Documentation for Data Pipeline Solution

# Overview

This document provides a detailed overview of the solution for the given task involving the design and implementation of a data pipeline for questionnaire data. The pipeline processes questionnaire data to produce clean, tabular data for researchers.

# Task Assumptions

- Questionnaire answers are uncleaned and unprocessed.  
- Only fully completed questionnaires are included in the research data sets.  
- New questionnaire data is available daily.  
- Regular data quality reports are required.  
- Quarterly datasets are provided to researchers.

# Solution Components

## High-Level Design

- Data Ingestion Pipeline: Ingests new questionnaire data daily from the cloud-based environment.  
- Data Cleaning Pipeline: Processes and cleans the ingested data, ensuring only fully completed questionnaires are included.  
- Data Aggregation Pipeline: Aggregates data quarterly for research purposes.  
- Data Quality Reporting Pipeline: Generates regular reports on data quality.

## Proof of Concept

- Scripts: Python scripts to perform data ingestion, cleaning, and orchestration.  
- Workflow Management: Instructions provided for running the scripts with or without a workflow manager (Dagster/Airflow/Prefect).

# Data Pipeline Stages:

## Data Ingestion:

Function: Downloads new questionnaire data daily from the cloud environment.

```python  
import requests  
import json  
  
def ingest\_data():  
 url = "cloud\_environment\_url"  
 response = requests.get(url)  
 data = response.json()  
 with open('data/synthetic\_questionnaire.json', 'w') as file:  
 json.dump(data, file)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 ingest\_data()  
```

## Data Ingestion for Local Environment:

The function `load\_data` reads and loads JSON data from two specified file paths.

The `questionnaire\_path` parameter is the file path to the JSON file containing the questionnaire data that you want to load. This function reads the JSON data from this file and returns it

The `answers\_path` parameter in the `load\_data` function is the file path to the JSON file containing the answers to the questionnaire. When the function is called, it will read the data from this file to load the answers The function `load\_data` returns two variables: `questionnaire` and `answers`.

```python

def load\_data(questionnaire\_path, answers\_path):

    with open(questionnaire\_path, "r") as file:

        questionnaire = json.load(file)

    with open(answers\_path, "r") as file:

        answers = json.load(file)

    return questionnaire, answers

```

## Transform Data:

The function `transform\_answers\_to\_df` takes a list of answers, extracts the 'pid' and 'answers' data from each entry, adds the 'pid' to the 'answers' data, and returns a pandas DataFrame.

The `transform\_answers\_to\_df` function is designed to transform a list of answers into a pandas DataFrame. However, the `pd` module is not imported in the code snippet provided. You will need to import pandas as pd at the beginning of your script for this function to work correctly

A DataFrame containing the answers data with an additional column 'pid' representing the participant ID.

```python  
  
def transform\_answers\_to\_df(answers):  
 answer\_list = []

    for entry in answers:

        pid = entry["pid"]

        answer\_data = entry["answers"]

        answer\_data["pid"] = pid

        answer\_list.append(answer\_data)

    return pd.DataFrame(answer\_list)

```

## Clean Data:

The function `clean\_data` takes a DataFrame and a list of question IDs, drops records where not all questions have been answered, and returns the cleaned DataFrame.

The `df` parameter is typically a pandas DataFrame containing the data that needs to be cleaned.

The `question\_ids` parameter is a list of question IDs that should have non-missing values in the DataFrame. The cleaned DataFrame with incomplete records removed.

```python  
# Ensure all questions have been answered

for question\_id in question\_ids:

df = df[df[question\_id].notna()] # Remove records where any question is not answered

# Further remove rows where any value is 'NA'

df.replace('NA', pd.NA, inplace=True)

df.dropna(inplace=True)

return df  
```

## Save Data:

The function `save\_data` saves a DataFrame to a CSV file at the specified output path and prints a message confirming the completion of data processing.

In the `save\_data` function provided, the DataFrame `df` is being saved to a CSV file. The `output\_path` parameter is the file path where the cleaned data will be saved after processing. It is the location where the CSV file containing the cleaned data will be stored

```python  
  
  
def save\_data(df, output\_path):

df.to\_csv(output\_path, index=False)

    print(f"Data processing complete. Cleaned data saved to '{output\_path}'.")

```

# Workflow Management:

Using Prefect:

```python  
from prefect import task, Flow  
  
@task  
def load\_data():  
 # Implementation of data ingestion  
 pass  
  
@task  
def transform\_answers\_to\_df():  
 # Implementation of data cleaning  
 pass  
  
@task  
def clean\_data():  
 # Implementation of data cleaning  
 pass  
  
@flow  
def health\_study\_data\_pipeline():  
 # The function `health\_study\_data\_pipeline` processes health study data by loading, transforming, cleaning, and saving it.  
 pass

```  
**Installation and Running Instructions:**

## Using Dockerfile:

Create a Dockerfile with the necessary dependencies. Build and run the Docker container.

## Description:

The data pipeline processes questionnaire data, ensuring it is clean and ready for research. The pipeline consists of ingestion, cleaning, aggregation, and quality reporting steps.

## Instructions:

## Setup:

# 1. Open the project in VSCode:

# Open Visual Studio Code (VSCode).

# Use the File menu to open the project directory where your code is located.

# Docker Setup: Ensure Docker is installed: Download and install Docker from [Docker's official website](https://www.docker.com/get-started).

# 2. Create the `Dockerfile`: Ensure you have a file named `Dockerfile` in your project directory with the following content:

# ```dockerfile

# FROM python:3.9-slim # Use the official Python image from the Docker Hub

# WORKDIR /app # Set the working directory in the container

# COPY requirements.txt /app/ # Copy the requirements file into the container

# RUN pip install --no-cache-dir -r requirements.txt # Install the required Python packages

# COPY . /app # Copy the entire project directory into the container

# CMD ["python", "prefect\_flow.py"] # Command to run the Prefect flow

```

# 3. Open a terminal and navigate to your project directory:

# ```bash

# cd /path/to/your/project

# ```

# Replace `/path/to/your/project` with the actual path to your project directory.

# 4. Build the Docker image:

# Run the following command to build the Docker image:

# ```bash

# docker build -t health\_study\_pipeline .

# ```

# 5. Run the Docker container:

# After the image is built, start a container from the image:

# On Windows:

# ```bash

# docker run health\_study\_pipeline

# ```

# 6. Verify the output:

# After running the Docker container, you should see the log output indicating the flow is running and the data processing is complete. The cleaned data should be saved in the `cleaned\_questionnaire\_data.csv` file in your project directory.

# Data Files:

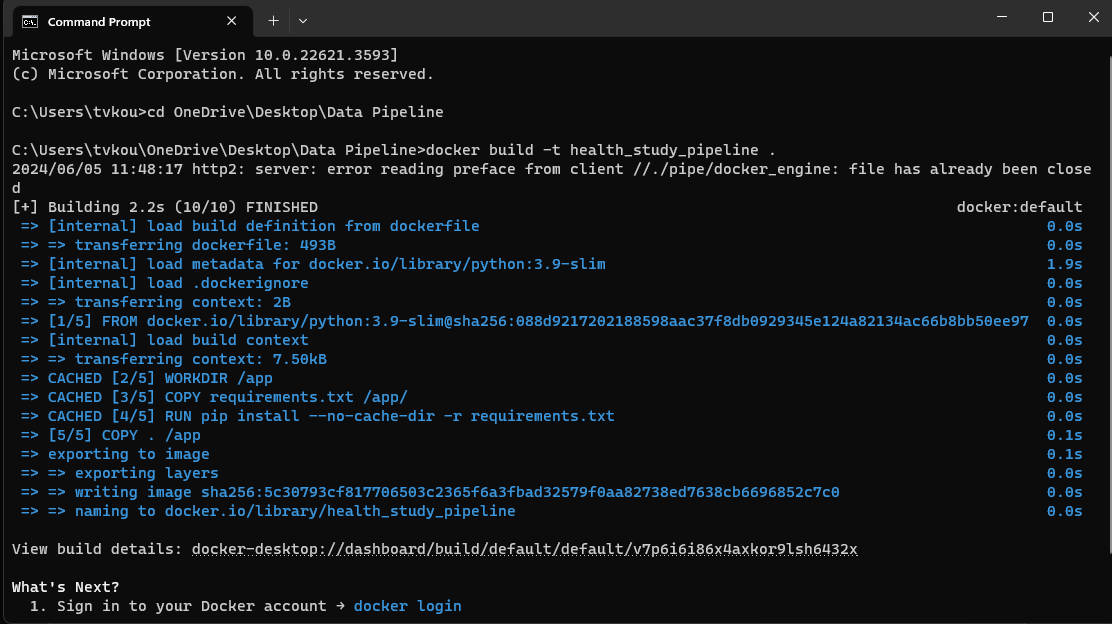
# `-synthetic\_questionnaire.json`: Contains the questionnaire structure and questions.

# - `synthetic\_answers.json`: Contains the answers provided by participants.

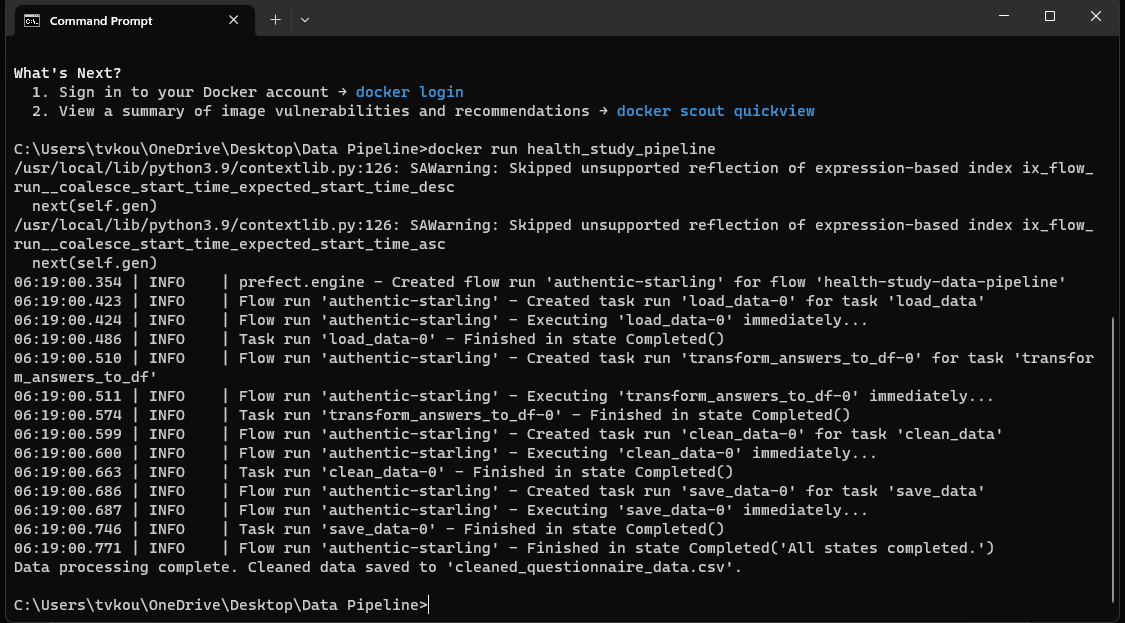
# Demo:

Pre-generated images or a live demo to showcase the proof of concept.

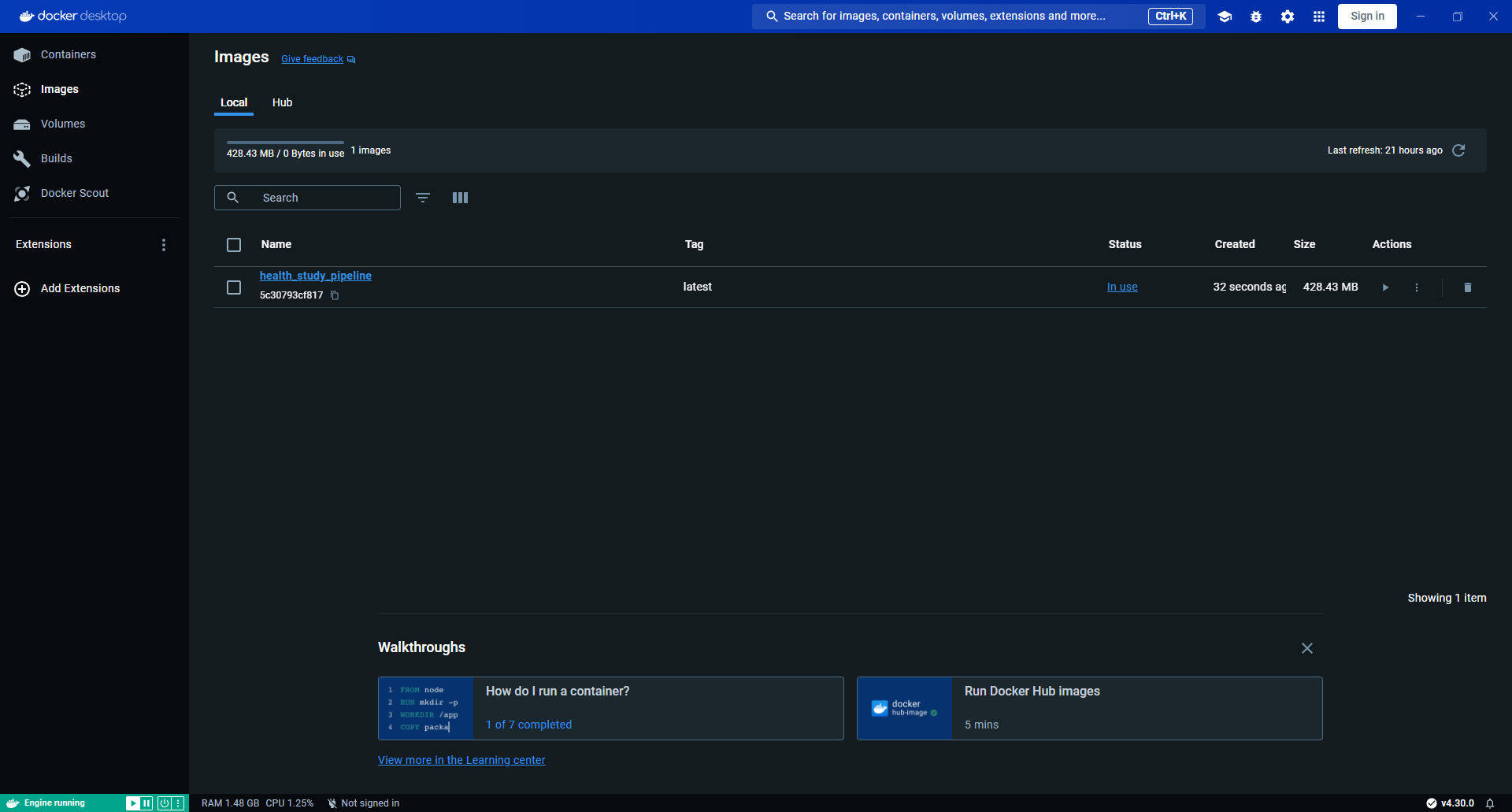
1.Docker Image Build:

****

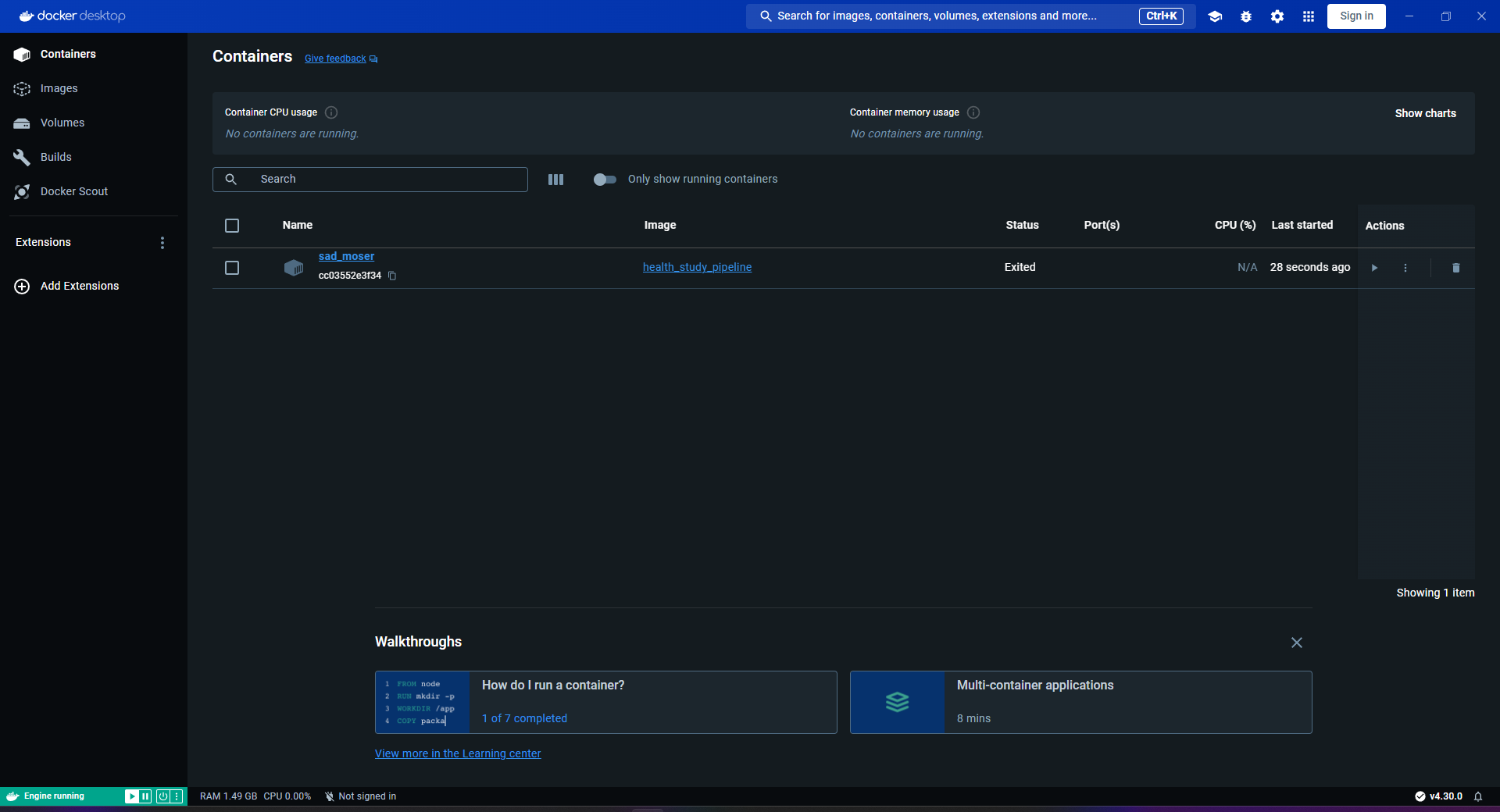
2.Docker Image Run:



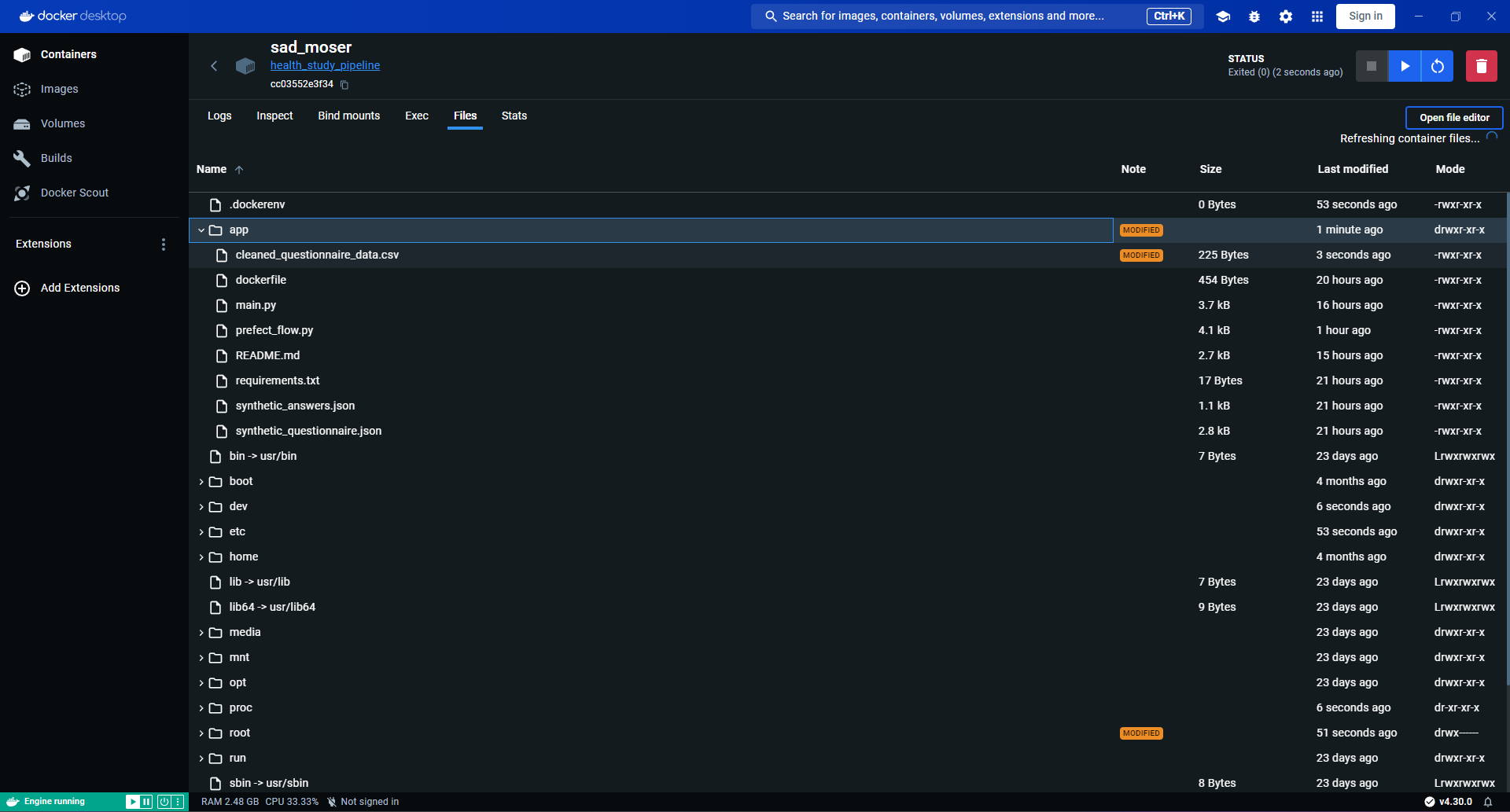
3.Docker Desktop Image:



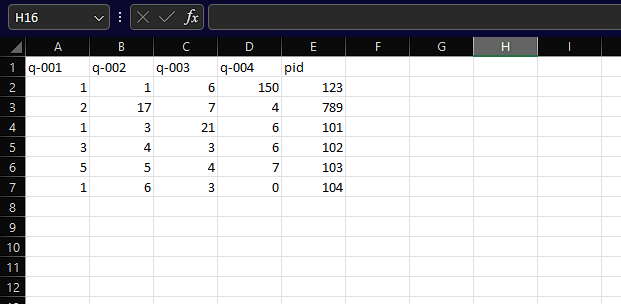
4.Docker Desktop Containers:



5.Final App Directory:



6.Final Cleaned CSV File:



# Questions and Answers

## How did you find the task? What was challenging? What was easy?

Finding the Task: The task was engaging and well-defined, providing a clear objective and steps to follow. It was a good balance between practical implementation and theoretical design.

Challenging Aspects:

- Ensuring data quality and completeness was challenging, particularly the need to filter out incomplete questionnaires.

- Designing a scalable solution that can handle daily data ingestion and quarterly aggregation.

- Balancing the implementation detail within the given timeframe while ensuring the solution is comprehensive and robust.

Easy Aspects:

- Setting up the environment and basic script structure was straightforward.

- Implementing the initial data ingestion and cleaning steps was relatively easy with the provided data format.

## How would you test your code? What practices would you use to ensure your pipeline scales? How would you set up CI testing?

Testing Code:

- Implement unit tests(pytest) for each function in the scripts to ensure they work as expected.

- Use mock data to simulate different scenarios, including edge cases and incomplete data.

- Perform integration testing to ensure that all components of the pipeline work together seamlessly.

Scaling Practices:

- Use distributed processing frameworks like Apache Spark to handle large datasets efficiently.

- Optimize data processing steps to minimize bottlenecks and improve performance.

- Implement robust error handling and retry mechanisms to ensure the pipeline's resilience.

CI Testing:

- Set up a CI/CD pipeline using tools like GitHub Actions or Jenkins to automate testing and deployment.

- Integrate automated tests to run on every commit to catch issues early.

- Include performance tests to monitor the pipeline's scalability and efficiency.

## The size of the data means that our processing occurs using Spark which we access through Azure Databricks. How would you amend your approach in this context.

Amendments for Spark on Azure Databricks:

- Convert the data processing scripts to use PySpark instead of pandas to leverage Spark's distributed processing capabilities.

- Use Azure Databricks notebooks to develop, test, and deploy the PySpark scripts.

- Utilize Databricks workflows for orchestrating the data pipelines, ensuring they run efficiently in the distributed environment.

- Store intermediate data in Azure Data Lake Storage Gen 2 (ADLS) to ensure scalability and accessibility.

- Optimize Spark configurations (e.g., partitioning, caching) to improve performance for large datasets.

## When it comes to building the pipelines, we would expect several engineers to be contributing to the codebase. What ways of working would you want the team to be using? What would you want the team to avoid doing?

Ways of Working:

- Use version control (e.g., Git) to manage code changes and facilitate collaboration.

- Conduct regular code reviews to ensure code quality and knowledge sharing.

- Implement a consistent coding style and enforce it through linters and code formatting tools.

- Document code thoroughly to make it easier for team members to understand and maintain.

- Encourage pair programming and knowledge sharing sessions to foster collaboration and continuous learning.

Things to Avoid:

- Avoid working in isolation without regular communication and synchronization with the team.

- Avoid merging untested or partially tested code into the main branch.

- Avoid neglecting documentation, as it is crucial for maintaining and scaling the codebase.

- Avoid creating monolithic scripts; instead, modularize code to improve maintainability and testability.

## What logging would you expect to be in place for the production system? How might you use these logs?

Expected Logging:

- Detailed logs at each step of the pipeline, including data ingestion, processing, and aggregation.(logging library of python/CloudWatch/Graphana)

- Error logs capturing exceptions and failures, along with relevant context for troubleshooting.

- Performance logs to monitor the execution time and resource usage of each pipeline step.

- Audit logs to track changes in the pipeline configuration and data access.

Using Logs:

- Use centralized log management tools (e.g., ELK stack) to aggregate and analyze logs.

- Set up alerts for critical errors or performance degradation to enable quick response. (datadog)

- Regularly review logs to identify patterns and potential improvements in the pipeline.

- Use logs to generate reports on pipeline health, performance, and data quality for stakeholders.

# Conclusion:

This documentation provides a comprehensive overview of the solution for the data pipeline task, including high-level design, proof of concept, scripts, workflow management, and additional considerations for testing, scaling, and team collaboration.