Clase 04: Pipeline de desarrollo geoespacial

De los datos a la solución en producción

Profesor: Francisco Parra O.

27 de agosto de 2025

USACH - Ingeniería Civil en Informática

Agenda

Flujo de trabajo geoespacial

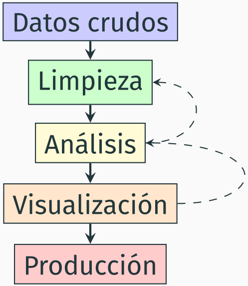
Pipeline completo de desarrollo

Fases del desarrollo:

1. **Adquisición** de datos
2. **Limpieza** y validación
3. **Análisis** espacial
4. **Visualización**
5. **Deployment**

Stack tecnológico típico:

- Python/R para análisis
- PostGIS para almacenamiento
- QGIS para exploración
- Web frameworks para deployment



Estructura de proyecto geoespacial

Organización recomendada:

```
proyecto_geo/
          data/
                raw/           # Datos originales
                processed/     # Datos limpios
                cache/         # Resultados cache
          src/
                etl/          # Extract-Transform-Load
                analysis/     # An lisis espacial
                api/          # REST API
                visualization/ # Mapas y gr ficos
          notebooks/        # Jupyter notebooks
          tests/           # Tests unitarios
          config/          # Configuraci n
          docker/          # Contenedores
          docs/           # Documentaci n
```

Archivo de configuración:

```yaml
# config.yaml
database:
  host: localhost
  port: 5432
  name: geodatabase
  user: ${DB_USER}
  password: ${DB_PASS}

apis:
  osm_overpass:
    url: "https://overpass-api.de"
    timeout: 30
  google_maps:
    key: ${GOOGLE_API_KEY}

cache:
  enabled: true
  ttl: 3600  # segundos

crs:
  default: "EPSG:4326"
  local: "EPSG:32719"  # UTM 19S
```

Control de versiones para datos espaciales

Git + DVC (Data Version Control):

```
# Instalar DVC
pip install dvc

# Inicializar DVC en proyecto Git
dvc init

# Agregar archivo grande
dvc add data/comunas_chile.gpkg

# Commit cambios
git add data/comunas_chile.gpkg.dvc
git commit -m "Add comunas dataset"

# Push a storage remoto (S3, GCS, etc)
dvc remote add -d myremote s3://bucket/path
dvc push
```

Mejores prácticas:

- No versionar datos ¿ 100MB en Git
- Usar '.gitignore' para datos locales
- Documentar origen y fecha de datos
- Mantener checksums de archivos

.gitignore típico:

```
# Datos
data/raw/*
data/cache/*
*.gpkg
*.tif

# Temporales
*.qgz~
.ipynb_checkpoints/

# Credenciales
.env
config/secrets.yaml
```

Conexión a fuentes de datos reales

APIs geoespaciales: OpenStreetMap

OSMnx - Python:

```python
import osmnx as ox
import geopandas as gpd

# Configurar OSMnx
ox.config(use_cache=True, log_console=True)

# Obtener red vial de Santiago
place = "Santiago, Chile"
G = ox.graph_from_place(place,
                        network_type='drive')

# Convertir a GeoDataFrame
nodes, edges = ox.graph_to_gdfs(G)

# Obtener edificios
buildings = ox.geometries_from_place(
    place,
    tags={'building': True}
)

# Obtener amenities espec ficos
tags = {'amenity': ['hospital', 'school']}
amenities = ox.geometries_from_place(
    place, tags
)

```

Overpass API - Query directa:

```python
import requests
import geopandas as gpd

# Query Overpass QL
query = """
[out:json][timeout:25];
area["name"="Santiago"]->.searchArea;
(
  node["amenity"="hospital"](area.searchArea);
  way["amenity"="hospital"](area.searchArea);
);
out geom;
"""

# Ejecutar query
overpass_url = "http://overpass-api.de/api/interpreter"
response = requests.get(
    overpass_url,
    params={'data': query}
)

data = response.json()
# Convertir a GeoDataFrame...

```

PostGIS: Base de datos espacial

Conexión y consultas:

```python
from sqlalchemy import create_engine
import geopandas as gpd
import pandas as pd

# Crear conexi n
engine = create_engine(
    'postgresql://user:pass@localhost/geodata'
)

# Leer tabla espacial
sql = """
SELECT
    c.nombre,
    c.poblacion,
    c.geom,
    COUNT(h.id) as num_hospitales
FROM
    comunas c
LEFT JOIN
    hospitales h ON ST_Contains(c.geom, h.geom)
GROUP BY
    c.id, c.nombre, c.poblacion, c.geom
"""

gdf = gpd.read_postgis(sql, engine,
                       geom_col='geom')
```

Operaciones espaciales en DB:

```sql
-- Crear  ndice  espacial
CREATE INDEX idx_comunas_geom
ON comunas USING GIST(geom);

-- Buffer de 1km alrededor de metro
CREATE TABLE areas_metro AS
SELECT
    id,
    nombre,
    ST_Buffer(geom::geography, 1000)::geometry as buffer
FROM estaciones_metro;

-- Encontrar comunas vecinas
SELECT
    a.nombre as comuna,
    b.nombre as vecina
FROM
    comunas a, comunas b
WHERE
    ST_Touches(a.geom, b.geom)
    AND a.id < b.id;
```

APIs de datos gubernamentales

IDE Chile - WFS:

```python
import geopandas as gpd
from owslib.wfs import WebFeatureService

# Conectar a IDE Chile
wfs_url = "http://www.ide.cl/geoserver/wfs"
wfs = WebFeatureService(url=wfs_url, version='2.0.0')

# Listar capas disponibles
list(wfs.contents)

# Obtener divisi n administrativa
response = wfs.getfeature(
    typename='division_politica:comunas',
    bbox=(-71, -34, -70, -33),  # RM
    srsname='EPSG:4326',
    outputFormat='json'
)

# Leer como GeoDataFrame
import json
comunas = gpd.GeoDataFrame.from_features(
    json.loads(response.read())
)
```

APIs REST - SII, INE:

```python
# Datos INE - Censo
import requests
import pandas as pd

# API INE (ejemplo)
url = "https://api.ine.cl/datos/censo2017"
params = {
    'region': '13',
    'indicador': 'poblacion',
    'formato': 'json'
}
response = requests.get(url, params=params)
censo_data = pd.DataFrame(response.json())

# Geocodificar direcciones SII
from geopy.geocoders import Nominatim

geolocator = Nominatim(user_agent="my_app")

def geocode_address(address):
    try:
        location = geolocator.geocode(
            f"{address}, Santiago, Chile"
        )
        return location.latitude, location.longitude
    except:
        return None, None
```

Google Maps y otras APIs comerciales

Google Maps Platform:

```python
import googlemaps
from datetime import datetime

# Cliente con API key
gmaps = googlemaps.Client(key='YOUR_API_KEY')

# Geocoding
geocode_result = gmaps.geocode(
    'Av. Libertador Bernardo O\'Higgins 3363, Santiago'
)

# Places API - buscar hospitales cercanos
places_result = gmaps.places_nearby(
    location=(-33.45, -70.65),
    radius=2000,
    type='hospital'
)

# Distance Matrix - tiempos de viaje
origins = [(-33.45, -70.65), (-33.44, -70.64)]
destinations = [(-33.46, -70.66)]

matrix = gmaps.distance_matrix(
    origins,
    destinations,
    mode="driving",
    departure_time=datetime.now()
)
```

Mapbox - Isócronas:

```python
import requests

# Isochrone API
url = "https://api.mapbox.com/isochrone/v1/mapbox/
    driving"

params = {
    'coordinates': '-70.65,-33.45',
    'contours_minutes': '5,10,15',
    'polygons': 'true',
    'access_token': 'YOUR_TOKEN'
}

response = requests.get(url, params=params)
isochrones = response.json()

# Convertir a GeoDataFrame
import geopandas as gpd
from shapely.geometry import shape

features = []
for feature in isochrones['features']:
    geom = shape(feature['geometry'])
    props = feature['properties']
    features.append({'geometry': geom,
                     'minutes': props['contour']})

gdf = gpd.GeoDataFrame(features)
```

Análisis espacial aplicado

Geocodificación masiva

Pipeline de geocodificación:

```python
import pandas as pd
import geopandas as gpd
from geopy.geocoders import Nominatim
from geopy.extra.rate_limiter import RateLimiter
import time

# Cargar direcciones
df = pd.read_csv('direcciones.csv')

# Configurar geocoder con rate limiting
geolocator = Nominatim(user_agent="myapp")
geocode = RateLimiter(geolocator.geocode,
                      min_delay_seconds=1)

# Funci n robusta de geocodificaci n
def geocode_df(df, address_col):
    locations = []
    for idx, row in df.iterrows():
        try:
            location = geocode(row[address_col])
            if location:
                locations.append({
                    'lat': location.latitude,
                    'lon': location.longitude,
                    'address': row[address_col]
                })
        except Exception as e:
            print(f"Error en {row[address_col]}: {e}")
            locations.append({'lat': None, 'lon': None})
    return pd.DataFrame(locations)
```

Validación y corrección:

```python
# Validar resultados
def validate_geocoding(gdf, comuna_bounds):
    """Verificar que puntos est n en comuna"""
    valid = []
    for idx, row in gdf.iterrows():
        point = row.geometry
        comuna = comuna_bounds[
            comuna_bounds.contains(point)
        ]
        if len(comuna) > 0:
            valid.append(True)
        else:
            valid.append(False)
    return valid

# Geocoding inverso para verificar
from geopy.geocoders import reverse

def verify_address(lat, lon, expected_comuna):
    location = geolocator.reverse(f"{lat}, {lon}")
    address = location.raw['address']

    if expected_comuna.lower() in
       address.get('city', '').lower():
        return True
    return False
```

Análisis de áreas de influencia

Buffer vs Isócronas reales:

```python
import osmnx as ox
import networkx as nx
import geopandas as gpd
from shapely.geometry import Point

# Red vial
G = ox.graph_from_place('Santiago, Chile',
                        network_type='drive')
G = ox.project_graph(G)

# Punto de inter s (ej: hospital)
hospital = Point(-70.65, -33.45)
hospital_proj = ox.project_gdf(
    gpd.GeoDataFrame([1], geometry=[hospital],
                    crs='EPSG:4326')
).geometry[0]

# Nodo m s cercano
hospital_node = ox.nearest_nodes(
    G, hospital_proj.x, hospital_proj.y
)

# Is crona de 10 minutos
travel_speed = 40  # km/h
trip_time = 10 * 60  # 10 min en segundos
meters = travel_speed * 1000 / 60 / 60 * trip_time

# Subgrafo alcanzable
subgraph = nx.ego_graph(G, hospital_node,
                        radius=meters,
```

Análisis de accesibilidad:

```python
# Calcular accesibilidad a servicios
def calculate_accessibility(puntos, servicios,
                            max_distance=2000):
    """
    Calcula m tricas de accesibilidad
    """
    results = []

    for idx, punto in puntos.iterrows():
        # Distancia al servicio m s cercano
        distances = servicios.distance(punto.geometry)
        min_dist = distances.min()

        # N mero de servicios en radio
        within = distances <= max_distance
        count = within.sum()

        #  ndice  de accesibilidad
        if min_dist > 0:
            accessibility = count / (min_dist / 1000)
        else:
            accessibility = count * 10

        results.append({
            'id': idx,
            'min_distance': min_dist,
            'services_count': count,
            'accessibility_index': accessibility
        })
```

Clustering espacial

DBSCAN espacial:

```python
from sklearn.cluster import DBSCAN
import numpy as np
import geopandas as gpd

# Preparar datos
coords = np.array([[p.x, p.y] for p in gdf.geometry])

# DBSCAN con distancia en metros
kms_per_radian = 6371.0088
epsilon = 0.5 / kms_per_radian  # 500 metros

db = DBSCAN(eps=epsilon, min_samples=5,
            algorithm='ball_tree',
            metric='haversine').fit(np.radians(coords))

# Asignar clusters
gdf['cluster'] = db.labels_

# An lisis de clusters
cluster_stats = gdf.groupby('cluster').agg({
    'precio': ['mean', 'std', 'count'],
    'superficie': 'mean',
    'geometry': lambda x: x.unary_union.centroid
})

# Visualizar hotspots
hotspots = gdf[gdf['cluster'] != -1]
```

K-means ponderado:

```python
from sklearn.cluster import KMeans

# Preparar features espaciales y atributos
X = np.column_stack([
    coords,  # ubicaci n
    gdf['precio'].values / 1e6,  # normalizado
    gdf['m2'].values / 100
])

# Ponderar componentes
weights = [1.0, 1.0, 0.5, 0.3]  # x,y,precio,m2
X_weighted = X * weights

# K-means
kmeans = KMeans(n_clusters=5, random_state=42)
gdf['segment'] = kmeans.fit_predict(X_weighted)

# Centros de clusters
centers = kmeans.cluster_centers_ / weights
center_points = [Point(c[0], c[1]) for c in centers]

# Pol gonos de Voronoi para  reas  de mercado
from scipy.spatial import Voronoi
vor = Voronoi(centers[:, :2])
```

Interpolación espacial

Kriging para valores continuos:

```python
from pykrige.ok import OrdinaryKriging
import numpy as np

# Datos de entrada
points = gdf[['x', 'y', 'precio_m2']].values
lons = points[:, 0]
lats = points[:, 1]
values = points[:, 2]

# Crear grid de interpolaci n
grid_lon = np.linspace(lons.min(), lons.max(), 100)
grid_lat = np.linspace(lats.min(), lats.max(), 100)

# Ordinary Kriging
OK = OrdinaryKriging(lons, lats, values,
                     variogram_model='spherical',
                     verbose=False,
                     enable_plotting=False)

z, ss = OK.execute('grid', grid_lon, grid_lat)

# Convertir a raster
import rasterio
from rasterio.transform import from_origin

transform = from_origin(west, north, pixel_size, pixel_size)
with rasterio.open('interpolated.tif', 'w',
                  driver='GTiff', height=z.shape[0],
                  width=z.shape[1], count=1,
                  dtype=z.dtype, crs='EPSG:4326',
```

IDW (Inverse Distance Weight):

```python
def idw_interpolation(points, values,
                      grid_points, power=2):
    """
    Interpolaci n IDW simple
    """
    interpolated = []

    for grid_point in grid_points:
        # Distancias a todos los puntos
        distances = np.sqrt(
            (points[:, 0] - grid_point[0])**2 +
            (points[:, 1] - grid_point[1])**2
        )

        # Evitar divisi n por cero
        distances[distances == 0] = 1e-10

        # Pesos inversos
        weights = 1 / distances**power
        weights /= weights.sum()

        # Valor interpolado
        value = np.sum(weights * values)
        interpolated.append(value)

    return np.array(interpolated)

# Aplicar a grid regular
grid_x, grid_y = np.meshgrid(
    np.linspace(min_x, max_x, 50),
```

Optimización y escalabilidad

Manejo eficiente de grandes datasets

Chunking y procesamiento por lotes:

```python
import geopandas as gpd
import pandas as pd
from shapely import wkt

# Leer en chunks
chunk_size = 10000
chunks = []

for chunk in pd.read_csv('huge_dataset.csv',
                         chunksize=chunk_size):
    # Procesar chunk
    chunk['geometry'] = chunk['wkt'].apply(wkt.loads)
    gdf_chunk = gpd.GeoDataFrame(chunk,
                                 crs='EPSG:4326')

    # Operaci n espacial en chunk
    gdf_chunk = gdf_chunk.to_crs('EPSG:32719')
    gdf_chunk['area'] = gdf_chunk.area

    # Filtrar y guardar resultado
    filtered = gdf_chunk[gdf_chunk['area'] > 1000]
    chunks.append(filtered)

# Combinar resultados
result = pd.concat(chunks, ignore_index=True)

# Guardar en formato eficiente
result.to_parquet('processed_data.parquet')
```

Dask para paralelización:

```python
import dask_geopandas as dgpd
import dask.dataframe as dd

# Leer dataset particionado
ddf = dgpd.read_parquet(
    'huge_dataset.parquet',
    npartitions=8
)

# Operaciones lazy (no se ejecutan a n)
ddf = ddf.to_crs('EPSG:32719')
ddf['buffer_100m'] = ddf.buffer(100)

# Spatial join paralelo
other_ddf = dgpd.read_file('polygons.gpkg',
                          npartitions=4)
joined = dgpd.sjoin(ddf, other_ddf,
                    how='inner',
                    predicate='intersects')

# Ejecutar y obtener resultado
with dask.config.set(scheduler='threads'):
    result = joined.compute()

# O guardar sin cargar en memoria
joined.to_parquet('joined_results/')
```

Índices espaciales y caché

R-tree para búsquedas rápidas:

```python
from rtree import index
import pickle
import hashlib

class SpatialCache:
    def __init__(self):
        self.idx = index.Index()
        self.cache = {}

    def add_features(self, features_gdf):
        """Agregar features al  ndice """
        for idx, row in features_gdf.iterrows():
            bounds = row.geometry.bounds
            self.idx.insert(idx, bounds)
            self.cache[idx] = row

    def query_area(self, bbox):
        """B squeda r pida por bbox"""
        candidates = list(self.idx.intersection(bbox))
        return [self.cache[i] for i in candidates]

    def nearest(self, point, n=5):
        """N vecinos m s cercanos"""
        coords = (point.x, point.y, point.x, point.y)
        nearest = list(self.idx.nearest(coords, n))
        return [self.cache[i] for i in nearest]

    def save(self, filename):
        """Persistir cach """
        with open(filename, 'wb') as f:
```

Memoization de operaciones costosas:

```python
from functools import lru_cache
import hashlib

def geometry_hash(geom):
    """Hash  nico  para geometr a"""
    wkb = geom.wkb
    return hashlib.md5(wkb).hexdigest()

@lru_cache(maxsize=1000)
def expensive_buffer_operation(geom_hash, distance):
    """Operaci n costosa con cach """
    # Reconstruir geometr a del hash
    geom = cache_dict[geom_hash]

    # Operaci n costosa
    result = geom.buffer(distance)
    for i in range(10):
        result = result.simplify(0.01)
        result = result.buffer(-distance/20)
        result = result.buffer(distance/20)

    return result

# Uso con cach
geom_id = geometry_hash(polygon)
result = expensive_buffer_operation(geom_id, 100)
```

Optimización de consultas PostGIS

Índices y particionamiento:

```sql
--  ndices  espaciales y de atributos
CREATE INDEX idx_spatial ON propiedades
USING GIST(geom);

CREATE INDEX idx_precio ON propiedades(precio);
CREATE INDEX idx_comuna ON propiedades(comuna_id);

--  ndice  compuesto
CREATE INDEX idx_spatial_price ON propiedades
USING GIST(geom, precio_range);

-- Particionamiento por regi n
CREATE TABLE propiedades_rm PARTITION OF propiedades
FOR VALUES IN (13);

CREATE TABLE propiedades_v PARTITION OF propiedades
FOR VALUES IN (5);

-- Clustering espacial para mejor performance
CLUSTER propiedades USING idx_spatial;

-- Materializar vistas complejas
CREATE MATERIALIZED VIEW mv_stats_comuna AS
SELECT
    c.id, c.nombre,
    COUNT(p.id) as total_propiedades,
    AVG(p.precio) as precio_promedio,
    ST_Union(p.geom) as coverage
FROM comunas c
LEFT JOIN propiedades p ON ST_Contains(c.geom, p.geom)
```

Query optimization:

```sql
-- Usar ST_DWithin en vez de buffer
-- MALO:
SELECT * FROM points
WHERE ST_Intersects(
    geom,
    ST_Buffer(target_point, 1000)
);

-- BUENO:
SELECT * FROM points
WHERE ST_DWithin(
    geom,
    target_point,
    1000
);

-- Simplificar para visualizaci n
SELECT
    id,
    nombre,
    ST_SimplifyPreserveTopology(
        geom,
        10  -- tolerancia
    ) as geom_simple
FROM comunas;

-- Usar && para pre-filtrar
SELECT * FROM a, b
WHERE a.geom && b.geom  -- bbox check
AND ST_Intersects(a.geom, b.geom);
```

Deployment de soluciones

API geoespacial:

```python
from fastapi import FastAPI, Query
from pydantic import BaseModel
import geopandas as gpd
from shapely.geometry import Point
import json

app = FastAPI(title="GeoAPI")

# Cache de datos
comunas_gdf = gpd.read_file("comunas.gpkg")

class LocationRequest(BaseModel):
    lat: float
    lon: float

@app.get("/api/comuna")
async def get_comuna(lat: float, lon: float):
    """Obtener comuna de un punto"""
    point = Point(lon, lat)

    for idx, comuna in comunas_gdf.iterrows():
        if comuna.geometry.contains(point):
            return {
                "comuna": comuna['nombre'],
                "region": comuna['region'],
                "poblacion": int(comuna['poblacion'])
            }

    return {"error": "Punto fuera de Chile"}
```

```python
@app.get("/api/nearest")
async def nearest_services(
    lat: float,
    lon: float,
    service_type: str,
    limit: int = 5
):
    """Servicios m s cercanos"""
    point = Point(lon, lat)

    services = load_services(service_type)
    services['distance'] = services.distance(point)
    nearest = services.nsmallest(limit, 'distance')

    return {
        "type": service_type,
        "results": [
            {
                "name": row['name'],
                "distance": round(row['distance'], 2),
                "address": row['address']
            }
            for _, row in nearest.iterrows()
        ]
    }

# Ejecutar con: uvicorn main:app --reload
```

App interactiva:

```python
import streamlit as st
import geopandas as gpd
import folium
from streamlit_folium import st_folium

st.set_page_config(page_title="GeoAnalytics",
                   layout="wide")

st.title("Dashboard Geoespacial")

# Sidebar para controles
with st.sidebar:
    st.header("Filtros")

    comuna = st.selectbox("Comuna",
                         comunas_gdf['nombre'].unique())

    precio_min, precio_max = st.slider(
        "Rango de precio",
        0, 1000000000,
        (100000000, 500000000)
    )

    year = st.slider("A o", 2020, 2024, 2024)

# Filtrar datos
filtered = propiedades[
    (propiedades['comuna'] == comuna) &
    (propiedades['precio'].between(precio_min, precio_max)) &
    (propiedades['year'] == year)
```

```python
# M tricas
col1, col2, col3 = st.columns(3)
with col1:
    st.metric("Total propiedades",
              len(filtered))
with col2:
    st.metric("Precio promedio",
              f"${filtered['precio'].mean():,.0f}")
with col3:
    st.metric("M2 promedio",
              f"{filtered['m2'].mean():.1f}")

# Mapa interactivo
m = folium.Map(location=[-33.45, -70.65],
               zoom_start=11)

for idx, row in filtered.iterrows():
    folium.CircleMarker(
        [row['lat'], row['lon']],
        radius=5,
        popup=f"${row['precio']:,.0f}",
        color='red',
        fill=True
    ).add_to(m)

st_folium(m, width=700, height=450)
```

Docker para aplicaciones geo

Dockerfile multi-stage:

```dockerfile
# Stage 1: Build dependencies
FROM python:3.9-slim as builder

RUN apt-get update && apt-get install -y \
    gdal-bin \
    libgdal-dev \
    gcc \
    g++ \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app
COPY requirements.txt .

ENV GDAL_CONFIG=/usr/bin/gdal-config
RUN pip install --no-cache-dir -r requirements.txt

# Stage 2: Runtime
FROM python:3.9-slim

RUN apt-get update && apt-get install -y \
    gdal-bin \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app
COPY --from=builder /usr/local/lib/python3.9/site-packages \
                    /usr/local/lib/python3.9/site-packages

COPY . .

EXPOSE 8000
```

docker-compose.yml:

```yaml
version: '3.8'

services:
  postgis:
    image: postgis/postgis:14-3.2
    environment:
      POSTGRES_DB: geodata
      POSTGRES_USER: geouser
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - pgdata:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  api:
    build: .
    environment:
      DB_HOST: postgis
      DB_NAME: geodata
      DB_USER: geouser
      DB_PASSWORD: ${DB_PASSWORD}
    ports:
      - "8000:8000"
    depends_on:
      - postgis
    volumes:
      - ./data:/app/data

volumes:
  pgdata:
```

Deployment en la nube

AWS - Terraform:

```
# RDS PostGIS
resource "aws_db_instance" "postgis" {
  engine         = "postgres"
  engine_version = "14.6"
  instance_class = "db.t3.micro"

  allocated_storage = 20
  storage_type     = "gp3"

  db_name  = "geodata"
  username = "geouser"
  password = var.db_password

  # Habilitar PostGIS
  enabled_cloudwatch_logs_exports = ["postgresql"]
}

# Lambda para procesamiento
resource "aws_lambda_function" "geo_processor" {
  function_name = "geo-processor"
  runtime      = "python3.9"
  handler      = "handler.main"

  layers = [
    "arn:aws:lambda:region:770693421928:layer:Klayers-p39-gdal:1"
  ]

  environment {
    variables = {
      DB_HOST = aws_db_instance.postgis.endpoint
```

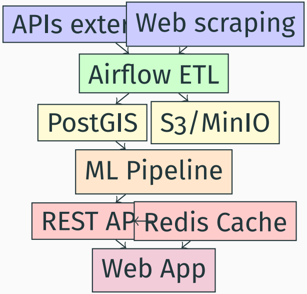
GitHub Actions CI/CD:

```
name: Deploy

on:
  push:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2

    - name: Setup Python
      uses: actions/setup-python@v2
      with:
        python-version: ’3.9’

    - name: Install GDAL
      run: |
        sudo apt-get update
        sudo apt-get install -y gdal-bin

    - name: Test
      run: |
        pip install -r requirements.txt
        pytest tests/

  deploy:
    needs: test
    runs-on: ubuntu-latest
```

Caso práctico: Sistema de valoración inmobiliaria

Arquitectura completa:

1. **Ingesta**: APIs (SII, OSM, Portal Inmobiliario)
2. **Storage**: PostGIS + S3 para imágenes
3. **Processing**: Airflow para ETL diario
4. **Analytics**: Jupyter Hub para DS
5. **API**: FastAPI con Redis cache
6. **Frontend**: React + Mapbox GL



Features espaciales:

- Distancia a metro/paraderos
- Densidad de servicios
- Índice de vegetación (NDVI)
- Contaminación acústica

Mejores prácticas y recomendaciones

Desarrollo:

- Usar ambientes virtuales (venv, conda)
- Documentar dependencias espaciales
- Tests con datos sintéticos
- Validar geometrías siempre
- Logging detallado de operaciones

Performance:

- Simplificar geometrías para web
- Usar tiles vectoriales para mapas
- Cachear resultados costosos
- Índices espaciales SIEMPRE
- Proyecciones locales para cálculos

Producción:

- Monitoreo de queries lentas
- Backups de datos espaciales
- Rate limiting en APIs
- CDN para tiles de mapas
- Healthchecks de servicios geo

Errores comunes:

- Mezclar CRS sin transformar
- No validar geometrías
- Ignorar índices espaciales
- Cargar todo en memoria

Recursos para proyectos

Datos Chile:

- IDE Chile
- SIIT - BCN
- Datos.gob.cl
- CEDEUS
- INE - Censo y cartografía

Herramientas recomendadas:

- QGIS para exploración
- DBeaver para PostGIS
- Jupyter para prototipado
- Postman para probar APIs
- Docker para desarrollo

Templates y boilerplates:

- GeoPandas examples
- Fiona recipes
- Leaflet demos
- Kepler.gl

Documentación esencial:

- PostGIS.net
- GeoPandas.org
- OSMnx documentation
- Shapely manual
- GDAL/OGR cookbook

Cierre

¿Preguntas?

Para sus proyectos:

Enfóquense en el pipeline completo desde datos hasta visualización

Próxima clase:

Análisis espacial y geoestadística