

PDF Layout – Week 5: Cloud and API Deployment

Title: Week 5: Cloud and API Deployment

Name: Radhika Ranchhodhbhai Diyora

Batch Code: LISUM51

Submission Date: 4 December 2025

Submitted To: Data Glacier

1. Project Overview

This project demonstrates deployment of a trained Iris classification model as a **web application** and **API** using Flask and Render (cloud).

Features:

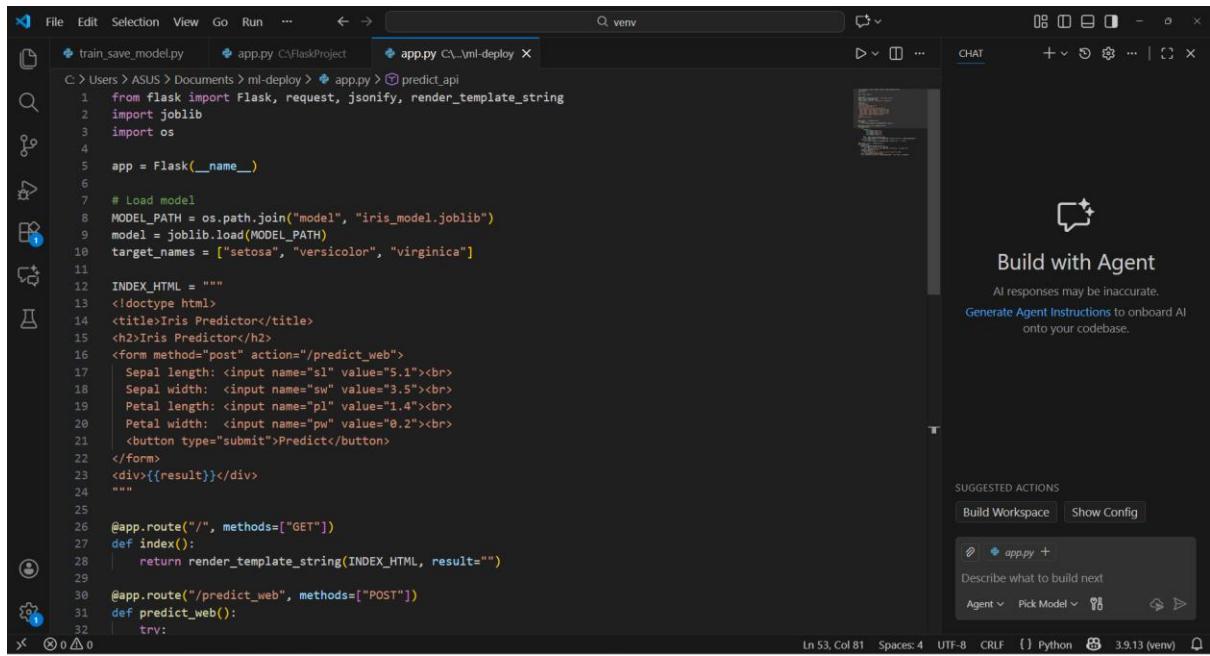
- Users can predict Iris species using a web form.
 - API endpoint allows JSON input for programmatic predictions.
 - Deployment is done using **Docker** on **Render**.
-

2. Steps of Deployment

Step 1 – Prepare Model & Flask App

- Trained Iris model saved as model/iris_model.joblib.
- Flask app (app.py) created with:
 - Web form endpoint / and /predict_web
 - API endpoint /predict

Screenshot 1: (screenshot of app.py showing Flask routes here)

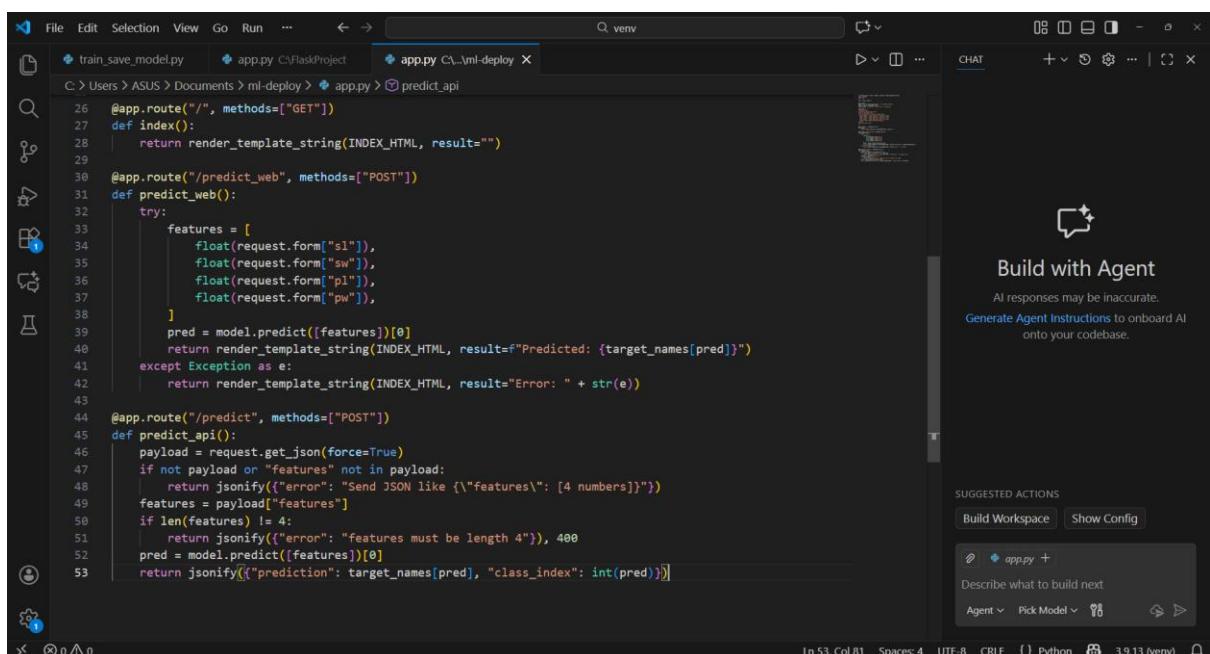


The screenshot shows a code editor interface with three tabs: train_save_model.py, app.py, and app.py. The app.py tab is active, displaying Python code for a Flask application. The code includes routes for a web interface and a REST API, handling POST requests to predict Iris flower species based on input features. The interface includes a sidebar for 'Build with Agent' and 'SUGGESTED ACTIONS'.

```

C:\> Users > ASUS > Documents > ml-deploy > app.py > predict_api
1  from flask import Flask, request, jsonify, render_template_string
2  import joblib
3  import os
4
5  app = Flask(__name__)
6
7  # Load model
8  MODEL_PATH = os.path.join("model", "iris_model.joblib")
9  model = joblib.load(MODEL_PATH)
10 target_names = ["setosa", "versicolor", "virginica"]
11
12 INDEX_HTML = """
13 <!doctype html>
14 <title>Iris Predictor</title>
15 <h2>Iris Predictor</h2>
16 <form method="post" action="/predict_web">
17   Sepal length: <input name="sl" value="5.1"><br>
18   Sepal width: <input name="sw" value="3.5"><br>
19   Petal length: <input name="pl" value="1.4"><br>
20   Petal width: <input name="pw" value="0.2"><br>
21   <button type="submit">Predict</button>
22 </form>
23 <div>{{result}}</div>
24 """
25
26 @app.route("/", methods=["GET"])
27 def index():
28     return render_template_string(INDEX_HTML, result="")
29
30 @app.route("/predict_web", methods=["POST"])
31 def predict_web():
32     try:
33         features = [
34             float(request.form["sl"]),
35             float(request.form["sw"]),
36             float(request.form["pl"]),
37             float(request.form["pw"]),
38         ]
39         pred = model.predict([features])[0]
40         return render_template_string(INDEX_HTML, result=f"Predicted: {target_names[pred]}")
41     except Exception as e:
42         return render_template_string(INDEX_HTML, result="Error: " + str(e))
43
44 @app.route("/predict", methods=["POST"])
45 def predict_api():
46     payload = request.get_json(force=True)
47     if not payload or "features" not in payload:
48         return jsonify({"error": "Send JSON like {\"features\": [4 numbers]}"})
49     features = payload["features"]
50     if len(features) != 4:
51         return jsonify({"error": "features must be length 4"}), 400
52     pred = model.predict([features])[0]
53     return jsonify({"prediction": target_names[pred], "class_index": int(pred)})

```



The second screenshot shows the same code editor interface with the code for app.py. The code has been modified to handle JSON input for the prediction API. It checks if the payload contains 'features' and if its length is 4. If not, it returns an error message. Otherwise, it performs the prediction and returns the result as JSON.

Step 2 – Dockerize the App

- Created a Dockerfile with required packages and **start command** using gunicorn.

Screenshot 2: (screenshot of Dockerfile content showing CMD with \$PORT here)

```

# Use official Python slim image
FROM python:3.11-slim

# Set working directory
WORKDIR /app

# Copy requirements and install
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy all files
COPY . .

# Expose port (optional, Render uses $PORT)
EXPOSE 8080

# Start the Flask app using Render's assigned $PORT
CMD ["sh", "-c", "gunicorn -w 4 -b 0.0.0.0:$PORT app:app"]

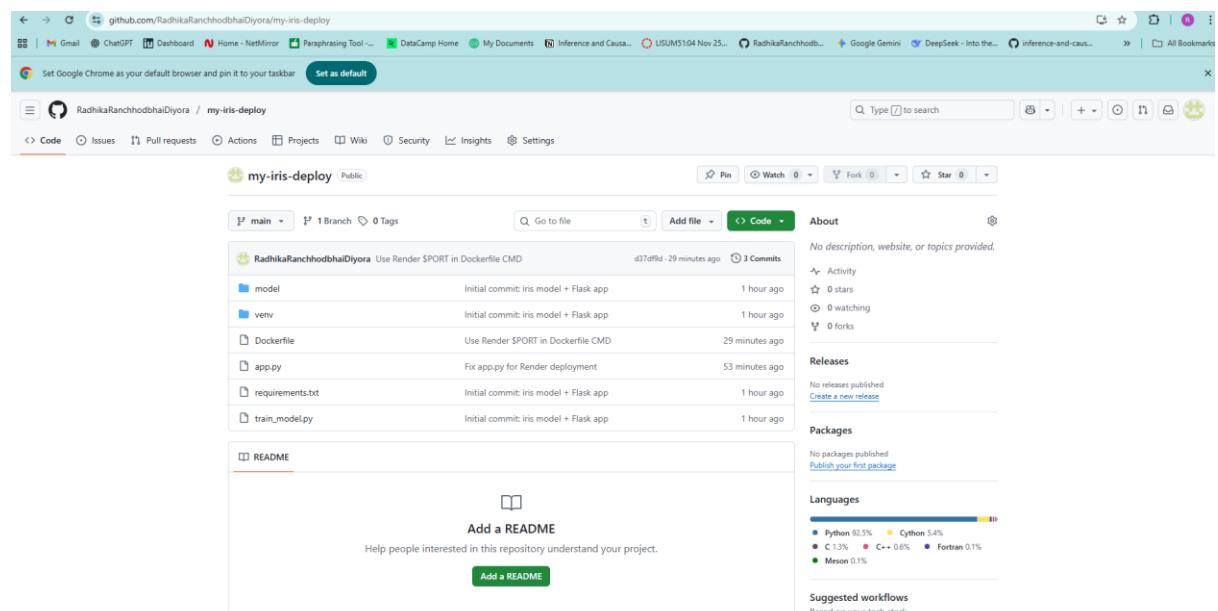
```

Step 3 – Push Code to GitHub

- All code pushed to:

<https://github.com/RadhikaRanchhodhaiDiyora/my-iris-deploy>

Screenshot 3: (Screenshot of GitHub repo showing files and latest commit here)



Step 4 – Deploy on Render

- Connected GitHub repo to Render.
- Render built Docker image and started web service.
- URL of deployed app:

<https://my-iris-deploy.onrender.com>

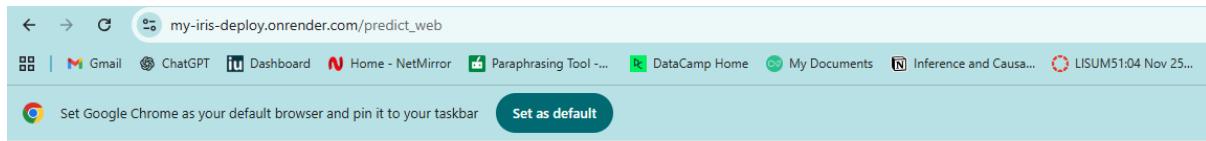
Screenshot 4: (screenshot of Render dashboard showing deployed service and logs here)

The screenshot shows the Render dashboard interface. On the left, there's a sidebar with links like 'Dashboard', 'My project', 'Overview', 'Settings', 'Changelog', 'Invite a friend', 'Contact support', and 'Render Status'. The main area is titled 'My project' and has a sub-section 'Production'. It displays a table with one row for 'my-iris-deploy'. The columns are 'SERVICE NAME', 'STATUS', 'RUNTIME', 'REGION', and 'UPDATED'. The service name is 'my-iris-deploy', status is 'Deployed' (with a green checkmark icon), runtime is 'Docker', region is 'Oregon', and it was updated 2min ago. There are also buttons for '+ Add environment' and '+ New service'.

Step 5 – Test Web App

- Open URL in browser.
- Enter sample features:
 - Sepal length: 5.1
 - Sepal width: 3.5
 - Petal length: 1.4
 - Petal width: 0.2
- Click **Predict** → result displayed.

Screenshot 5: (screenshot of browser showing HTML form and prediction result here)



Iris Predictor

Sepal length:
Sepal width:
Petal length:
Petal width:

Set as default

Step 6 – Test API

- Command used in terminal (CMD/PowerShell):

```
curl -X POST -H "Content-Type: application/json" -d "{\"features\": [5.1,3.5,1.4,0.2]}" https://my-iris-deploy.onrender.com/predict
```

- Response:

```
{"class_index":0,"prediction":"setosa"}
```

Screenshot 6: (screenshot of CMD showing API JSON response here)

```
C:\Users\ASUS\Documents\ml-deploy>git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 578 bytes | 578.00 KiB/s, done.
Total 3 (delta 1), reused 2 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/RadhikaRanchohodhaiDiyora/my-iris-deploy.git
 d37df9d..6c491bd main -> main

C:\Users\ASUS\Documents\ml-deploy>curl -X POST -H "Content-Type: application/json" -d "{\"features\": [5.1,3.5,1.4,0.2]}" https://my-iris-deploy.onrender.com/predict
{"class_index":0,"prediction":"setosa"}
```

3. Conclusion

- Successfully deployed a machine learning model as a **web app and API**.
- Both **web interface** and **API endpoint** are working as expected.
- All files and deployment can be verified via **GitHub repo** and **Render URL**.

