Airline Ticket Shopping

Problem Statement : Airlines / travel agencies / analysts need real-time data to optimize pricing / demand forecasting & market trends.

Dynamic price optimize : Demand surges, competitor, user search pattern

Personalised recommendations : tailored made flight options

Demand forecasting → Supply chain, airport timings

Market trend analysis → Identifying popular routes, peak travel time, emerging travel destinations

Technical problems

* Ingesting → High volume (10Tb) → Continuously streamed

Data Acquisition

→ GDS (Global Distribution System)

↳ Sabre, Travelport, Amadeus

→ Air India API

→ Metasearch engines → Kayak, Skyscanner, google flights

Data

Semi-structured → JSON, XML

Structured → flight no, origin, dest, date time, price, cabin class, no. of passengers

Unstructured format → search queries, ip addresses


```
{
  "timestamp": "2024-01-26T10:30:00Z",
  "search_id": "unique_search_123",
  "user_id": "anon_user_456",
  "origin_airport": "JFK",
  "destination_airport": "LAX",
  "departure_date": "2024-03-15",
  "return_date": "2024-03-22",
  "cabin_class": "economy",
  "number_of_passengers": 1,
  "currency": "USD",
  "price_offers": [
    {
      "airline": "AA",
      "flight_number": "AA123",
      "price": 350.00,
      "stops": 0
    },
    {
      "airline": "UA",
      "flight_number": "UA456",
      "price": 380.50,
      "stops": 1
    }
  ],
  "user_location": {
    "ip_address": "x.x.x.x",
    "country": "US"
  },
  "device_info": {
    "user_agent": "Mozilla/...",
    "platform": "Web"
  }
}
```


* Can one search generate multiple offers from diff airlines
* User behaviour ↳ tracks user ids
* origin-destination pair

AWS Services

- Amazon Kinesis Data Streams → Real-time ingestion of streaming data

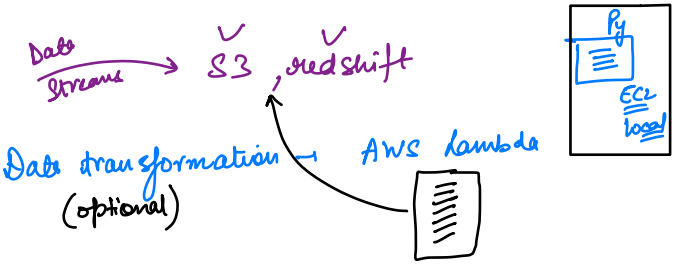
https://aws.amazon.com/kinesis/data-streams/

Use-case

GDS, airline api → Kinesis

- Amazon Kinesis Data Firehose

load streaming data into some data lake, data store



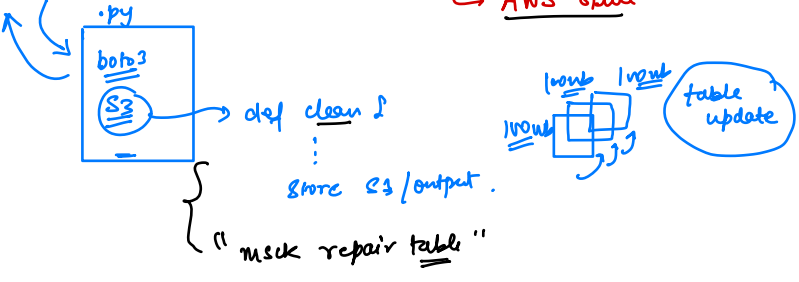
- → Run code without provisioning or managing servers
- → Can be triggered by data Streams, S3 event
- → Scales automatically
- → pay-per-execution

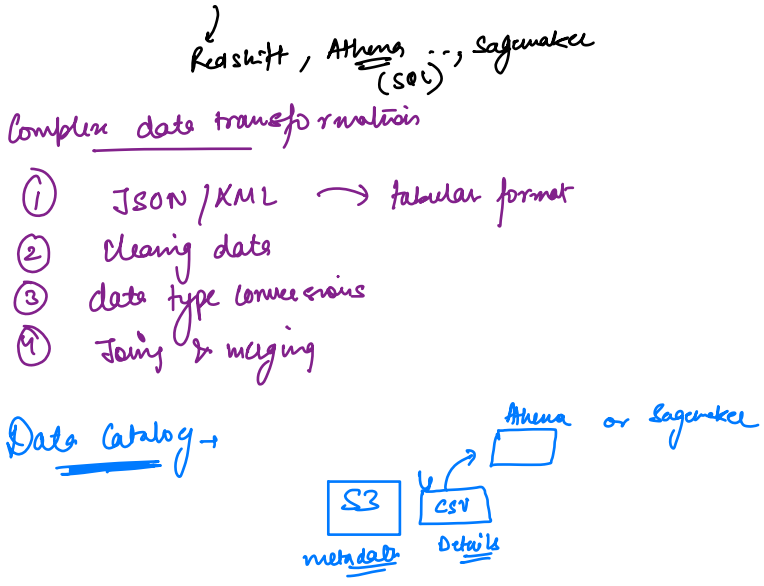
S3

- Extract, transform & load

Serverless data integration service for ETL

↳ AWS Glue





Data Preprocessing Steps (using Kinesis, Firehose, Lambda, Glue, S3):

Ingest Raw Data (Kinesis Data Streams): Data sources push streaming data messages into Kinesis Data Streams.

Persist Raw Data (Kinesis Data Firehose to S3): Configure Kinesis Data Firehose to subscribe to the Kinesis Data Stream and deliver raw data in batches to an S3 bucket (Raw Data Lake - Landing Zone). Consider using Lambda for *minimal*, lightweight inline transformation if absolutely necessary within Firehose (e.g., basic format normalization).

Data Cataloging (AWS Glue Crawler): Set up a Glue Crawler to crawl the S3 bucket where raw data lands. The crawler will automatically infer the schema of the raw data (JSON/XML) and create tables in the Glue Data Catalog.

Data Transformation and Cleaning (AWS Glue ETL Jobs):

Create Glue ETL jobs (using PySpark or Scala) to:

Read data from the Raw Data Lake (S3) using the Glue Data Catalog tables.

Perform data cleaning: Handle missing values, standardize formats, validate data types, remove duplicates.

Transform data: Flatten nested JSON structures into tabular formats, convert data types, extract relevant fields, enrich data (e.g., derive day of week from date).

Convert data to efficient columnar formats like Parquet. Parquet is optimized for analytical queries in S3.

Write the cleaned and transformed data to another S3 bucket (Curated Data Lake - Processed/ Staging Zone) in Parquet format.

5. Catalog Curated Data (AWS Glue Crawler): Set up another Glue Crawler to crawl the S3 bucket where the curated data is stored and update the Glue Data Catalog with metadata for the curated data tables (Parquet format).

Feature Engineering

Flight specific → Origin, Dest, Departure date, return, class, airline,
PNR,
day
week
year
weekend
seasonal
no of stop,
duration of flights
price

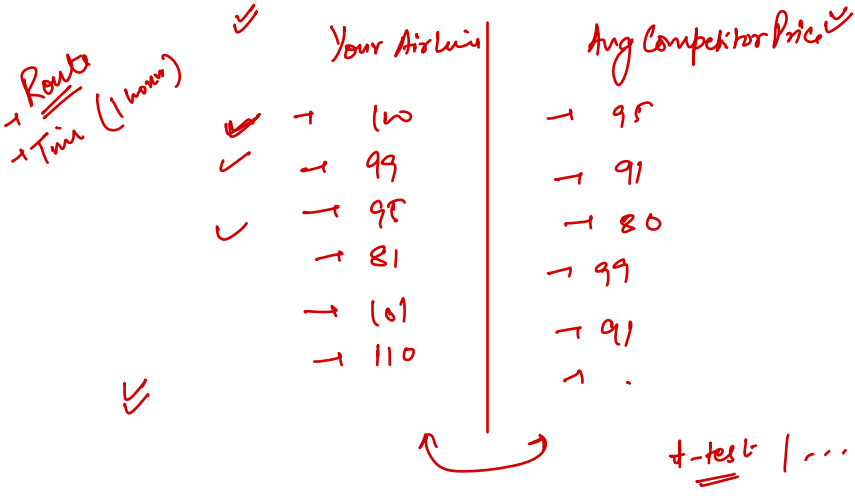
Transaction,
PNR, credit card, discount, points ...

Contextual features
↳ user location, ip address, Device type,
{ Groundtruth } location intelligence
app, browser, OS version

Derived features
↳ Route popular
today's booking
with
20 days MA

→ Price volatility ↳ historic flight fluctuation
Std ↑
→ Demand indicator → Search volume

→ Competitor pricing features → Price diff



Model Selection, training, evaluation & hyperparameter tuning

→ Airline Ticket Price prediction → Regression

→ Demand forecasting → predict the no. of bookings / search volume for a particular route (High / med / low)

(origin – destination) pairs (Reg / Classification)

→ Anomaly detection → Identify Unusual price spikes or patterns that could indicate market disruption

(unsupervised)

{Isolation forest}

→ Personalised recommendation

Training, Evaluation, Hyperparameter Tuning (using SageMaker):

Prepare Training Data: Load the engineered features from the Analytics Data Lake in S3. Split data into training, validation, and test sets.
SageMaker Notebook Instance: Launch a SageMaker Notebook instance (e.g., `ml.m5.xlarge` or larger based on data size).
Data Exploration and Preprocessing in Notebook: Use the SageMaker notebook to:
Connect to S3 and load the feature data.
Perform further data exploration, visualization, and any final preprocessing steps specific to the chosen model (e.g., scaling numerical features, one-hot encoding categorical features).
Model Training (SageMaker Training Job):
Choose an ML algorithm (e.g., XGBoost). SageMaker provides pre-built containers for popular algorithms.
Configure a SageMaker Training Job:
Input Data: Specify the S3 path to the training data.
Output Path: Specify the S3 path to store the trained model artifacts.
Instance Type and Count: Choose the EC2 instance type and number of instances for distributed training (e.g., `ml.m5.xlarge`, instance count based on data size and model complexity).
Algorithm Specification: Select the XGBoost algorithm container.
Hyperparameters: Set initial hyperparameters for the XGBoost model.
Launch the SageMaker Training Job. SageMaker will provision the training infrastructure, train the model, and store the trained model artifacts (model weights, configuration) in S3.

Model Evaluation (SageMaker Processing Job or within Notebook):
After training, evaluate the model's performance on the validation and test datasets.
Metrics for Regression: RMSE (Root Mean Squared Error), MAE (Mean Absolute Error), R-squared.
Use SageMaker Processing Jobs or perform evaluation directly in the notebook using the test data and the trained model.
Hyperparameter Tuning (SageMaker Automatic Model Tuning):
If model performance is not satisfactory, use SageMaker Automatic Model Tuning to optimize hyperparameters.
Define the hyperparameter ranges to search (e.g., `learning_rate`, `max_depth`, `n_estimators` for XGBoost).
Specify the objective metric to optimize (e.g., minimize RMSE).
Configure the tuning strategy (e.g., Bayesian Optimization) and the number of tuning jobs to run.
SageMaker will automatically launch multiple training jobs with different hyperparameter combinations and find the best set of hyperparameters that optimize the objective metric.

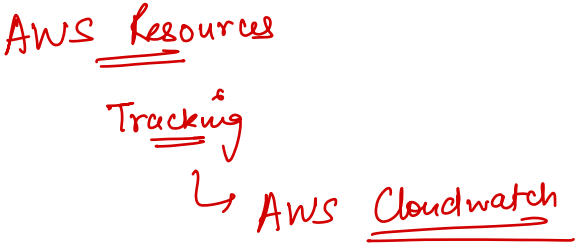
Model Deployment

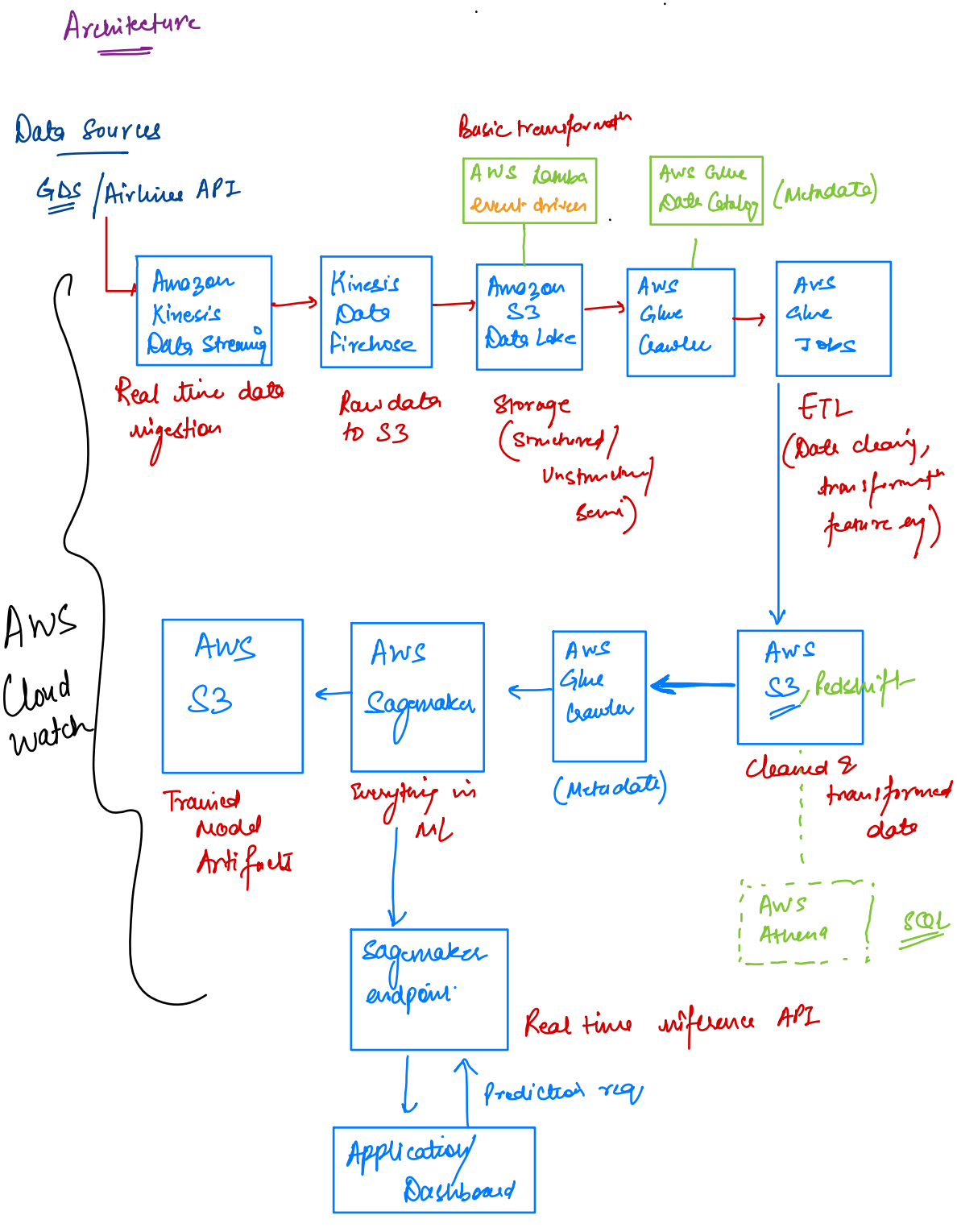
Deployment Options with SageMaker:

Real-time Inference Endpoint (SageMaker Endpoint):

Use Case: For low-latency, online predictions. Ideal for real-time price prediction when a user searches for a flight.
Functionality: Deploys the trained model as a REST API endpoint. You send prediction requests to the endpoint, and it returns predictions in real-time.
Instance Types: Choose instance types optimized for inference (e.g., `ml.m5.large` or `ml.c5.large` depending on latency and throughput requirements).
Auto Scaling: Configure auto-scaling for the endpoint to handle varying prediction request loads.
Batch Inference (SageMaker Batch Transform):

Use Case: For offline predictions on large datasets. Useful for tasks like generating daily or weekly price forecasts for all routes, or for batch scoring of historical data.
Functionality: Processes a batch of input data stored in S3 and writes the predictions to another S3 location. More cost-effective for offline prediction tasks.




Architecture
Data Sources
GDS / Airlines API
Basic transformation
AWS Lambda
event driven
AWS Glue
Data Catalog
(Metadata)
Amazon
Kinesis
Data Streams
Kinesis
Data
Firehose
Amazon
S3
Data Lake
AWS
Glue
Crawler
AWS
Glue
Jobs
Real time data
ingestion
Raw data
to S3
Storage
(Structured/
Unstructured/
Semi)
ETL
(Data cleaning,
transformation
feature eng)
AWS
Cloud
watch
AWS
S3
AWS
Sagemaker
AWS
Glue
Crawler
AWS
S3, Redshift
Trained
Model
Artifacts
Everything in
ML
(Metadata)
Cleaned &
transformed
data
AWS
Athena
SQL
Sagemaker
endpoint
Real time inference API
Prediction req
Application
Dashboard

Example Glue job

```
# PySpark code within an AWS Glue ETL Job

from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, col, from_json, get_json_object, to_date
from pyspark.sql.types import StructType, StructField, StringType, FloatType, ArrayType, TimestampType

# Initialize Spark Session (already available in Glue ETL environment)
spark = SparkSession.builder.appName("AirlineDataETL").getOrCreate()

# 1. Read Raw Data from S3 using Glue Data Catalog table
raw_data_dyf = glueContext.create_dynamic_frame.from_catalog(  # Using Glue's DynamicFrame for schema flexibility with
JSON
    database="your_raw_data_catalog_db",
    table_name="your_raw_data_table_name"
)

raw_data_df = raw_data_dyf.toDF() # Convert to Spark DataFrame for easier operations

# 2. Transformations: Flatten price_offers array and select fields

flattened_df = raw_data_df.select(
    col("search_id"),
    col("timestamp"),
    col("origin_airport"),
    col("destination_airport"),
    col("departure_date"),
    explode("price_offers").alias("price_offer") # Explode the array into rows
).select( # Select fields after explode - nested structure becomes flattened
    col("search_id"),
    col("timestamp"),
    col("origin_airport"),
    col("destination_airport"),
    col("departure_date"),
    col("price_offer.airline").alias("airline"),
    col("price_offer.flight_number").alias("flight_number"),
    col("price_offer.price").alias("price")
)


# 3. Load to Curated Data Lake in S3 (Parquet format)
curated_s3_path = "s3://your-curated-data-lake-bucket/airline_data_parquet/"

flattened_df.write.mode("overwrite").parquet(curated_s3_path) # Write DataFrame to S3 as Parquet

print("Glue ETL job completed successfully!")

# You might optionally update the Glue Data Catalog for the curated data here too using Glue Catalog APIs within the script
```