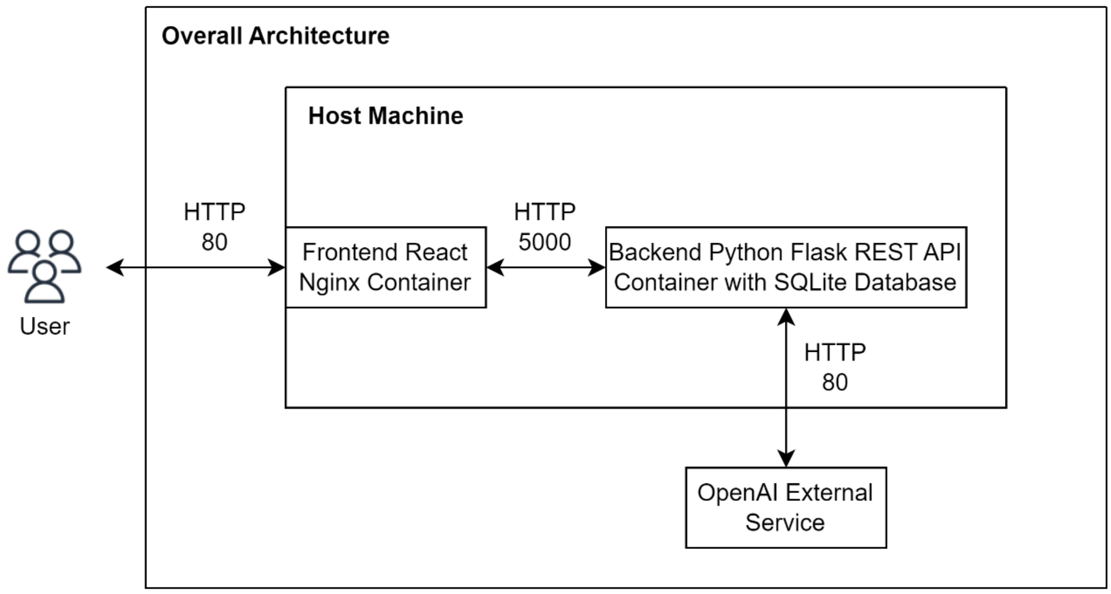
Table of Contents

1. Architecture Overview and Explanation


Overall Architecture
Host Machine
User
HTTP 80
Frontend React Nginx Container
HTTP 5000
Backend Python Flask REST API Container with SQLite Database
HTTP 80
OpenAI External Service

The architecture for this code repository is designed with the following considerations:

- **Simplicity**
  Given the requirements of SQLite as the primary database, it is assumed that the application should be lightweight. Therefore, the application is built with minimal number of containers: one for frontend and one for backend. By being in the same docker network created by docker compose, inter-container communication is enabled. Torch CPU packages instead of Torch GPU packages are also used to reduce the size of the resulting Backend Python Flask container.
- **Security**
  The containers are designed such that only one port of the host machine needs to be exposed. In secured environment, all other ports can possibly be blocked with a firewall and this application can still work. This is enabled via Nginx's reverse proxy capability. Appropriate configurations are set in the Frontend React Nginx container to forward relevant requests to the Backend Python Flask container. There is no need to map the exposed port of the Backend container to the host machine as long as the two containers are in the same docker network.

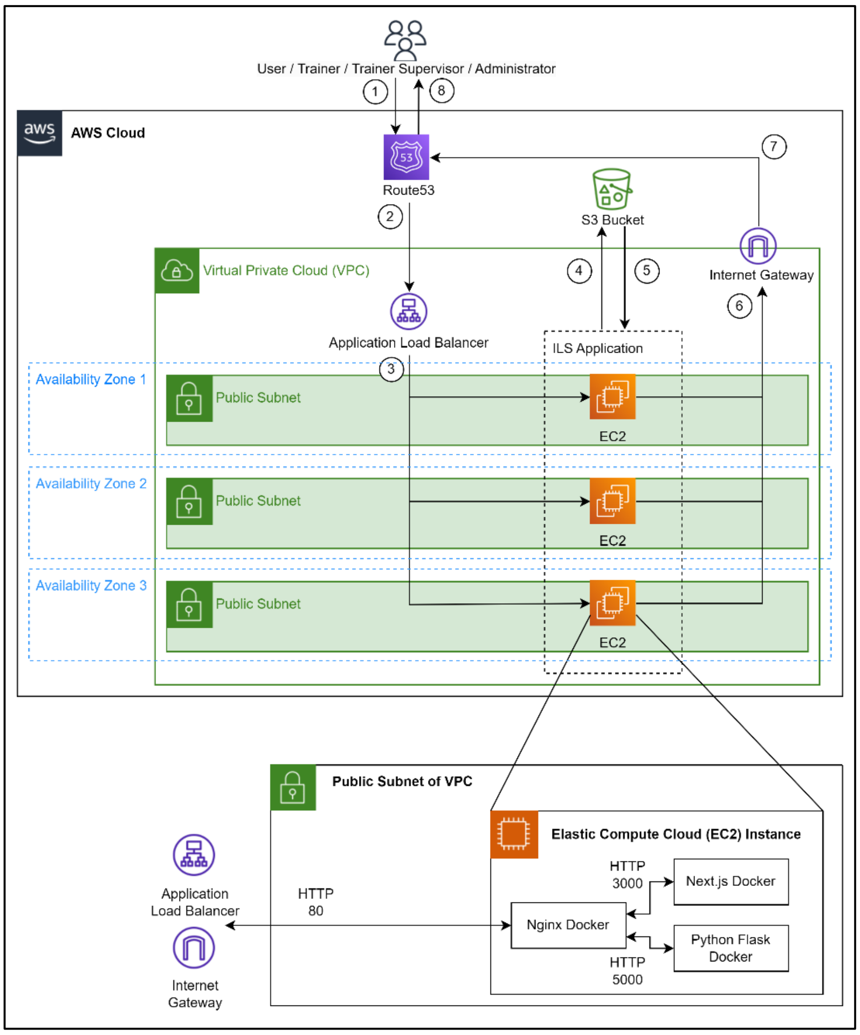
2. Further Details on Endpoints

Each of the below REST API endpoints are prefixed with a root */api/*. For example, the */health* REST API endpoint can be accessed from the local Flask application at *http://localhost:5000/api/health*.

- /health
    - This is a simple endpoint that checks if the Flask application is running.
- /transcribe
    - This endpoint transcribes .mp3 files or a .zip of .mp3 files.
    - Files that are not .mp3 of .zip will be rejected.
    - Threads are used to allow transcribing to be done in the background while returning a response to the user that the transcription task is in progress, making the response time faster as a result.
    - Each thread stores the final transcribed text in the SQLite database.
    - The statuses of the transcription tasks are also stored in the SQLite database. When a task is being handled in the background by a thread, the status stored in the database is “In Progress”.
- /transcriptions
    - Limits and offsets are set as parameters for this endpoint to enable the option of reducing the number of transcription records in a request.
    - This leads to improve performance as unnecessary transactions with large amounts of data can be avoided. This also leads to better user experience as users can view the data incrementally via pagination features in the React app.
    - The pagination can be seen with sample data by connecting to the *db_filled.db* by configuring the DB_NAME environment variable for the backend container.
- /search
    - Similar limit and offset parameters from the */transcriptions* endpoint are also set here. There is data binding in the React app search bar that allows the search results in the table to be populated in real-time, providing a seamless user experience as a result.
- /postprocess (Bonus Feature)
    - This is a bonus feature to enable postprocessing of transcriptions. Spelling errors for Singaporean food such as “young” in “young tofu” from the Sample 3 audio file can possibly be corrected with the help of ChatGPT.
    - A generator object is used to enable the streaming text effect which is commonly seen in https://chatgpt.com/. This helps to provide a better user experience by allowing users to see the progress of the transcription postprocessing in real time.

3. Possible Future Enhancements for Production

- In contrast with the current repository's 2-container setup, a multi-container setup with separate containers for Nginx reverse proxy and React frontend can be considered for better separation of concerns and independent scalability of each container based on each container's demand.
- Each endpoint can also be hosted in individual containers for even further separation of concerns in a microservices architecture.
- Either the 2-container setup or a multi-container setup can be deployed into the cloud as seen from the below AWS diagram I used in a paper published with my SMU professors at the Machine Learning with Applications (MLWA) journal.
- The SQLite database can be replaced with a more production-ready database such as MySQL, MSSQL, or PostgreSQL to allow for high availability. NoSQL databases such as MongoDB or S3 as seen in the AWS diagram is also possible.
- The application can also be secured with an SSL certificate to enable HTTPS traffic. With SSL termination capability via Nginx or the Application Load Balancer seen in the AWS diagram on the right, SSL processing is offloaded from the application servers, resulting in minimal code change required for this code repository.


User / Trainer / Trainer Supervisor / Administrator
1
8
AWS Cloud
Route53
7
2
S3 Bucket
Virtual Private Cloud (VPC)
4
5
Internet Gateway
6
Application Load Balancer
ILS Application
3
Availability Zone 1
Public Subnet
EC2
Availability Zone 2
Public Subnet
EC2
Availability Zone 3
Public Subnet
EC2
Public Subnet of VPC
Elastic Compute Cloud (EC2) Instance
HTTP 3000
Next.js Docker
Application Load Balancer
HTTP 80
Nginx Docker
Python Flask Docker
HTTP 5000
Internet Gateway