Exercise Instructions for the Interviewee

1. Overview

You are tasked with implementing the system architecture that supports a detailed API for executing Quantum Circuits. The API endpoints—with their inputs and outputs—are defined below. Your primary goal is to design and implement the underlying architecture that manages asynchronous processing, task integrity (i.e., ensuring that no tasks are lost), and containerized deployment.

Please read the entire document carefully before starting the exercise to ensure you have a clear understanding of the requirements and expectations.

---

2. API Requirements

POST /tasks

Description:

This endpoint receives a quantum circuit payload and initiates its asynchronous processing. It then returns a task ID that can be used to track the processing status.

Input: A JSON payload with the key "qc" and a string value containing a serialized quantum circuit in QASM3 format.

Example Input (QC example in the bottom of this exercise):

```
None
    {

      "qc": "<serialized_quantum_circuit_in_qasm3>"

    }
```

Output: A JSON response containing a unique task identifier and a confirmation message.

Example Output:

```
None
{
  "task_id": "12345",
  "message": "Task submitted successfully."
}
```

GET /tasks/<id>

Description:

This endpoint retrieves the results of a previously submitted quantum circuit using its unique task identifier.

Input: A URL parameter `<id>` representing a unique task identifier.

Example Request:

- GET /tasks/12345

Output: A JSON response based on the task's state:

- If the task is completed:

```
None
{
  "status": "completed",
  "result": {"0": 512, "1": 512}
}
```

- If the task is still processing:

```
None
{
```

```
  "status": "pending",
  "message": "Task is still in progress."
}
```

- If the task is not found:

```
None
{
  "status": "error",
  "message": "Task not found."
}
```

---

3. System Architecture Focus

Your challenge is to design and implement the underlying system architecture to support the API described above. Your solution should address the following:

- **Asynchronous Processing & Task Integrity:**
  Ensure that tasks are processed asynchronously and that no submitted tasks are lost during the processing lifecycle.

- **Containerization & Orchestration:**
  Your implementation must use Docker Compose to containerize and orchestrate all components of the system.

- **Robustness:**
  Implement appropriate error handling and logging mechanisms as needed to support a production-like environment.

The specifics of how you design and implement these components are left to your discretion.

---

4. Additional Requirements

- **Technology Stack:**
  Use Python 3.9+ and a lightweight web framework (such as Flask or FastAPI).

- **Code Quality:**
  Follow best practices for readability, modularity, and maintainability.

---

5. Deliverable Checklist

Ensure your submission includes:

- **Source Code:**
  - API server implementation.
  - Components for asynchronous task processing.
- **Containerization:**
  - Dockerfiles for all components.
  - A `docker-compose.yml` file that defines and orchestrates the API server, background processing components, and any necessary supporting services.
- **Documentation:**
  - A README file with setup instructions, architectural design decisions, and usage guidelines.
- **Tests:**
  - Integration tests covering task submission, processing, and result retrieval.

---

6. Timeframe & Submission

- **Overall Time:** You have 48 hours to complete the exercise.
- **Submission Instructions:**
  - **Format:** Provide a link to a public or private Git repository (e.g., GitHub, GitLab) or submit a ZIP archive.
  - **Content Requirements:** Include all source code, container configurations, documentation, and tests as described above.
  - **Final Check:** Verify that your solution can be built and run using a straightforward command (e.g., `docker-compose up`).

---

7. Example: Creating and Executing a Quantum Circuit

Below is an example code snippet for creating and executing a basic Quantum Circuit using Qiskit.

Python

```python
from qiskit import QuantumCircuit

from qiskit.providers.aer import AerSimulator


NUM_SHOTS = 1024


def create_basic_quantum_circuit() -> QuantumCircuit:

    qc = QuantumCircuit(2, 2)

    qc.h(0)                     # Hadamard on qubit 0

    qc.cx(0, 1)                 # CNOT from qubit 0 to qubit 1

    qc.measure([0, 1], [0, 1])  # Measure both qubits into the
two classical bits

    return qc


def execute_circuit(qc: QuantumCircuit) -> dict:

    simulator = AerSimulator()

    job = simulator.run(qc, shots=NUM_SHOTS)

    result = job.result()

    return result.get_counts()
```

Hint: Use `qiskit.qasm3` to convert your Quantum Circuit to a string for serialization and back to a QuantumCircuit object for deserialization. Refer to Qiskit's documentation for detailed usage.