Deploy Automático

Etapa 0 - Planejamento
Etapa 1 - Preparar a EC2
ETAPA 2 — Criar chave SSH para o GitHub conectar-se na EC2
ETAPA 3 — Configurar os **GitHub Secrets**
ETAPA 4 — Criar o Workflow `.github/workflows/deploy.yml`
ETAPA 5 — Repetir para todos os servidores
Garantia da eficácia da arquitetura
    Testes de carga para as requisições na API
    Quantas conexões ao banco cada requisição gera
    Teste de carga do servidor do broker
    Teste de carga de inserções no MongoDB
    Teste de carga de inserções no PostgreSQL
    Caso haja maior expansão, o que terá que ser ajustado?
Conclusão

Etapa 0 - Planejamento

1. Ambiente:
    - Sistema Operacional: Linux (Ubuntu)
    - Cloud: AWS Academy Learner Lab (restrições: até 9 instâncias EC2 e 32 vCPUs)
    - Região: us-east-1
2. Demanda esperada:
    - Usuários simultâneos:
        - Atualmente: 100 usuários totais (~20 simultâneos)
        - Em 1 ano: estimativa de 1000 usuários (~200 simultâneos)
        - Justificativa: aplicação voltada ao público geral com possível uso em tempo real.
    - Estações meteorológicas integradas:
        - Inicialmente: 500
        - Em 1 ano: 1.000
        - Em 2 anos: 10.000
        - Justificativa: crescimento planejado da rede de sensores, exigindo maior capacidade de ingestão e processamento de dados ao longo do tempo.
3. Arquitetura de servidores:
    - Decidimos por 6 servidores distintos, e a decisão foi baseada em princípios de separação de responsabilidades, escalabilidade e facilidade de manutenção.

Servidor	Função	Justificativa Técnica
Servidor 1	Banco de dados relacional (PostgreSQL)	**- Isolamento de dados críticos**: manter o banco separado reduz os riscos de travamentos ou lentidão causados por outros serviços (como API ou processamento de dados). **- Performance dedicada**: com mais de 10.000 estações no futuro, o volume de inserções será alto. Ter o banco isolado permite alocar mais recursos (RAM, IOPS) se necessário. **- Facilidade de backup**: um banco separado pode ser copiado ou restaurado com menos impacto nos demais serviços.

Servidor	Função	Justificativa Técnica
		- Conexões simultaneas que ele permite: 100 por padrão mas pode ser alterado
Servidor 2	Banco de dados não relacional (MongoDB)	Usaremos o Atlas pelos seguintes motivos: **-** Escalabilidade e alta disponibilidade integradas sem precisar configurar réplica ou sharding manualmente. **-** Backup automático e ferramentas de monitoramento em tempo real. **-** Reduz a responsabilidade de gerenciamento e aumenta a confiabilidade em produção. **-** Facilita o acesso externo seguro e permite maior foco na lógica de ingestão e análise. **-** Está dando conta da nossa demanda.
Servidor 3	API (FastAPI)	**- Escalabilidade horizontal**: se a quantidade de usuários simultâneos crescer (de 20 para 200, por exemplo), podemos duplicar ou distribuir mais instâncias só desse servidor sem afetar o banco ou os processamentos.
Servidor 4	Processamento de dados	**- Carga computacional diferente**: processamento de dados mais pesado (limpeza, agregação, análise), consumindo CPU e memória de forma intermitente ou contínua. **- Evita gargalo na API**: se esses processamentos rodarem na mesma máquina que a API, podem travar ou atrasar respostas ao usuário final. **- Escalável de forma independente**: se o número de estações crescer de 1.000 para 10.000, o volume de dados a processar cresce também. Ter esse serviço isolado permite escalar só essa parte, sem precisar refazer toda a infraestrutura.
Servidor 5	Broker MQTT	**- Conexões persistentes:** com até 2.000 estações simultâneas (20% das 10.000 previstas), o broker precisa lidar com milhares de conexões TCP ativas. Isolar esse serviço evita sobrecarga em servidores de API ou banco. **- Baixa latência:** o MQTT é um protocolo leve e exige resposta rápida; manter o broker dedicado garante performance e previsibilidade. **- Segurança e controle:** permite configurar autenticação, ACLs e criptografia (TLS) sem interferir nos outros serviços. **- Escalabilidade:** caso a carga aumente, o broker pode ser substituído por uma solução em cluster (como EMQX ou HiveMQ) sem afetar os demais componentes do sistema.
Servidor 6	Frontend (React)	**- Limitação de espaço das instâncias da AWS:** cada instância não aguenta mais de um serviço então por isso cada um foi separado em uma instância. **- Instância com 4GB de RAM:** o Frontend usa muitos recursos então essa instância teve que ter mais RAM.

Etapa 1 - Preparar a EC2

1. Conectar-se à instância:

```
ssh -i weatherdataservicekey.pem ubuntu@<IP-EC2>
```

2. Instalar docker e docker compose na instância:

```
sudo apt update -y
 sudo apt install -y docker.io
```

```
sudo systemctl start docker
sudo systemctl enable docker
sudo usermod -aG docker ubuntu

sudo curl -L "https://github.com/docker/compose/releases/download/v2.24.2/docker-compose-$(uname -s)-$(unar
sudo chmod +x /usr/local/bin/docker-compose
```

3. Clonar o repositório correspondente ao serviço que será executado na instância:

```
git clone https://github.com/Code-Nine-FTC/WeatherDataService.git
```

ETAPA 2 — Criar chave SSH para o GitHub conectar-se na EC2

1. Na máquina local, rodar o comando:

```
ssh-keygen -t rsa -b 4096 -C "github-deploy"
```

   - Isso cria:
     - `~/.ssh/id_rsa` (privada)
     - `~/.ssh/id_rsa.pub` (pública)

2. Copiar a chave pública para a EC2:

```
ssh -i weatherdataservicekey.pem ubuntu@<IP-EC2>
 mkdir -p ~/.ssh
 echo "<conteúdo do id_rsa.pub>" >> ~/.ssh/authorized_keys
 chmod 700 ~/.ssh
 chmod 600 ~/.ssh/authorized_keys
```

ETAPA 3 — Configurar os GitHub Secrets

1. No repositório → **Settings > Secrets and variables > Actions** → "New repository secret":

   - Criar:

EC2_HOST	IP público da EC2
EC2_SSH_KEY	Conteúdo do id_rsa (chave privada)

ETAPA 4 — Criar o Workflow `.github/workflows/deploy.yml`

1. Criar o seguinte arquivo no repositório:

```
name: Deploy to EC2

 on:
   push:
     branches:
       - main

 jobs:
   deploy:
     runs-on: ubuntu-latest

     steps:
```

```
    - name: Deploy via SSH
      uses: appleboy/ssh-action@v0.1.6
      with:
        host: ${{ secrets.EC2_HOST }}
        username: ubuntu
        key: ${{ secrets.EC2_SSH_KEY }}
        script: |
          cd /home/ubuntu/WeatherDataService
          git pull origin main
          docker compose up -d --build
```

ETAPA 5 — Repetir para todos os servidores

1. Realizar os mesmos passos para o banco de dados, frontend e serviço de processamento.

Garantia da eficácia da arquitetura

Testes de carga para as requisições na API

1. Instalar k6 (ferramenta para teste de carga)

```
sudo apt install k6
```

2. Criar loadtest.js

```
import http from 'k6/http';
 import { sleep } from 'k6';

 export let options = {
    vus: 200, // 200 usuários simultâneos
  duration: '1m',
 };

 export default function () {
  http.get('http://<IP-EC2>:8000/');
  sleep(1);
 }
```

3. Rodar

```
k6 run loadtest.js
```

4. Resultado do teste de carga:

```
 TOTAL RESULTS

   checks_total.....: 8218    133.307997/s
   checks_succeeded.....: 100.00% 8218 out of 8218
   checks_failed.....: 0.00%   0 out of 8218

   ✓ login status is 200
   ✓ status is 200

   HTTP
   http_req_duration.....: avg=,468.01ms min=120.68ms med=243.96ms max=5.22s p(9
```

```
  { expected_response:true }.....: avg=468.01ms min=120.68ms med=243.96ms max=5.22s
http_req_failed.....: 0.00%  0 out of 8218
http_reqs.....: 8218   133.307997/s

EXECUTION
iteration_duration.....: avg=1.47s    min=1.12s    med=1.24s    max=6.37s p(90)=1.56s
iterations.....: 8217   133.291776/s
vus.....: 120    min=120      max=200
vus_max.....: 200    min=200      max=200

NETWORK
data_received.....: 2.5 MB 41 kB/s
data_sent.....: 1.9 MB 31 kB/s
```

Ou seja, teste para 200 usuários simultâneos passou (não houve nenhuma falha e o tempo de cada requisição foi de em média 468ms, o que não é muito alto).

Quantas conexões ao banco cada requisição gera

- SQL alchemy usa pool de conexões para que cada requisição não gere uma conexão nova no banco e não fique lento
- Configuração do SQL alchemy para 20 usuários simultâneos:

```
create_async_engine(
    Settings().DATABASE_URL,
  poolclass=NullPool if Settings().TEST_ENV else None, # Se em modo teste: cada requisição gera 1 conexão
  pool_size=10, # Nº de requisições permanentes (quantas conexões ficam sempre abertas)
  max_overflow=20, # Se o pool estiver cheio, podem ser criadas 20 conexões temporárias extras (em picos de uso)
  pool_timeout=30, # Se não tiver conexão livre, espera até 30s antes de dar erro
)
```

- Configuração do SQL alchemy para 200 usuários simultâneos:

```
create_async_engine(
  DATABASE_URL,
  pool_size=150,
  max_overflow=50,
  pool_timeout=30
)
```

- Por padrão, o PostgreSQL aceita até 100 conexões simultâneas, então a seguinte configuração deve ser alterada usando o comando psql dentro do container:

```
ALTER SYSTEM SET max_connections = 200;
```

- **Confirmar funcionamento utilizando testes de carga**

Teste de carga do servidor do broker

```
https://github.com/krylovsk/mqtt-benchmark
```

```
~/go/bin/mqtt-benchmark
    --broker tcp://52.22.65.168:1883
    --clients 200
    --count 10000
    --size 256
```

```
    --qos 1
    --topic teste/carga
    --format text
```

Resultados para 10000 estações mandando mensagens:

```
========== TOTAL (100) ==========
Total Ratio:               1.000 (100000/100000)
Total Runtime (sec):       1000.666
Average Runtime (sec):     1000.633
Msg time min (ms):         0.215
Msg time max (ms):         13.210
Msg time mean mean (ms):   1.882
Msg time mean std (ms):    0.297
Average Bandwidth (msg/sec): 0.999
Total Bandwidth (msg/sec):   99.937
```

O que significa que passou nos testes pois:

- Todas as mensagens foram enviadas
- Mesmo o tempo mais alto ainda foi baixo
- 100 mensagens por segundo enviadas com sucesso

Teste de carga de inserções no MongoDB

```
from pymongo import MongoClient
import time

client = MongoClient("mongodb+srv://ninecodek9:hC5EPmsZw2VoJS2S@cluster0.sfjpedo.mongodb.net/?retryWrites=tr
collection = client.db.station_data

start = time.time()

for i in range(10000):
    doc = {"station_id": i, "value": 42, "timestamp": time.time()}
    collection.insert_one(doc)

end = time.time()
print(f"Tempo total: {end - start:.2f} segundos")
```

Fazer otimizações no código para inserções mais rápidas:

- Insert many em vez de insert one

```
collection.insert_many(list_of_docs)
```

- Configurar pool de conexões

```
MongoClient(uri, maxPoolSize=100)
```

Resultado sem otimizações:

```
Tempo total: 316.28 segundos
```

O que dá conta, já que as estações mandarão dados a cada 15 minutos.

Teste de carga de inserções no PostgreSQL

```
import asyncio
import asyncpg
import time

DB_URL = "postgresql+asyncpg://codenine:codenine2025@34.231.36.192:5432/tecsus"

NUM_INSERCOES = 10000
BATCH_SIZE = 500  # insere 500 por vez

async def preparar_banco(conn):
    await conn.execute("""
        CREATE TABLE IF NOT EXISTS sensor_data (
            id SERIAL PRIMARY KEY,
            station_id INT,
            valor FLOAT,
            timestamp TIMESTAMPTZ DEFAULT now()
        );
    """)

async def inserir_dados(conn):
    inicio = time.time()

    for i in range(0, NUM_INSERCOES, BATCH_SIZE):
        batch = [(j, 42.0) for j in range(i, i + BATCH_SIZE)]
        await conn.executemany(
            "INSERT INTO sensor_data (station_id, valor) VALUES ($1, $2);",
            batch
        )

    fim = time.time()
    print(f"Inseridos {NUM_INSERCOES} registros em {fim - inicio:.2f} segundos")

async def main():
    conn = await asyncpg.connect(DB_URL)
    await preparar_banco(conn)
    await inserir_dados(conn)
    await conn.close()

asyncio.run(main())
```

Resultado:

```
Inseridos 10000 registros em 5.18 segundos
```

Ou seja, inserções eficazes.

Caso haja maior expansão, o que terá que ser ajustado?

O primeiro servidor a sofrer problemas será o da API ao atingir ~300 usuários simultâneos. Aproximadamente 20% das requisições passam a dar erro.

Se esse ponto for atingido, mudaremos a instância de t2.micro para t3.medium.

Conclusão

Com esse fluxo, a cada push na branch main, o GitHub Actions conecta automaticamente à EC2 via SSH, atualiza o código e realiza o deploy com Docker Compose. Isso garante entregas contínuas sem intervenção manual.

Foram também realizados testes de carga nas diferentes instâncias do sistema, simulando usuários simultâneos acessando os serviços e estações publicando grandes quantidades de dados. Os resultados demonstraram estabilidade, ausência de falhas e tempos de resposta dentro dos padrões esperados, mesmo sob condições intensas de uso. Isso confirma que a arquitetura proposta está preparada para lidar com o volume previsto de acessos e processamentos, com margem para expansão conforme a demanda cresça.