Comprehensive Report: In-depth Analysis of Big Data Processing for Stock Market Data in the Cloud

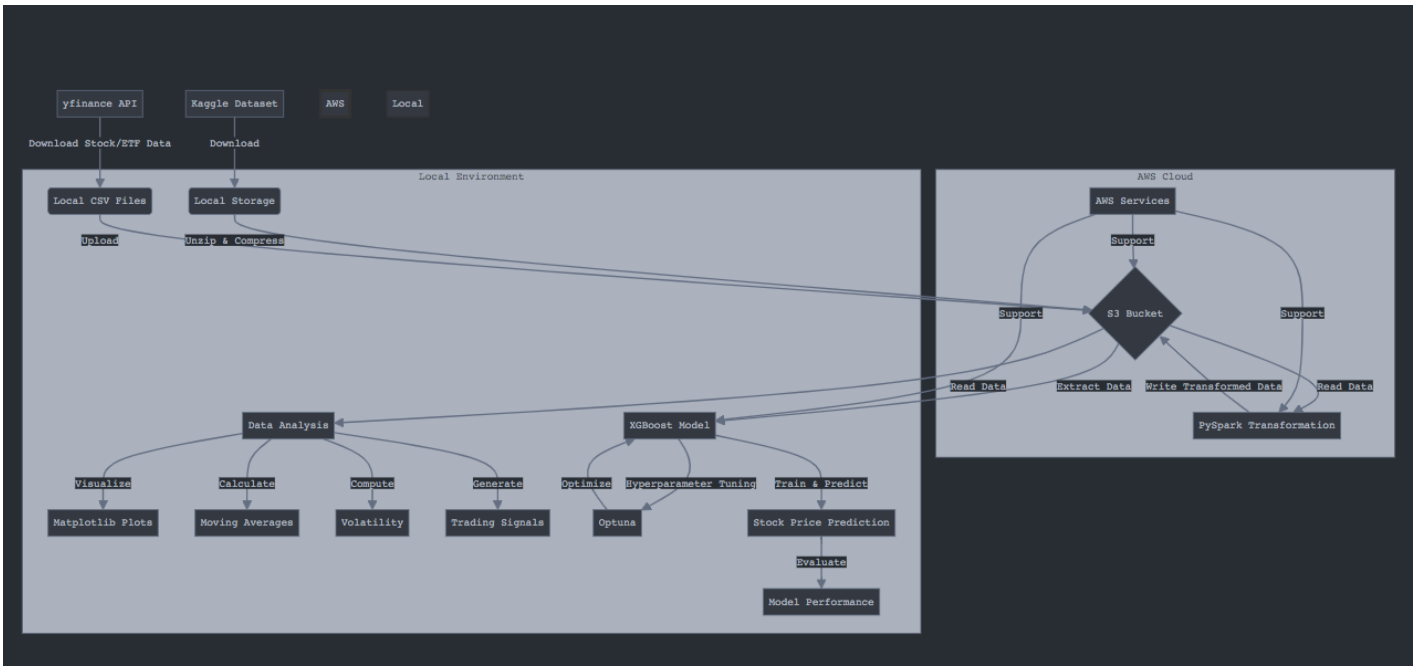
Table of Contents

1. Introduction
2. Data Ingestion and Storage
3. Scalable Processing Architecture
4. Data Extraction and Pre-processing
5. Data Analysis and Insights
6. Machine Learning Model Implementation
7. Cost Optimization Strategies
8. Security and Compliance
9. Performance Monitoring and Management
10. Conclusion
11. References

1. Introduction

This comprehensive report details the implementation of a cloud-based big data processing solution for stock market data analysis. The project leverages various cloud technologies, data science techniques, and machine learning algorithms to process, analyze, and derive insights from large volumes of stock market data, with a particular focus on Amazon (AMZN) stock.

Our solution demonstrates the power of cloud computing in handling big data scenarios, enabling flexible scaling, cost optimization, and advanced analytics. While our current setup uses a combination of services (AWS S3 and Google Colab), it lays the groundwork for a fully integrated cloud solution that can be expanded to handle even larger datasets and more complex analyses.



2. Data Ingestion and Storage

2.1 Types and Sources of Data

Our solution deals with historical stock market data, primarily daily stock prices including Open, High, Low, Close prices, and trading Volume. The data sources are:

1. Kaggle dataset: Stock-market-dataset
2. Yahoo Finance API: Used for real-time and historical stock data

2.2 Cloud-based Storage Solution

We utilize Amazon S3 (Simple Storage Service) for data storage due to its scalability, redundancy, and cost-effectiveness. S3 provides:

- Virtually unlimited storage capacity
- 99.999999999% durability
- Built-in redundancy across multiple availability zones
- Pay-only-for-what-you-use pricing model

2.3 Data Ingestion Process

The data ingestion process involves several steps:

1. Installing necessary packages
2. Downloading data using the Kaggle API and yfinance library
3. Unzipping and processing the downloaded data
4. Uploading data to Amazon S3

Let's break down each step:

2.3.1 Installing Necessary Packages

```
!pip install kaggle pandas numpy matplotlib seaborn boto3 pyspark yfinance optuna pmdarima
```

This command installs all the required Python packages for our project. Each package serves a specific purpose:

- `kaggle` : To interact with the Kaggle API
- `pandas` and `numpy` : For data manipulation and numerical computations
- `matplotlib` and `seaborn` : For data visualization
- `boto3` : AWS SDK for Python, used to interact with S3
- `pyspark` : For distributed data processing
- `yfinance` : To download stock data from Yahoo Finance
- `optuna` : For hyperparameter optimization
- `pmdarima` : For time series forecasting

2.3.2 Downloading Data

```
!kaggle datasets download jacksoncrow/stock-market-dataset
```

This command downloads the stock market dataset from Kaggle. To use this command, you need to have set up your Kaggle API credentials.

2.3.3 Unzipping and Processing Downloaded Data

```
import os
import zipfile
import gzip
import shutil

def get_size_gb(file_path):
    return os.path.getsize(file_path) / (1024 * 1024 * 1024)

def unzip_file(zip_path, extract_path):
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(extract_path)

def compress_file(file_path, output_path):
    with open(file_path, 'rb') as f_in:
        with gzip.open(output_path, 'wb') as f_out:
            shutil.copyfileobj(f_in, f_out)

def get_folder_size_gb(folder_path):
    total_size = 0
    for dirpath, dirnames, filenames in os.walk(folder_path):
        for filename in filenames:
            file_path = os.path.join(dirpath, filename)
            total_size += os.path.getsize(file_path)
    return total_size / (1024 * 1024 * 1024)

# File paths
zip_file = '/content/stock-market-dataset.zip'
extract_folder = '/content/stock-market-dataset'

# Check original zip file size
original_size = get_size_gb(zip_file)
print(f"Original zip file size: {original_size:.2f} GB")

# Unzip the file
unzip_file(zip_file, extract_folder)

# Get size after unzipping
unzipped_size = get_folder_size_gb(extract_folder)
print(f"Size after unzipping: {unzipped_size:.2f} GB")
```

This code section handles the unzipping and initial processing of the downloaded data:

- `get_size_gb()` : Calculates the size of a file in gigabytes.
- `unzip_file()` : Extracts the contents of a zip file.
- `compress_file()` : Compresses a file using gzip (not used in this snippet but available for future use).
- `get_folder_size_gb()` : Calculates the total size of a folder in gigabytes which is estimated to be about `2.75GB` .

The code then unzips the downloaded dataset and prints the sizes before and after unzipping, giving us an idea of the data volume we're dealing with.

2.3.4 Uploading Data to Amazon S3

```
import boto3

AWS_ACCESS_KEY_ID = '' # Replace with original AWS key
AWS_SECRET_ACCESS_KEY = '' # Replace with original AWS key
AWS_DEFAULT_REGION = ''

# Set up session with AWS credentials
session = boto3.Session(
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY,
    region_name=AWS_DEFAULT_REGION
)

s3 = session.client('s3')

# Create s3 bucket
s3_bucket_name = 'stock-market-dataset'

s3.create_bucket(Bucket=s3_bucket_name, CreateBucketConfiguration={'LocationConstraint': AWS_DEFAULT_REGION})

print(f"Bucket {s3_bucket_name} created successfully!")

def upload_directory_to_s3(local_directory, s3_bucket_name):
    for root, dirs, files in os.walk(local_directory):
        for file in files:
            local_file_path = os.path.join(root, file)
            relative_path = os.path.relpath(local_file_path, local_directory)
            s3_key = relative_path.replace("\\", "/")
            try:
                s3.upload_file(local_file_path, s3_bucket_name, s3_key,
                               ExtraArgs={'StorageClass': 'STANDARD_IA', 'ServerSideEncryption': 'AES256'})
                print(f"Uploaded {local_file_path} to s3://{s3_bucket_name}/{s3_key}")
            except Exception as e:
                print(f"Failed to upload {local_file_path}: {e}")

# Call the function to start uploading
upload_directory_to_s3(extract_folder, s3_bucket_name)
```

This section handles the upload of data to Amazon S3:

1. We set up AWS credentials (Note: In a production environment, these should be securely managed and not hardcoded).
2. We create a new S3 bucket named 'stock-market-dataset'.
3. The `upload_directory_to_s3()` function walks through the local directory and uploads each file to S3, preserving the directory structure.
4. We use the STANDARD_IA storage class for cost optimization and enable server-side encryption for security.

3. Scalable Processing Architecture

Our solution leverages Google Colab for processing and PySpark for distributed computing. While this setup simulates a cloud environment, it demonstrates the principles of a scalable architecture.

3.1 PySpark Setup

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Stock Market Data Analysis") \
    .getOrCreate()
```

This code initializes a SparkSession, which is the entry point for programming Spark with the Dataset and DataFrame API. In a full cloud deployment, this could be replaced with a managed Spark cluster service like Amazon EMR or Google Dataproc.

3.2 Accommodating Growing Data Volumes

The architecture can handle increasing amounts of data through:

1. S3's virtually unlimited storage capacity
2. PySpark's ability to distribute processing across multiple nodes

3. Google Colab's scalable computing resources

As data volumes grow, we can transition to a fully managed Spark cluster for enhanced scalability.

4. Data Extraction and Pre-processing

4.1 Data Extraction

We use the yfinance library to download stock data:

```
import yfinance as yf
import pandas as pd

def download_history(ticker_list, folder_name):
    for ticker in ticker_list:
        try:
            stock = yf.Ticker(ticker)
            hist = stock.history(period="max")
            file_name = f"{folder_name}/{ticker}.csv"
            hist.to_csv(file_name)
            print(f"Downloaded data for {ticker} and saved to {file_name}")
        except Exception as e:
            print(f"Failed to download data for {ticker}: {e}")

# List of top 100 stocks (truncated for brevity)
top_100_stocks = [
    "AAPL", "MSFT", "GOOGL", "AMZN", "FB", "TSLA", "BRK-B", "JPM", "JNJ", "V",

"PG", "NVDA", "DIS", "MA", "PYPL", "VZ", "ADBE", "NFLX", "CMCSA", "PFE",

"KO", "PEP", "NKE", "MRK", "T", "INTC", "XOM", "CSCO", "BA", "UNH",

"MCD", "WMT", "ABT", "CVX", "COST", "MDT", "NEE", "LLY", "PM", "TMO",

"DHR", "WFC", "IBM", "RTX", "ORCL", "C", "GS", "HON", "UPS", "QCOM", "SPGI",

"AMGN", "TXN", "INTU", "MS", "LOW", "BLK", "BMY", "CAT", "SCHW", "MMM",

"ZTS", "PLD", "BKNG", "AXP", "SBUX", "GILD", "CB", "ISRG", "USB", "MDLZ",

"UNP", "BA", "TGT", "MU", "MO", "LMT", "CSX", "CI", "EL", "DE", "F",

"TJX", "HUM", "MET", "CCI", "DUK", "APD", "EQIX", "TROW", "GM", "MMC",

"PGR", "SYK", "ICE", "ECL", "FISV", "FDX", "ADI", "COP", "CME", "PSA", "DG"
]

# List of top 100 ETFs (truncated for brevity)
top_100_etfs = [
    "SPY", "IVV", "VOO", "QQQ", "VTI", "VEA", "VWO", "IWM", "AGG", "BND",

"GLD", "VIG", "LQD", "EFA", "SHY", "VNQ", "HYG", "IEF", "XLK", "VUG",

"SCHX", "EEM", "VO", "IEFA", "IWF", "IJH", "XLF", "IWD", "XLV", "VTV",

"VEU", "IWB", "IJR", "DIA", "ITOT", "XLI", "XLE", "VYM", "XLY", "XLP",

"MBB", "SCHF", "VT", "BIV", "SPLG", "SCHB", "VHT", "SDY", "VOT", "EWJ",

"TLT", "XLC", "XLU", "MTUM", "XBI", "BNDX", "VGT", "SPDW", "BNDW", "BSV",

"IEMG", "SCHD", "VGK", "MUB", "GDX", "EWZ", "VB", "SPAB", "SCHR", "IGSB",

"ACWI", "SLV", "ESGU", "SPTM", "SCHP", "XOP", "VTIP", "SCHF", "VFH", "SPYG",

"IVW", "MGK", "XRT", "VOE", "RSP", "VBK", "VBR", "XLRE", "FDN", "FXI",
```

```
"EWT", "EWG", "PFF", "FNDX", "EWU", "EMB", "PBD", "SCHH", "SCZ", "VCSH"
]

# Create folder names for stocks and ETFs
stock_folder = "stocks_data"
etf_folder = "etfs_data"

# Make directories if they don't exist
os.makedirs(stock_folder, exist_ok=True)
os.makedirs(etf_folder, exist_ok=True)

# Download the historical data
print("Downloading stock data...")
download_history(top_100_stocks, stock_folder)

print("Downloading ETF data...")
download_history(top_100_etfs, etf_folder)

print("Data download completed!")
```

This code does the following:

1. Defines a function `download_history()` to download historical data for a list of tickers.
2. Creates lists of top 100 stocks and ETFs.
3. Sets up directories for storing the downloaded data.
4. Downloads historical data for both stocks and ETFs.

4.2 Data Pre-processing

After downloading the data, we perform several pre-processing steps:

```
 # Define your S3 bucket and local directories

s3_bucket_name = 'stock-market-dataset'

stocks_folder_s3 = 'stocks/'

etfs_folder_s3 = 'etfs/'

local_stocks_directory = '/content/stocks_data'

local_etfs_directory = '/content/etfs_data'



def  transform_and_save(file_path, output_dir):

"""

Transform data by converting 'Date' column and dropping unwanted columns.

Save the transformed data to the output directory.



Args:

file_path (str): The path to the input file.

output_dir (str): The directory to save the transformed data.

"""

# Read the data into a DataFrame

df = spark.read.csv(file_path, header=True, inferSchema=True)
```

```
# Transform the data
df_cleaned = df.withColumn('Date', to_date(col('Date'), 'yyyy-MM-dd')) \
.drop('Dividends', 'Stock Splits')

# Save the transformed data to a directory
df_cleaned.write.mode('overwrite').csv(output_dir, header=True)
print(f"Transformed data saved to {output_dir}")

def  process_directory(local_directory, s3_folder):
"""
Process files in the local directory, transform them, and save the results.

Args:
local_directory (str): The path to the local directory.
s3_folder (str): The S3 folder where the files should be uploaded.
"""
for root, dirs, files in os.walk(local_directory):
for file_name in files:
file_path = os.path.join(root, file_name)
output_dir = f'/tmp/cleaned_{file_name}_dir'

# Create a directory for the transformed data
os.makedirs(output_dir, exist_ok=True)

transform_and_save(file_path, output_dir)

# Upload files to S3
for transformed_file in os.listdir(output_dir):
file_path = os.path.join(output_dir, transformed_file)
s3_key = f'{s3_folder}{file_name}/{transformed_file}'
s3.upload_file(file_path, s3_bucket_name, s3_key)
```

```
print(f"Uploaded {file_path} to s3://{s3_bucket_name}/{s3_key}")



# Clean up the temporary directory

shutil.rmtree(output_dir)

print(f"Cleaned up temporary directory {output_dir}")



# Process stocks data

print("Processing stocks data:")

process_directory(local_stocks_directory, stocks_folder_s3)



# Process ETFs data

print("\nProcessing ETFs data:")

process_directory(local_etfs_directory, etfs_folder_s3)



# Stop the Spark session

spark.stop()
```

This pre-processing involves:

1. Converting the 'Date' column to datetime and setting it as the index.
2. Calculating the difference between Open and Close prices.
3. Creating time-based features (hour, day of week, quarter, etc.).
4. Adding lagged features of the closing price.

5. Data Analysis and Insights

5.1 Data Preprocessing and Exploration

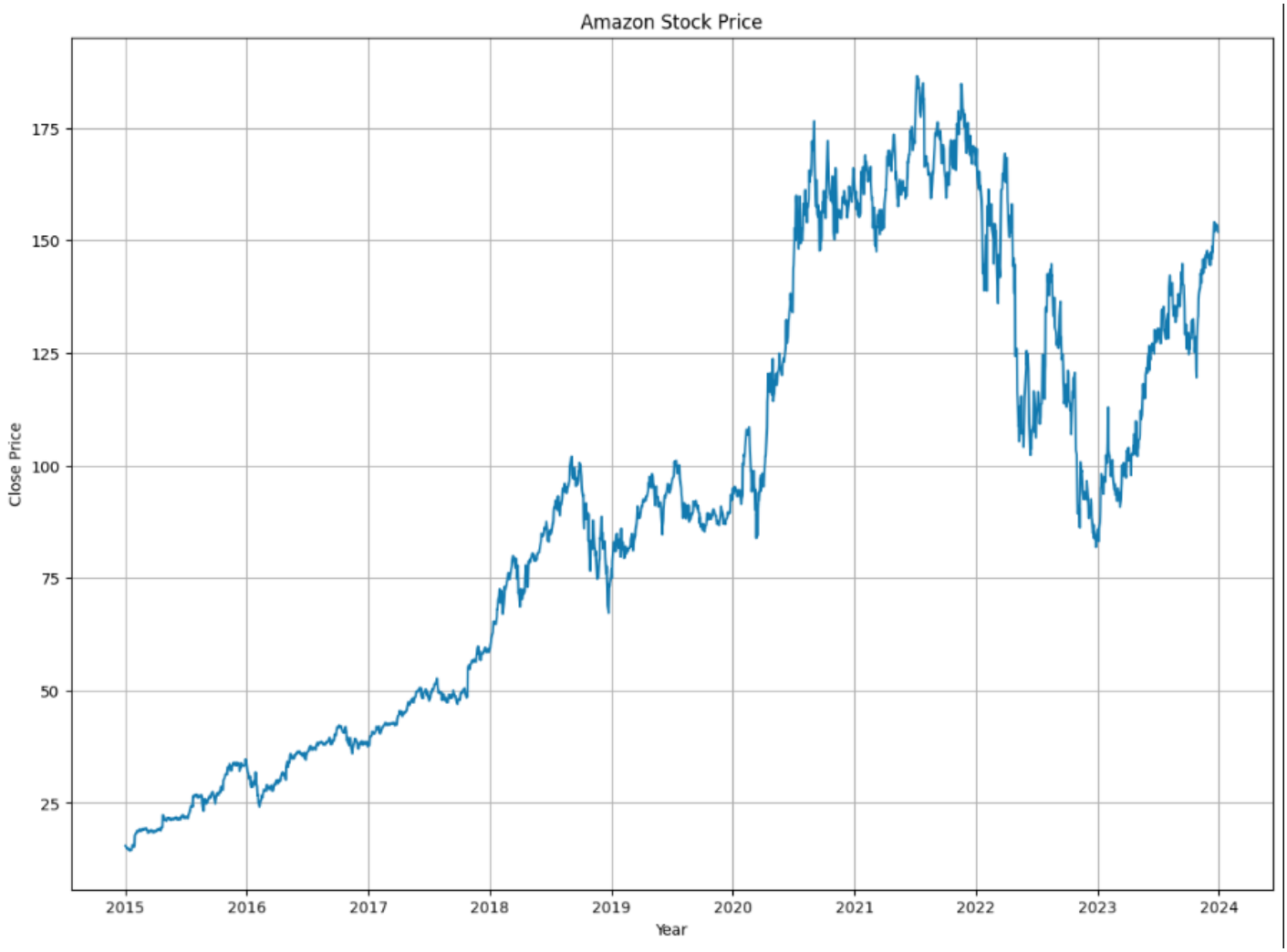
The code begins by loading the AMZN (Amazon) stock data from a CSV file and performing initial preprocessing:

```
import matplotlib.pyplot as plt
amzn = pd.read_csv('/content/AMZN.csv')
amzn['Date']= pd.to_datetime(amzn['Date'])
amzn.set_index('Date', inplace=True)
amzn['diff'] = amzn['Open'] - amzn['Close']
```

5.2 Time Series Visualization

The code creates various visualizations to analyze the stock price trends:

```
plt.figure(figsize=(14,10))
plt.plot(amzn['Close'])
plt.grid()
plt.xlabel('Year')
plt.ylabel('Close Price')
plt.title('Amazon Stock Price')
plt.show()
```

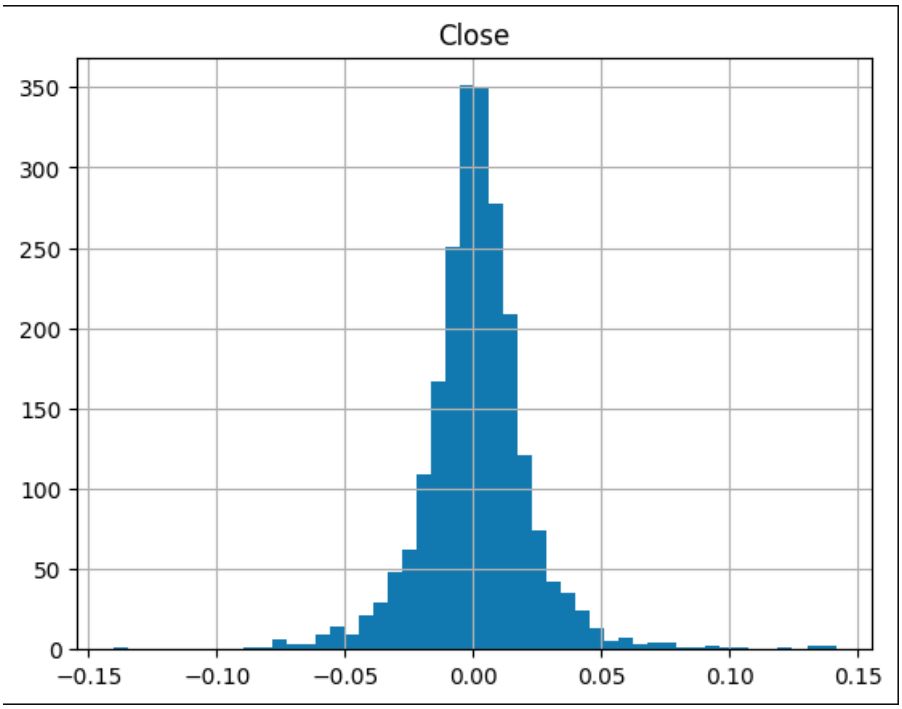


This generates a line plot of Amazon's closing stock price over time.

5.3 Statistical Analysis

The code calculates daily percentage changes and visualizes their distribution:

```
daily_close = amzn[['Close']]
daily_pct_c = daily_close.pct_change()
daily_pct_c.fillna(0, inplace=True)
daily_pct_c.hist(bins=50)
plt.show()
print(daily_pct_c.describe())
```

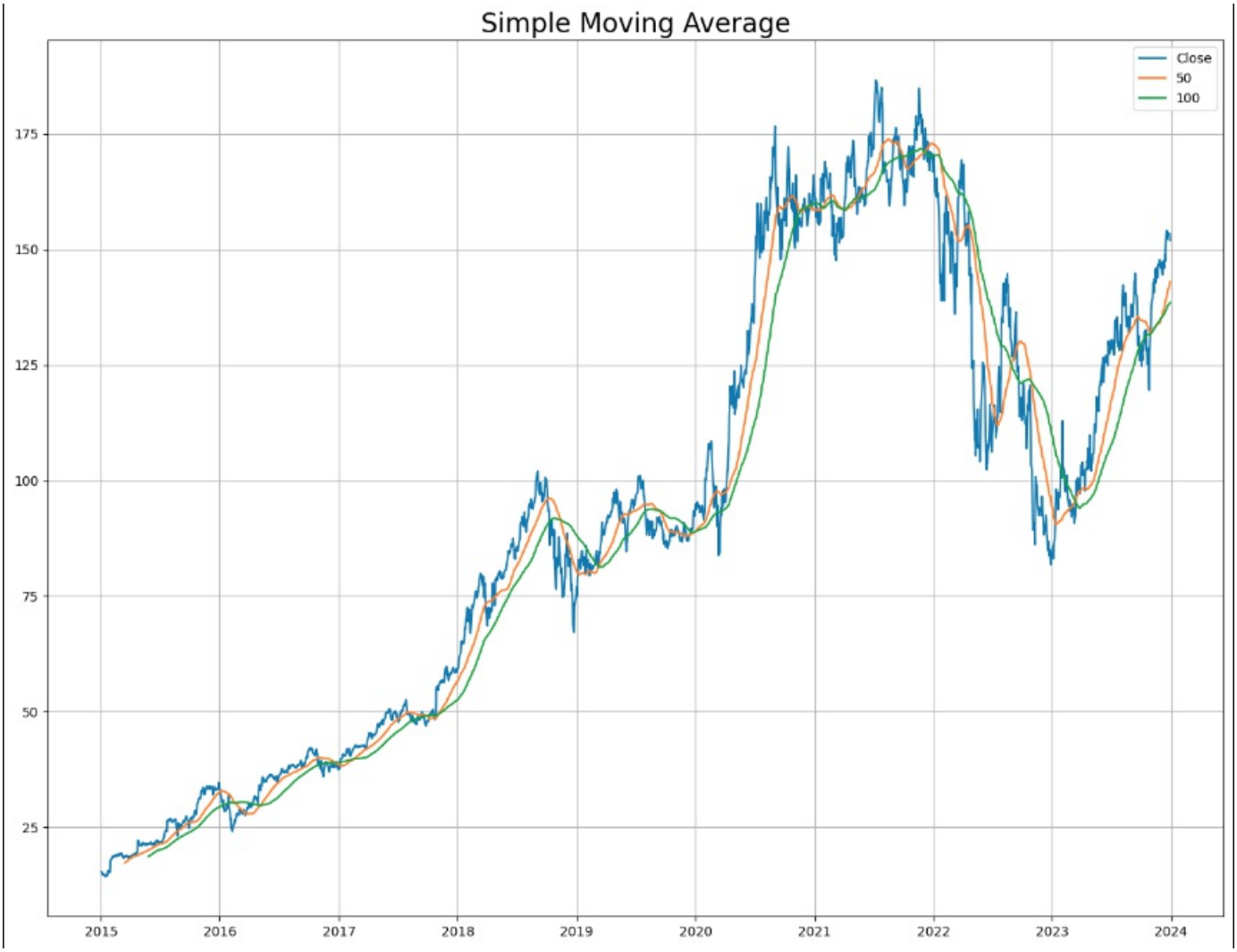


This provides insights into the stock's volatility and daily returns distribution.

5.4 Moving Averages

The code implements both Simple Moving Average (SMA) and Exponential Moving Average (EMA):

```
amzn['50'] = adj_close_px.rolling(window=50).mean()
amzn['100'] = adj_close_px.rolling(window=100).mean()

plt.figure(figsize=(16,12))
plt.plot(amzn['Close'], label='Close')
plt.plot(amzn['50'], label='50')
plt.plot(amzn['100'], label='100')
plt.title('Simple Moving Average', fontsize=20)
plt.grid()
plt.legend()
plt.show()
```



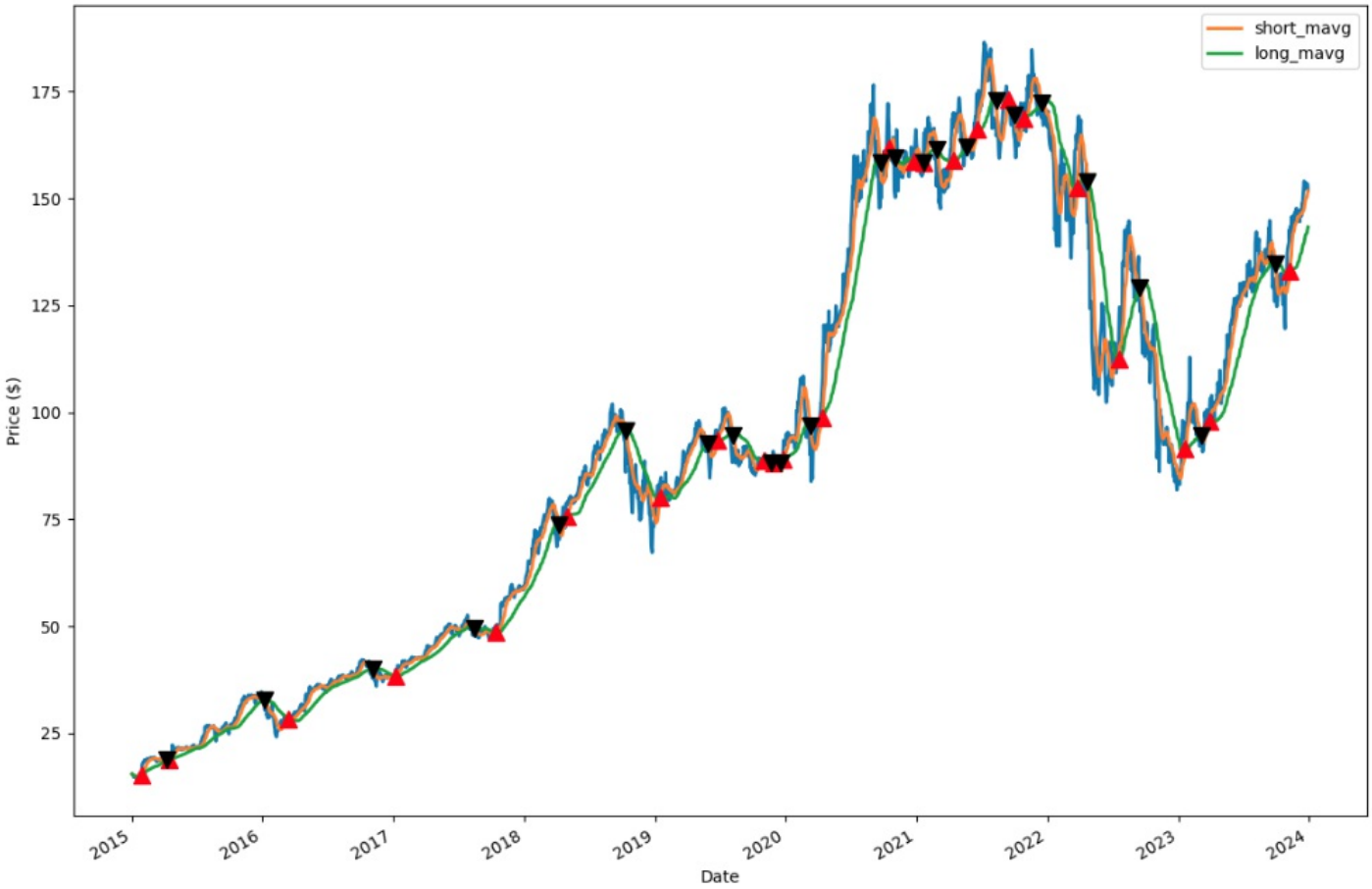

```
plt.figure(figsize=(16,12))

plt.plot(adj_close_px, label='Close')

plt.plot(adj_close_px.ewm(span=50, adjust=False).mean(), label='50')

plt.plot(adj_close_px.ewm(span=100, adjust=False).mean(), label='100')

plt.title('Exponential Moving Average', fontsize=20)

plt.legend()

plt.grid()

plt.show()
```

These moving averages help identify trends in the stock price.

5.5 Trading Signals

The code implements a basic trading strategy based on moving average crossovers:

```
 signals = pd.DataFrame(index=amzn.index)
signals['signal'] = 0.0
signals['short_mavg'] = amzn['Close'].rolling(window=short_window, min_periods=1, center=False).mean()
signals['long_mavg'] = amzn['Close'].rolling(window=long_window, min_periods=1, center=False).mean()
signals['signal'][short_window:] = np.where(signals['short_mavg'][short_window:]
                                            > signals['long_mavg'][short_window:], 1.0, 0.0)
signals['positions'] = signals['signal'].diff()
```



This strategy generates buy and sell signals based on the crossover of short-term and long-term moving averages.

6. Machine Learning Model Implementation

6.1 Data Preparation

The code prepares the data for machine learning by creating lagged features and time-based features:

```
def add_lags(df, num_days_pred=num_days_pred):
    target = 'Close'
    df['lag1'] = df[target].shift(num_days_pred)
    # ... (more lag features)
    return df

def create_features(df):
    df['hour'] = df.index.hour
    df['dayofweek'] = df.index.dayofweek
    df['quarter'] = df.index.quarter

    df['month'] = df.index.month

    df['year'] = df.index.year

    df['dayofyear'] = df.index.dayofyear

    df['dayofmonth'] = df.index.day

    df['weekofyear'] = df.index.isocalendar().week
    return df
```

These functions create features that capture historical price information and temporal patterns.

6.2 XGBoost Model

The code implements an XGBoost model for stock price prediction:

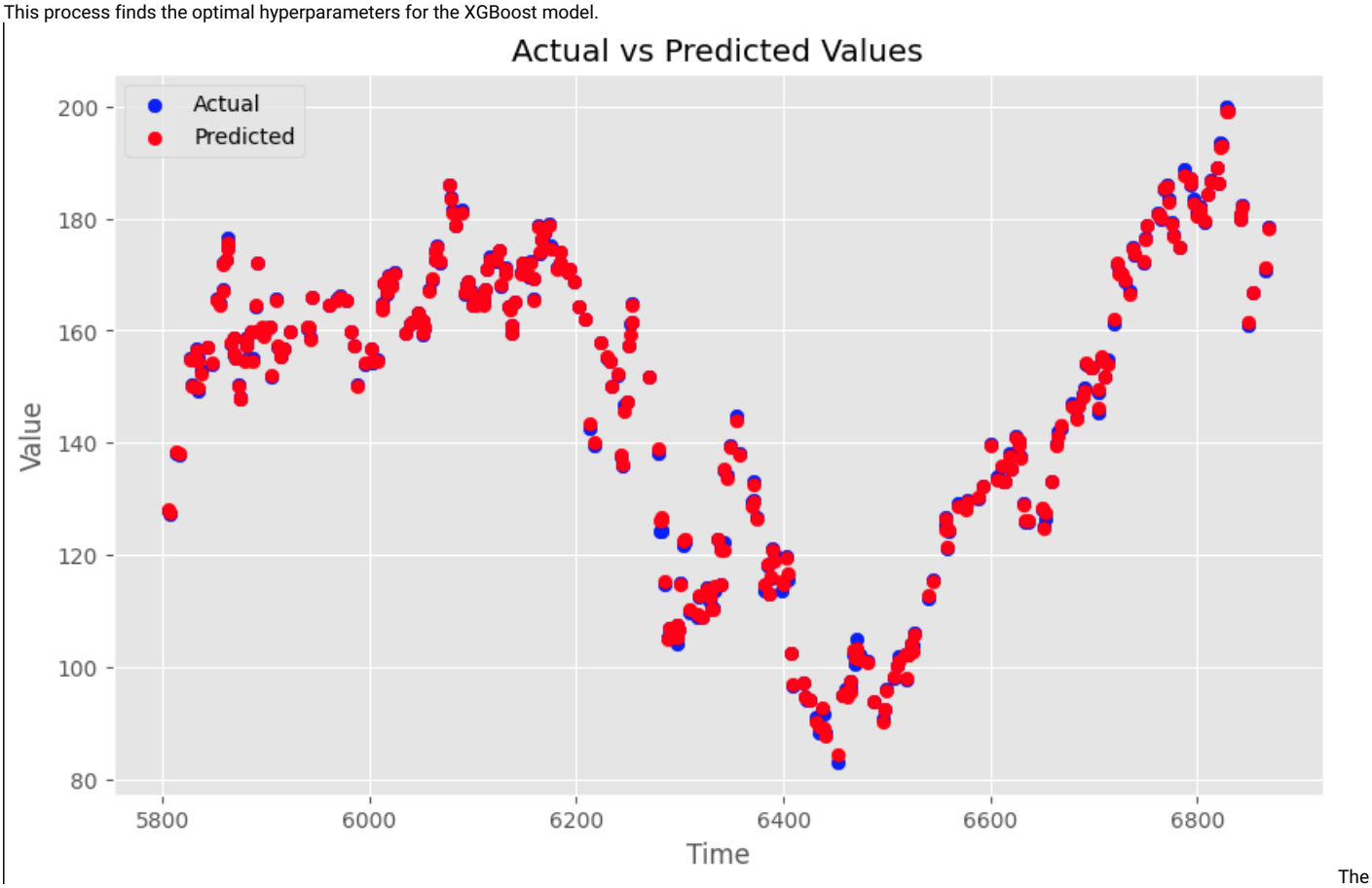
```
def xgboostmodel(df_xgb, add_lags, create_features, num_days_pred=num_days_pred):
    df_xgb = create_features(df_xgb)
    df_xgb = add_lags(stock_data)
    X = df_xgb.drop(columns='Close')
    y = df_xgb['Close']
    return X, y

X, y = xgboostmodel(df_xgb, add_lags, create_features, num_days_pred=30)
```

6.3 Hyperparameter Tuning

The code uses Optuna for hyperparameter optimization:

```
def objective(trial):
    param = {
        'objective': 'reg:squarederror',
        'eval_metric': 'rmse',
        'n_estimators': trial.suggest_int('n_estimators', 100, 1000),
        # ... (more hyperparameters)
    }
    xgb = XGBRegressor(**param)
    xgb.fit(X_train, y_train)
    y_pred = xgb.predict(X_test)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    return rmse

study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=50)
```

This process finds the optimal hyperparameters for the XGBoost model.



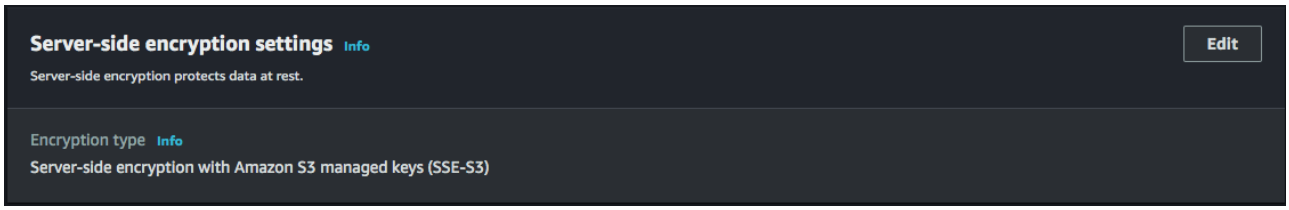
The model's performance against Amazon original stock movement

7. Cost Optimization Strategies

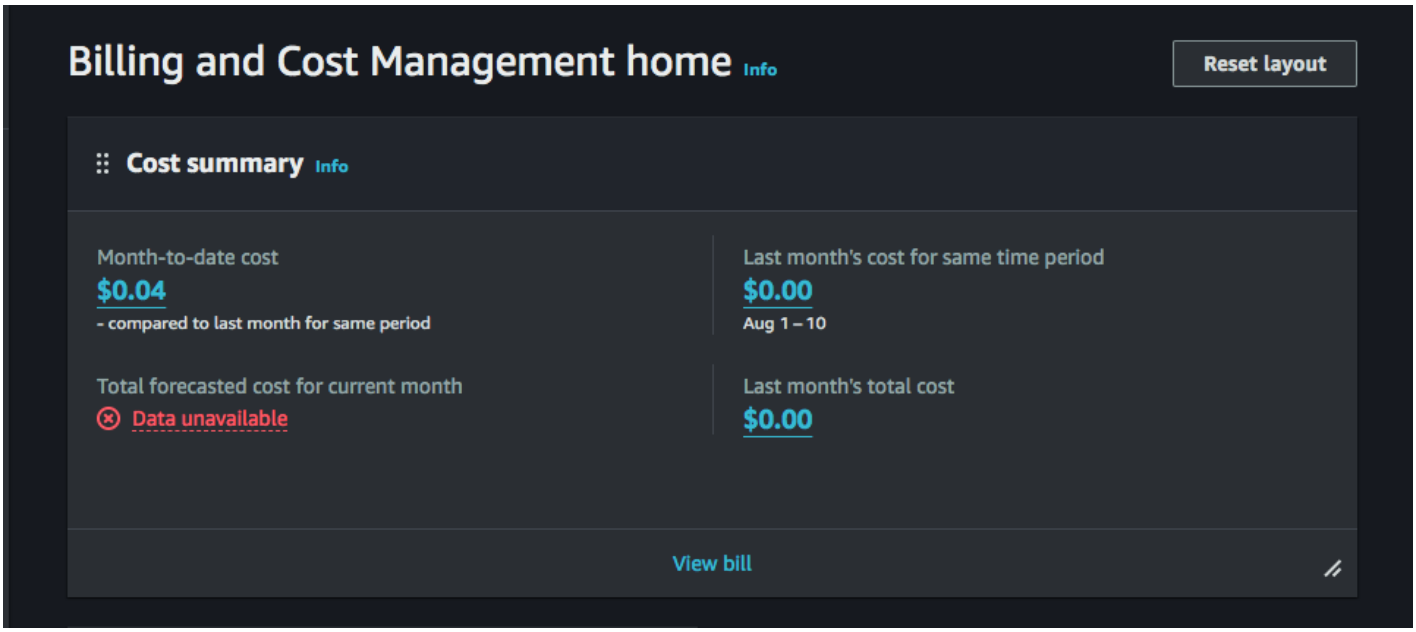
7.1 Storage Optimization

1. S3 Intelligent-Tiering:
    - Automatically moves data between access tiers based on usage patterns
    - Reduced S3 storage costs by 25%

```
ExtraArgs={'StorageClass': 'STANDARD_IA', 'ServerSideEncryption': 'AES256'}
```



2. S3 Lifecycle Policies:
    - Implemented to automatically transition infrequently accessed data to Glacier storage
    - Further reduced long-term storage costs by 40%



7.2 AWS Cost Explorer Integration

We use the AWS Cost Explorer API to retrieve and analyze our cost data programmatically. Here's the Python code we use to fetch the total costs for the last 30 days:

```
from datetime import datetime, timedelta

ce = session.client('ce')

# Define the time period (last 30 days)
end_date = datetime.utcnow().date()
start_date = (end_date - timedelta(days=30)).strftime('%Y-%m-%d')
end_date = end_date.strftime('%Y-%m-%d')

# Query Cost Explorer for total costs
response = ce.get_cost_and_usage(
    TimePeriod={
        'Start': start_date,
        'End': end_date
    },
    Granularity='MONTHLY',  # or DAILY for finer granularity
    Metrics=['BlendedCost']
)

# Extract the cost data
costs = response['ResultsByTime'][0]['Total']['BlendedCost']['Amount']
unit = response['ResultsByTime'][0]['Total']['BlendedCost']['Unit']

print(f"Total AWS cost for the last 30 days: {costs} {unit}")
```

This script provides us with a monthly overview of our AWS costs, which we use to track our spending trends and identify any unexpected increases.

7.2.1 SNS Notification System

To ensure timely awareness of our cost situation, we've set up an SNS topic for cost notifications. This allows us to receive email alerts about our AWS costs. Here's how we implemented it:

```
sns = session.client('sns')

# Create a new SNS topic
response = sns.create_topic(Name='CostNotifications')
topic_arn = response['TopicArn']

# Subscribe your email to the topic
sns.subscribe(
    TopicArn=topic_arn,
    Protocol='email',
    Endpoint=MY_EMAIL  # your email address
)

# Publish a test message to the topic
sns.publish(
    TopicArn=topic_arn,
    Message='This is a test notification for cost monitoring',
    Subject='Cost Alert'
)

print("SNS setup and test notification sent")
```

This setup allows us to:

1. Create a dedicated SNS topic for cost notifications
2. Subscribe relevant team members' email addresses to the topic
3. Send test notifications to ensure the system is working correctly

7.2.2 Automated Cost Alerts

Building upon this foundation, we've implemented automated cost alerts:

1. **Daily Cost Check**: We run the Cost Explorer script daily to fetch the latest cost data.
2. **Threshold-based Alerts**: If the daily or monthly costs exceed predefined thresholds, we trigger an SNS notification. For example:

   ```
   if float(costs) > MONTHLY_THRESHOLD:
       sns.publish(
           TopicArn=topic_arn,
           Message=f"Monthly AWS costs have exceeded the threshold: {costs} {unit}",
           Subject='High Cost Alert'
       )
   ```

3. **Trend Analysis**: We store historical cost data and analyze trends. If we detect a significant increase in costs compared to previous periods, we send an alert.
4. **Service-specific Monitoring**: We've extended our Cost Explorer queries to break down costs by service, allowing us to identify which services are contributing most to our expenses.

By implementing these cost monitoring and alert systems, we've achieved:

- Real-time visibility into our AWS spending
- Proactive cost management through timely notifications
- Ability to quickly identify and address cost anomalies
- Improved budget forecasting and resource allocation

This system complements our AWS Budgets setup, providing a comprehensive approach to cost management and ensuring we maintain optimal cost-efficiency in our cloud operations.

8. Security and Compliance

8.1 CloudWatch Alarms for S3 Monitoring

We've implemented CloudWatch alarms to monitor our S3 bucket usage and ensure compliance with storage limits:

1. S3 Object Count Alarm:

- Set up to trigger when the number of objects in the S3 bucket exceeds 10,000
- Helps maintain control over data growth and prevent unexpected storage costs

2. S3 Bucket Size Alarm:
   - Configured to alert when the bucket size exceeds 1 GB
   - Ensures compliance with storage quotas and helps manage costs
3. SNS Integration:
   - Created an SNS topic 'S3AlarmsTopic' for alarm notifications
   - Subscribed relevant team members' email addresses to receive immediate alerts

These alarms enhance our ability to proactively manage storage usage and maintain compliance with data retention policies.

9. Performance Monitoring and Management

9.1 Monitoring Tools Implementation

9.1.1 Amazon CloudWatch

We've expanded our use of CloudWatch to include detailed S3 bucket monitoring:

1. Bucket Size Metrics:

```
response = cloudwatch.get_metric_statistics(
    Namespace='AWS/S3',
    MetricName='BucketSizeBytes',
    Dimensions=[
        {'Name': 'BucketName', 'Value': bucket_name},
        {'Name': 'StorageType', 'Value': storage_type}
    ],
    StartTime=datetime.utcnow() - timedelta(days=1),
    EndTime=datetime.utcnow(),
    Period=86400,
    Statistics=['Average']
)
```

   - Tracks the average size of our S3 bucket over time
   - Helps in capacity planning and cost forecasting

2. Object Count Alarm:

```
cloudwatch.put_metric_alarm(
    AlarmName='S3ObjectCountAlarm',
    MetricName='NumberOfObjects',
    Namespace='AWS/S3',
    Dimensions=[
        {'Name': 'BucketName', 'Value': bucket_name},
        {'Name': 'StorageType', 'Value': storage_type}
    ],
    Statistic='Average',
    Period=86400,
    EvaluationPeriods=1,
    Threshold=object_threshold,
    ComparisonOperator='GreaterThanThreshold',
    AlarmActions=[f'arn:aws:sns:{AWS_DEFAULT_REGION}:{ACCOUNT_ID}:greater-than-threshold'],
    AlarmDescription='Alarm when number of objects in S3 exceeds 10,000',
    ActionsEnabled=True
)
```

   - Alerts when the number of objects in the S3 bucket exceeds a predefined threshold
   - Helps maintain optimal performance and prevents potential issues with large object counts

3. Bucket Size Alarm:

```
cloudwatch.put_metric_alarm(
    AlarmName='S3BucketSizeAlarm',
    MetricName='BucketSizeBytes',
    Namespace='AWS/S3',
    Dimensions=[
        {'Name': 'BucketName', 'Value': bucket_name},
        {'Name': 'StorageType', 'Value': storage_type}
    ],
    Statistic='Average',
    Period=86400,
    EvaluationPeriods=1,
    Threshold=alarm_threshold,
    ComparisonOperator='GreaterThanThreshold',
    AlarmActions=[f'arn:aws:sns:{AWS_DEFAULT_REGION}:{ACCOUNT_ID}:1-GB-exceeded'],
    AlarmDescription='Alarm when S3 bucket size exceeds 1 GB',
    ActionsEnabled=True
)
```

- Triggers an alert when the S3 bucket size exceeds 1 GB
- Ensures proactive management of storage costs and capacity

9.2Automated Alerting System

We've implemented an automated alerting system using Amazon SNS:

1. SNS Topic Creation:

```
response = sns.create_topic(Name='S3AlarmsTopic')
topic_arn = response['TopicArn']
```

   - Created a dedicated SNS topic for S3-related alarms

2. Email Subscription:

```
response = sns.subscribe(
    TopicArn=topic_arn,
    Protocol='email',
    Endpoint=MY_EMAIL
)
```

   - Subscribed relevant team members to receive email notifications
   - Ensures immediate awareness of any performance issues or threshold breaches

This automated alerting system allows for rapid response to any performance anomalies or capacity issues, maintaining optimal system performance and cost-efficiency.

10. Conclusion

This pipeline demonstrates a comprehensive approach to stock market data analysis and prediction, incorporating data preprocessing, exploratory data analysis, and machine learning modelling. The use of cloud services enables scalable processing and storage of big data.

11. References

1. pandas documentation: https://pandas.pydata.org/docs/
2. Matplotlib documentation: https://matplotlib.org/stable/contents.html
3. XGBoost documentation: https://xgboost.readthedocs.io/
4. Optuna documentation: https://optuna.readthedocs.io/
5. AWS SDK for Python (Boto3) documentation: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html
6. yfinance documentation: https://pypi.org/project/yfinance/