### **Part 1: Project Setup and Environment**

A well-structured project is crucial for reproducibility and maintainability. This setup ensures that the project can be run consistently across different machines.1

#### **1.1. Directory Structure**

Start by creating a root folder for your project. Inside, create the following subdirectories. This organization separates concerns, making the project easier to navigate.2

ai\_transport\_project/  
├──.venv/ # Virtual environment files (will be created)  
├── data/  
│ ├── raw/ # Raw GTFS zip files  
│ └── processed/ # Cleaned or intermediate data  
├── notebooks/ # Jupyter notebooks for exploration and analysis  
├── src/  
│ ├── \_\_init\_\_.py  
│ ├── data/  
│ │ ├── \_\_init\_\_.py  
│ │ └── load\_data.py # Scripts for data ingestion and preprocessing  
│ ├── models/  
│ │ ├── \_\_init\_\_.py  
│ │ ├── cluster.py # K-Means clustering model  
│ │ ├── forecast.py # ARIMA forecasting model  
│ │ └── optimize.py # NetworkX graph optimization  
│ └── app.py # The Streamlit web application  
├── tests/ # Unit and integration tests  
├──.dockerignore # Files to ignore in Docker context  
├──.gitignore # Files to ignore for Git  
├── docker-compose.yml # Docker configuration for services  
├── Dockerfile # Docker configuration for the web app  
└── requirements.txt # Python package dependencies

#### **1.2. Virtual Environment**

Using a virtual environment is essential to isolate project dependencies and avoid conflicts with other Python projects on your system.4

1. **Navigate to your project's root directory:**  
   Bash  
   cd ai\_transport\_project
2. **Create a virtual environment using venv:**  
   Bash  
   python -m venv.venv
3. **Activate the environment:**
   * **On macOS/Linux:**  
     Bash  
     source.venv/bin/activate
   * **On Windows:**

.venv\Scripts\activate```Your shell prompt will now be prefixed with (.venv), indicating the environment is active.

#### **1.3. Dependency Management**

Create a requirements.txt file in the root directory. This file will list all the Python libraries your project needs. Start with the following core packages:

**requirements.txt**

pandas  
geopandas  
sqlalchemy  
psycopg2-binary  
scikit-learn  
statsmodels  
networkx  
streamlit  
pydeck

Install these packages into your active virtual environment using pip 6:

Bash

pip install -r requirements.txt

### **Part 2: Geospatial Database Setup with Docker and PostGIS**

We will use Docker to run a PostgreSQL database with the PostGIS extension. This containerized approach ensures a consistent and isolated database environment that is easy to set up and manage.7

1. **Create a docker-compose.yml file** in your project's root directory. This file defines the services, networks, and volumes for your application.9  
   **docker-compose.yml**  
   YAML  
   version: '3.8'  
     
   services:  
    db:  
    image: postgis/postgis:14-3.3  
    container\_name: postgis\_db  
    environment:  
    - POSTGRES\_USER=user  
    - POSTGRES\_PASSWORD=password  
    - POSTGRES\_DB=gtfs\_db  
    ports:  
    - "5432:5432"  
    volumes:  
    - postgis\_data:/var/lib/postgresql/data  
    restart: always  
     
   volumes:  
    postgis\_data:  
   * image: postgis/postgis:14-3.3: Specifies the official PostGIS Docker image.
   * environment: Sets the username, password, and database name.
   * ports: Maps port 5432 on your local machine to port 5432 in the container.
   * volumes: Creates a persistent volume named postgis\_data to store the database contents, ensuring your data is saved even if the container is removed.9
2. **Start the database service.** From your project's root directory, run:  
   Bash  
   docker compose up -d  
     
   The -d flag runs the container in detached mode (in the background). Your PostGIS database is now running and ready to accept connections.

### **Part 3: Data Ingestion and Processing**

Now, we'll write a Python script to load your GTFS data into the PostGIS database.

1. **Acquire GTFS Data:** Download a GTFS zip file from a transit agency and place it in the data/raw/ directory.
2. **Create the Ingestion Script:** In src/data/load\_data.py, write the following code. This script will connect to the database, read the GTFS files using pandas, convert spatial data into GeoDataFrames, and upload everything to PostGIS.10  
   **src/data/load\_data.py**  
   Python  
   import pandas as pd  
   import geopandas as gpd  
   from sqlalchemy import create\_engine  
   from zipfile import ZipFile  
   import os  
     
   # --- Configuration ---  
   GTFS\_ZIP\_PATH = 'data/raw/your\_gtfs\_feed.zip'  
   DB\_CONNECTION\_STRING = 'postgresql://user:password@localhost:5432/gtfs\_db'  
     
   def load\_gtfs\_to\_postgis(gtfs\_path, db\_conn\_str):  
    """  
    Loads GTFS data from a zip file into a PostGIS database.  
    """  
    engine = create\_engine(db\_conn\_str)  
    print("Database engine created.")  
     
    # GTFS files to load  
    files\_to\_load = [  
    'agency.txt', 'routes.txt', 'trips.txt', 'calendar.txt',  
    'calendar\_dates.txt', 'stop\_times.txt', 'stops.txt', 'shapes.txt'  
    ]  
     
    with ZipFile(gtfs\_path) as myzip:  
    for file in files\_to\_load:  
    if file in myzip.namelist():  
    table\_name = os.path.splitext(file)  
    print(f"Processing {file} -> table '{table\_name}'...")  
     
    with myzip.open(file) as f:  
    df = pd.read\_csv(f)  
     
    # Handle spatial data for stops and shapes  
    if table\_name == 'stops':  
    gdf = gpd.GeoDataFrame(  
    df,  
    geometry=gpd.points\_from\_xy(df.stop\_lon, df.stop\_lat),  
    crs="EPSG:4326" # WGS84 CRS  
    )  
    gdf.to\_postgis(table\_name, engine, if\_exists='replace', index=False)  
    elif table\_name == 'shapes':  
    gdf = gpd.GeoDataFrame(  
    df,  
    geometry=gpd.points\_from\_xy(df.shape\_pt\_lon, df.shape\_pt\_lat),  
    crs="EPSG:4326"  
    )  
    gdf.to\_postgis(table\_name, engine, if\_exists='replace', index=False)  
    else:  
    df.to\_sql(table\_name, engine, if\_exists='replace', index=False)  
     
    print(f"Successfully loaded {table\_name} to PostGIS.")  
    else:  
    print(f"Warning: {file} not found in zip archive.")  
     
   if \_\_name\_\_ == '\_\_main\_\_':  
    load\_gtfs\_to\_postgis(GTFS\_ZIP\_PATH, DB\_CONNECTION\_STRING)  
   * geopandas.points\_from\_xy: Creates point geometries from longitude and latitude columns.12
   * crs="EPSG:4326": Sets the Coordinate Reference System to WGS 84, the standard for GPS data.
   * to\_postgis: A GeoDataFrame method to write spatial data directly to PostGIS.13
3. **Run the script** from the project's root directory:  
   Bash  
   python src/data/load\_data.py  
     
   Your GTFS data is now stored in your PostGIS database.

### **Part 4: Core Modeling Logic**

This section covers the implementation of the project's three main analytical components.

#### **4.1. Demand Clustering (K-Means)**

In src/models/cluster.py, we'll write a function to fetch stop data, apply K-Means clustering, and identify high-demand zones.

**src/models/cluster.py**

Python

import geopandas as gpd  
from sklearn.cluster import KMeans  
from sklearn.preprocessing import StandardScaler  
from sqlalchemy import create\_engine  
  
def cluster\_stops(db\_conn\_str, n\_clusters=10):  
 """  
 Performs K-Means clustering on transit stops.  
 """  
 engine = create\_engine(db\_conn\_str)  
   
 # Load stops data from PostGIS  
 sql = "SELECT stop\_id, stop\_name, geometry FROM stops;"  
 stops\_gdf = gpd.read\_postgis(sql, engine, geom\_col='geometry', crs="EPSG:4326")  
   
 # Feature Engineering: Use coordinates as features  
 features = stops\_gdf[['geometry']].copy()  
 features['lon'] = features.geometry.x  
 features['lat'] = features.geometry.y  
   
 # Scale features  
 scaler = StandardScaler()  
 scaled\_features = scaler.fit\_transform(features[['lon', 'lat']])  
   
 # K-Means clustering  
 kmeans = KMeans(n\_clusters=n\_clusters, random\_state=42, n\_init=10)  
 stops\_gdf['cluster'] = kmeans.fit\_predict(scaled\_features)  
   
 print(f"Clustered {len(stops\_gdf)} stops into {n\_clusters} clusters.")  
   
 # Optional: Save clustered data back to DB  
 # stops\_gdf[['stop\_id', 'cluster']].to\_sql('stop\_clusters', engine, if\_exists='replace', index=False)  
   
 return stops\_gdf

#### **4.2. Flow Forecasting (ARIMA)**

In src/models/forecast.py, we'll prepare time-series data and train an ARIMA model. This is a simplified example; a real implementation would involve more complex data aggregation and parameter tuning.

**src/models/forecast.py**

Python

import pandas as pd  
from statsmodels.tsa.arima.model import ARIMA  
from sqlalchemy import create\_engine  
import pickle  
  
def train\_arima\_model(db\_conn\_str, cluster\_id=0):  
 """  
 Trains an ARIMA model for a specific cluster's passenger flow.  
 """  
 engine = create\_engine(db\_conn\_str)  
   
 # This SQL is a placeholder for a complex query that would aggregate  
 # trip arrivals per hour for a given cluster.  
 sql = f"""  
 SELECT   
 TO\_CHAR(TO\_TIMESTAMP(st.arrival\_time, 'HH24:MI:SS'), 'HH24') as hour,  
 COUNT(st.trip\_id) as trip\_count  
 FROM stop\_times st  
 JOIN stops s ON st.stop\_id = s.stop\_id  
 -- JOIN stop\_clusters sc ON s.stop\_id = sc.stop\_id WHERE sc.cluster = {cluster\_id}  
 GROUP BY hour  
 ORDER BY hour;  
 """  
 # For this example, we'll create dummy data.  
 # In a real scenario, you would execute the SQL query:  
 # hourly\_flow = pd.read\_sql(sql, engine)  
   
 # Dummy time series data  
 data = {'trip\_count': }  
 hourly\_flow = pd.DataFrame(data)  
  
 # Fit ARIMA model  
 # Parameters (p,d,q) would be determined by ACF/PACF analysis  
 model = ARIMA(hourly\_flow['trip\_count'], order=(5, 1, 0))  
 model\_fit = model.fit()  
   
 print(model\_fit.summary())  
   
 # Save the trained model  
 with open(f'data/processed/arima\_model\_cluster\_{cluster\_id}.pkl', 'wb') as pkl:  
 pickle.dump(model\_fit, pkl)  
   
 print(f"ARIMA model for cluster {cluster\_id} trained and saved.")  
 return model\_fit

#### **4.3. Route Optimization (NetworkX)**

In src/models/optimize.py, we build the graph and find the shortest path using Dijkstra's algorithm.15

**src/models/optimize.py**

Python

import networkx as nx  
import geopandas as gpd  
from sqlalchemy import create\_engine  
import pandas as pd  
  
def build\_transit\_graph(db\_conn\_str):  
 """  
 Builds a NetworkX graph from GTFS data.  
 """  
 engine = create\_engine(db\_conn\_str)  
   
 # Load stops as nodes  
 stops\_gdf = gpd.read\_postgis("SELECT stop\_id, geometry FROM stops;", engine, geom\_col='geometry')  
   
 G = nx.DiGraph()  
 for \_, row in stops\_gdf.iterrows():  
 G.add\_node(row['stop\_id'], pos=(row.geometry.x, row.geometry.y))  
   
 # Load transit segments as edges  
 # This query gets consecutive stops on the same trip and calculates travel time  
 sql\_edges = """  
 WITH ranked\_stops AS (  
 SELECT  
 trip\_id,  
 stop\_id,  
 EXTRACT(EPOCH FROM TO\_TIMESTAMP(arrival\_time, 'HH24:MI:SS')) as arrival\_seconds,  
 stop\_sequence  
 FROM stop\_times  
 )  
 SELECT  
 t1.stop\_id as source,  
 t2.stop\_id as target,  
 (t2.arrival\_seconds - t1.arrival\_seconds) as travel\_time  
 FROM ranked\_stops t1  
 JOIN ranked\_stops t2 ON t1.trip\_id = t2.trip\_id AND t1.stop\_sequence = t2.stop\_sequence - 1  
 WHERE (t2.arrival\_seconds - t1.arrival\_seconds) > 0;  
 """  
 edges\_df = pd.read\_sql(sql\_edges, engine)  
   
 # Add edges with travel time as weight  
 for \_, row in edges\_df.iterrows():  
 G.add\_edge(row['source'], row['target'], weight=row['travel\_time'])  
   
 print(f"Graph built with {G.number\_of\_nodes()} nodes and {G.number\_of\_edges()} edges.")  
 return G  
  
def find\_optimal\_route(graph, start\_stop, end\_stop):  
 """  
 Finds the shortest path between two stops using Dijkstra's algorithm.  
 """  
 try:  
 path = nx.dijkstra\_path(graph, source=start\_stop, target=end\_stop, weight='weight')  
 length = nx.dijkstra\_path\_length(graph, source=start\_stop, target=end\_stop, weight='weight')  
 print(f"Shortest path found with length (seconds): {length}")  
 return path  
 except nx.NetworkXNoPath:  
 print("No path found between the specified stops.")  
 return None

### **Part 5: Deployment with a Streamlit Web App**

Finally, we'll create an interactive dashboard to use our model. Streamlit is excellent for quickly building data-centric web apps.17

1. **Create the application script src/app.py:**  
   **src/app.py**  
   Python  
   import streamlit as st  
   import pandas as pd  
   import geopandas as gpd  
   from sqlalchemy import create\_engine  
   from src.models.optimize import build\_transit\_graph, find\_optimal\_route  
     
   # --- App Configuration ---  
   st.set\_page\_config(page\_title="Public Transport Optimizer", layout="wide")  
   DB\_CONNECTION\_STRING = 'postgresql://user:password@db:5432/gtfs\_db' # Use service name 'db'  
     
   # --- Caching Data Loading ---  
   @st.cache\_resource  
   def load\_data():  
    """  
    Loads graph and stops data, caching the result.  
    """  
    engine = create\_engine(DB\_CONNECTION\_STRING)  
    graph = build\_transit\_graph(DB\_CONNECTION\_STRING)  
    stops\_df = pd.read\_sql("SELECT stop\_id, stop\_name, stop\_lat, stop\_lon FROM stops", engine)  
    return graph, stops\_df  
     
   st.title("AI-Powered Public Transport Route Optimizer")  
     
   # --- Load Data ---  
   try:  
    G, stops = load\_data()  
   except Exception as e:  
    st.error(f"Failed to connect to the database and load data. Please ensure the database container is running. Error: {e}")  
    st.stop()  
     
   # --- User Inputs ---  
   st.sidebar.header("Route Planner")  
   start\_stop\_name = st.sidebar.selectbox("Select Start Stop", options=stops['stop\_name'].unique())  
   end\_stop\_name = st.sidebar.selectbox("Select End Stop", options=stops['stop\_name'].unique())  
     
   if st.sidebar.button("Find Optimal Route"):  
    start\_stop\_id = stops[stops['stop\_name'] == start\_stop\_name]['stop\_id'].iloc  
    end\_stop\_id = stops[stops['stop\_name'] == end\_stop\_name]['stop\_id'].iloc  
     
    st.write(f"Finding route from \*\*{start\_stop\_name}\*\* to \*\*{end\_stop\_name}\*\*...")  
     
    # In a full implementation, you would load the ARIMA model,  
    # get a forecast for the selected time, and update graph weights here.  
     
    path = find\_optimal\_route(G, start\_stop\_id, end\_stop\_id)  
     
    if path:  
    st.success("Optimal Route Found!")  
     
    # Prepare data for map visualization  
    path\_df = stops[stops['stop\_id'].isin(path)].copy()  
    path\_df['order'] = path\_df['stop\_id'].apply(path.index)  
    path\_df = path\_df.sort\_values('order')  
     
    st.subheader("Route Sequence:")  
    st.dataframe(path\_df[['stop\_name', 'stop\_id']])  
     
    st.subheader("Route on Map:")  
    st.map(path\_df, latitude='stop\_lat', longitude='stop\_lon', size=50)  
    else:  
    st.error("Could not find a route between the selected stops.")
2. **Create a Dockerfile for the Streamlit app:** This file tells Docker how to build an image containing your web application and its dependencies.8  
   **Dockerfile**  
   Dockerfile  
   # Use the official Python image as a base  
   FROM python:3.9-slim  
     
   # Set the working directory  
   WORKDIR /app  
     
   # Copy the requirements file and install dependencies  
   COPY requirements.txt.  
   RUN pip install --no-cache-dir -r requirements.txt  
     
   # Copy the source code into the container  
   COPY./src /app/src  
   COPY./data /app/data  
     
   # Expose the port Streamlit runs on  
   EXPOSE 8501  
     
   # Command to run the Streamlit app  
   CMD ["streamlit", "run", "src/app.py"]
3. **Update docker-compose.yml to include the web app:** Add a new service for the Streamlit app and make it depend on the database.  
   **docker-compose.yml (Updated)**  
   YAML  
   version: '3.8'  
     
   services:  
    db:  
    image: postgis/postgis:14-3.3  
    container\_name: postgis\_db  
    environment:  
    - POSTGRES\_USER=user  
    - POSTGRES\_PASSWORD=password  
    - POSTGRES\_DB=gtfs\_db  
    ports:  
    - "5432:5432"  
    volumes:  
    - postgis\_data:/var/lib/postgresql/data  
    restart: always  
     
    web:  
    build:.  
    container\_name: streamlit\_app  
    ports:  
    - "8501:8501"  
    depends\_on:  
    - db  
    restart: always  
     
   volumes:  
    postgis\_data:  
   * build:.: Tells Docker Compose to build the image from the Dockerfile in the current directory.
   * depends\_on: - db: Ensures the database container starts before the web app container.
4. **Launch the entire application stack:**  
   Bash  
   docker compose up --build  
     
   This command will build the image for your web app and start both the database and web app containers. Open your web browser and navigate to http://localhost:8501 to see your interactive dashboard in action.

#### Works cited

1. How to Structure Python Projects - Dagster, accessed on October 17, 2025, <https://dagster.io/blog/python-project-best-practices>
2. 5 Tips for Structuring Your Data Science Projects - KDnuggets, accessed on October 17, 2025, <https://www.kdnuggets.com/5-tips-structuring-data-science-projects>
3. How to organize your Python data science project - GitHub Gist, accessed on October 17, 2025, <https://gist.github.com/ericmjl/27e50331f24db3e8f957d1fe7bbbe510>
4. Python Virtual Environment - GeeksforGeeks, accessed on October 17, 2025, <https://www.geeksforgeeks.org/python/python-virtual-environment/>
5. 12. Virtual Environments and Packages — Python 3.14.0 ..., accessed on October 17, 2025, <https://docs.python.org/3/tutorial/venv.html>
6. Install packages in a virtual environment using pip and venv, accessed on October 17, 2025, <https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/>
7. A Docker Tutorial for Beginners, accessed on October 17, 2025, <https://docker-curriculum.com/>
8. Setting Up Docker for Python Projects: A Step-by-Step Guide - GeeksforGeeks, accessed on October 17, 2025, <https://www.geeksforgeeks.org/python/setting-up-docker-for-python-projects-a-step-by-step-guide/>
9. Use containers for Python development - Docker Docs, accessed on October 17, 2025, <https://docs.docker.com/guides/python/develop/>
10. Producing Data - General Transit Feed Specification, accessed on October 17, 2025, <https://gtfs.org/resources/producing-data/>
11. public-transport/gtfs-via-postgres: Process GTFS Static ... - GitHub, accessed on October 17, 2025, <https://github.com/public-transport/gtfs-via-postgres>
12. GeoPandas Tutorial: An Introduction to Geospatial Analysis - DataCamp, accessed on October 17, 2025, <https://www.datacamp.com/tutorial/geopandas-tutorial-geospatial-analysis>
13. geopandas.GeoDataFrame.to\_postgis, accessed on October 17, 2025, <https://geopandas.org/en/stable/docs/reference/api/geopandas.GeoDataFrame.to_postgis.html>
14. 7. Using GeoPandas — Spatial Data Management with PostgreSQL and PostGIS, accessed on October 17, 2025, <https://postgis.gishub.org/chapters/geopandas.html>
15. geopandas.read\_postgis — GeoPandas 0.8.2 documentation, accessed on October 17, 2025, <https://geopandas.org/en/v0.8.2/reference/geopandas.read_postgis.html>
16. Building Lightweight Geospatial Data Viewers with StreamLit and PyDeck | by Joseph George Lewis | Python in Plain English, accessed on October 17, 2025, <https://python.plainenglish.io/building-lightweight-geospatial-data-viewers-with-streamlit-and-pydeck-de1e0fbd7ba7>
17. Get started with Streamlit - Streamlit Docs, accessed on October 17, 2025, <https://docs.streamlit.io/get-started>
18. Building a dashboard in Python using Streamlit - Streamlit Blog, accessed on October 17, 2025, <https://blog.streamlit.io/crafting-a-dashboard-app-in-python-using-streamlit/>
19. Step-by-Step Guide to Deploying Machine Learning Models with ..., accessed on October 17, 2025, <https://machinelearningmastery.com/step-by-step-guide-to-deploying-machine-learning-models-with-fastapi-and-docker/>