Project Abstract: An Automated Investment Insights Dashboard

Student: Brandyn Ewanek **Matriculation:** 9216750

Part 1: Conceptual Overview and Architectural Journey

Project Concept and Goals

In today's fast-paced financial markets, investors require timely, consolidated, and actionable data to make informed decisions. This project addresses this need by creating a comprehensive Investment Insights Dashboard. The core concept is to develop an automated system that aggregates disparate financial data points—including historical stock performance, analyst price targets, and real-time market sentiment—into a single, interactive visualization platform. The primary goal is to empower a user by replacing time-consuming manual data gathering with a "single pane of glass" view, enabling them to efficiently analyze market trends, evaluate individual equities, and understand the sentiment surrounding key companies and global economic events.

The dashboard is designed to source data from multiple APIs, process it into a consistent format, and present it through a leading business intelligence tool. Key data sources include the Yahoo Finance API for quantitative metrics like historical stock prices and analyst ratings, and Google's Gemini API for generating sentiment scores based on recent company-specific and global financial news. The final output is an interactive Power BI dashboard that allows for dynamic filtering and exploration of the collected data.

The "Making Of": A Journey from Azure Serverless to AWS Virtual Machine

The initial architectural design for this project was centered on Microsoft Azure. This decision was strategically made based on the platform's significant and growing popularity within the Canadian technology landscape. As an aspiring data science professional in Canada, gaining proficiency in Azure's ecosystem, was identified as a valuable career asset. The proposed solution involved using Azure Functions, a serverless compute service, to run the Python data collection script on a daily schedule. This serverless approach was theoretically ideal due to its cost-effectiveness (paying only for execution time) and scalability.

However, this initial approach encountered a significant and persistent roadblock. Upon deployment, the Azure Function would crash immediately on invocation, providing literally no logs or errors. For several months, troubleshooting efforts were focused on a perceived issue with the Python programming model, specifically the differences and migration complexities between Azure Functions' v2 and v3 frameworks. This led to a prolonged cycle of refactoring code, adjusting dependencies, and researching programming model intricacies, all without resolving the core instability.

The project reached a turning point with the decision to pivot to Amazon Web Services (AWS) to diagnose the problem in a different environment. The chosen architecture on AWS was

intentionally simpler and more direct: an EC2 virtual machine. By setting up the Python environment on this VM, the script could be executed with full visibility into its resource consumption and dependency behavior. This move provided the crucial insight that had been elusive on Azure. The AWS environment immediately revealed that the collection of Python libraries required for the project (`pandas`, `yfinance`, `google-generativeai`, `awswrangler`, etc.) resulted in a dependency package size that was too large for the constraints of a standard serverless function environment.

This realization clarified the entire problem: the issue was not the Python code's logic or the Azure programming model (v2 vs. v3), but a fundamental architectural mismatch. The complexity and size of the script's dependencies were better suited for a more flexible environment like a virtual machine, where there are no restrictive package size limits. This journey underscored a critical and not easily forgot lesson in cloud architecture: understanding the trade-offs between serverless functions (ideal for lightweight, fast-executing tasks) and virtual machines (ideal for applications with large dependencies or complex, long-running processes).

Part 2: Final Technical Approach on AWS

The successful architecture is a robust, event-driven pipeline built entirely on AWS, designed for automation, scalability, and data integrity.

Data Flow and High-Level Architecture

1. **Scheduling & Automation:** A `cron` job on the EC2 instance triggers the Python script daily at 11:00 AM Toronto time on weekdays. This provides reliable, scheduled execution. An alternative cloud-native approach using Amazon EventBridge was also designed as a future enhancement.

2. **Compute & Data Acquisition:** An Amazon EC2 instance (t2.micro) runs a central Python script (`stockCollectionFunction.py`). This script uses the `yfinance` library to pull quantitative stock data and the `google-generativeai` library to make API calls to Gemini for sentiment analysis scores.

3. **Data Processing & Storage:** The script uses the `pandas` library to structure the collected data into DataFrames. It then uses the `awswrangler` library to write this data as CSV files into an Amazon S3 bucket (`stock-data-collection-bucket`). A dual-save strategy is employed:
    - **Archive:** A dated copy of the daily data is saved to an `archive/` folder to maintain a complete historical record (e.g., `archive/financial_metrics/YYYY-MM-DD_financial_metrics.csv`).

    - **Latest:** The full appended historical data is saved to a `latest/` folder, overwriting the file each day (e.g., `latest/financial_metrics_table/financial_metrics.csv`). This provides a stable endpoint for the BI tool.
4. **Schema Definition & Querying:**
    - **AWS Glue Crawler:** To avoid manual `CREATE TABLE` errors and handle schema inference robustly, a Glue Crawler is configured to run against the S3 paths of the "latest" CSV files. It automatically detects the schema (column names and data types) and creates corresponding tables in the AWS Glue Data Catalog.
    - **Amazon Athena:** This serverless query engine uses the tables defined by the Glue Crawler to run standard SQL queries directly on the CSV files stored in S3. This decouples the query engine from the data storage and is highly cost-effective, as billing is per query.
5. **Visualization & Reporting:**
    - **Microsoft Power BI** connects to Amazon Athena as an ODBC data source.
    - Power BI sends SQL queries to Athena, which then reads the "latest" data from S3 and returns the results.
    - This populates the visuals on the dashboard, providing the end-user with an interactive and up-to-date view of the investment data.

Key Component Details

- **Compute (Amazon EC2):** A lightweight `t2.micro` instance running Amazon Linux 2023 provides the necessary environment. A Python virtual environment (`venv`) is used to isolate dependencies, preventing conflicts. An **IAM Role** is attached to the instance, granting it secure, key-free access to read and write to the designated S3 bucket.

- **Automation (`cron`):** The `crontab` on the EC2 instance is configured to run the script on a schedule (`0 15 * * 1-5` for 11 AM EDT). The command line explicitly sources the `~/.bashrc` file (to load the Gemini API key) and activates the Python virtual environment before running the script, ensuring the correct context.

- **Security (AWS IAM):** The project utilizes multiple IAM roles and users based on the principle of least privilege:
    - An **IAM Role for EC2** grants the virtual machine permissions to access S3.
    - A separate **IAM Role for AWS Glue** grants the crawler permissions to read S3 and write to the Glue Data Catalog.
    - A dedicated **IAM User** with programmatic access keys is created for Power BI, granting it `AmazonAthenaFullAccess` and specific S3 read permissions.

- **Infrastructure as Code (IaC):** The entire AWS infrastructure—including the S3 bucket, all IAM roles and policies, the EC2 instance with its UserData setup script, security group, Glue Database, and Glue Crawlers—has been defined in a comprehensive **AWS**

CloudFormation template (`infrastructure.yaml`). This allows for the entire backend to be deployed, updated, or recreated in an automated and repeatable manner.

Conclusion

This project successfully achieved its goal of creating a fully automated data pipeline and insights dashboard. The architectural pivot from Azure Functions to an AWS EC2 instance was a critical and insightful part of the development process, highlighting the practical trade-offs between serverless and virtual machine-based architectures, especially when dealing with complex dependencies. The final AWS architecture is robust, scalable, and leverages cloud-native services like S3, IAM, Glue, and Athena to create an efficient and cost-effective solution.