To address the challenges presented in the Data Engineering Case Study for AdvertiseX, we will design a comprehensive solution. Here's an outline of the steps involved, along with the technologies and tools we will use:

1. **Data Ingestion:**

- **Technologies:** Apache Kafka, Apache NiFi, AWS S3
- **Approach:**
- Use Apache Kafka for real-time data ingestion of ad impressions (JSON) and bid requests (Avro).
- Use Apache NiFi for ingesting click and conversion data (CSV) in batch mode.
- Store raw data in AWS S3 for further processing.

2. **Data Processing:**

- Technologies: Apache Spark, AWS Glue
- Approach:
- Use Apache Spark to process and transform data.
- Handle data validation, filtering, and deduplication using Spark.
- Correlate ad impressions with clicks and conversions.

3. **Data Storage and Query Performance:**

- **Technologies:** Amazon Redshift, AWS Athena
- **Approach:**
- Store processed data in Amazon Redshift for efficient querying.
- Use AWS Athena for ad hoc queries and analysis.
- Optimize Redshift for analytical queries by designing appropriate schema and indexing strategies.

4. **Error Handling and Monitoring:**

- **Technologies:** AWS CloudWatch, Apache Airflow
- **Approach:**
- Implement monitoring using AWS CloudWatch to detect anomalies and discrepancies.
- Set up alerting mechanisms to notify the team of any data quality issues.
- Use Apache Airflow for workflow management and scheduling, ensuring that data pipelines run smoothly and are monitored for failures.

Let's implement the solution step by step. Below is a high-level implementation plan and sample code snippets for each step:

Step 1: Data Ingestion

Apache Kafka Configuration

- Set up Kafka topics for ad impressions and bid requests.
- Create a producer to send ad impression data (JSON) to Kafka.
- Create a producer to send bid request data (Avro) to Kafka.

```
from kafka import KafkaProducer
import json
import avro.schema
import avro.io
import io

# Kafka producer for JSON data (ad impressions)
producer_json = KafkaProducer(bootstrap_servers='localhost:9092', value_serializer=lambda v: json.dumps(v).encode('utf-8'))

# Kafka producer for Avro data (bid requests)
schema = avro.schema.Parse(open("bid_request_schema.avsc", "r").read())
producer_avro = KafkaProducer(bootstrap_servers='localhost:9092')

# Example ad impression data
ad_impression = {
  "ad_creative_id": "123",
  "user_id": "abc",
  "timestamp": "2024-05-24T12:34:56Z",
  "website": "example.com"
}

# Send ad impression data to Kafka
producer_json.send('ad_impressions', ad_impression)

# Example bid request data
bid_request = {
    "user_info": {"user_id": "abc"},
    "auction_details": {"auction_id": "456"},
    "ad_targeting": {"criteria": "example_criteria"}
}

# Serialize Avro data and send to Kafka
bytes_writer = io.BytesIO()
encoder = avro.io.BinaryEncoder(bytes_writer)
datum_writer = avro.io.DatumWriter(schema)
datum_writer.write(bid_request, encoder)
raw_bytes = bytes_writer.getvalue()

producer_avro.send('bid_requests', raw_bytes)
```

Apache NiFi Configuration

- Configure NiFi to ingest click and conversion data from CSV files.

Use NiFi's processors such as GetFile, PutS3Object, and ConvertRecord to handle CSV ingestion and save raw data to AWS S3.

Step 2: Data Processing

Apache Spark Job

- Use Spark to process data, ensuring validation, filtering, deduplication, and correlation.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, from_json, to_timestamp

# Initialize Spark session
spark = SparkSession.builder.appName("AdDataProcessing").getOrCreate()

# Load ad impressions data from Kafka
ad_impressions_df = spark.readStream.format("kafka").option("kafka.bootstrap.servers", "localhost:9092").option("subscribe", "ad_impressions").load()
ad_impressions_df = ad_impressions_df.selectExpr("CAST(value AS STRING) as
json").select(from_json(col("json"), schema).alias("data")).select("data.*")
```

# Load bid requests data from Kafka

```
bid_requests_df = spark.readStream.format("kafka").option("kafka.bootstrap.servers",
"localhost:9092").option("subscribe", "bid_requests").load()

bid_requests_df = bid_requests_df.selectExpr("CAST(value AS BINARY) as
avro").select(from_avro(col("avro"), schema).alias("data")).select("data.*")
```

# Load click and conversion data from S3

```
clicks_conversions_df = spark.read.csv("s3a://advertisex-data/clicks_conversions/*.csv",
header=True)
```

# Data transformation and correlation logic

```
ad_impressions_df = ad_impressions_df.withColumn("timestamp",
to_timestamp(col("timestamp")))

clicks_conversions_df = clicks_conversions_df.withColumn("event_timestamp",
to_timestamp(col("event_timestamp")))
```

```
# Correlate ad impressions with clicks and conversions

correlated_data_df = ad_impressions_df.join(clicks_conversions_df, "user_id", "left_outer")
```

# Write processed data to Redshift

```
correlated_data_df.write.format("jdbc").option("url", "jdbc:redshift://redshift-
cluster:5439/dev").option("dbtable", "ad_campaign_data").option("user",
"awsuser").option("password", "password").save()
```

Step 3: Data Storage and Query Performance

Amazon Redshift and Athena

- Store processed data in Amazon Redshift.
- Use AWS Athena for querying.

-- Example Redshift table creation

```
CREATE TABLE ad_campaign_data (
  ad_creative_id VARCHAR,
  user_id VARCHAR,
  timestamp TIMESTAMP,
  website VARCHAR,
  event_timestamp TIMESTAMP,
  conversion_type VARCHAR
);
```

```
-- Example query to analyze campaign performance
SELECT ad_creative_id, COUNT(*) AS impressions, SUM(CASE WHEN conversion_type IS NOT NULL THEN 1 ELSE 0 END) AS conversions
FROM ad_campaign_data
GROUP BY ad_creative_id;
```

Step 4: Error Handling and Monitoring

AWS CloudWatch and Apache Airflow

- Set up CloudWatch for monitoring and alerting.
- Use Airflow for orchestrating and scheduling data pipelines.

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.utils.dates import days_ago

# Define the DAG
dag = DAG('advertisex_data_pipeline', description='AdvertiseX Data Pipeline', schedule_interval='@hourly', start_date=days_ago(1))

# Define tasks
task1 = BashOperator(task_id='run_spark_job', bash_command='spark-submit --master yarn ad_data_processing.py', dag=dag)
task2 = BashOperator(task_id='monitor_data_quality', bash_command='python monitor_data_quality.py', dag=dag)
```

```
# Task dependencies
task1 >> task2
```

By following this approach, we can build a scalable and efficient data ingestion and processing system for AdvertiseX. The code snippets above provide a starting point, and further refinements and optimizations can be made based on the specific requirements and data volumes.