then act as the main node.

For Apache Airflow pipelines, if any of the pipelines fail, the pipeline can be re-triggered for the batch that it failed for.

1. **Delivery Guarantees:**

* The usage of Kafka idempotent writes guarantees the uniqueness of every record written it
* Retry mechanism at the producer along with Kafka idempotent writes guarantees all the records are written at least once in the kafka topic. Coupled with the multi node cluster guarantees the high availability of the Kafka Cluster
* The RDS can have multiple read replicas configured, which in case of a failure act as the main source of the database.