Machine Learning Engineering & AI

Bootcamp Capstone

Step 11: Deploy Your Application to Production

17 April 2025

Outline of Steps to Deploy Fine-Tuned Model as a Web Application

Below is a high-level outline of steps I executed to deploy as a web application the Depth Anything MDE model which I had fine-tuned. In summary, I created a backend Flask application, a frontend Javascript HTML user interface, used a Dockerfile to containerize the deployment package along with model checkpoints and supplementary scripts, and then tagged-pushed-run the container on Google Cloud Run. I created many of the scripts by asking ChatGPT to provide a baseline script which I then may or may not have to modify to suit my specific needs.

Below is a comprehensive outline of the entire process—from developing the Flask backend and frontend HTML GUI through containerization with Docker to deploying on Google Cloud Run. Diagrams and a project structure overview are provided for clarity.

---

1. Building the Flask Backend (api.py)

Steps:

- **Create the Flask Application:**
  An api.py file was created to define the REST API using Flask. Key endpoints include:
  - **"/"**: Serves the GUI page (using render_template from a templates folder).
  - **"/infer"**: Accepts a POST request (with an image and additional form fields such as options and representation), processes the image (using the Depth Anything MDE model), and returns the processed depth map image.
  - The Flask application adapts to the runtime port set by Cloud Run.

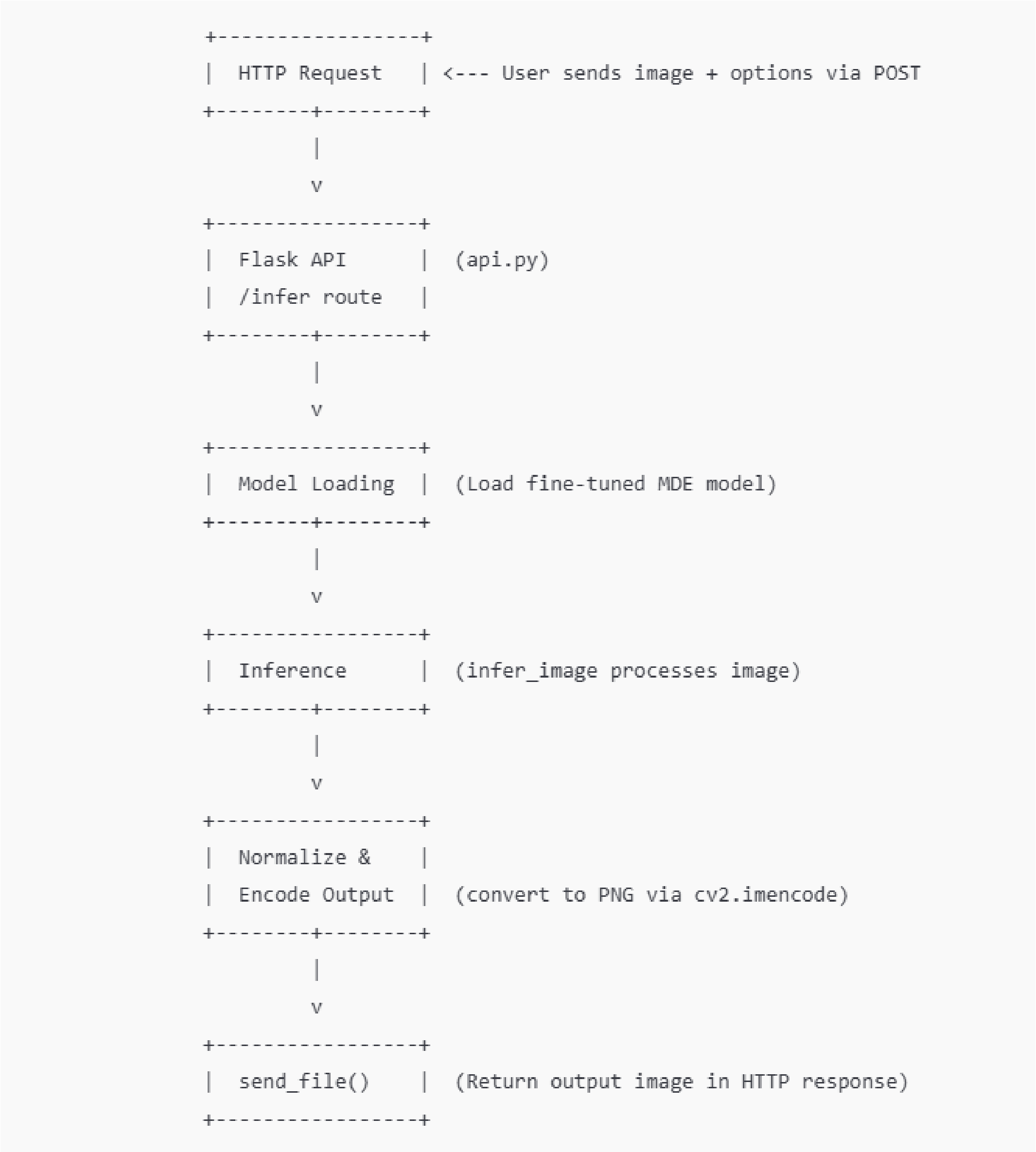
- **Model Loading and Inference:**
  The backend loads a fine-tuned Depth Anything MDE model (using PyTorch) from a checkpoint file. The inference method processes the input image, normalizes the depth map, and returns it as a PNG image via Flask's send_file function.

- **Error Handling and Logging:**
  The script logs errors and uses JSON responses for error messages, ensuring that clients receive useful feedback when something goes wrong.

- **Test Flask backend with curl command**
  Use curl command to test application locally to confirm that:
  - Image uploads work.

- Depth maps are produced as expected.

Figure 1: Flask Backend Flow Diagram



```
                +-----------------+
                |  HTTP Request   | <--- User sends image + options via POST
                +--------+--------+
                         |
                         v
                +-----------------+
                |  Flask API      |  (api.py)
                |  /infer route   |
                +--------+--------+
                         |
                         v
                +-----------------+
                |  Model Loading  |  (Load fine-tuned MDE model)
                +--------+--------+
                         |
                         v
                +-----------------+
                |  Inference      |  (infer_image processes image)
                +--------+--------+
                         |
                         v
                +-----------------+
                |  Normalize &    |
                |  Encode Output  |  (convert to PNG via cv2.imencode)
                +--------+--------+
                         |
                         v
                +-----------------+
                |  send_file()    |  (Return output image in HTTP response)
                +-----------------+
```

2. Creating the HTML Frontend GUI

Steps:

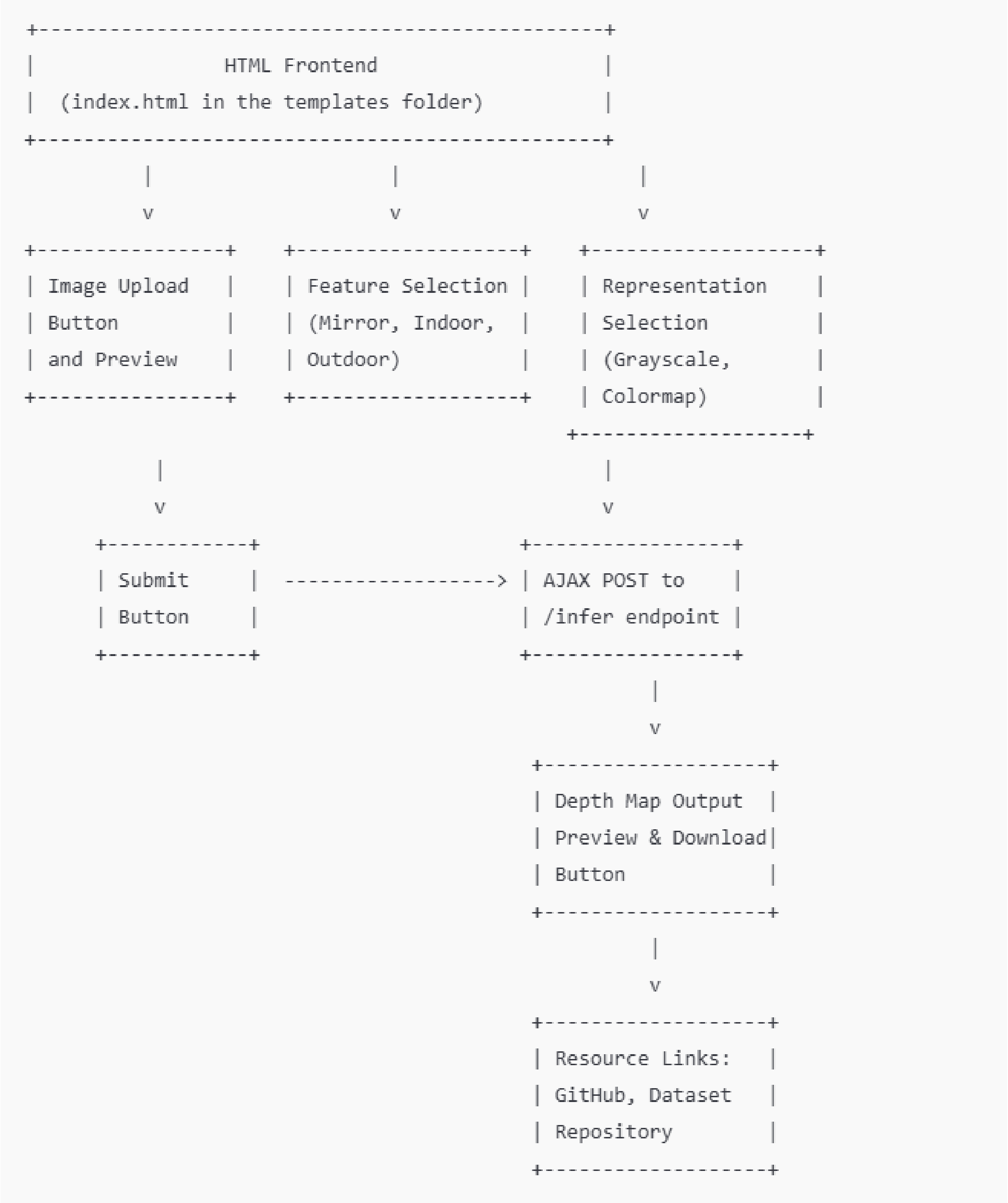
- **Design the Interface:**
  An HTML file (e.g., index.html) was created and placed in a templates folder that serves as the GUI. The GUI includes:

    1. **Image Upload Button:** An <input type="file"> element.

2. **Display of the Uploaded Image:** An <img> element that shows the user's selected image.
3. **Feature Selection:** Three buttons allow the user to choose among "mirror", "indoor art", and "outdoor art".
4. **Depth Map Representation Options:** Two buttons to select between "grayscale" and "colormap".
5. **Submit Button:** Initiates the API call via a JavaScript fetch() POST request.
6. **Display of the Output Depth Map:** An <img> element shows the returned depth map.
7. **Download Button:** Allows the user to download the depth map image.
8. **Resource Links:** Hyperlinks to Depth Anything v2, GitHub repository with deployment package, and the Google Cloud repository for the dataset.

- **Integrate JavaScript:**
  The embedded JavaScript handles:
  - File reading and preview.
  - Option selection for feature and representation.
  - Sending an AJAX POST request (using fetch()) with the file and form parameters.
  - Displaying the API's response and enabling the download of the processed image.
  - The JavaScript front-end uses a relative URL, ensuring it works correctly regardless of the domain or port assigned in production.

Figure 2: GUI Component Diagram



```
+------------------------------------------------+
|                HTML Frontend                   |
|  (index.html in the templates folder)          |
+------------------------------------------------+
          |                    |                    |
          v                    v                    v
+----------------+    +-------------------+    +-------------------+
| Image Upload   |    | Feature Selection |    | Representation    |
| Button         |    | (Mirror, Indoor,  |    | Selection         |
| and Preview    |    | Outdoor)          |    | (Grayscale,       |
+----------------+    +-------------------+    | Colormap)         |
                                              +-------------------+
           |                                     |
           v                                     v
      +------------+                      +-----------------+
      | Submit     |  ------------------> | AJAX POST to    |
      | Button     |                      | /infer endpoint |
      +------------+                      +-----------------+
                                                   |
                                                   v
                                           +-------------------+
                                           | Depth Map Output  |
                                           | Preview & Download|
                                           | Button            |
                                           +-------------------+
                                                   |
                                                   v
                                           +-------------------+
                                           | Resource Links:   |
                                           | GitHub, Dataset   |
                                           | Repository        |
                                           +-------------------+
```

---

3. Containerizing the Application with Docker

Steps:

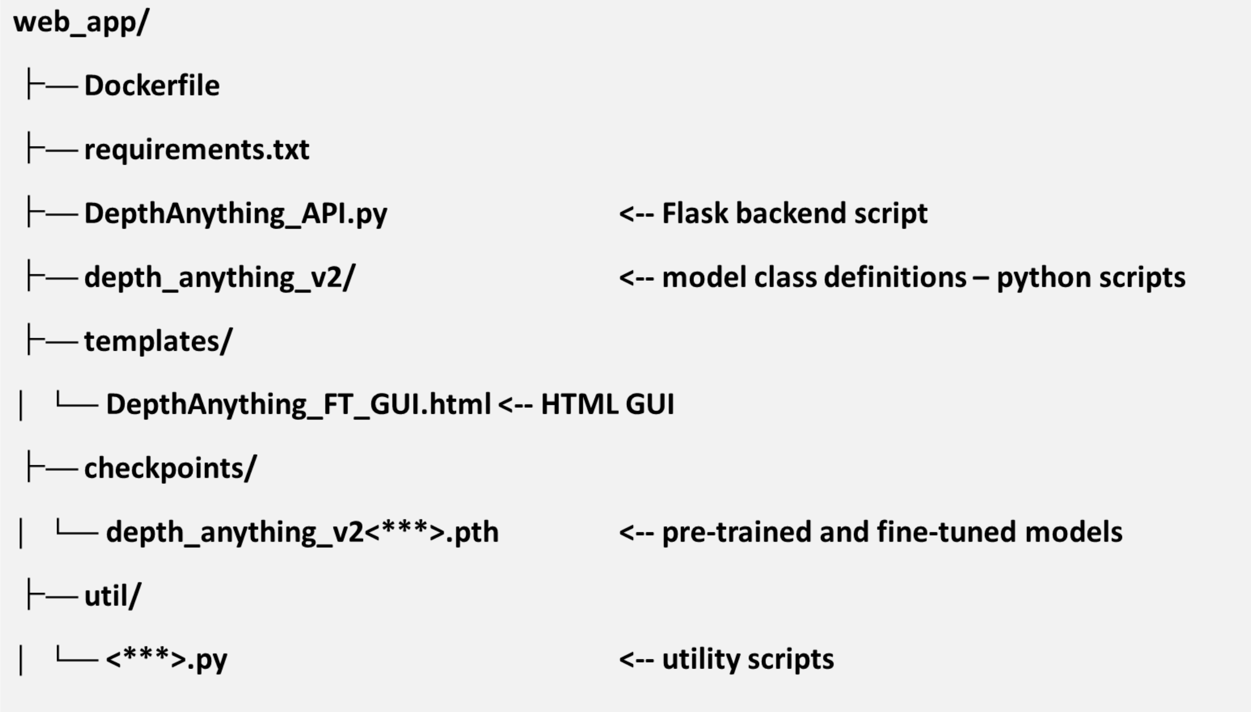
- **Organization of Project Directory (web_app):**

The project folder includes (a):

- Dockerfile – The file that describes how to build the project container.
- requirements.txt – All Python dependencies.
- Flask backend script (e.g., DepthAnything_API.py).
- templates folder – Contains the Javascript frontend HTML file (DepthAnything_FT_GUI.html)
- checkpoints folder – Pre-trained and fine-tuned models
- util folder – Contains utility and supplementary scripts

- **Create a Dockerfile:**
  A Dockerfile was created to:
  - Uses a lightweight Python base image (e.g., python:3.9-slim).
  - Sets environment variables to disable .pyc file creation and enable unbuffered output.
  - Sets the working directory to /app inside the container.
  - Copies the application files (including the Flask API script and the templates folder) into the container.
  - Installs required packages from requirements.txt.
  - Exposes the appropriate port (e.g., 5000) as a default.
  - Document and expose the expected default port without hard-coding production values.
  - Defines a CMD to run the Flask application (for example, CMD ["python", "DepthAnything_API.py"] since my Flask app is named as such).

Figure 3: Project Structure Diagram



```
web_app/
 ├── Dockerfile
 ├── requirements.txt
 ├── DepthAnything_API.py              <-- Flask backend script
 ├── depth_anything_v2/                <-- model class definitions – python scripts
 ├── templates/
 │   └── DepthAnything_FT_GUI.html <-- HTML GUI
 ├── checkpoints/
 │   └── depth_anything_v2<***>.pth    <-- pre-trained and fine-tuned models
 ├── util/
 │   └── <***>.py                      <-- utility scripts
```

Link 1: GitHub repository with deployment package

https://github.com/kentheman4AI/SB-Capstone-Project-Monocular-Depth-Estimation

- **Build and Run the Docker Container:**

    1. **Build the Docker Image:**

```
docker build -t myapp .
```

    2. **Run the Docker Container (to test locally):**

```
docker run -p 5000:5000 myapp
```

- This maps port 5000 inside the container (where the Flask app listens) to port 5000 on a host.
- A curl command was used to test application locally after running Dockerfile

---

4. Deploying on Google Cloud Run

Steps:

1. **Set Up Google Cloud:**
    - Create a Google Cloud project and enable billing.
    - Enable Cloud Run and Container Registry (or Artifact Registry) APIs.
    - Install and configure the Google Cloud SDK.

2. **Push the Docker Image:**
    - Tag the image with the Google Cloud Run project ID (prime-boulevard-455921-k7):

```
docker tag myapp gcr.io/PRIME-BOULEVARD-455921-K7/myapp:latest
```

    - Push it to the Container Registry:

```
docker push gcr.io/PRIME-BOULEVARD-455921-K7/myapp:latest
```

3. **Deploy to Cloud Run:**
    - Deploy the container using the following command (chosen region is us-central1):

```
gcloud run deploy myapp \
  --image gcr.io/PRIME-BOULEVARD-455921-K7/myapp:latest \
  --platform managed \
  --region us-central1 \
  --allow-unauthenticated
```

    - Cloud Run will provide a public URL for the running service.

4. **Test the Deployed Application**

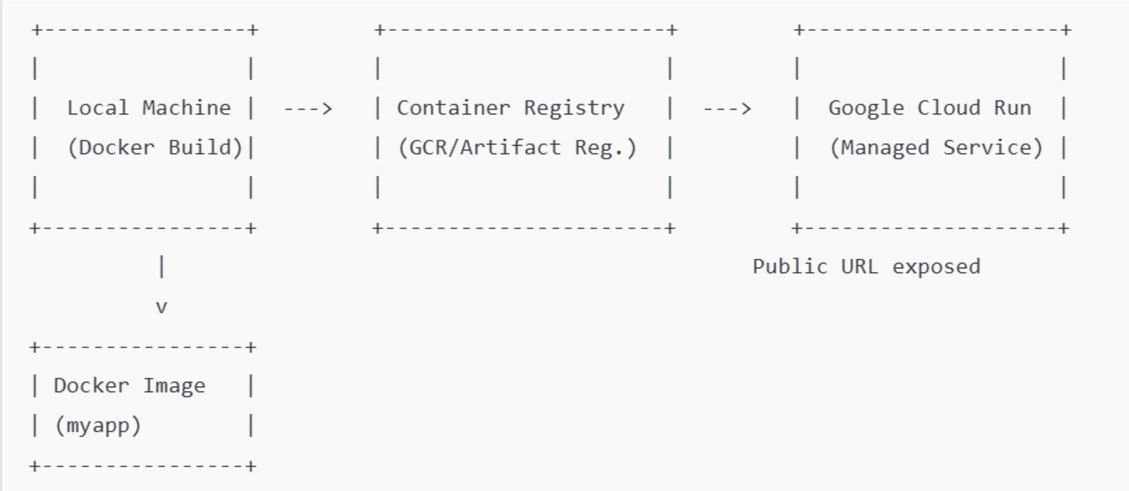
a) **Access the Application:**

Open the provided URL in a browser. A GUI (the HTML interface) served by the Flask application is seen.

b) **Test Functionality:**

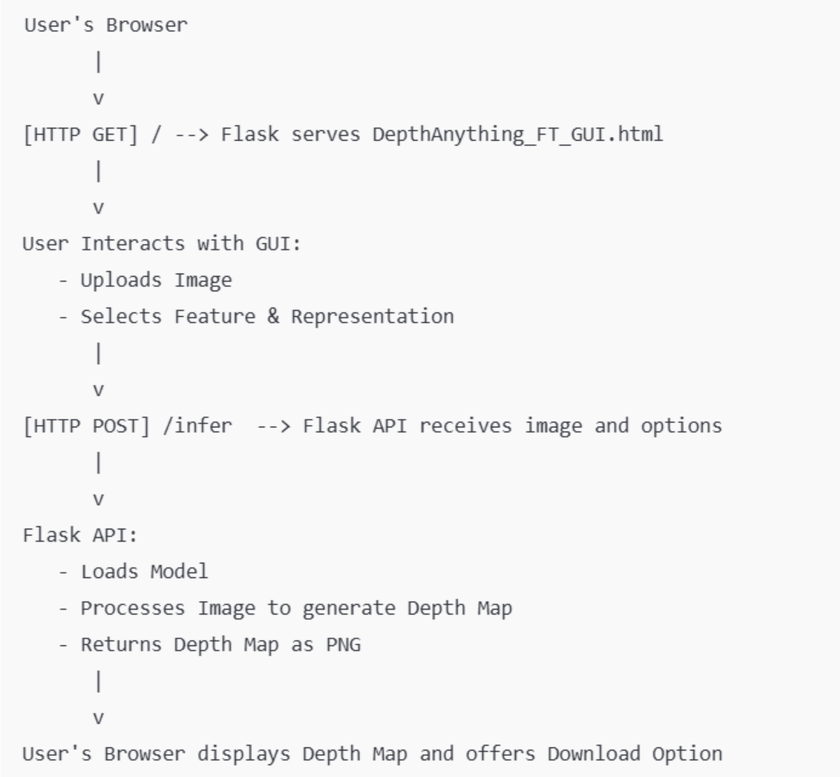
- Upload an image, select the feature and depth representation options.
- Submit the image.
- Verify that the API returns the depth map and that a user can view and download it.
- Check the resource links (GitHub and dataset repository) to ensure they point to the correct locations.

Figure 4: Cloud Run Deployment Flow



```
+----------------+        +----------------------+        +--------------------+
|                |        |                      |        |                    |
|  Local Machine |  --->  | Container Registry   |  --->  |  Google Cloud Run  |
|  (Docker Build)|        | (GCR/Artifact Reg.)  |        |  (Managed Service) |
|                |        |                      |        |                    |
+----------------+        +----------------------+        +--------------------+
         |                                             Public URL exposed
         v
+----------------+
| Docker Image   |
| (myapp)        |
+----------------+
```

Figure 5: End-to-End Application Flow



```
User's Browser
      |
      v
[HTTP GET] / --> Flask serves DepthAnything_FT_GUI.html
      |
      v
User Interacts with GUI:
   - Uploads Image
   - Selects Feature & Representation
      |
      v
[HTTP POST] /infer  --> Flask API receives image and options
      |
      v
Flask API:
   - Loads Model
   - Processes Image to generate Depth Map
   - Returns Depth Map as PNG
      |
      v
User's Browser displays Depth Map and offers Download Option
```

Summary of the End-to-End Process

1. **Flask Backend (api.py / DepthAnything_API.py):**
   - Develop a REST API using Flask.
   - Integrate model inference logic using PyTorch.
   - Return results (e.g., depth map image) using send_file.

2. **HTML Frontend GUI:**
   - Build an interactive HTML page (placed in the templates folder) that allows image upload, option selection (features and representation), and displays results.
   - Include JavaScript to send POST requests to the Flask API and handle the response (display and download).

3. **Containerization with Docker:**
   - Write a Dockerfile that sets up a Python environment, installs dependencies, and copies the application code.
   - Build the Docker image and run it locally to ensure it works.

4. **Google Cloud Run Deployment:**
   - Push the Docker image to Google Container Registry.
   - Deploy the container image to Google Cloud Run.

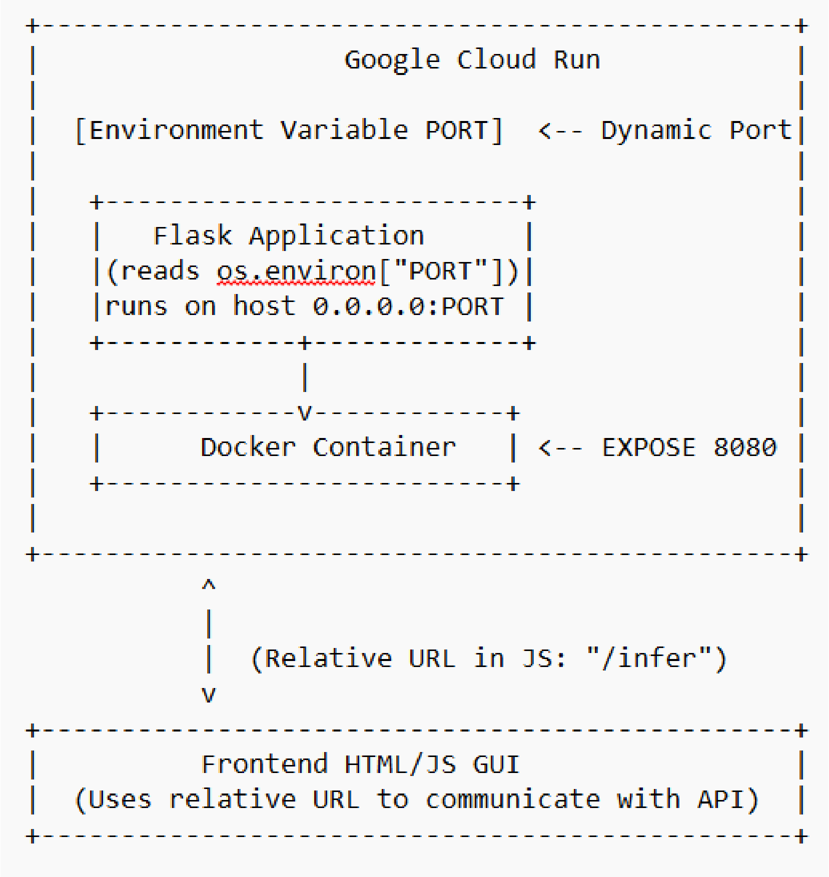
- Obtain a public URL to access the web application.
- Test the deployment using a browser or curl.

Notes:

- The Flask application adapts to the runtime port set by Cloud Run.
    - The Dockerfile documents and exposes the expected default port without hard-coding production values.
    - The JavaScript front-end uses a relative URL, ensuring it works correctly regardless of the domain or port assigned in production.

These steps form a complete pipeline—from local development to production deployment—ensuring the web application is portable, scalable, and accessible on the cloud.

Summary Diagram



```
+-----------------------------------------------+
|                   Google Cloud Run            |
|                                               |
|  [Environment Variable PORT]  <-- Dynamic Port|
|                                               |
|   +--------------------------+                |
|   |   Flask Application      |                |
|   |(reads os.environ["PORT"])|                |
|   |runs on host 0.0.0.0:PORT |                |
|   +------------+-------------+                |
|                |                              |
|   +------------v------------+                 |
|   |      Docker Container   | <-- EXPOSE 8080 |
|   +-------------------------+                 |
|                                               |
+-----------------------------------------------+
           ^
           |
           |  (Relative URL in JS: "/infer")
           v
+-----------------------------------------------+
|          Frontend HTML/JS GUI                 |
|  (Uses relative URL to communicate with API)  |
+-----------------------------------------------+
```