DevOps Implementation of a Containerized Web Scraping Application

Overview:

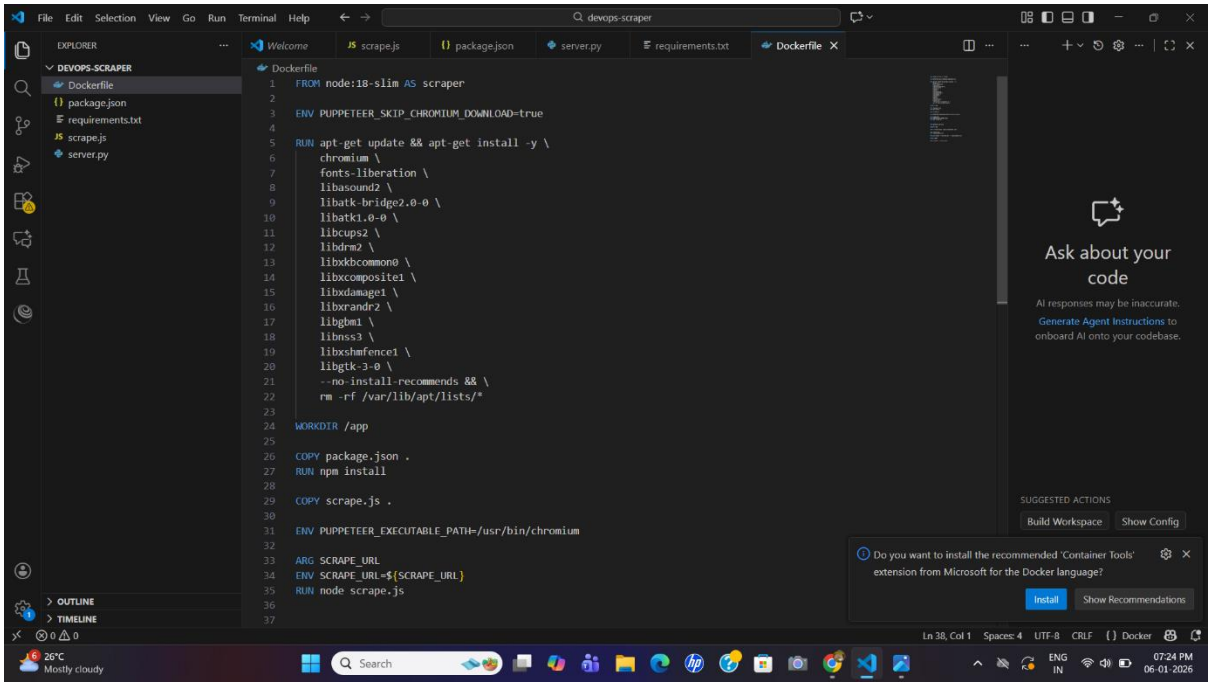
The project creates and executes a containerized web scrapping application using Docker. Basic website data is scraped using a Node.js service with Puppeteer, and the extracted output is served through a Python flask. Docker multi-stage builds are used to separate the scraping and server layers, resulting in a lightweight and efficient final image. The application demonstrates the practical use of Docker for application packaging and deployment, as it runs successfully inside a container and is accessible using a local web interface.

Tools Used:

- **Docker & Docker Desktop –** For containerization and multi-stage builds
- **Node.js –** To run the web scraping service
- **Puppeteer –** For automated web scraping using a headless browser
- **Python –** To develop the backend service
- **Flask –** To serve scraped data through a REST API
- **Chromium –** Headless browser used by Puppeteer
- **Visual Studio Code –** Development environment
- **Web Browser (Chrome) –** To access the API output

Assignment Explanation:

Dockerfile – Stage 1: Web Scraping (Node.js & Puppeteer)



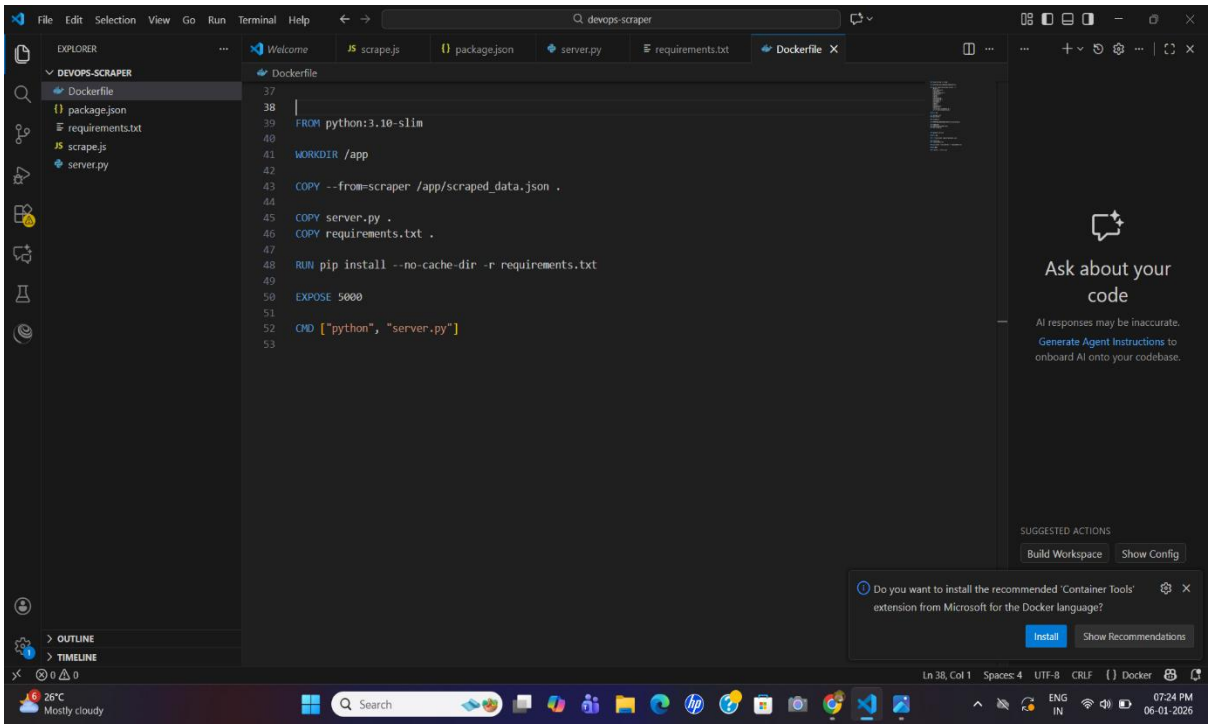
This stage of the Dockerfile is responsible for performing the web scraping operation using Node.js and Puppeteer. The base image node:18-slim is used to keep the image lightweight while supporting Node.js execution.

Puppeteer requires a browser environment, so Chromium and its required system libraries are installed using apt-get. The environment variable is set to skip the default Chromium download and instead use the system-installed browser.

The working directory is set to /app, and Node.js dependencies are installed using npm install. The scraping script (scrape.js) is then copied into the container. An environment variable is used to pass the target website URL dynamically.

Finally, the Node.js script is executed, which launches a headless browser, scrapes the webpage title and first heading, and saves the output as a JSON file. This output is later used by the second stage of the Docker build.

Dockerfile – Stage 2: Flask Server (Python)



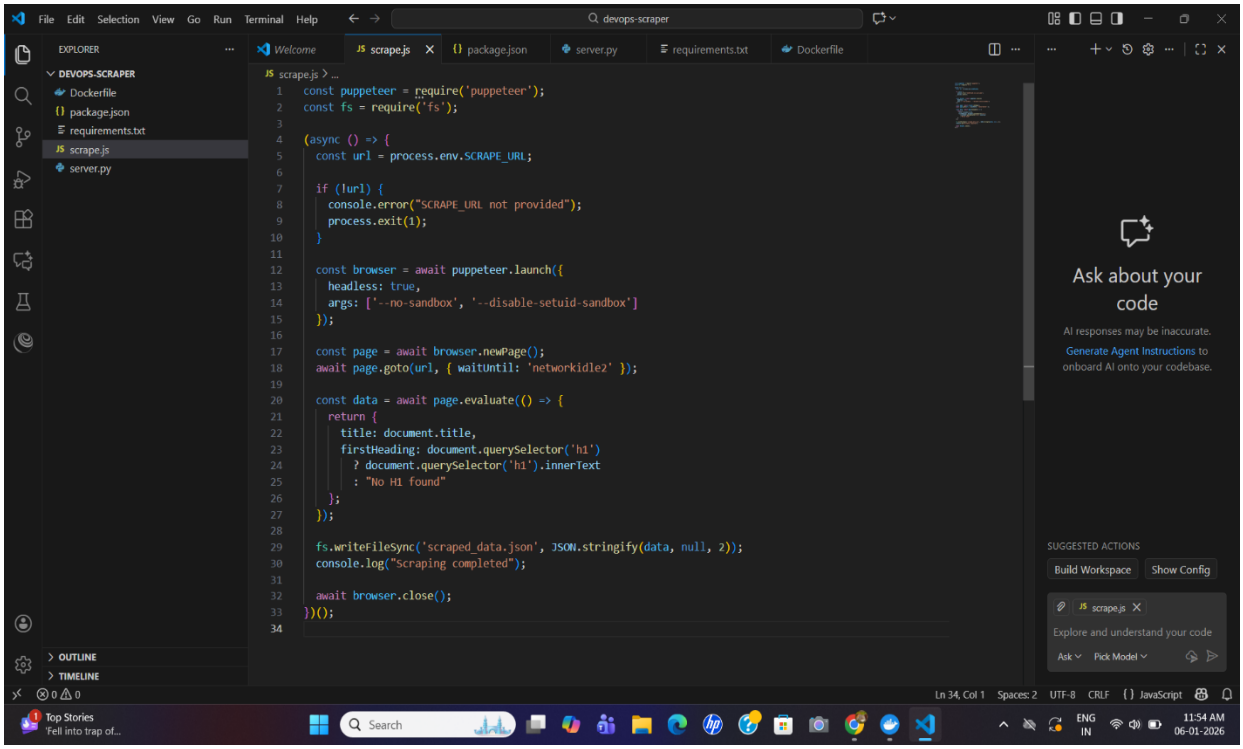
This stage of the Dockerfile is responsible for serving the scraped data using a Python Flask application. The base image python:3.10-slim is used to keep the final container lightweight while supporting Python execution.

The working directory is set to /app. The scraped JSON file generated in the first (Node.js) stage is copied into this stage using a multi-stage build approach. This avoids carrying unnecessary Node.js and browser dependencies into the final image.

The Flask server file (server.py) and the dependency file (requirements.txt) are then copied into the container. Required Python libraries are installed using pip, ensuring only essential packages are included.

Port 5000 is exposed to allow external access to the Flask application. Finally, the container starts the Flask server using the Python command, making the scraped website data accessible through a local web interface.

Web Scraping Script (scrape.js)



This JavaScript file is responsible for scraping data from a given website using the Puppeteer library. Puppeteer is a Node.js tool that controls a headless Chromium browser to interact with web pages programmatically.

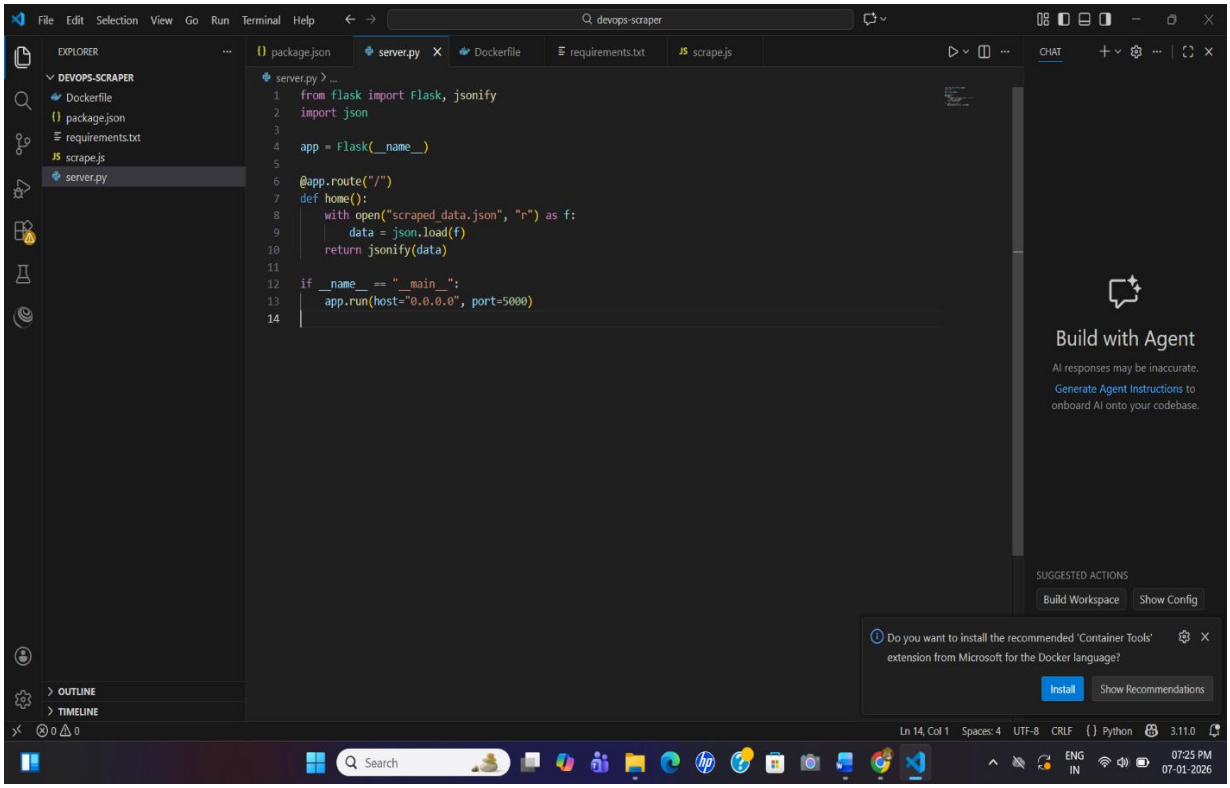
The script first reads the target website URL from an environment variable called SCRAPE_URL. This makes the script flexible, as different websites can be scraped without changing the code. If the URL is not provided, the program stops execution and displays an error message.

A headless browser is then launched with sandbox restrictions disabled to ensure smooth execution inside a Docker container. A new page is opened and the browser navigates to the given URL, waiting until all network requests are completed.

Once the page is fully loaded, the script extracts two pieces of information: the page title and the first heading (<h1>). If no heading is found, a default message is returned. The scraped data is saved into a JSON file named scraped_data.json, which is later used by the Python Flask server.

Finally, the browser is closed after successful scraping to free system resources.

Flask Server Script (server.py)



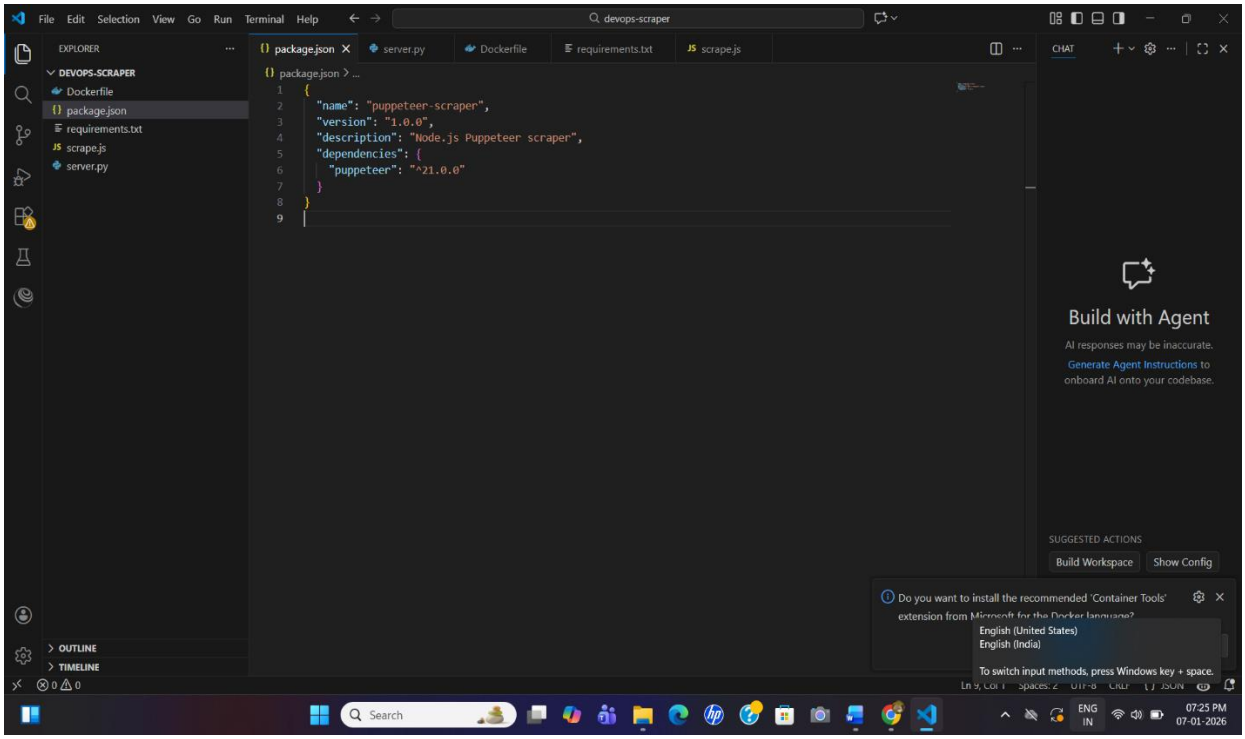
This Python file implements a simple web server using the Flask framework. Its main purpose is to serve the data that was scraped by the Node.js Puppeteer script and stored in a JSON file.

The Flask application starts by importing the required modules, including Flask for creating the web server and json for reading the scraped data file. When the server starts, it looks for the scraped_data.json file in the application directory.

A single route (/) is defined in the application. When a user accesses this route through a web browser, the server reads the contents of the JSON file and returns the data as a JSON response. This allows users to easily view the scraped website title and heading through a web interface.

The server runs on port 5000 and is configured to listen on all network interfaces, making it accessible from outside the Docker container. This setup demonstrates how scraped data can be exposed using a lightweight Python API.

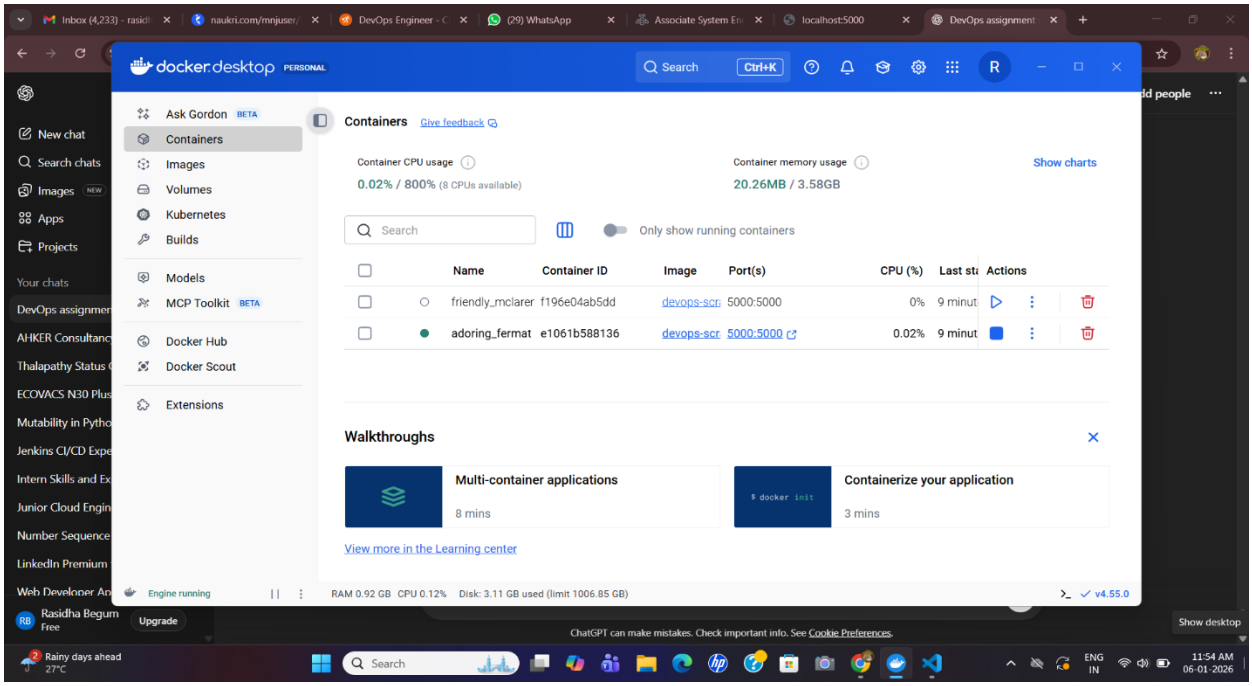
Node.js Configuration File (package.json)



This package.json file defines the configuration and dependencies required for the Node.js scraping component of the project. It specifies basic project details such as the project name, version, and a short description indicating that the application is a Puppeteer-based web scraper.

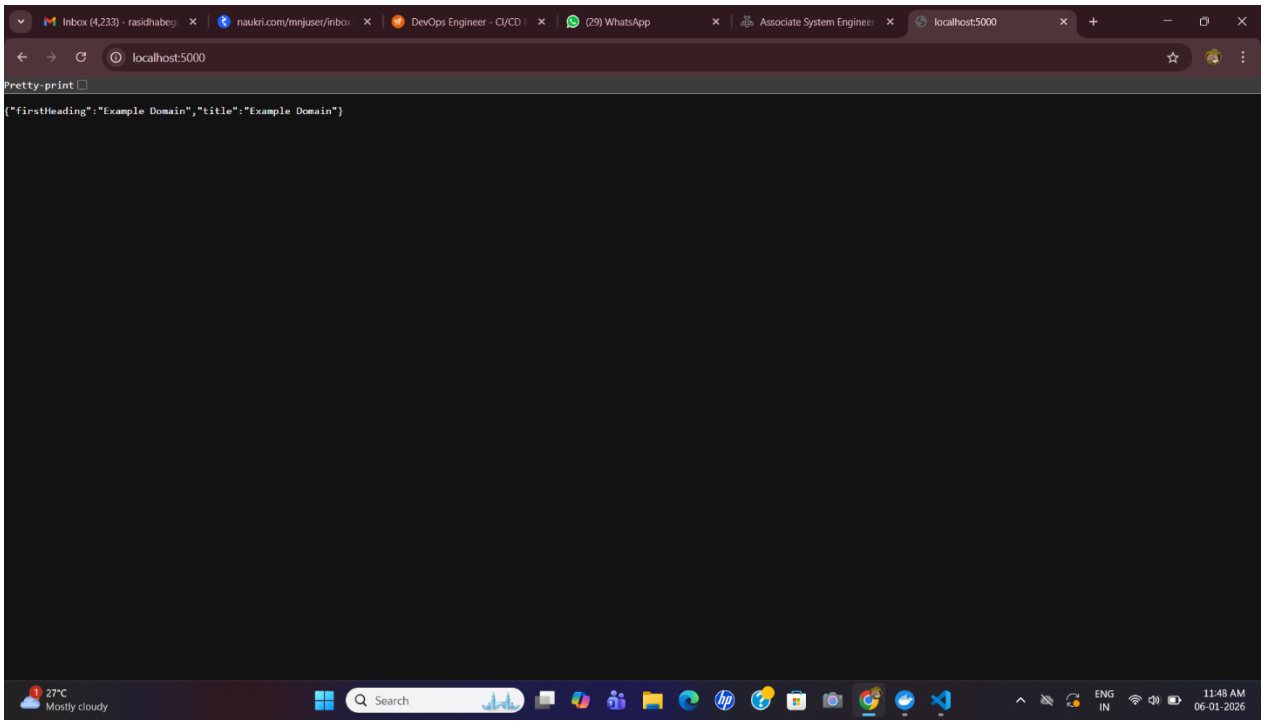
The most important part of this file is the dependencies section, where Puppeteer is listed. Puppeteer is a Node.js library used to control a headless Chrome browser, which enables automated web page loading and data extraction. During the Docker build process, this file is used to install all required Node.js dependencies inside the container.

Step 2: Docker Container Execution



This screen shows the running Docker containers for the project in Docker Desktop. It confirms that the application image is successfully deployed as a container and is actively running, with port **5000** mapped to the host system for accessing the Flask web service through a browser.

Step 3: Final Output Verification



This screenshot shows the final output of the application accessed through **http://localhost:5000**. The Flask server successfully displays the scraped website data in JSON format, including the page title and first heading, confirming that the containerized scraping and API services are working correctly.

Challenges Faced:

- Docker image build failed due to network DNS issues.
- Puppeteer failed initially because Chromium was not installed.
- Port 5000 was not accessible at first.

These issues were resolved by reinstalling Docker, configuring DNS, and rebuilding the image.