Overview of the Problem Statement and Solution

Problem Statement:

Analyzing customer data to segment them into clusters, defining groups of the most affluent and likely-to-spend customers. This facilitates targeted marketing and product adjustments to meet customer-specific needs. This analysis helps identify who spends more and who waits for offers to make purchases, enabling the presentation of tailored products to each user. This facilitates targeted marketing and product adjustments to meet customer-specific needs.

Proposed Solution:

- Analyze the data to identify the most relevant features and understand their relationships to the target objective.
- Apply preprocessing techniques such as handling missing values, outlier treatment, standardization (e.g., using StandardScaler), and dimensionality reduction techniques like PCA.
- Use these prepared features to achieve an optimal clustering outcome that reveals actionable customer segments for targeted marketing strategies.
- Perform customer segmentation using clustering techniques.
- Only for demonstration: Create labels as attributes to enhance the demonstration of various modeling and training methodologies

Key Steps, Challenges, and Resolutions

Key Steps:

1. Data preprocessing (handling missing values, scaling, feature transformation, etc.).
2. Different types of Analysis
3. Clustering for segmentation, subsequently to generate labels to demonstrate other different strategies.
4. Model training and evaluation.
5. Implementation of Dev-Staging-Prod environments for robust MLOps pipelines.
6. Real-time or near-real-time requirement

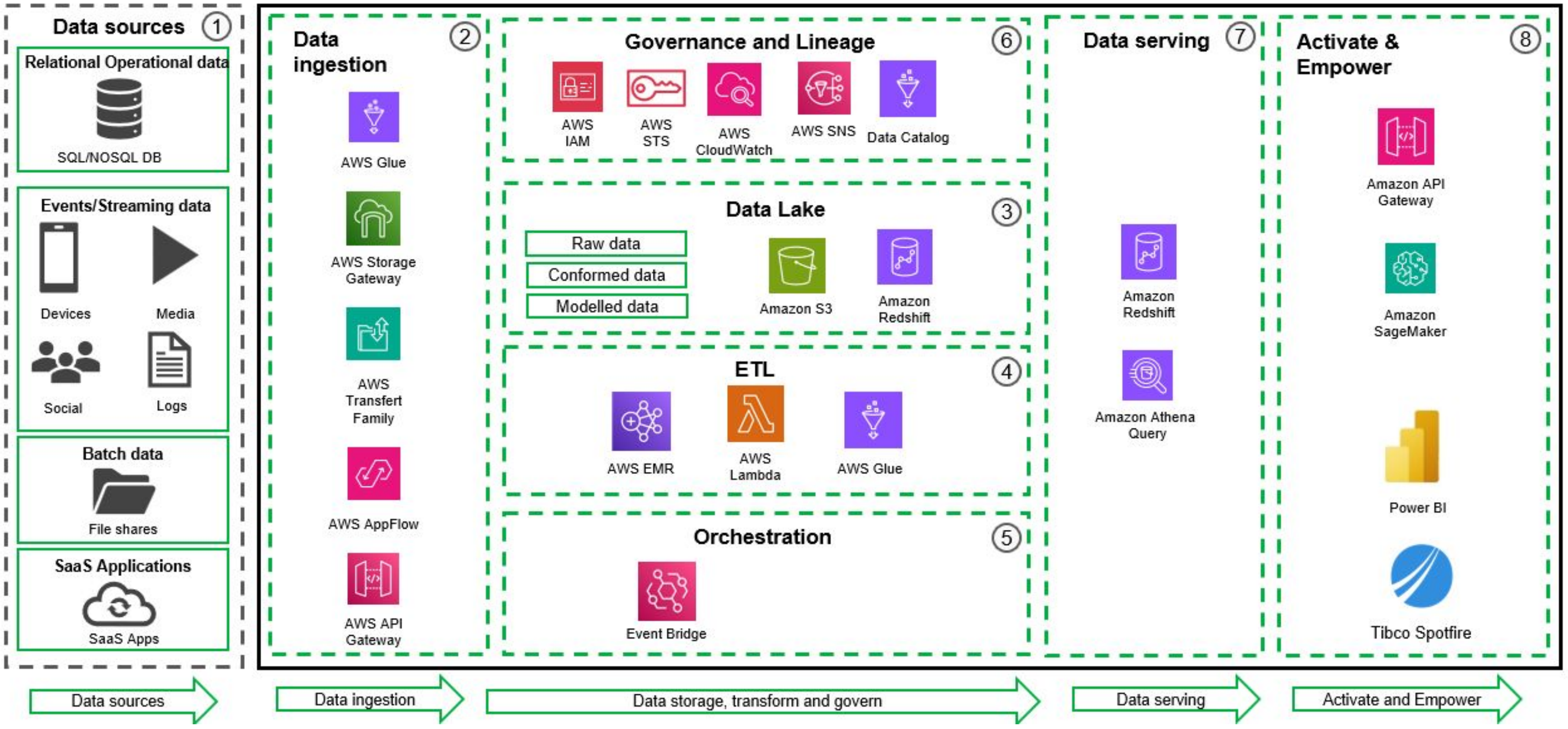
Challenges:

- Handling noisy and incomplete data.
- Managing features with different scales and units.
- Ensuring scalability and reproducibility across environments.

Resolutions:

- Utilized robust preprocessing pipelines.
- Applied feature engineering to standardize and transform data.
- Structured environments for development, testing, and production to isolate and validate changes.
- Implementation of a distributed solution with PySpark on cloud platform such as Databricks

SCENARIO 1 - FULL CLOUD AWS



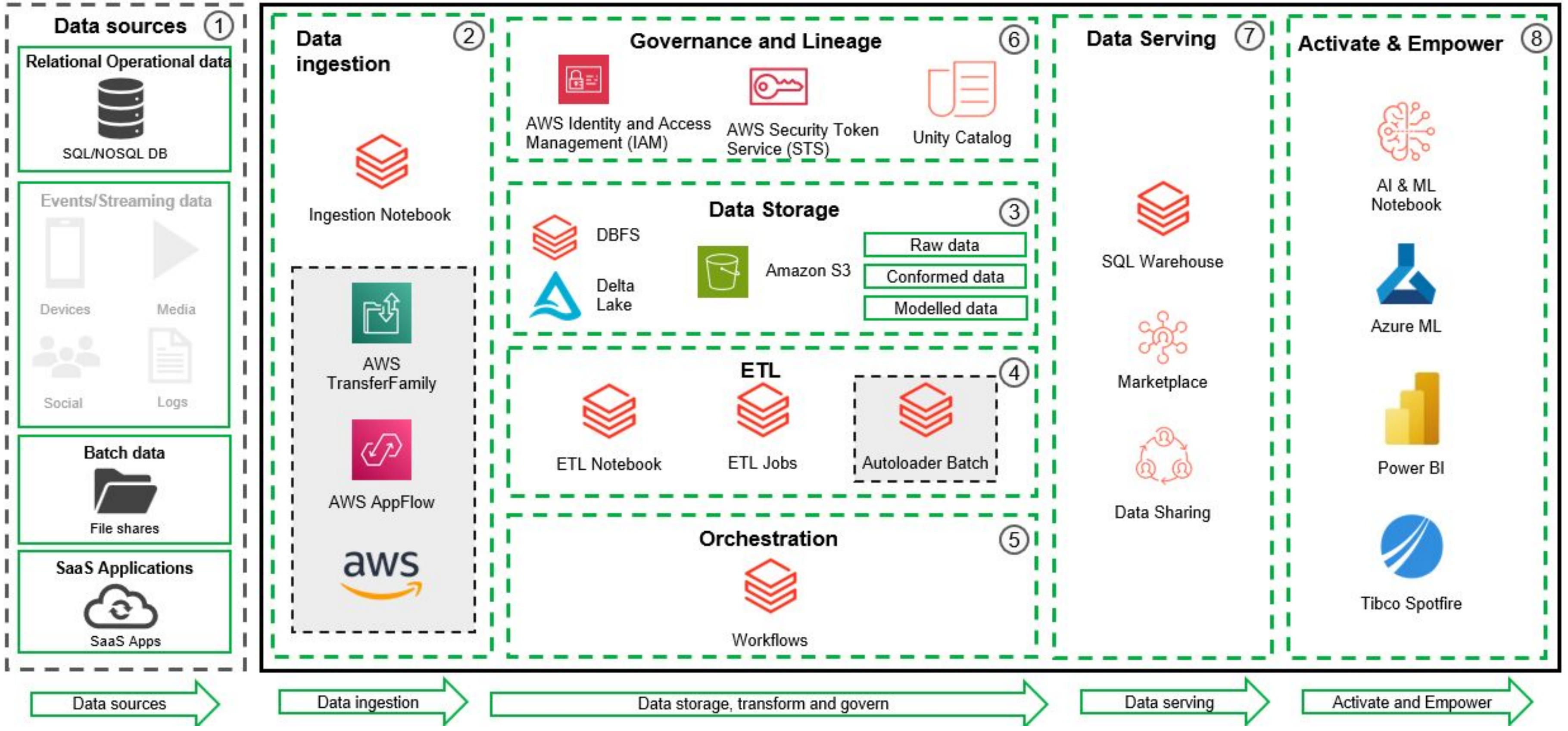
AWS Full Cloud Architecture:

The architecture comprises:

1. **Data Sources**: Relational databases, event streams, batch data, and SaaS applications.
   - Example: SQL/NoSQL **databases**, **Kafka** for event data, **AWS AppFlow** for SaaS integration.
2. **Data Ingestion**:
   - **AWS Glue**: For ETL pipelines.
   - **AWS Storage Gateway**: For batch data transfer.
   - **AWS Transfer Family**: Secure file transfers.
   - **Amazon API Gateway**: For real-time data ingestion.
3. **Data Lake**:
   - **Amazon S3**: Stores raw, conformed, and modeled data.
   - Organized into Bronze (raw), Silver (cleaned), and Gold (aggregated) tiers.
   - **Amazon Redshift**
4. **ETL**:
   - **AWS Lambda** and **AWS EMR**: Processing and transformation tasks.
5. **Orchestration**:
   - **AWS EventBridge**: Triggers workflows for ingestion, transformation, and serving.
6. **Governance and Lineage**:
   - **AWS IAM and CloudWatch**: Secure access and monitoring.
   - **AWS STS e SNS:** security-token-service, simple-notification-service
7. **Data Serving**:
   - **Amazon Redshift and Athena**: Query and analytics.
8. **Activate & Empower**:
   - **Amazon SageMaker**: For model training and deployment.
   - **API Gateway**: Hosting prediction services.

DATABRICKS + AWS

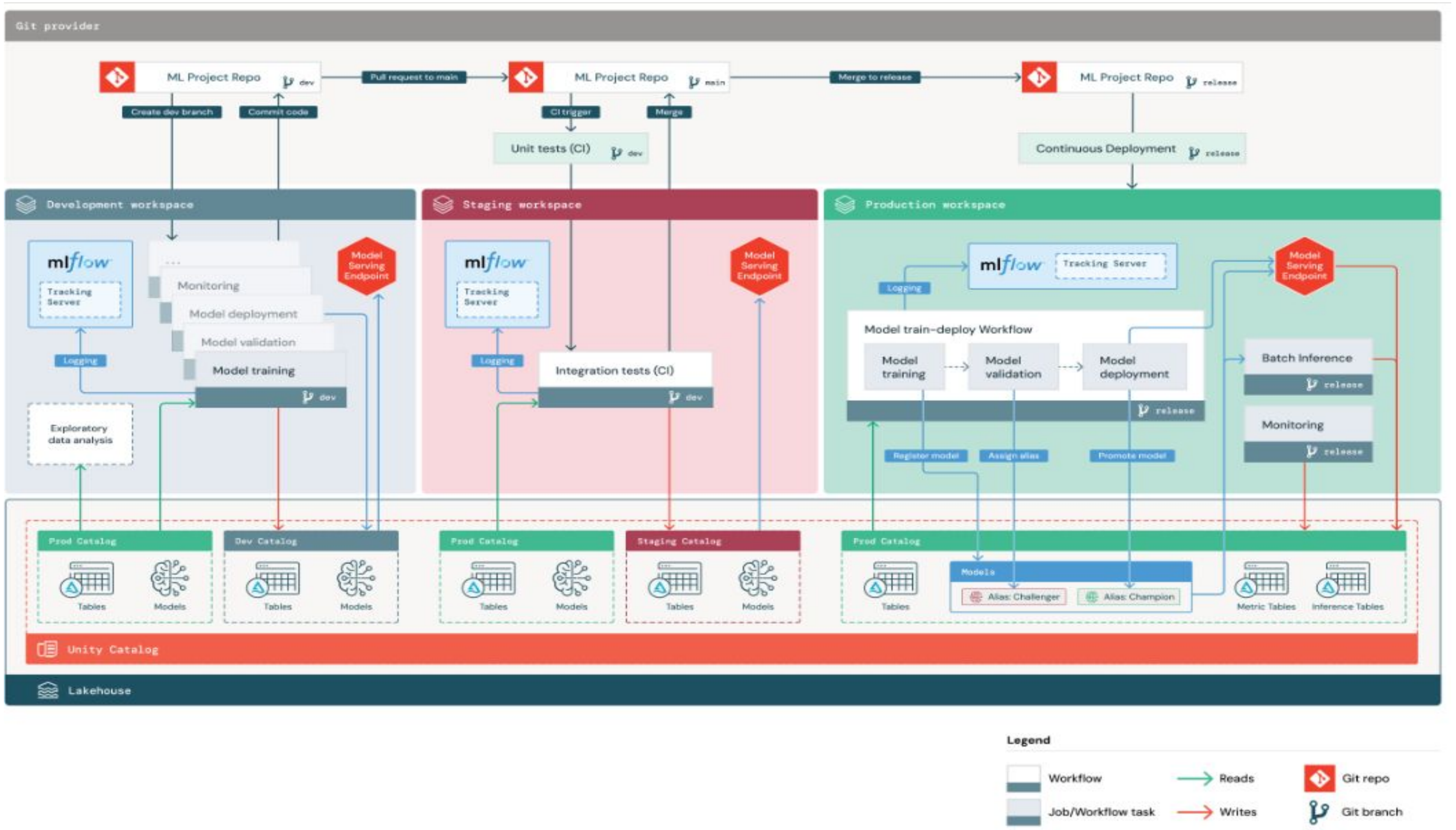
The use of AWS tools to make ingestion is optional and closely related to Databricks Autolaoder. In fact, the Autolaoder allows with triggers to start ETL processes as soon as a file is deposited on a storage location.



Databricks + AWS Architecture:

1. **Data Sources**:
   - Same as AWS architecture.
2. **Data Ingestion**:
   - **Databricks Autoloader**: Efficiently ingests files from cloud storage.
   - **AWS AppFlow and Transfer Family**: Additional ingestion options.
3. **Data Storage**:
   - **DBFS**: Databricks File System for processing.
   - **Delta Lake**: Bronze, Silver, Gold schema for data organization.
4. **ETL**:
   - **ETL Notebooks and Jobs**: Automated workflows for transformation.
5. **Orchestration**:
   - **Databricks Workflows**: For managing pipeline tasks.
6. **Governance and Lineage**:
   - **Unity Catalog**: Centralized data and model management.
7. **Data Serving**:
   - **Databricks SQL Warehouse**: Analytics and querying.
8. **Activate & Empower**:
   - **Azure ML**: Model training and inference.
   - **Power BI and Tibco Spotfire**: Visualization tools.

Organization of Assets: Dev-Staging-Prod Environments



1. Dev Catalog (development):

- For assets in development.
- More open access to allow developers to experiment freely.

2. Staging Catalog (pre-production):

- Used for assets in testing and validation.
- Used as an area for writing resources produced during the code test in the staging environment before introduction into production environment
- This catalog could also be used to store more permanent pre-production the activities to reflect the production environment during integration tests
- Data and templates stored in the staging catalog may be temporary, or cleaned on a regular basis
- Limited access to writing, as well as administrators and services directors
- Read access should be enabled for users who may need to debug integration tests

3. Prod Catalog (production):

- Contains assets used in production.
- Access to write to the prod catalog is typically limited to a small number of administrators or service principals

Within each catalog, assets can be further organized into schema that reflect the level of data processing, according to Medallion Architecture:

- **Bronze**:
  - Represents raw data, imported directly from the original sources without any processing.
  - Used to store data in its most authentic form, preserving all original information.

- **Silver**:
  - Contains already cleaned or partially processed data, derived from the raw bronze level data.
  - Errors, null or duplicate values are removed and data is standardized.

- **Gold**:
  - Includes fully transformed, enriched and aggregated data.
  - Ready for advanced analysis, reporting or training of machine learning models.
  - Can contain the feature tables ready for model training. This structure facilitates quality control at each stage of the data pipeline

Model Explanation

Model Choice:

The problem was approached as an **unsupervised learning** task to represent scenarios without labeled data. Clustering was selected to segment customers and create meaningful groupings. These cluster labels were later used for illustrative purposes in supervised modeling.

Why K-Means Was Chosen:

K-Means, a **partitioning clustering method**, was chosen for its simplicity and scalability. It works by:

- Dividing data into k clusters using centroids to minimize within-cluster variance.
- Using the Elbow and Silhouette methods to determine the optimal number of clusters.

This approach identified meaningful customer segments:

- High-value customers for targeted marketing.
- Price-sensitive customers driven by discounts.
- Moderate spenders with balanced purchasing behavior.

Data Selection and Preprocessing

Dataset:

This dataset containing attributes like income, spending habits, and promotional responses. (all details are available on test.ipynb)

Justification:

- Rich features for customer segmentation.
- Balanced distribution across multiple dimensions.

Preprocessing Steps:

1. **Handle Missing Values**: Removed rows with missing income values.
2. **Outlier Management**: Outliers were treated by capping values beyond the 95th percentile.
3. **Feature Transformation**: Converted birth year to age.
4. **Standardization**: Applied StandardScaler to scale numerical attributes.
5. **Feature Selection**: Focused on spending and demographic attributes most relevant to segmentation.

Challenges Addressed:

- Missing values managed through elimination.
- Outliers minimized via capping and scaling.
- Standardization improved clustering performance.
- Feature engineering enhanced interpretability. Save specific features into **Feature Store**

How the Model Addresses Real-Time or Low-Latency Requirements:

To handle real-time or low-latency requirements effectively, the following best practices were employed:

1. **Optimized Infrastructure**:
   - Leveraging serverless computing platforms such as AWS Lambda or Databricks Model Serving for scaling predictions on demand.
   - Using GPU-enabled instances (e.g., AWS EC2 with GPUs) for high-performance inference during peak loads.
2. **Efficient Data Handling**:
   - Utilizing online feature stores (e.g., **AWS SageMaker Feature Store** or **Databricks Feature Store**) to precompute and store freque accessed features, reducing computation overhead at runtime.
3. **Model Optimization**:
   - Employing lightweight, optimized models (e.g., Logistic Regression or quantized versions of neural networks) to minimize latency.
   - Using frameworks like ONNX to streamline inference across different platforms.
4. **Asynchronous Processing**:
   - Implementing asynchronous inference pipelines to decouple request-response handling, enabling faster response times for critical requests.
5. **Caching Mechanisms**:
   - Caching results of frequently requested predictions to avoid redundant computations and reduce latency.
6. **Parallel and Distributed Processing with Spark**:
   - Leveraging Apache Spark for distributed data processing and parallelism to handle large-scale real-time data streams efficiently.
   - Utilizing **Spark Structured Streaming** for real-time data ingestion and transformation with minimal latency.
   - Combining Spark's in-memory processing capabilities with libraries like MLlib for scalable real-time model inference.
7. **Monitoring and Feedback Loops**:
   - Continuous monitoring of model performance and response times using tools like AWS CloudWatch or Databricks MLflow.
   - Incorporating feedback loops to dynamically adjust resource allocation based on traffic patterns.

By adopting these strategies, the system ensures scalable, reliable, and low-latency predictions for real-time use cases.

To enhance efficiency and scalability, a **Feature Store** was integrated into the project pipeline. This component facilitated:

1. **Centralized Feature Management**:
   - Storing and managing preprocessed features like standardized income, total expenditure, and customer age.
   - Ensuring consistency across different stages of the pipeline (training, validation, and inference).
2. **Real-Time Feature Retrieval**:
   - Features precomputed and stored in the Feature Store were accessible in real time, significantly reducing latency during prediction.
3. **Collaboration and Reusability**:
   - Features developed in the development environment (e.g., Dev) were stored and reused seamlessly in Staging and Production.

Implementation Example:

- **AWS SageMaker Feature Store**:
  - Preprocessed data was ingested into the Feature Store with metadata, making it queryable for downstream tasks.
  - Features were retrieved dynamically during model inference using APIs.
- **Databricks Feature Store**:
  - Built-in integration with Delta Lake allowed versioning and lineage tracking for features.
  - Features were directly utilized in Databricks Model Serving for consistent real-time predictions.

All the code relating to the "Feature Store" can be viewed within the jupyter file (last two cells)