Software Development Engineer Assessment

Note: Please attach your solutions in one ".zip" folder. Solutions will be built and run against multiple test cases. You are expected to complete this assessment with Python 3.x. Should we detect any sign of plagiarism, candidates will be rejected immediately.

Scenario

You are contributing to the development of **ZypherAI**, an innovative platform for Machine Learning model deployment and inference. Your current task is to create a simple web application that allows users to run predictions with machine learning models.

This application, instead of running predictions using actual ML models, simulates the prediction process. It calls a mock function that runs a prediction and returns a randomized result:

```
import time
import random
from typing import Dict

def mock_model_predict(input: str) -> Dict[str, str]:
        time.sleep(random.randint(10, 17)) # Simulate processing delay
        result = str(random.randint(1000, 20000))
        output = {"input": input, "result": result}
        return output
```

Objective

Your goal is to create a simple web application server using a Python framework of your choice, such as FastAPI or Flask. The expected request and response body content type of this web application server is **application/json**

Part A – Synchronous Model Prediction

- Develop a web application server that operates on **localhost** port **8080**.
- Implement a synchronous **/predict** POST endpoint.
- This endpoint should take user input, invoke the mock_model_predict function, and return its output to the user with a 200 status code.

A sample POST request to the **/predict** endpoint might look like:

```
POST /predict HTTP/1.1
Host: localhost:8080
Content-Type: application/json
{
        "input": "Sample input data for the model"
}
```

The response from the server upon receiving the above request could be:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
        "input": "Sample input data for the model",
        "result": "1234"
}
```

Part B – Asynchronous Model Prediction

- Enhance the **/predict** POST endpoint from Part A to support asynchronous request processing.
- The endpoint should recognize the **Async-Mode** request header to activate asynchronous processing.
- In asynchronous mode, immediately respond to the user with a 202 status code and a unique **prediction_id**.
- After responding to the user, call the **mock_model_predict** function using the provided input and save results that can be retrieved later.

A sample POST request to the **/predict** endpoint for asynchronous processing might look like this:

```
POST /predict HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Async-Mode: true
{
        "input": "Sample input data for the model"
}
```

The response from the server upon receiving the above asynchronous request would be:

```
HTTP/1.1 202 Accepted
Content-Type: application/json
{
        "message": "Request received. Processing asynchronously.",
        "prediction_id": "abc123" // A unique ID generated for this prediction request
}
```

Additional Endpoint for Asynchronous Results

- Implement a **/predict/{prediction_id}** GET endpoint.
- This endpoint allows users to retrieve prediction results for a specific **prediction_id**.
- If the **prediction_id** is not valid, the endpoint should return a 404 status code response.
- If the **prediction_id** is being processed, the endpoint should return a 400 status code response.

To retrieve the result for a specific prediction, a user would send a GET request like this:

```
GET /predict/abc123 HTTP/1.1
Host: localhost:8080
```

If the prediction is still being processed, response from the server would look like:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
{
        "error": "Prediction is still being processed."
}
```

If the prediction id is not valid, response from the server would look like:

```
HTTP/1.1 404 Not Found
Content-Type: application/json
{
        "error": "Prediction ID not found."
}
```

If the prediction results are available, response from the server would look like:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
        "prediction_id": "abc123",
        "output": {"input": "Sample input data for the model", "result": "5678"}
}
```

Pro-Tip

- Consider using an in-built Python queue or an external queue implementation (e.g., Redis Streams or RabbitMQ) for managing asynchronous predictions.
- Explore background task processing in Python.
- It's okay to temporarily store prediction results in memory. This means they don't need to be persisted, especially if the server restarts or shuts down.

Part C - Dockerfile

This role requires you to be proficient with containerization technologies such as Docker. Therefore,

- You need to provide a Dockerfile for building and running your web application server.
- Should you decide to use an external queue implementation in Part B, you can include a docker compose file that runs your server's image and the external queue system's image together.

Extra Credit

- Utilize Python type hints for clearer code.
- Document your web application endpoints and any complex logic.
- Add inline comments to explain your thought process and decisions.
- Use an **external queue system** (e.g., Redis Streams or RabbitMQ) to handle background task processing and asynchronous predictions at scale.
- Demonstrate Docker proficiency by optimizing the Docker image size and configuration.
- Include a **README.md** file documenting your approach, instructions to run, assumptions you have made, and any alternative approach you decided not to pursue.
- Keep in mind the space and time complexity of your solutions. This role requires you to design and develop highly efficient and optimized systems that can scale with minimal resource consumption.