GeoData Pipeline and POI Delivery System

1. Data Ingestion & Processing Pipeline

Step 1: Choosing the Data Sources

Objective: Pull geospatial data from various sources, such as OpenStreetMap (OSM), government datasets, or private POI databases.

Data Formats: Common formats include GeoJSON, Shapefiles, and CSV for geospatial data. We use WKT/WKB for handling geometries in the pipeline, as supported by Snowflake.

Why Snowflake?

- Snowflake supports geospatial data types and functions.
- Scalability: Snowflake scales automatically to handle huge datasets efficiently.

Step 2: Setting Up Snowflake for Geospatial Data

Create a Snowflake Database and Schema to organize the data:

```
CREATE DATABASE GeoData;
CREATE SCHEMA GeoData.POI;
```

Upload the Data: Use Snowflake's PUT command or integrate with cloud storage (S3, GCS) for ingestion. Example using OSM data:

```
CREATE TABLE POI (
    id INT,
    name STRING,
    location GEOGRAPHY,
    metadata STRING
);
```

Insert the data into Snowflake using SQL to convert data into Snowflake's GEOGRAPHY type:

```
INSERT INTO POI SELECT id, name, ST_GeographyFromText('POINT(' || lon || ' ' || lat || ')') AS location, metadata FROM @data/poi.csv;
```

Step 3: Organizing Data with Quad Trees

Why Quad Trees? They optimize spatial queries by dividing the map into quadrants recursively, ensuring efficient search and filtering.

Custom Function for Quad Tree Encoding in Snowflake using geohashing:

```
ALTER TABLE POI ADD COLUMN geohash STRING;
UPDATE POI SET geohash = ST_Geohash(location, 12);
```

2. API Services for Map Data & POIs

Step 1: Designing the API

Objective: Expose map and POI data via APIs to allow various clients to query the data.

Example Python/Flask API Setup:

```
from flask import Flask, request, jsonify
import snowflake.connector

app = Flask(__name__)
```

```
def get_snowflake_connection():
    conn = snowflake.connector.connect(
        user='your_user',
        password='your_password',
        account='your_account',
        warehouse='COMPUTE_WH',
        database='GeoData',
        schema='POI'
    )
    return conn


@app.route('/api/poi', methods=['GET'])
def get_poi():
    conn = get_snowflake_connection()
    lat = request.args.get('lat')
    lon = request.args.get('lon')
    radius = request.args.get('radius', default=1000)
    query = f'SELECT name, metadata FROM POI WHERE ST_DWithin(location, ST_GeographyFromText("POINT({lon} {lat})"), {radius});'
    cursor = conn.cursor()
    cursor.execute(query)
    results = cursor.fetchall()
    return jsonify(results)


if __name__ == '__main__':
    app.run(debug=True)
```

Step 2: Containerizing the API with Docker

Why containerize? To ensure the API runs consistently across different environments.

Dockerfile:

```
FROM python:3.9-slim


WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt


COPY . .

CMD ["python", "app.py"]
```

Build and run the container:

```
docker build -t geodata-api .

docker run -d -p 5000:5000 geodata-api
```

3. 3D Map Rendering & Visualization

Step 1: Setting up a WebGL 3D Map

Objective: Visualize map data using WebGL with Three.js.

Sample 3D map rendering code using Three.js:

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>3D Map Visualization</title>
    <style>body { margin: 0; } canvas { display: block; }</style>
</head>
<body>
    <script src="https://threejs.org/build/three.js"></script>
    <script>
        const scene = new THREE.Scene();
        const camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight,
0.1, 1000);
        const renderer = new THREE.WebGLRenderer();
        renderer.setSize(window.innerWidth, window.innerHeight);
        document.body.appendChild(renderer.domElement);

        const geometry = new THREE.PlaneGeometry(100, 100);
        const material = new THREE.MeshBasicMaterial({color: 0x00ff00});
        const plane = new THREE.Mesh(geometry, material);
        scene.add(plane);

        const poiMarker = new THREE.BoxGeometry(1, 1, 1);
        const markerMaterial = new THREE.MeshBasicMaterial({ color: 0xff0000 });
        const marker = new THREE.Mesh(poiMarker, markerMaterial);
        marker.position.set(10, 10, 5);
        scene.add(marker);
        camera.position.z = 50;

        function animate() {

            requestAnimationFrame(animate);

            renderer.render(scene, camera);

        }

        animate();

    </script>

</body>

</html>
```

4. Performance Monitoring & Optimization

Step 1: Setting Up Metrics with Prometheus

Monitor API performance (latency, request rates).

Install Prometheus:

```
docker run -d --name prometheus -p 9090:9090 prom/prometheus
```

Add metrics in your Flask app using prometheus-flask-exporter:

```
pip install prometheus-flask-exporter
```

Update Flask code:

```
from prometheus_flask_exporter import PrometheusMetrics

metrics = PrometheusMetrics(app)
```

5. Code Reviews & Continuous Integration

Step 1: CI/CD Pipeline

Set up a CI/CD pipeline using GitHub Actions or Jenkins.

GitHub Actions YAML:

```
name: CI
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Set up Python
      uses: actions/setup-python@v2
    - name: Install dependencies
      run: pip install -r requirements.txt
    - name: Lint
      run: flake8 .
```

6. Product Release Cycle & QA

Ensure QA testing with end-to-end tests using Selenium or Postman. Conduct load testing using Locust to stress-test APIs.