Case Study – Fullstack Developer

Context

InRisk Labs builds climate-risk platforms that ingest, store, and visualize weather/climate data. This case simulates a minimal version of that workflow using a public weather API and cloud object storage.

Problem Statement

Build and deploy a small full-stack weather explorer that:

1. Fetches **historical daily weather** for a user-chosen location and date range (Open-Meteo API).
2. **Stores** the raw JSON **only in a cloud bucket** (**GCS or AWS S3**).
3. Exposes a web dashboard to trigger fetch/store, list stored files, view a file, and visualize temps.

Part A – Backend (API + Cloud)

Tech Stack

- Language/Framework: **Python** (Flask or FastAPI)
- Cloud Storage: **Google Cloud Storage (GCS) or AWS S3**
- Deployment: **GCP Cloud Run** (or AWS Lambda/API Gateway/App Runner)

Required Endpoints

1. **POST** */store-weather-data*

```
{
 "latitude": <float>,
 "longitude": <float>,
 "start_date": "YYYY-MM-DD",
 "end_date": "YYYY-MM-DD"
}
```

Behavior:

- Validate inputs:
  - latitude ∈ [-90, 90], longitude ∈ [-180, 180]
  - dates valid; start_date ≤ end_date; **range ≤ 31 days**
- Call Open-Meteo daily-history with at least:
  - temperature_2m_max, temperature_2m_min,
  - apparent_temperature_max, apparent_temperature_min
- Store full API JSON to chosen bucket with name:

```
weather_<lat>_<lon>_<start>_<end>_<timestamp>.json
```

- Return *{"status": "ok", "file": "<stored_file_name>"}*

2. **GET** */list-weather-files*

- Behavior: List objects in the bucket
- Response:

```
{
 "files": [
  {"name": "<file>", "size": <bytes>, "created_at": "ISO8601"}
 ]
}
```

3. **GET** */weather-file-content/{file}*

- Behavior***:*** Fetch and return the JSON of {file} from the bucket.
- If missing/invalid → **404** with: *{"status": "error", "message": "not found"}*

Backend Quality Expectations

- Proper validation, clear error messages, correct HTTP status codes (400/404/5xx).
- Simple, modular structure (routes, storage client, validation).
- Efficient bucket listing (SDK methods; avoid brute scans).
- CORS enabled for your frontend.

Part B – Frontend (Dashboard)

Tech Stack

- **React or Next.js**
- **Tailwind CSS**

Features

1. **Input Panel**
    - Fields: Latitude, Longitude, Start Date, End Date
    - Actions:
        - Fetch & Store Data → POST /store-weather-data
        - Show loading/error states and returned file name
2. **Stored Files**
    - Browse Stored Files → GET /list-weather-files
    - Click a file to load content → GET /weather-file-content/{file}
    - Display basic metadata if available
3. **Data Visualization**
    - Line chart of daily max/min temperature
    - Table of the same daily variables
    - Pagination: 10/20/50 rows

UX Requirements

- Responsive layout (desktop/tablet/mobile)
- Clear loading/error handling
- Avoid excessive external API calls (work off stored files)

Submission Guidelines

1. Code Repository:

- Host your code in a public GitHub repository.
- Include a clear README.md with setup instructions, libraries used, and your design approach.

2. Live Demo:

- Deploy your app using Vercel, Netlify, or any free hosting service.
- Submit both the GitHub repo link and live demo URL. Submissions without live, accessible URLs will not be evaluated.

Evaluation Criteria

1. **Full-Stack Integration**
    - Frontend correctly calls backend
    - End-to-end flow: input → fetch/store → list → visualize
2. **Backend Engineering**
    - Correct Open-Meteo usage and variable selection
    - Proper cloud bucket storage, retrieval, naming
    - Clear validation and error handling
3. **Frontend Engineering & UI**
    - Clean, responsive Tailwind UI
    - Useful chart + paginated table

   - Good loading/error states

4. **Code Quality**
   - Structure, readability, modularity
   - Sensible abstractions and comments
   - Testability (even if minimal tests)

5. **Practicality & Performance**
   - Efficient API usage and bucket operations
   - Reasonable limits (≤31-day fetch)
   - Handles empty/edge cases gracefully