

```
import os
from math import cos, radians

import tempfile
import zipfile
import shutil
import folium
import geopandas as gpd
import numpy as np
import pandas as pd
import base64
import plotly.express as px
import plotly.graph_objects as go
import matplotlib
import matplotlib.pyplot as plt
import branca.colormap as cm
from rasterio.transform import array_bounds
from pyproj import Transformer
import pymannkendall as mk
import requests
import streamlit as st
import io
import sys
from folium import plugins
from folium.plugins import LocateControl, MarkerCluster
from plotly.subplots import make_subplots
from prophet import Prophet
from scipy import stats
from matplotlib import path as mpath
```

Imports de Ciencia de Datos y Análisis

```
from scipy.interpolate import Rbf, griddata
from shapely.geometry import Point
from shapely.geometry import LineString, MultiLineString
from shapely.geometry import MultiPolygon, Polygon
from statsmodels.tsa.seasonal import seasonal_decompose
from streamlit_folium import st_folium, folium_static
```

```
import modules.life_zones as lz
import modules.land_cover as lc
```

Módulos Internos

```
from modules.config import Config
import modules.analysis as analysis
```

```

# Importar funciones de análisis (Manejo de errores por si faltan)
try:
    from modules.analysis import (calculate_climatic_indices,
        calculate_duration_curve,
        calculate_hydrological_balance,
        calculate_hypsometric_curve,
        calculate_morphometry,
        calculate_percentiles_extremes,
        calculate_return_periods, calculate_spei,
        calculate_water_balance_turc,
        classify_holdridge_point,
        estimate_temperature,
        generate_life_zone_raster)

except ImportError:
    # Dummies para evitar crash visual si falta backend

def calculate_morphometry(g):
    return {
        "area_km2": 100,
        "perimetro_km": 50,
        "alt_prom_m": 1500,
        "pendiente_prom": 15,
    }

def calculate_hydrological_balance(p, t, g):
    return {"P": p, "ET": p * 0.6, "Q_mm": p * 0.4, "Vol": (p * 0.4 * 100) / 1000}

def calculate_duration_curve(ts, c, a):
    return None

def calculate_climatic_indices(ts, a):
    return {}

def calculate_hypsometric_curve(g):
    return None

# PESTAÑA DE BIENVENIDA (PÁGINA DE INICIO RENOVARADA)
# =====
def display_welcome_tab():
    # CSS para ajustar márgenes
    st.markdown(
        """
<style>
.block-container { padding-top: 1rem; }
h1 { margin-top: -3rem; }
        """)

```

```

</style>
"""
unsafe_allow_html=True,
)

st.title(f"Bienvenido a {Config.APP_TITLE}")
st.caption(
    "Sistema de Información Hidroclimática Integrada para la Gestión Integral del Agua y la Biodiversidad en
el Norte de la Region Andina"
)

# Pestañas de Inicio
tab_intro, tab_clima, tab_modulos, tab_aleph = st.tabs(
[
    "Presentación del Sistema",
    "Climatología Andina",
    "Módulos y Capacidades",
    "El Aleph",
]
)

```

--- PESTAÑA 1: PRESENTACIÓN (SIN LOGO) ---

with tab_intro:

Usamos el ancho completo ahora

st.markdown(

"""

Origen y Visión

****SIHCLI-POTER**** nace de la necesidad imperativa de integrar datos, ciencia y tecnología para la toma de decisiones informadas en el territorio.

En un contexto de variabilidad climática creciente, la gestión del recurso hídrico y el ordenamiento territorial requieren herramientas que transformen datos dispersos en conocimiento accionable.

Este sistema no es solo un repositorio de datos; es un ****cerebro analítico**** diseñado para procesar, modelar y visualizar la complejidad hidrometeorológica de la región Andina.

Su arquitectura modular permite desde el monitoreo en tiempo real hasta la proyección de escenarios de cambio climático a largo plazo.

Aplicaciones Clave

- * ****Gestión del Riesgo:**** Alertas tempranas y mapas de vulnerabilidad ante eventos extremos (sequías e inundaciones).

- * ****Planeación Territorial (POT):**** Insumos técnicos para la zonificación ambiental y la gestión de cuencas.

- * ****Agricultura de Precisión:**** Calendarios de siembra basados en pronósticos estacionales y zonas de vida.

- * ****Investigación:**** Base de datos depurada y herramientas estadísticas para estudios académicos.

```
--  
**Versión:** 2.0 (Cloud-Native) | **Desarrollado por:** omejia - POTER.  
****  
)  
  
# --- PESTAÑA 2: CLIMATOLOGÍA ANDINA ---  
with tab_clima:  
    st.markdown(  
        """  
        #### La Danza del Clima en los Andes  
        La región Andina es un mosaico climático de una complejidad fascinante.  
        Aquí, la geografía no es solo un escenario, sino un actor protagonista que esculpe el clima kilómetro a kilómetro.  
  
        **La Verticalidad como Destino:**  
        En los Andes, viajar hacia arriba es como viajar hacia los polos.  
        En pocos kilómetros lineales, pasamos del calor húmedo de los valles interandinos (bosque seco tropical) a la neblina perpetua de los bosques de niebla, y finalmente al gélido silencio de los páramos y las nieves perpetuas. Esta **zonificación altitudinal** (bien descrita por Holdridge) define la vocación del suelo y la biodiversidad.  
  
        **El Pulso de Dos Océanos:**  
        Somos un país anfibio, respirando la humedad que llega tanto del Pacífico (Chocó Biogeográfico) como de la Amazonía.  
        Los vientos alisios chocan contra nuestras cordilleras, descargando su humedad en las vertientes orientales y creando "fábricas de agua" que alimentan nuestros grandes ríos.  
  
        **La Variabilidad (ENSO):**  
        Este sistema complejo no es estático. Está sometido al latido irregular del Pacífico Ecuatorial:  
        * **El Niño (Fase Cálida):** Cuando el océano se calienta, la atmósfera sobre nosotros se estabiliza, las nubes se disipan y la sequía amenaza, trayendo consigo el riesgo de incendios y desabastecimiento.  
        * **La Niña (Fase Fría):** Cuando el océano se enfriá, los vientos se aceleran y la humedad se condensa con furia, desbordando ríos y saturando laderas.  
  
        Entender esta climatología no es solo leer termómetros; es comprender la interacción dinámica entre la montaña, el viento y el océano.  
        """  
    )  
  
# --- PESTAÑA 3: MÓDULOS ---  
with tab_modulos:  
    st.markdown(  
        """  
        #### Arquitectura del Sistema  
        SIHCLI-POTER está estructurado en módulos especializados interconectados:  
        """
```

1. ** 🚨 Monitoreo (Tiempo Real):**
 - * Tablero de control con las últimas lecturas de estaciones telemétricas.
 - * Alertas inmediatas de umbrales críticos.
 2. ** 🌎 Distribución Espacial:**
 - * Mapas interactivos para visualizar la red de monitoreo.
 - * Análisis de cobertura espacial y densidad de datos.
 3. ** 🌌 Pronóstico Climático & ENSO:**
 - * Integración directa con el **IRI (Columbia University)** para pronósticos oficiales de El Niño/La Niña.
 - * Modelos de predicción local (Prophet, SARIMA) y análisis de probabilidades.
 4. ** 🔍 Tendencias y Riesgo:**
 - * Análisis estadístico de largo plazo (Mann-Kendall) para detectar si llueve más o menos que antes.
 - * Mapas de vulnerabilidad hídrica interpolados.
 5. ** 🛫 Satélite y Sesgo:**
 - * Comparación de datos de tierra vs. reanálisis satelital (ERA5-Land).
 - * Herramientas para corregir y llenar series históricas.
 6. ** 🌱 Zonas de Vida y Cobertura:**
 - * Cálculo automático de la clasificación de Holdridge.
 - * Análisis de uso del suelo y cobertura vegetal.
-
-)

```
# --- PESTAÑA 4: EL ALEPH ---  
with tab_aleph:  
    c_text, c_img = st.columns([3, 1])  
    with c_text:  
        st.markdown(  
            .....  
            > *"Borges y el Aleph: La metáfora perfecta de la información total."*  
  
        #### Fragmento de "El Aleph"  
  
        "... Todo lenguaje es un alfabeto de símbolos cuyo ejercicio presupone un pasado que los  
        interlocutores comparten;  
        ¿cómo transmitir a los otros el infinito Aleph, que mi temerosa memoria apenas abarca? (...)  
  
        En la parte inferior del escalón, hacia la derecha, vi una pequeña esfera tornasolada, de casi  
        intolerable fulgor.
```

Al principio la creí giratoria; luego comprendí que ese movimiento era una ilusión producida por los vertiginosos espectáculos que encerraba.

El diámetro del Aleph sería de dos o tres centímetros, pero el espacio cósmico estaba ahí, sin disminución de tamaño.

Cada cosa (la luna del espejo, digamos) era infinitas cosas, porque yo la veía claramente desde todos los puntos del universo.

Vi el populoso mar, vi el alba y la tarde, vi las muchedumbres de América,
vi una plateada telaraña en el centro de una negra pirámide, vi un laberinto roto (era Londres),
vi interminables ojos inmediatos escrutándose en mí como en un espejo, vi todos los espejos del
planeta y ninguno me reflejó...

**Vi el engranaje del amor y la modificación de la muerte, vi el Aleph, desde todos los puntos,
vi en el Aleph la tierra, y en la tierra otra vez el Aleph y en el Aleph la tierra, vi mi cara y mis vísceras, vi
tu cara, y sentí vértigo y lloré,
porque mis ojos habían visto ese objeto secreto y conjectural, cuyo nombre usurpan los hombres, pero
que ningún hombre ha mirado: el inconcebible universo."**

— *Jorge Luis Borges (1945)*

.....

)

with c_img:

st.info(

"El Aleph del tiempo, del clima, del agua, de la biodiversidad, ... del terri-torio."

)

1. FUNCIONES AUXILIARES

--- HELPER: GEOLOCALIZACIÓN MANUAL PARA PLOTLY ---

def _get_user_location_sidebar(key_suffix=""):

"""Agrega controles en el sidebar para ubicar al usuario en mapas Plotly."""

with st.sidebar.expander(f"📍 Mi Ubicación ({key_suffix})", expanded=False):

st.caption(

"Ingrese coordenadas para ver su ubicación en los mapas estáticos (Zonas de Vida, Isoyetas, etc)."

)

Usamos key_suffix para hacer únicos los keys

u_lat = st.number_input(

"Latitud:", value=6.25, format=".4f", step=0.01, key=f"u_lat_{key_suffix}"

)

u_lon = st.number_input(

"Longitud:",

value=-75.56,

```

        format=".4f",
        step=0.01,
        key=f"u_lon_{key_suffix}",
    )
    show_loc = st.checkbox(
        "Mostrar en mapa", value=False, key=f"show_loc_{key_suffix}"
    )

if show_loc:
    st.success(f"📍 Ubicación activa:\nLat: {u_lat}\nLon: {u_lon}")
    return (u_lat, u_lon)
return None

def _plot_panel_regional(rng, meth, col, tag, u_loc, df_long, gdf_stations):
    """Helper para graficar un panel regional (A o B)."""
    mask = (df_long[Config.YEAR_COL] >= rng[0]) & (df_long[Config.YEAR_COL] <= rng[1])
    df_sub = df_long[mask]
    df_avg = _calcular_promedios_reales(df_sub)

    if df_avg.empty:
        col.warning(f"Sin datos para {rng}")
        return

    if Config.STATION_NAME_COL not in df_avg.columns:
        df_avg = df_avg.reset_index()

    df_m = pd.merge(df_avg, gdf_stations, on=Config.STATION_NAME_COL).dropna(
        subset=["latitude", "longitude"]
    )

    if len(df_m) > 2:
        bounds = [
            df_m.longitude.min() - 0.1,
            df_m.longitude.max() + 0.1,
            df_m.latitude.min() - 0.1,
            df_m.latitude.max() + 0.1,
        ]
        gx, gy, gz = _run_interp(df_m, meth, bounds)

    if gz is not None:
        # Mapa Plotly (Isoyetas)
        fig = go.Figure(
            go.Contour(
                z=gz.T,
                x=gx[:, 0],

```

```

y=gy[0,:],
colorscale="Viridis",
colorbar=dict(title="mm"),
contours=dict(start=0, end=5000, size=200),
)
)

# Estaciones
fig.add_trace(
    go.Scatter(
        x=df_m.longitude,
        y=df_m.latitude,
        mode="markers",
        marker=dict(color="black", size=5),
        text=df_m[Config.STATION_NAME_COL],
        hoverinfo="text",
        showlegend=False,
    )
)

# --- CAPA USUARIO (Estrella Roja) ---
if u_loc:
    fig.add_trace(
        go.Scatter(
            x=[u_loc[1]],
            y=[u_loc[0]],
            mode="markers+text",
            marker=dict(color="red", size=15, symbol="star"),
            text=["📍 TÚ"],
            textposition="top center",
            name="Tu Ubicación",
        )
    )

fig.update_layout(
    title=f"Ppt Media ({rng[0]}-{rng[1]})",
    margin=dict(l=0, r=0, b=0, t=30),
    height=350,
)
col.plotly_chart(fig, use_container_width=True)

# Mapa Interactivo (Folium)
with col.expander(
    f"🔍 Ver Mapa Interactivo Detallado ({tag})", expanded=True
):

```

```

col.write(
    "Mapa navegable con detalles por estación. Haga clic en los puntos."
)

# Centrar mapa en usuario si existe, sino en el centro de los datos
if u_loc:
    center_lat, center_lon = u_loc
    zoom = 10
else:
    center_lat = (bounds[2] + bounds[3]) / 2
    center_lon = (bounds[0] + bounds[1]) / 2
    zoom = 8

m = folium.Map(
    location=[center_lat, center_lon],
    zoom_start=zoom,
    tiles="CartoDB positron",
)
for _, row in df_m.iterrows():
    nombre = row[Config.STATION_NAME_COL]
    lluvia = row[Config.PRECIPITATION_COL]
    altura = row.get(Config.ALTITUDE_COL, "N/A")
    muni = row.get(Config.MUNICIPALITY_COL, "N/A")

    html = f"""
<div style='font-family:sans-serif;font-size:13px;min-width:180px'>
    <h5 style='margin:0; color:#c0392b; border-bottom:1px solid #ccc; padding-
bottom:4px'>{nombre}</h5>
        <div style="margin-top:5px;"><b>Mun:</b> {muni}<br><b>Alt:</b> {altura} m</div>
        <div style='background-color:#f0f2f6; padding:5px; margin-top:5px; border-radius:4px;'>
            <b>Ppt Media:</b> {lluvia:.0f} mm<br>
        </div>
    </div>
    """
    popup = folium.Popup(
        folium.IFrame(html, width=220, height=160), max_width=220
    )
    folium.CircleMarker(
        [row["latitude"], row["longitude"]],
        radius=6,
        color="blue",
        fill=True,
        fill_color="cyan",
        fill_opacity=0.9,
        popup=popup,

```

```

        tooltip=f"{{nombre}}",
    ).add_to(m)

    # 1. Marcador de Usuario (Si existe)
    if u_loc:
        folium.Marker(
            [u_loc[0], u_loc[1]],
            icon=folium.Icon(color="black", icon="star"),
            tooltip="Tu Ubicación",
        ).add_to(m)

    # 2. Botón de Geolocalización (El ícono que pediste)
    LocateControl(auto_start=False).add_to(m)
    st_folium(
        m, height=350, use_container_width=True, key=f"folium_comp_{tag}"
    )

    # Botón GPS Nativo
    LocateControl(auto_start=False).add_to(m)
    st_folium(
        m, height=350, use_container_width=True, key=f"fol_comp_{tag}"
    )

@st.cache_data(ttl=3600)
def get_img_as_base64(url):
    """
    Descarga una imagen y la convierte a string Base64.
    Esto permite incrustarla directamente en el HTML, evitando bloqueos de hotlinking.
    """

    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36",
        "Referer": "https://google.com",
    }
    try:
        r = requests.get(url, headers=headers, timeout=10)
        if r.status_code == 200:
            # Codificar a Base64
            encoded = base64.b64encode(r.content).decode()
            return f"data:image/png;base64,{encoded}"
    except Exception as e:
        print(f"Error Base64: {e}")
    return None

def analyze_point_data(lat, lon, df_long, gdf_stations, gdf_municipios, gdf_subcuenca):

```

```

#####
Analiza un punto geográfico:
1. Toponimia (Municipio/Cuenca).
2. Datos Históricos (Interpolados).
3. Variables Ambientales (Raster).
#####

# NOTA: Point ya está importado globalmente, no lo redefinimos aquí.

# Importaciones locales seguras (expandidas para evitar error E701)
try:
    import modules.land_cover as lc
except ImportError:
    lc = None

try:
    import modules.life_zones as lz
except ImportError:
    lz = None

try:
    import pymannkendall as mk
except ImportError:
    mk = None

# Configuración
Config = None
try:
    from modules.config import Config as Cfg
    Config = Cfg
except Exception:
    pass

results = {}
# Usamos Point del scope global
point_geom = Point(lon, lat)

# 1. CONTEXTO GEOGRÁFICO
results["Municipio"] = "Desconocido"
results["Cuenca"] = "Fuera de cuencas principales"

try:
    if gdf_municipios is not None and not gdf_municipios.empty:
        matches = gdf_municipios[gdf_municipios.contains(point_geom)]
        if not matches.empty:
            results["Municipio"] = matches.iloc[0].get("nombre", "Sin Nombre")

```

```

if gdf_subcuenca is not None and not gdf_subcuenca.empty:
    matches_c = gdf_subcuenca[gdf_subcuenca.contains(point_geom)]
    if not matches_c.empty:
        results["Cuenca"] = matches_c.iloc[0].get("nombre", "Sin Nombre")
    except Exception as e:
        print(f"Error espacial: {e}")

# 2. INTERPOLACIÓN (Simplificada)
results["Ppt_Media"] = 0
results["Tendencia"] = 0

try:
    if not gdf_stations.empty:
        # Lógica simple de proximidad si no hay interpolación compleja
        # Aquí puedes reactivar tu lógica IDW completa si la necesitas
        pass
except Exception:
    pass

# 3. RASTERS (ALTITUD Y COBERTURA)
results["Altitud"] = 1500
results["Cobertura"] = "No disponible"

try:
    import rasterio

    # A. Altitud
    if Config and hasattr(Config, "DEM_FILE_PATH"):
        if os.path.exists(Config.DEM_FILE_PATH):
            try:
                with rasterio.open(Config.DEM_FILE_PATH) as src:
                    val_gen = src.sample([(lon, lat)])
                    val = next(val_gen)[0]
                    if val > -1000:
                        results["Altitud"] = int(val)
            except Exception:
                pass

    # B. Cobertura (Módulo Centralizado)
    if Config and hasattr(Config, "LAND_COVER_RASTER_PATH"):
        if lc:
            results["Cobertura"] = lc.get_land_cover_at_point(
                lat, lon, Config.LAND_COVER_RASTER_PATH
            )

except Exception as e:

```

```

results["Cobertura"] = f"Error: {str(e)}"

# 4. ZONA DE VIDA
try:
    if lz and hasattr(lz, "classify_life_zone_alt_ppt"):
        z_id = lz.classify_life_zone_alt_ppt(results["Altitud"], results["Ppt_Media"])
        results["Zona_Vida"] = lz.holdridge_int_to_name_simplified.get(z_id, "Desconocido")
    else:
        results["Zona_Vida"] = "Módulo LZ no disponible"
except Exception:
    results["Zona_Vida"] = "Error cálculo LZ"

return results


def get_weather_forecast_detailed(lat, lon):
    """
    Obtiene pronóstico detallado de Open-Meteo con 9 variables agrometeorológicas.
    """

    try:
        url = "https://api.open-meteo.com/v1/forecast"
        params = {
            "latitude": lat,
            "longitude": lon,
            "daily": [
                "temperature_2m_max",
                "temperature_2m_min",
                "precipitation_sum",
                "relative_humidity_2m_mean",
                "surface_pressure_mean",
                "et0_fao_evapotranspiration",
                "shortwave_radiation_sum",
                "wind_speed_10m_max",
            ],
            "timezone": "auto",
        }
        response = requests.get(url, params=params, timeout=5)
        data = response.json()

        daily = data.get("daily", {})
        if not daily:
            return pd.DataFrame()

        # Crear DataFrame
        df = pd.DataFrame(
            {

```

```

        "Fecha": pd.to_datetime(daily.get("time", [])),
        "T. Máx (°C)": daily.get("temperature_2m_max", []),
        "T. Mín (°C)": daily.get("temperature_2m_min", []),
        "Ppt. (mm)": daily.get("precipitation_sum", []),
        "HR Media (%)": daily.get("relative_humidity_2m_mean", []),
        "Presión (hPa)": daily.get("surface_pressure_mean", []),
        "ET0 (mm)": daily.get("et0_fao_evapotranspiration", []),
        "Radiación SW (MJ/m2)": daily.get("shortwave_radiation_sum", []),
        "Viento Máx (km/h)": daily.get("wind_speed_10m_max", []),
    }
)
return df
except Exception:
    return pd.DataFrame()

def create_enso_chart(enso_data):
    """
    Genera el gráfico avanzado de ENSO con franjas de fondo para las fases (El Niño/La Niña).
    """

    if (
        enso_data is None
        or enso_data.empty
        or Config.ENSO_ONI_COL not in enso_data.columns
    ):
        return go.Figure().update_layout(title="Datos ENSO no disponibles", height=300)

    # Preparar datos
    data = (
        enso_data.copy()
        .sort_values(Config.DATE_COL)
        .dropna(subset=[Config.ENSO_ONI_COL])
    )

    # Definir colores de fondo según el valor ONI
    conditions = [data[Config.ENSO_ONI_COL] >= 0.5, data[Config.ENSO_ONI_COL] <= -0.5]
    colors = ["rgba(255, 0, 0, 0.2)", "rgba(0, 0, 255, 0.2)"]
    data["color"] = np.select(conditions, colors, default="rgba(200, 200, 200, 0.2)")

    y_min = data[Config.ENSO_ONI_COL].min() - 0.5
    y_max = data[Config.ENSO_ONI_COL].max() + 0.5

    fig = go.Figure()

    # 1. Barras de Fondo (Fases)
    fig.add_trace(

```

```

go.Bar(
    x=data[Config.DATE_COL],
    y=[y_max - y_min] * len(data),
    base=y_min,
    marker_color=data["color"],
    width=86400000 * 30, # Ancho aprox de 1 mes en ms
    hoverinfo="skip",
    showlegend=False,
    name="Fase",
)
)

# 2. Línea Principal (ONI)
fig.add_trace(
    go.Scatter(
        x=data[Config.DATE_COL],
        y=data[Config.ENSO_ONI_COL],
        mode="lines",
        line=dict(color="black", width=2),
        name="Anomalía ONI",
    )
)

# 3. Líneas de Umbral
fig.add_hline(
    y=0.5,
    line_dash="dash",
    line_color="red",
    annotation_text="Umbral El Niño (+0.5)",
)
fig.add_hline(
    y=-0.5,
    line_dash="dash",
    line_color="blue",
    annotation_text="Umbral La Niña (-0.5)",
)
fig.add_hline(y=0, line_width=1, line_color="black")

# 4. Leyenda Personalizada
fig.add_trace(
    go.Scatter(
        x=[None],
        y=[None],
        mode="markers",
        marker=dict(symbol="square", size=10, color="rgba(255, 0, 0, 0.5)"),
        name="El Niño",
    )
)

```

```

        )
    )
fig.add_trace(
    go.Scatter(
        x=[None],
        y=[None],
        mode="markers",
        marker=dict(symbol="square", size=10, color="rgba(0, 0, 255, 0.5)'),
        name="La Niña",
    )
)
fig.add_trace(
    go.Scatter(
        x=[None],
        y=[None],
        mode="markers",
        marker=dict(symbol="square", size=10, color="rgba(200, 200, 200, 0.5)'),
        name="Neutral",
    )
)
fig.update_layout(
    title="Fases del Fenómeno ENSO y Anomalía ONI (Histórico)",
    yaxis_title="Anomalía ONI (°C)",
    xaxis_title="Fecha",
    height=500,
    hovermode="x unified",
    yaxis_range=[y_min, y_max],
    barmode="overlay",
    legend=dict(orientation="h", yanchor="bottom", y=1.02, xanchor="right", x=1),
)
return fig

```

1. FUNCIONES AUXILIARES DE PARSEO Y DATOS

```

def parse_spanish_date(x):
    if isinstance(x, str):
        x = x.lower().strip()
    trans = {
        "ene": "Jan",
        "feb": "Feb",
        "mar": "Mar",
        "abr": "Apr",

```

```

    "may": "May",
    "jun": "Jun",
    "jul": "Jul",
    "ago": "Aug",
    "sep": "Sep",
    "oct": "Oct",
    "nov": "Nov",
    "dic": "Dec",
}
for es, en in trans.items():
    if es in x:
        x = x.replace(es, en)
        break
try:
    return pd.to_datetime(x, format="%b-%y")
except:
    return pd.to_datetime(x, errors="coerce")
return pd.to_datetime(x, errors="coerce")

```

```

# 2. FUNCIONES PRINCIPALES DE VISUALIZACIÓN
# -----
# -----
# NUEVA FUNCIÓN: CONEXIÓN CON IRI (COLUMBIA UNIVERSITY)
# -----
try:
    from modules.iri_api import (fetch_iri_data, process_iri_plume,
                                process_iri_probabilities)
except ImportError:
    # Evita que la app se rompa si el archivo iri_api.py aún no se ha creado o cargado
    fetch_iri_data = None

# NUEVA FUNCIÓN: VISUALIZACIÓN DEL PRONÓSTICO OFICIAL IRI/CPC
# Columbia University
# -----
def display_iri_forecast_tab():
    st.subheader("🌐 Pronóstico Oficial ENSO (IRI - Columbia University)")

    # --- SECCIÓN EDUCATIVA (NUEVA CAJA DESPLEGABLE) ---
    with st.expander(
        "📘 Conceptos, Metodología e Importancia (Pronóstico ENSO - IRI)",
        expanded=False,
    ):
        st.markdown(

```

.....

Este módulo se conecta directamente a los servidores del **International Research Institute for Climate and Society (IRI)**.

Los datos se actualizan mensualmente (aprox. el día 19) y representan el estándar global para la predicción de El Niño/La Niña.

1. Definición

El **Pronóstico ENSO del IRI** (International Research Institute for Climate and Society) es el estándar global para monitorear el fenómeno El Niño-Oscilación del Sur. Recopila y armoniza las predicciones de más de 20 instituciones científicas de todo el mundo (NASA, NOAA, JMA, ECMWF, etc.).

2. Metodología

El pronóstico se basa en la región **Niño 3.4** (Pacífico Ecuatorial Central) y combina dos tipos de modelos:

* ** Modelos Dinámicos:** Usan supercomputadoras para simular las leyes físicas del océano y la atmósfera (ej. NCEP CFSv2, ECMWF). Son mejores para predicciones a largo plazo.

* ** Modelos Estadísticos:** Usan patrones históricos y matemáticas para proyectar el futuro. Son eficientes para el corto plazo.

3. Interpretación de los Gráficos

* ** La "Pluma" (Spaghetti Plot):** Muestra la incertidumbre. Cada línea es una opinión científica distinta.

* **Línea Negra Gruesa:** Es el promedio de todos los modelos (Consenso). Suele ser el predictor más confiable.

* **Umbrales:** Si el promedio supera **+0.5°C**, se prevé **El Niño**. Si baja de **-0.5°C**, se prevé **La Niña**.

* ** Probabilidades:** Muestra el porcentaje de certeza de que ocurra cada evento (Niño, Niña o Neutral) en cada trimestre venidero.

4. Utilidad en Colombia

El ENSO es el principal modulador del clima en Colombia:

* **El Niño:** Generalmente asociado a disminución de lluvias, sequías y altas temperaturas.

* **La Niña:** Generalmente asociada a aumento de lluvias, inundaciones y deslizamientos.

5. Fuente Oficial

Datos provistos directamente vía FTP seguro por el [IRI / Columbia University Climate School](<https://iri.columbia.edu/our-expertise/climate/forecasts/enso/current/>).

.....

)

```
# 1. Verificar credenciales y módulo
```

```
if fetch_iri_data is None:
```

```
    st.error(
```

```
        "⚠ Falta el módulo 'modules/iri_api.py' o hubo un error al importarlo."
```

```
)
```

```
return
```

```

# 2. Cargar Datos (Pluma y Probabilidades)
with st.spinner("Conectando con FTP seguro de IRI (Columbia University)..."):
    json_plume = fetch_iri_data("enso_plumes.json")
    json_probs = fetch_iri_data("enso_cpc_prob.json")

if not json_plume or not json_probs:
    st.warning(
        "No se pudieron cargar los datos. Verifica tu conexión a internet o las credenciales en
'.streamlit/secrets.toml'."
    )
    return

# 3. Procesar Datos
plume_data = process_iri_plume(json_plume)
df_probs = process_iri_probabilities(json_probs)

if not plume_data or df_probs.empty:
    st.error("Datos recibidos pero con formato inesperado o vacíos.")
    return

# --- VISUALIZACIÓN ---
tab_plume, tab_prob = st.tabs([
    "📈 Pluma de Modelos (SST)", "📊 Probabilidades (%)"
])

# GRÁFICO 1: PLUMA DE MODELOS (Plume Plot)
with tab_plume:
    # Título descriptivo con fecha
    forecast_date_str = f"{plume_data['month_idx']+1}/{plume_data['year']}"
    st.markdown(f"**Emisión del Pronóstico:** {forecast_date_str}")

    fig = go.Figure()
    seasons = plume_data["seasons"]

    # Umbrales
    fig.add_shape(
        type="line",
        x0=seasons[0],
        x1=seasons[-1],
        y0=0.5,
        y1=0.5,
        line=dict(color="red", width=1, dash="dash"),
        name="Umbral Niño",
    )
    fig.add_shape(

```

```

        type="line",
        x0=seasons[0],
        x1=seasons[-1],
        y0=-0.5,
        y1=-0.5,
        line=dict(color="blue", width=1, dash="dash"),
        name="Umbral Niña",
    )

all_values = []
for model in plume_data["models"]:
    color = (
        "rgba(100, 200, 100, 0.6)"
        if model["type"] == "Statistical"
        else "rgba(150, 150, 150, 0.6)"
    )

    # Recortar valores
    y_vals = model["values"][: len(seasons)]

    fig.add_trace(
        go.Scatter(
            x=seasons,
            y=y_vals,
            mode="lines",
            name=model["name"],
            line=dict(color=color, width=1),
            showlegend=True, # <--- CAMBIO: Leyenda visible para cada modelo
            hoverinfo="name+y",
        )
    )
    all_values.append(y_vals)

# --- CORRECCIÓN MATEMÁTICA Y PROMEDIO ---
try:
    # 1. Encontrar longitud máxima
    max_len = max(len(v) for v in all_values) if all_values else 0

    # 2. Limpiar matriz: Convertir 'None' a 'np.nan' y llenar huecos
    clean_matrix = []
    for v in all_values:
        # Convertimos None -> np.nan (float)
        row_clean = [val if val is not None else np.nan for val in v]
        # Rellenamos si falta longitud
        padding = [np.nan] * (max_len - len(row_clean))
        clean_matrix.append(row_clean + padding)

```

```

# 3. Crear array float explícito (evita el error de tipos mixtos)
arr = np.array(clean_matrix, dtype=float)

# 4. Calcular promedio ignorando NaNs
avg_vals = np.nanmean(arr, axis=0)[: len(seasons)]

fig.add_trace(
    go.Scatter(
        x=seasons,
        y=avg_vals,
        mode="lines+markers",
        name="PROMEDIO MULTIMODELO",
        line=dict(color="black", width=4),
        marker=dict(size=8),
        showlegend=True,
    )
)
except Exception as e:
    st.warning(f"No se pudo calcular la línea de promedio: {e}")

fig.update_layout(
    title=f"Predicción Anomalía SST - Niño 3.4 (Emisión: {forecast_date_str})", # <--- CAMBIO: Fecha en
    título
    yaxis_title="Anomalía de Temperatura (°C)",
    xaxis_title="Trimestre",
    height=600,
    hovermode="x unified",
    showlegend=True,
    legend=dict( # <--- CAMBIO: Configuración de leyenda a la derecha
        yanchor="top",
        y=1,
        xanchor="left",
        x=1.02,
        font=dict(size=10),
        traceorder="normal",
    ),
    margin=dict(r=150), # Margen derecho para que quupa la leyenda
)
st.plotly_chart(fig)
st.caption(
    "🔴 Umbral El Niño (+0.5°C) | 🔵 Umbral La Niña (-0.5°C). Líneas grises: Modelos Dinámicos. Líneas
verdes: Estadísticos."
)

```

GRÁFICO 2: PROBABILIDADES

```

with tab_prob:
    st.markdown(
        f"##### Probabilidad Oficial (Emisión: {plume_data['month_idx']+1}/{plume_data['year']}))"
    )
    colors = {"La Niña": "#00008B", "Neutral": "#808080", "El Niño": "#DC143C"}

    fig_bar = go.Figure()
    for evento in ["La Niña", "Neutral", "El Niño"]:
        fig_bar.add_trace(
            go.Bar(
                x=df_probs["Trimestre"],
                y=df_probs[evento],
                name=evento,
                marker_color=colors[evento],
                text=df_probs[evento].apply(lambda x: f"{x}%"),
                textposition="auto",
            )
        )

    fig_bar.update_layout(
        barmode="stack",
        title=f"Consenso Probabilístico CPC/IRI ({plume_data['year']}",
        yaxis_title="Probabilidad (%)",
        height=500,
        yaxis=dict(range=[0, 100]),
        legend=dict(
            orientation="h", yanchor="bottom", y=1.02, xanchor="right", x=1
        ),
    )
    st.plotly_chart(fig_bar, use_container_width=True)
    st.dataframe(df_probs.set_index("Trimestre"))

# CENTRO DE MONITOREO Y TIEMPO REAL (DASHBOARD)
# -----
def display_realtime_dashboard(df_long, gdf_stations, gdf_filtered, **kwargs):
    st.header("⚠️ Centro de Monitoreo y Tiempo Real")

    tab_fc, tab_sat, tab_alert = st.tabs([
        "🌙 Pronóstico Semanal", "📡 Satélite en Vivo", "📊 Alertas Históricas"
    ])

    # --- SUB-PESTAÑA 1: PRONÓSTICO COMPLETO ---
    with tab_fc:
        if gdf_filtered is None or gdf_filtered.empty:

```

```

st.warning("⚠ Seleccione al menos una estación en el menú lateral.")
return

# Selector de Estación
estaciones_list = sorted(gdf_filtered[Config.STATION_NAME_COL].unique())
sel_st = st.selectbox("Estación para Pronóstico:", estaciones_list)

if sel_st:
    st_dat = gdf_filtered[gdf_filtered[Config.STATION_NAME_COL] == sel_st].iloc[
        0
    ]

    # Intentar obtener pronóstico
    df_forecast = pd.DataFrame()
    try:
        # Importamos aquí para evitar ciclos si no se usa
        from modules.openmeteo_api import get_weather_forecast_detailed

        with st.spinner("Consultando modelos meteorológicos globales..."):
            lat = (
                st_dat["latitude"]
                if "latitude" in st_dat
                else st_dat.geometry.y
            )
            lon = (
                st_dat["longitude"]
                if "longitude" in st_dat
                else st_dat.geometry.x
            )
            df_forecast = get_weather_forecast_detailed(lat, lon)
    except Exception as e:
        st.error(f"Error consultando pronóstico: {e}")

    if not df_forecast.empty:
        # 1. TARJETAS DE RESUMEN (HOY)
        td = df_forecast.iloc[0] # Datos de hoy/ahora
        c1, c2, c3, c4 = st.columns(4)
        c1.metric(
            "🌡️ T. Máx/Mín",
            f"{td.get('T. Máx (°C)', '--')}/{td.get('T. Mín (°C)', '--)}°C"
        )
        c2.metric("🌧️ Lluvia Hoy", f"{td.get('Ppt. (mm)', 0):.1f} mm")
        c3.metric("💨 Viento Máx", f"{td.get('Viento Máx (km/h)', 0):.1f} km/h")
        c4.metric(
            "☀️ Radiación",
            f"{td.get('Radiación SW (MJ/m²)', 0):.1f} MJ/m²"
        )

```

```

)
# 2. GRÁFICO PRINCIPAL (Climograma)
st.markdown("#### 🌡️ Temperatura y Precipitación (7 Días)")

fig = make_subplots(specs=[[{"secondary_y": True}]])

# Lluvia (Barras - Eje Derecha)
fig.add_trace(
    go.Bar(
        x=df_forecast["Fecha"],
        y=df_forecast["Ppt. (mm)"],
        name="Lluvia (mm)",
        marker_color="#4682B4",
        opacity=0.6,
    ),
    secondary_y=True,
)

# Temperatura (Líneas - Eje Izquierda)
fig.add_trace(
    go.Scatter(
        x=df_forecast["Fecha"],
        y=df_forecast["T. Máx (°C)"],
        name="T. Máx",
        line=dict(color="#FF4500", width=2),
    ),
    secondary_y=False,
)

fig.add_trace(
    go.Scatter(
        x=df_forecast["Fecha"],
        y=df_forecast["T. Mín (°C)"],
        name="T. Mín",
        line=dict(color="#1E90FF", width=2),
        fill="tonexty", # Relleno entre líneas
    ),
    secondary_y=False,
)

# Layout Ajustado para evitar cortes
fig.update_layout(
    height=450,
    hovermode="x unified",
    legend=dict(

```

```

        orientation="h", # Horizontal
        yanchor="bottom",
        y=1.02, # Arriba del gráfico
        xanchor="right",
        x=1,
    ),
    margin=dict(l=50, r=50, t=50, b=50),
)

# Ejes
fig.update_yaxes(
    title_text="Temperatura (°C)", secondary_y=False, showgrid=True
)
fig.update_yaxes(
    title_text="Precipitación (mm)",
    secondary_y=True,
    showgrid=False,
    range=[0, max(df_forecast["Ppt. (mm)"].max() * 3, 10)],
)

st.plotly_chart(fig)

# 3. GRÁFICOS SECUNDARIOS
st.markdown("#### 🌬️ Condiciones Atmosféricas")
col_g1, col_g2 = st.columns(2)

with col_g1:
    # Humedad y Presión
    fig_atm = make_subplots(specs=[[{"secondary_y": True}]])
    fig_atm.add_trace(
        go.Scatter(
            x=df_forecast["Fecha"],
            y=df_forecast["HR Media (%)"],
            name="Humedad",
            line=dict(color="teal"),
        ),
        secondary_y=False,
    )
    fig_atm.add_trace(
        go.Scatter(
            x=df_forecast["Fecha"],
            y=df_forecast.get(
                "Presión (hPa)", [1013] * len(df_forecast)
            ),
            name="Presión",
            line=dict(color="purple", dash="dot"),
        ),
        secondary_y=True,
    )

```

```

        ),
        secondary_y=True,
    )

fig_atm.update_layout(
    title="Humedad y Presión",
    height=350,
    legend=dict(orientation="h", y=-0.2),
)
fig_atm.update_yaxes(title_text="HR (%)", secondary_y=False)
fig_atm.update_yaxes(
    title_text="hPa", secondary_y=True, showgrid=False
)
st.plotly_chart(fig_atm, use_container_width=True)

with col_g2:
    # Energía y Agua (Radiación + ET0)
    fig_nrg = make_subplots(specs=[[{"secondary_y": True}]])
    fig_nrg.add_trace(
        go.Bar(
            x=df_forecast["Fecha"],
            y=df_forecast["Radiación SW (MJ/m2)"],
            name="Radiación",
            marker_color="gold",
        ),
        secondary_y=False,
    )
    fig_nrg.add_trace(
        go.Scatter(
            x=df_forecast["Fecha"],
            y=df_forecast["ET0 (mm)"],
            name="Evapotranspiración",
            line=dict(color="green"),
        ),
        secondary_y=True,
    )

    fig_nrg.update_layout(
        title="Energía y Ciclo del Agua",
        height=350,
        legend=dict(orientation="h", y=-0.2),
    )
    fig_nrg.update_yaxes(title_text="MJ/m2", secondary_y=False)
    fig_nrg.update_yaxes(
        title_text="mm", secondary_y=True, showgrid=False
    )

```

```

st.plotly_chart(fig_nrg, use_container_width=True)

# 4. TABLA DETALLADA
with st.expander("Ver Tabla de Datos Completa"):
    st.dataframe(df_forecast)
else:
    st.info(
        "No se pudo obtener el pronóstico para esta ubicación. Intente más tarde."
    )

# --- SUB-PESTAÑA 2: SATÉLITE (ESTABILIZADA) ---
with tab_sat:
    st.subheader("Observación Satelital")

    # Controles
    c_sat1, c_sat2 = st.columns([1, 3])
    with c_sat1:
        sat_mode = st.radio(
            "Modo:",
            ["Animación (Visible)", "Mapa Interactivo (Lluvia/Nubes)"],
            index=1,
        )
        show_stations_sat = st.checkbox("Mostrar Estaciones", value=True)

    with c_sat2:
        if sat_mode == "Animación (Visible)":
            # GIF Oficial NOAA (GeoColor) - Muy estable
            st.image(
                "https://cdn.star.nesdis.noaa.gov/GOES16/ABI/GIFS/GOES16-ABI-GEOCOLOR-1000x1000.gif",
                caption="GOES-16 GeoColor (Tiempo Real)",
                use_column_width=True,
            )
        else:
            # Mapa Interactivo
            try:
                # Usamos OpenStreetMap por estabilidad, centrado en la zona de interés
                m = folium.Map(
                    location=[6.2, -75.5], zoom_start=7, tiles="OpenStreetMap"
                )

                # Capa de Radar de Lluvia (RainViewer - Cobertura Global y Rápida)
                folium.TileLayer(
                    tiles="https://tile.rainviewer.com/nowcast/now/256/{z}/{x}/{y}/2/1_1.png",
                    attr="RainViewer",
                    name="Radar de Lluvia (Tiempo Real)",
                    overlay=True,
                )
            except Exception as e:
                st.error(f"Error al cargar el mapa: {e}")

```

```

        opacity=0.7,
    ).add_to(m)

    # Capa de Nubes (Infrarrojo) - Opcional, si RainViewer falla
    folium.TileLayer(
        tiles="https://mesonet.agron.iastate.edu/cache/tile.py/1.0.0/goes-east-ir-4km-
900913/{z}/{x}/{y}.png",
        attr="IEM/NOAA",
        name="Nubes Infrarrojo",
        overlay=True,
        opacity=0.5,
        show=False, # Oculta por defecto para no saturar
    ).add_to(m)

    # Mostrar Estaciones (Lo que pediste recuperar)
    if (
        show_stations_sat
        and gdf_filtered is not None
        and not gdf_filtered.empty
    ):
        for _, row in gdf_filtered.dropna(
            subset=["latitude", "longitude"]
        ).iterrows():
            folium.CircleMarker(
                location=[row["latitude"], row["longitude"]],
                radius=3,
                color="red",
                fill=True,
                fill_opacity=1,
                tooltip=row[Config.STATION_NAME_COL],
            ).add_to(m)

    # --- GEOLOCALIZADOR NATIVO DE FOLIUM ---
    LocateControl(auto_start=False).add_to(
        m
    ) # <--- AQUÍ ESTÁ EL BOTÓN DE GPS

    folium.LayerControl().add_to(m)
    st_folium(m, height=600, width="100%")
    st.caption(
        " ● Radar: RainViewer. ⛅ Nubes: GOES-16. | 🌈 Usa el botón de GPS en el mapa para
ubicarte."
    )
except Exception as e:
    st.error(f"Error cargando el mapa satelital: {e}")

```

```

# --- SUB-PESTAÑA 3: ALERTAS ---
with tab_alert:
    if df_long is not None:
        umb = st.slider("Umbral (mm):", 0, 1000, 300)
        alts = df_long[df_long[Config.PRECIPITATION_COL] > umb]
        st.metric("Eventos Extremos", len(alts))
    if not alts.empty:
        st.dataframe(
            alts.sort_values(Config.PRECIPITATION_COL, ascending=False).head(
                100
            ),
        )
    )

def display_spatial_distribution_tab(
    user_loc, interpolacion, df_long, df_complete, gdf_stations, gdf_filtered,
    gdf_municipios, gdf_subcuenca, gdf_predios, df_enso, stations_for_analysis,
    df_anual_melted, df_monthly_filtered, analysis_mode, selected_regions,
    selected_municipios, selected_months, year_range, start_date, end_date, **kwargs
):
    # Importación necesaria para evitar el bloqueo del mapa
    from folium.plugins import MarkerCluster

    # Inicializar estado
    if "selected_point" not in st.session_state:
        st.session_state.selected_point = None

    st.markdown("### 🌎 Distribución Espacial y Análisis Puntual")

    tab_mapa, tab_avail, tab_series = st.tabs(["📍 Mapa Interactivo", "📊 Disponibilidad", "📅 Series Anuales"])

# --- PESTAÑA 1: MAPA INTERACTIVO ---
with tab_mapa:
    # 1. Configuración de Vista
    c_zoom, c_manual = st.columns([2, 1])
    location_center = [6.5, -75.5] # Default Antioquia
    zoom_level = 8

    with c_zoom:
        escala = st.radio("📍 Zoom Rápido:", ["Colombia", "Antioquia", "Región Actual"], horizontal=True)
        if escala == "Colombia": location_center, zoom_level = [4.57, -74.29], 6
        elif escala == "Antioquia": location_center, zoom_level = [7.0, -75.5], 8
        elif escala == "Región Actual" and not gdf_filtered.empty:
            try:

```

```

c = gdf_filtered.dissolve().centroid
location_center, zoom_level = [c.geometry[0].y, c.geometry[0].x], 9
except: pass

with c_manual:

    with st.expander("📍 Ingresar Coordenadas", expanded=False):
        lat_in = st.number_input("Latitud", value=float(location_center[0]), format=".5f")
        lon_in = st.number_input("Longitud", value=float(location_center[1]), format=".5f")
        if st.button("Analizar Coordenadas"):
            st.session_state.selected_point = {"lat": lat_in, "lon": lon_in}

# 2. CREACIÓN DEL MAPA
m = folium.Map(location=location_center, zoom_start=zoom_level, control_scale=True)

# Capas y Fondos
folium.TileLayer('cartodbpositron', name='Mapa Claro (Default)').add_to(m)
folium.TileLayer('openstreetmap', name='Callejero (OSM)').add_to(m)
try:
    folium.TileLayer(
        tiles='https://server.arcgisonline.com/ArcGIS/rest/services/World_Imagery/MapServer/tile/{z}/{y}/{x}',
        attr='Esri', name='Satélite (Esri)'
    ).add_to(m)
except: pass

# Plugins
plugins.LocateControl(auto_start=False, position="topleft").add_to(m)
plugins.Fullscreen(position='topright').add_to(m)
plugins.Geocoder(position='topright').add_to(m)

# Capas Vectoriales (Optimizadas)
if gdf_municipios is not None and not gdf_municipios.empty:
    folium.GeoJson(
        gdf_municipios,
        name="Municipios",
        style_function=lambda x: {'fillColor': '#ffff00', 'color': 'gray', 'weight': 1, 'fillOpacity': 0.1},
        tooltip="Municipio"
    ).add_to(m)

if gdf_subcuenca is not None and not gdf_subcuenca.empty:
    folium.GeoJson(
        gdf_subcuenca,
        name="Subcuenca",
        style_function=lambda x: {'color': 'blue', 'weight': 2, 'fillOpacity': 0},
        tooltip="Subcuenca"
    ).add_to(m)

```

```

if gdf_predios is not None and not gdf_predios.empty:
    folium.GeoJson(
        gdf_predios,
        name="Predios",
        style_function=lambda x: {'color': 'red', 'weight': 2, 'fillOpacity': 0.2},
        tooltip="Predio"
    ).add_to(m)

# -----
# SOLUCIÓN AL BLOQUEO: MARKER CLUSTER
# Agrupamos los marcadores para no saturar el navegador
# -----
marker_cluster = MarkerCluster(name="Estaciones (Agrupadas)").add_to(m)

# 1. PRE-CÁLCULO DE ESTADÍSTICAS
stats_cache = {}
if not df_long.empty:
    try:
        # Detectar columna de código
        col_cod_long = next((c for c in ['Codigo', 'CODIGO', 'id_estacion', 'station_code'] if c in df_long.columns), df_long.columns[0])

        # Agrupamos por estación (Optimizado)
        grp = df_long.groupby(col_cod_long)[Config.PRECIPITATION_COL]
        medias = grp.mean()
        conteos = grp.count()

        for cod_stat, val_media in medias.items():
            anios = conteos[cod_stat] / 12
            stats_cache[str(cod_stat)] = {
                'media': f'{val_media:.1f} mm/mes',
                'hist': f'{anios:.1f} años'
            }
    except Exception as e:
        print(f"Nota: Estadísticas básicas no calculadas: {e}")

# 2. FUNCIÓN DE BÚSQUEDA FLEXIBLE
def get_fuzzy_col(row, aliases, default="N/A"):
    row_cols_lower = {c.lower(): c for c in row.index}
    for alias in aliases:
        for col_lower, col_real in row_cols_lower.items():
            if alias in col_lower:
                val = row[col_real]
                return str(val) if pd.notna(val) else default
    return default

```

```

# BUCLE DE ESTACIONES (Ahora añadiendo al Cluster)
if not gdf_filtered.empty:
    for _, row in gdf_filtered.iterrows():
        try:
            # Datos básicos
            nom = str(row[Config.STATION_NAME_COL])
            mun = str(row.get(Config.MUNICIPALITY_COL, 'Desconocido'))
            alt = str(row.get(Config.ALTITUDE_COL, 0))

            # ID y Subcuenca
            cod = get_fuzzy_col(row, ['codigo', 'id', 'serial', 'cod'], 'Sin ID')
            cue = get_fuzzy_col(row, ['subcuenca', 'cuenca', 'szh', 'vertiente', 'micro', 'zona'], 'N/A')

            # Estadísticas desde cache
            stat_data = stats_cache.get(cod, {'media': 'N/A', 'hist': 'N/A'})
            if stat_data['media'] == 'N/A':
                try: stat_data = stats_cache.get(str(int(float(cod))), {'media': 'N/A', 'hist': 'N/A'})
                except: pass

            precip = stat_data['media']
            anios = stat_data['hist']

            # HTML Popup
            html_content = f"""
<div style="font-family: Arial, sans-serif; width: 260px; font-size: 12px;">
    <h4 style="margin: 0; color: #2c3e50; border-bottom: 2px solid #3498db; padding-bottom: 4px;">{nom}</h4>
    <div style="margin-top: 5px; color: #7f8c8d; font-size: 11px;"><b>ID:</b> {cod}</div>
    <br>
    <table style="width: 100%; border-collapse: collapse;">
        <tr style="border-bottom: 1px solid #eee;"><td><b>📍 Municipio:</b></td><td style="text-align:right;">{mun}</td></tr>
        <tr style="border-bottom: 1px solid #eee;"><td><b>_altitud:</b></td><td style="text-align:right;">{alt}</td></tr>
        <tr style="border-bottom: 1px solid #eee;"><td><b>💧 Subcuenca:</b></td><td style="text-align:right;">{cue}</td></tr>
        <tr style="border-bottom: 1px solid #eee;"><td><b>📊 P. Media:</b></td><td style="text-align:right;">{precip}</td></tr>
        <tr><td><b>📅 Histórico:</b></td><td style="text-align:right;">{anios}</td></tr>
    </table>
    <div style="margin-top: 10px; text-align: center; background-color: #f0f8ff; padding: 5px; border-radius: 4px;">
        <i style="color: #2980b9; font-size: 11px;">👉 Clic para ver gráficas abajo</i>
    </div>

```

```

</div>
"""

iframe = folium.IFrame(html_content, width=280, height=240)
popup = folium.Popup(iframe, max_width=280)

# AÑADIR AL CLUSTER EN VEZ DE AL MAPA DIRECTO
folium.Marker(
    [row.geometry.y, row.geometry.x],
    tooltip=f"{nom}",
    popup=popup,
    icon=folium.Icon(color="blue", icon="cloud", prefix='fa')
).add_to(marker_cluster)

except Exception:
    continue

# Control de capas
folium.LayerControl().add_to(m)

st.markdown("👉 **Haz clic en un marcador para ver detalles o en cualquier punto del mapa para ver el pronóstico.**")

# Renderizar mapa
map_output = st_folium(m, width=None, height=600, returned_objects=["last_clicked"])

# Lógica de Clic
if map_output and map_output.get("last_clicked"):
    coords = map_output["last_clicked"]
    st.session_state.selected_point = {"lat": coords["lat"], "lng": coords["lng"]}

# 3. DASHBOARD DE PRONÓSTICO
if st.session_state.selected_point:
    lat = float(st.session_state.selected_point["lat"])
    lng = float(st.session_state.selected_point["lng"])

    st.markdown("---")
    st.subheader(f"📍 Análisis Puntual: {lat:.4f}, {lng:.4f}")

    # Verificación segura de la función externa
    if 'get_weather_forecast_detailed' in globals() or
    callable(kwags.get('get_weather_forecast_detailed')):
        func_forecast = kwags.get('get_weather_forecast_detailed') or
        globals().get('get_weather_forecast_detailed')

```

```

with st.spinner("Conectando con satélites meteorológicos..."):
    try:
        fc = func_forecast(lat, lng)
    except:
        fc = None

    if fc is not None and not fc.empty:
        # A. MÉTRICAS
        hoy = fc.iloc[0]
        m1, m2, m3, m4, m5 = st.columns(5)
        m1.metric("🌡️ Temp", f"{{(hoy['T. Máx (°C)']+hoy['T. Mín (°C)])/2:.1f}°C}")
        m2.metric("🌧️ Lluvia", f"{{hoy['Ppt. (mm)']}} mm")
        m3.metric("💧 Humedad", f"{{hoy['HR Media (%)']}}%")
        m4.metric("💨 Viento", f"{{hoy['Viento MÁX (km/h)']}} km/h")
        m5.metric("☀️ Radiación", f"{{hoy['Radiación SW (MJ/m²)']}} MJ/m²")

        # B. GRÁFICOS
        with st.expander("📈 Ver Gráficos Detallados (7 Días)", expanded=True):
            # 1. Temperatura y Lluvia
            fig = make_subplots(specs=[[{"secondary_y": True}]])
            fig.add_trace(go.Bar(x=fc['Fecha'], y=fc['Ppt. (mm)'], name="Lluvia", marker_color='blue',
            opacity=0.5), secondary_y=True)
            fig.add_trace(go.Scatter(x=fc['Fecha'], y=fc['T. Máx (°C)'], name="Máx",
            line=dict(color='red')), secondary_y=False)
            fig.add_trace(go.Scatter(x=fc['Fecha'], y=fc['T. Mín (°C)'], name="Mín", line=dict(color='cyan'),
            fill='tonexty'), secondary_y=False)
            fig.update_layout(title="Temperatura y Precipitación", height=350, hovermode="x unified")
            st.plotly_chart(fig, use_container_width=True)

            # 2. Atmósfera y Energía
            c_g1, c_g2 = st.columns(2)

            with c_g1: # Atmósfera
                fig_atm = make_subplots(specs=[[{"secondary_y": True}]])
                fig_atm.add_trace(go.Scatter(x=fc["Fecha"], y=fc["HR Media (%)"], name="Humedad %",
                line=dict(color="teal")), secondary_y=False)
                fig_atm.add_trace(go.Scatter(x=fc["Fecha"], y=fc["Presión (hPa)"], name="Presión",
                line=dict(color="purple", dash="dot")), secondary_y=True)
                fig_atm.update_layout(title="Atmósfera", height=300, hovermode="x unified")
                st.plotly_chart(fig_atm, use_container_width=True)

            with c_g2: # Energía
                fig_nrg = make_subplots(specs=[[{"secondary_y": True}]])
                fig_nrg.add_trace(go.Bar(x=fc["Fecha"], y=fc["Radiación SW (MJ/m²)"], name="Radiación",
                marker_color="orange"), secondary_y=False)

```

```

        fig_nrg.add_trace(go.Scatter(x=fc["Fecha"], y=fc["ET0 (mm)"], name="ET0",  

line=dict(color="green")), secondary_y=True)  

        fig_nrg.update_layout(title="Energía", height=300, hovermode="x unified")  

st.plotly_chart(fig_nrg, use_container_width=True)

# C. TABLA  

with st.expander(" Ver Tabla de Datos", expanded=False):  

    st.dataframe(fc)
else:  

    st.warning("⚠️ No se pudo obtener el pronóstico.")  

else:  

    st.info("El módulo de pronóstico no está vinculado en este contexto.")

# ======  

# PESTAÑA 2: DISPONIBILIDAD  

# ======  

with tab_avail:  

    c_title, c_sel = st.columns([2, 1])  

    with c_title:  

        st.markdown("#### 📊 Inventario y Continuidad de Datos")  

    with c_sel:  

        data_view_mode = st.radio(  

            "Vista de Datos:",  

            ["Observados (Con huecos)", "Interpolados (Simulación)"],  

            horizontal=True,  

            label_visibility="collapsed",
        )
    if df_long is not None and not df_long.empty:  

        df_to_plot = df_long.copy()

    if data_view_mode == "Interpolados (Simulación)":  

        if interpolacion == "No":  

            with st.spinner("Simulando relleno de datos..."):  

                try:  

                    from modules.data_processor import complete_series  

                    df_to_plot = complete_series(df_to_plot)
                except ImportError:  

                    st.warning("Módulo de interpolación no disponible.")
        else:  

            st.info("Los datos ya están interpolados globalmente.")

    avail = (
        df_to_plot[df_to_plot[Config.PRECIPITATION_COL].notna()]
        .groupby([Config.STATION_NAME_COL, Config.YEAR_COL])[Config.PRECIPITATION_COL]
)

```

```

    .count()
    .reset_index()
)
avail.rename(columns={Config.PRECIPITATION_COL: "Meses con Datos"}, inplace=True)

all_years = list(range(int(avail[Config.YEAR_COL].min()), int(avail[Config.YEAR_COL].max()) + 1))
all_stations = avail[Config.STATION_NAME_COL].unique()

full_idx = pd.MultiIndex.from_product([all_stations, all_years], names=[Config.STATION_NAME_COL,
Config.YEAR_COL])
avail_full = avail.set_index([Config.STATION_NAME_COL, Config.YEAR_COL]).reindex(full_idx,
fill_value=0).reset_index()

title_chart = "Continuidad de Información"

# FIX: use_container_width deprecation fix
fig_avail = px.density_heatmap(
    avail_full,
    x=Config.YEAR_COL,
    y=Config.STATION_NAME_COL,
    z="Meses con Datos",
    nbinsx=len(all_years),
    nbinsy=len(all_stations),
    color_continuous_scale=[(0, "white"), (0.01, "#ffcccc"), (0.5, "#ffaa00"), (1.0, "#006400")],
    range_color=[0, 12],
    title=title_chart,
    height=max(400, len(all_stations) * 20),
)
fig_avail.update_layout(xaxis_title="Año", yaxis_title="Estación",
coloraxis_colorbar=dict(title="Meses"), xaxis=dict(dtick=1), yaxis=dict(dtick=1))
st.plotly_chart(fig_avail, use_container_width=True)

# Métricas
c1, c2, c3 = st.columns(3)
total_months = len(all_years) * 12
actual_months = avail["Meses con Datos"].sum()
completeness = (actual_months / (len(all_stations) * total_months)) * 100 if len(all_stations) > 0 else
0

c1.metric("Total Estaciones", len(all_stations))
c2.metric("Rango de Años", f"{min(all_years)} - {max(all_years)}")
c3.metric("Completitud Global", f"{completeness:.1f}%")

with st.expander("Ver Tabla de Disponibilidad", expanded=False):
    pivot_avail = avail_full.pivot(index=Config.STATION_NAME_COL, columns=Config.YEAR_COL,
values="Meses con Datos")

```

```

        st.dataframe(pivot_avail.style.background_gradient(cmap="Greens", vmin=0,
vmax=12).format("{:.0f}"))
    else:
        st.warning("No hay datos cargados.")

# --- PESTAÑA 3: SERIES ANUALES ---
with tab_series:
    st.markdown("##### 📈 Series Históricas")
    if df_anual_melted is not None and not df_anual_melted.empty:
        fig = px.line(
            df_anual_melted,
            x=Config.YEAR_COL,
            y=Config.PRECIPITATION_COL,
            color=Config.STATION_NAME_COL,
            title="Precipitación Anual por Estación"
        )
        st.plotly_chart(fig, use_container_width=True)

    with st.expander("Ver Datos en Tabla"):
        pivot_anual = df_anual_melted.pivot(
            index=Config.YEAR_COL,
            columns=Config.STATION_NAME_COL,
            values=Config.PRECIPITATION_COL
        )
        st.dataframe(pivot_anual)
    else:
        st.warning("No hay datos suficientes para graficar.")

def display_graphs_tab(
    df_monthly_filtered, df_anual_melted, stations_for_analysis, **kwargs
):
    import geopandas as gpd
    import os

    # Ruta base donde están los datos
    DATA_PATH = "data/"

    # 1. Cargar el Parquet (Datos de Lluvia)
    if st.session_state.get('df_long') is None:
        try:
            @st.cache_data
            def load_parquet():
                # Buscamos en la carpeta data
                return pd.read_parquet(os.path.join(DATA_PATH, "datos_precipitacion_largos.parquet"))
        except:
            st.error("Error al cargar los datos de precipitación largos. Verifique la conexión a la base de datos o la ubicación del archivo parquet.")
    else:
        df_long = df_monthly_filtered.merge(st.session_state['df_long'], on='station_id')

```

```

        st.session_state['df_long'] = load_parquet()
    except Exception as e:
        st.error(f"Error cargando Parquet desde data/: {e}")

# 2. Cargar CSV Estaciones (Metadatos)
if st.session_state.get('gdf_stations') is None:
    try:
        @st.cache_data
        def load_stations():
            return pd.read_csv(os.path.join(DATA_PATH, "mapaCVENSO.csv"), sep=";", encoding="latin-1")

        st.session_state['gdf_stations'] = load_stations()
    except Exception as e:
        st.error(f"Error cargando CSV desde data/: {e}")

# 3. Cargar GeoJSON (Cuenca)
if st.session_state.get('gdf_subcuenca') is None:
    try:
        @st.cache_data
        def load_cuenca():
            return gpd.read_file(os.path.join(DATA_PATH, "SubcuencaAinfluencia.geojson"))

        st.session_state['gdf_subcuenca'] = load_cuenca()
    except Exception as e:
        st.error(f"Error cargando GeoJSON desde data/: {e}")
# ----

st.subheader("📊 Análisis Gráfico Detallado")

# Validación de datos
if df_monthly_filtered is None or df_monthly_filtered.empty:
    st.warning(
        "No hay datos para mostrar. Seleccione estaciones y rango de fechas."
    )
    return

# --- PREPARACIÓN DE DATOS ---
df_monthly_filtered["Mes"] = df_monthly_filtered[Config.MONTH_COL]
df_monthly_filtered["Año"] = df_monthly_filtered[Config.YEAR_COL]
meses_orden = {
    1: "Ene",
    2: "Feb",
    3: "Mar",
    4: "Abr",
    5: "May",
}

```

```

    6: "Jun",
    7: "Jul",
    8: "Ago",
    9: "Sep",
    10: "Oct",
    11: "Nov",
    12: "Dic",
}
df_monthly_filtered["Nombre_Mes"] = df_monthly_filtered["Mes"].map(meses_orden)

# Definición de Pestañas (Originales + Nuevas)
tab_names = [
    "1. Serie Anual",
    "2. Ranking Multianual",
    "3. Serie Mensual",
    "4. Ciclo Anual (Promedio)",
    "5. Análisis Estacional Detallado",
    "6. Distribución de Frecuencias",
    "7. Comparativa Multiescalar"
]
tabs = st.tabs(tab_names)

# -----
# 1. SERIE ANUAL
# -----
with tabs[0]:
    st.markdown("##### Precipitación Total Anual")

    # 1. Crear Figura (Asignar a variable específica)
    fig_anual = px.line(
        df_anual_melted,
        x=Config.YEAR_COL,
        y=Config.PRECIPITATION_COL,
        color=Config.STATION_NAME_COL,
        markers=True,
        labels={Config.PRECIPITATION_COL: "Lluvia (mm)", Config.YEAR_COL: "Año"},
    )

    # 2. Mostrar
    st.plotly_chart(fig_anual, use_container_width=True)

    # 3. Guardar en Memoria para el Reporte PDF (CRÍTICO)
    st.session_state["report_fig_anual"] = fig_anual

    # Descarga
    st.download_button(

```

```

    "⬇️ Descargar Datos Anuales (CSV)",
    df_anual_melted.to_csv(index=False).encode("utf-8"),
    "serie_anual.csv",
    "text/csv",
)

# -----
# 2. RANKING MULTIANUAL
# -----
with tabs[1]:
    st.markdown("##### Ranking de Precipitación Media")

    avg_ppt = (
        df_anual_melted.groupby(Config.STATION_NAME_COL)[Config.PRECIPITATION_COL]
        .mean()
        .reset_index()
    )
    col_val = "Precipitación Media (mm)"
    avg_ppt.rename(columns={Config.PRECIPITATION_COL: col_val}, inplace=True)

    c_sort, _ = st.columns([1, 2])
    with c_sort:
        sort_opt = st.radio(
            "Ordenar:",
            ["Mayor a Menor", "Menor a Mayor", "Alfabético"],
            horizontal=True,
            label_visibility="collapsed",
        )

    if sort_opt == "Mayor a Menor":
        avg_ppt = avg_ppt.sort_values(col_val, ascending=False)
    elif sort_opt == "Menor a Mayor":
        avg_ppt = avg_ppt.sort_values(col_val, ascending=True)
    else:
        avg_ppt = avg_ppt.sort_values(Config.STATION_NAME_COL)

    fig_rank = px.bar(
        avg_ppt,
        x=Config.STATION_NAME_COL,
        y=col_val,
        color=col_val,
        color_continuous_scale=px.colors.sequential.Blues,
        text_auto=".0f",
    )
    st.plotly_chart(fig_rank, use_container_width=True)

```

```

# Guardar
st.session_state["report_fig_ranking"] = fig_rank

st.download_button(
    "📥 Descargar Ranking",
    avg_ppt.to_csv(index=False).encode("utf-8"),
    "ranking.csv",
    "text/csv",
)

# -----
# 3. SERIE MENSUAL
# -----
with tabs[2]:
    st.markdown("##### Serie Histórica Mensual")

    col_opts, col_chart = st.columns([1, 4])
    with col_opts:
        showRegional = st.checkbox("Ver Promedio Regional", value=False)
        showMarkers = st.checkbox("Mostrar Puntos", value=False)

    with col_chart:
        fig_mensual = px.line(
            df_monthly_filtered,
            x=Config.DATE_COL,
            y=Config.PRECIPITATION_COL,
            color=Config.STATION_NAME_COL,
            markers=showMarkers,
            title="Precipitación Mensual",
        )

    if showRegional:
        reg_mean = (
            df_monthly_filtered.groupby(Config.DATE_COL)[
                Config.PRECIPITATION_COL
            ]
            .mean()
            .reset_index()
        )
        fig_mensual.add_trace(
            go.Scatter(
                x=reg_mean[Config.DATE_COL],
                y=reg_mean[Config.PRECIPITATION_COL],
                mode="lines",
                name="PROMEDIO REGIONAL",
                line=dict(color="black", width=3, dash="dash"),
            )
        )

```

```

        )
    )

st.plotly_chart(fig_mensual, use_container_width=True)

# Guardar
st.session_state["report_fig_mensual"] = fig_mensual

st.download_button(
    "⬇️ Descargar Mensual",
    df_monthly_filtered.to_csv(index=False).encode("utf-8"),
    "mensual.csv",
    "text/csv",
)
# -----
# 4. CICLO ANUAL
# -----
with tabs[3]:
    st.markdown("##### Régimen de Lluvias (Ciclo Promedio)")
    ciclo = (
        df_monthly_filtered.groupby([Config.STATION_NAME_COL, Config.MONTH_COL])[
            Config.PRECIPITATION_COL
        ]
        .mean()
        .reset_index()
    )

    fig_ciclo = px.line(
        ciclo,
        x=Config.MONTH_COL,
        y=Config.PRECIPITATION_COL,
        color=Config.STATION_NAME_COL,
        markers=True,
        labels={
            Config.MONTH_COL: "Mes",
            Config.PRECIPITATION_COL: "Lluvia Promedio (mm)",
        },
    )
    fig_ciclo.update_xaxes(tickmode="linear", tick0=1, dtick=1)
    st.plotly_chart(fig_ciclo, use_container_width=True)

# Guardar
st.session_state["report_fig_ciclo"] = fig_ciclo

st.download_button(

```

```

    "⬇️ Descargar Ciclo",
    ciclo.to_csv(index=False).encode("utf-8"),
    "ciclo.csv",
    "text/csv",
)

# -----
# 5. DISTRIBUCIÓN (CÓDIGO CORREGIDO)
# -----
with tabs[4]:
    st.markdown("##### Análisis Estadístico de Distribución")

    c1, c2, c3 = st.columns(3)
    with c1:
        data_src = st.radio(
            "Datos:",
            ["Anual (Totales)", "Mensual (Detalle)"],
            horizontal=True,
            key="dist_src",
        )
    with c2:
        chart_typ = st.radio(
            "Gráfico:",
            ["Violín", "Histograma", "ECDF"],
            horizontal=True,
            key="dist_type",
        )
    with c3:
        sort_ord = st.selectbox(
            "Orden:",
            ["Alfabético", "Mayor a Menor"], key="dist_sort"
        )

    df_plot = df_anual_melted if "Anual" in data_src else df_monthly_filtered

    cat_orders = {}
    if sort_ord != "Alfabético":
        medians = df_plot.groupby(Config.STATION_NAME_COL)[
            Config.PRECIPITATION_COL
        ].median()
        order_list = medians.sort_values(ascending=False).index.tolist()
        cat_orders = {Config.STATION_NAME_COL: order_list}

    if "Violín" in chart_typ:
        fig_dist = px.violin(
            df_plot,
            x=Config.STATION_NAME_COL,

```

```

y=Config.PRECIPITATION_COL,
color=Config.STATION_NAME_COL,
box=True,
points="all",
category_orders=cat_orders,
)
fig_dist.update_layout(showlegend=False)
elif "Histograma" in chart_typ:
    fig_dist = px.histogram(
        df_plot,
        x=Config.PRECIPITATION_COL,
        color=Config.STATION_NAME_COL,
        marginal="box",
        barmode="overlay",
        opacity=0.7,
        category_orders=cat_orders,
    )
else:
    fig_dist = px.ecdf(
        df_plot, x=Config.PRECIPITATION_COL, color=Config.STATION_NAME_COL
    )

fig_dist.update_layout(
    height=600, title=f"Distribución {data_src} - {chart_typ}"
)
st.plotly_chart(fig_dist, use_container_width=True)
st.session_state["report_fig_dist"] = fig_dist

# Tabla resumen rápida
with st.expander("Ver Resumen Estadístico"):
    desc = df_plot.groupby(Config.STATION_NAME_COL)[
        Config.PRECIPITATION_COL
    ].describe()
    st.dataframe(desc)

# -----
# 6. ANÁLISIS ESTACIONAL DETALLADO (CÓDIGO CORREGIDO)
# -----
with tabs[5]:
    st.markdown("#### 📈 Ciclo Anual Comparativo (Spaghetti Plot)")
    st.info(
        "Compara el comportamiento de cada año individual frente al promedio histórico."
    )
    sel_st_detail = st.selectbox(
        "Analizar Estación:", stations_for_analysis, key="st_detail_seasonal"
    )

```

```

    )

if sel_st_detail:
    df_st = df_monthly_filtered[
        df_monthly_filtered[Config.STATION_NAME_COL] == sel_st_detail
    ].copy()

c_hl, c_type = st.columns([1, 1])
with c_hl:
    years = sorted(df_st["Año"].unique(), reverse=True)
    hl_year = st.selectbox(
        "Resaltar Año:", [None] + years, key="hl_year_seasonal"
    )
with c_type:
    chart_mode = st.radio(
        "Tipo de Visualización:",
        ["Líneas (Spaghetti)", "Cajas (Variabilidad)"],
        horizontal=True,
        key="mode_seasonal",
    )

if chart_mode == "Líneas (Spaghetti)":  

    fig_multi = go.Figure()
    for yr in years:
        df_y = df_st[df_st["Año"] == yr].sort_values("Mes")
        color = "rgba(200, 200, 200, 0.4)"
        width = 1
        opacity = 0.5
        name = str(yr)
        show_leg = False
        if hl_year and yr == hl_year:
            color = "red"
            width = 4
            opacity = 1.0
            show_leg = True

        fig_multi.add_trace(
            go.Scatter(
                x=df_y["Nombre_Mes"],
                y=df_y[Config.PRECIPITATION_COL],
                mode="lines",
                name=name,
                line=dict(color=color, width=width),
                opacity=opacity,
                showlegend=show_leg,
                hoverinfo="name+y",
            )
        )

```

```

        )
    )

clim = (
    df_st.groupby("Nombre_Mes")[Config.PRECIPITATION_COL]
    .mean()
    .reindex(list(meses_orden.values()))
)
fig_multi.add_trace(
    go.Scatter(
        x=clim.index,
        y=clim.values,
        mode="lines+markers",
        name="Promedio Histórico",
        line=dict(color="black", width=3, dash="dot"),
        marker=dict(size=8, color="black"),
    )
)
fig_multi.update_layout(
    title=f"Ciclo Anual Comparativo - {sel_st_detail}",
    xaxis_title="Mes",
    yaxis_title="Precipitación (mm)",
    hovermode="x unified",
    height=500,
)
st.plotly_chart(fig_multi, use_container_width=True)
else:
    fig_box = px.box(
        df_st,
        x="Nombre_Mes",
        y=Config.PRECIPITATION_COL,
        category_orders={"Nombre_Mes": list(meses_orden.values())},
        color="Nombre_Mes",
        points="all",
        title=f"Variabilidad Mensual Histórica - {sel_st_detail}",
    )
    fig_box.update_layout(showlegend=False, height=500)
    st.plotly_chart(fig_box, use_container_width=True)

if hl_year:
    st.markdown(f"##### Detalle Año {hl_year} vs Promedio")
    df_y_hl = df_st[df_st["Año"] == hl_year].set_index("Nombre_Mes")[
        Config.PRECIPITATION_COL
    ]
    comp_df = pd.DataFrame(
        {"Año Seleccionado": df_y_hl, "Promedio Histórico": clim}
    )

```

```

        )
comp_df["Diferencia (%)"] = (
    (comp_df["Año Seleccionado"] - comp_df["Promedio Histórico"])
    / comp_df["Promedio Histórico"]
) * 100
st.dataframe(comp_df.style.format("{:.1f}"))

# 7. COMPARATIVA MULTIESCALAR (VERSIÓN FINAL CON DESCARGA Y METADATOS)
# -----
with tabs[6]:
    st.subheader("📊 Comparativa de Regímenes de Lluvia")

    # 1. MENSAJE INSPIRADOR (Solicitud 1)
    st.info(
        "💡 **Análisis Multiescalar:** Aquí puedes pasar de un análisis individual a un análisis "
        "comparativo multiescalar para entender diferencias climáticas entre zonas vecinas o lejanas."
    )

# Verificación de carga de datos
if (st.session_state.get('df_long') is not None and
    st.session_state.get('gdf_stations') is not None and
    st.session_state.get('gdf_subcuenca') is not None):
    try:
        # --- PREPARACIÓN DE DATOS (Mantenemos la lógica que ya funciona) ---
        df_datos = st.session_state['df_long'].copy()
        df_meta = st.session_state['gdf_stations'].copy()
        gdf_poligonos = st.session_state['gdf_subcuenca'].copy()

        col_codigo_datos = 'id_estacion'
        col_valor = 'precipitacion_mm'
        col_fecha = 'fecha' if 'fecha' in df_datos.columns else 'fecha_mes_año'
        col_codigo_meta = 'Id_estacio'
        col_municipio = 'municipio'
        col_region = 'SUBREGION'
        col_cuenca = 'SUBC_LBL'

        # A) Limpieza de Coordenadas
        for col_coord in ['Longitud_geo', 'Latitud_geo']:
            if df_meta[col_coord].dtype == 'object':
                df_meta[col_coord] = (
                    df_meta[col_coord]
                    .astype(str)
                    .str.replace(',', '.', regex=False)
                    .apply(pd.to_numeric, errors='coerce')
                )
    
```

```

        )

df_meta = df_meta.dropna(subset=['Longitud_geo', 'Latitud_geo'])

if not isinstance(df_meta, gpd.GeoDataFrame):
    df_meta = gpd.GeoDataFrame(
        df_meta,
        geometry=gpd.points_from_xy(df_meta['Longitud_geo'], df_meta['Latitud_geo']),
        crs="EPSG:4326"
    )

if df_meta.crs != gdf_poligonos.crs:
    gdf_poligonos = gdf_poligonos.to_crs(df_meta.crs)

# B) Cruce Espacial
df_meta_espacial = gpd.sjoin(
    df_meta,
    gdf_poligonos[['geometry', col_cuenca]],
    how="left",
    predicate="intersects"
)

# C) Unión Final
df_datos[col_codigo_datos] = df_datos[col_codigo_datos].astype(str)
df_meta_espacial[col_codigo_meta] = df_meta_espacial[col_codigo_meta].astype(str)

df_full = pd.merge(
    df_datos,
    df_meta_espacial[[col_codigo_meta, col_municipio, col_region, col_cuenca]],
    left_on=col_codigo_datos,
    right_on=col_codigo_meta,
    how='inner'
)

# D) Procesamiento Temporal (Traducción de Fechas)
fechas_str = df_full[col_fecha].astype(str).str.lower()
reemplazos = {
    'ene': 'Jan', 'feb': 'Feb', 'mar': 'Mar', 'abr': 'Apr',
    'may': 'May', 'jun': 'Jun', 'jul': 'Jul', 'ago': 'Aug',
    'sep': 'Sep', 'oct': 'Oct', 'nov': 'Nov', 'dic': 'Dec'
}
for mes_es, mes_en in reemplazos.items():
    fechas_str = fechas_str.str.replace(mes_es, mes_en, regex=False)

df_full[col_fecha] = pd.to_datetime(fechas_str, format='%b-%y', errors='coerce')
df_full['MES_NUM'] = df_full[col_fecha].dt.month

```

```

mapa_meses = {1:'Ene', 2:'Feb', 3:'Mar', 4:'Abr', 5:'May', 6:'Jun',
              7:'Jul', 8:'Ago', 9:'Sep', 10:'Oct', 11:'Nov', 12:'Dic'}
df_full['Nombre_Mes'] = df_full['MES_NUM'].map(mapa_meses)

# --- INTERFAZ VISUAL ---
col1, col2 = st.columns([1, 2])

with col1:
    nivel_analisis = st.radio(
        "Nivel de Agregación:",
        ["Municipio", "Cuenca", "Región"],
        key="radio_multi_nivel_final"
    )

    if nivel_analisis == "Municipio":
        columna_filtro = col_municipio
    elif nivel_analisis == "Cuenca":
        columna_filtro = col_cuenca
    else:
        columna_filtro = col_region

    opciones = sorted(df_full[columna_filtro].dropna().astype(str).unique())

with col2:
    seleccion = st.multiselect(
        f"Seleccione {nivel_analisis}s:",
        options=opciones,
        default=[opciones[0]] if len(opciones) > 0 else None
    )

    if seleccion:
        df_filtrado = df_full[df_full[columna_filtro].isin(seleccion)]

        # Agrupación para el gráfico
        df_agrupado = df_filtrado.groupby(['MES_NUM', 'Nombre_Mes',
                                            columna_filtro])[col_valor].mean().reset_index()
        df_agrupado = df_agrupado.sort_values('MES_NUM')

        fig = px.line(
            df_agrupado,
            x='Nombre_Mes',
            y=col_valor,
            color=columna_filtro,
            title=f"Patrón Medio Anual Comparativo ({nivel_analisis})",
            markers=True
        )

```

```

fig.update_layout(hovermode="x unified", legend=dict(orientation="h", y=1.1))
st.plotly_chart(fig)

# 2. BOTÓN DE DESCARGA (Solicitud 2)
# Preparamos los datos en formato CSV para la descarga
# Usamos pivot para que sea más legible en Excel (Meses como columnas)
df_export = df_agrupado.pivot(index=columna_filtro, columns='Nombre_Mes', values=col_valor)
# Reordenar columnas cronológicamente
cols_meses = [mapa_meses[i] for i in range(1, 13)]
df_export = df_export[cols_meses]

csv_data = df_export.to_csv().encode('utf-8-sig') # utf-8-sig para que Excel lea bien las tildes

st.download_button(
    label="⬇️ Descargar Datos del Gráfico (CSV)",
    data=csv_data,
    file_name=f"comparativa_{nivel_analisis.lower()}.csv",
    mime="text/csv",
)

# 3. FICHA METODOLÓGICA (Solicitud 3)
with st.expander("ℹ️ Metodología, Datos y Periodo Analizado"):
    # Calculamos el periodo real de los datos
    min_year = df_full[col_fecha].dt.year.min()
    max_year = df_full[col_fecha].dt.year.max()

    st.markdown(f"""
        **Ficha Técnica del Análisis:**

        * **Metodología:** Se calcula el promedio aritmético de la precipitación mensual de todas las estaciones ubicadas dentro de la geometría seleccionada ({nivel_analisis}).
        * **Tipo de Datos:** Precipitación acumulada mensual (mm). Datos procesados del archivo histórico.
        * **Fuente:** Red de monitoreo SIHCLIM / IDEAM.
        * **Periodo de Registro:** Enero {min_year} - Diciembre {max_year}.
        * **Procesamiento:** Las estaciones fueron asignadas espacialmente usando intersección geométrica.
        * Se aplicó control de calidad básico a las series temporales.
    """))

    else:
        st.warning("Seleccione al menos un elemento para generar el reporte.")

except Exception as e:

```

```

        st.error(f"Ocurrió un error en el procesamiento: {e}")
    else:
        st.warning("⚠️ Faltan datos cargados.")

def display_weekly_forecast_tab(stations_for_analysis, gdf_filtered, **kwargs):
    """Muestra el pronóstico semanal para una estación seleccionada."""
    st.subheader("🌦 Pronóstico a 7 Días (Open-Meteo)")

    if not stations_for_analysis:
        st.warning("Seleccione estaciones en el panel lateral primero.")
        return

    selected_station = st.selectbox(
        "Seleccionar Estación:", stations_for_analysis, key="wk_cast_sel"
    )

    if selected_station and gdf_filtered is not None:
        station_data = gdf_filtered[
            gdf_filtered[Config.STATION_NAME_COL] == selected_station
        ]
        if not station_data.empty:
            # Obtener lat/lon
            if "latitude" in station_data.columns:
                lat = station_data.iloc[0]["latitude"]
                lon = station_data.iloc[0]["longitude"]
            else:
                lat = station_data.iloc[0].geometry.y
                lon = station_data.iloc[0].geometry.x

            df = get_weather_forecast_simple(lat, lon)
            if not df.empty:
                st.dataframe(df)

                fig = go.Figure()
                fig.add_trace(
                    go.Scatter(
                        x=df["Fecha"],
                        y=df["Temp. Máx (°C)"],
                        name="Máx",
                        line=dict(color="red"),
                    )
                )
                fig.add_trace(
                    go.Scatter(

```

```

        x=df["Fecha"],
        y=df["Temp. Mín (°C)"],
        name="Mín",
        line=dict(color="blue"),
    )
)
st.plotly_chart(fig)
else:
    st.error("No se pudo obtener el pronóstico.")

def display_satellite_imagery_tab(gdf_filtered):
    """
    Muestra imágenes satelitales en tiempo real.
    Versión Robusta: Descarga segura de imágenes y mapas ligeros.
    """
    st.subheader("📡 Monitoreo Satelital (Tiempo Real)")

    tab_map, tab_anim = st.tabs([
        "🌐 Mapa de Nubes (Interactivo)", "▶️ Animación (Últimas Horas)"
    ])

    # --- TAB 1: MAPA INTERACTIVO ---
    with tab_map:
        col_map, col_info = st.columns([3, 1])
        with col_map:
            try:
                # Centrar mapa
                if gdf_filtered is not None and not gdf_filtered.empty:
                    if "latitude" not in gdf_filtered.columns:
                        gdf_filtered["latitude"] = gdf_filtered.geometry.y
                        gdf_filtered["longitude"] = gdf_filtered.geometry.x
                    center_lat = gdf_filtered["latitude"].mean()
                    center_lon = gdf_filtered["longitude"].mean()
            except:
                center_lat, center_lon = 6.0, -75.0

        m = folium.Map(location=[center_lat, center_lon], zoom_start=6)

        # 1. Base: CartoDB Positron (Carga muy rápido y es limpia)
        folium.TileLayer(
            tiles="CartoDB positron",
            attr="CartoDB",
            name="Mapa Base Claro",
            overlay=False,

```

```

).add_to(m)

# 2. Overlay: Nubes (GOES-16 IR) - NASA GIBS
# Usamos una URL WMS estándar que suele ser muy compatible
folium.raster_layers.WmsTileLayer(
    url="https://gibs.earthdata.nasa.gov/wms/epsg4326/best/wms.cgi",
    name="Nubes (Infrarrojo)",
    layers="GOES-East_ABI_Band13_Clean_Infrared",
    fmt="image/png",
    transparent=True,
    opacity=0.5,
    attr="NASA GIBS",
).add_to(m)

# 3. Estaciones
if gdf_filtered is not None and not gdf_filtered.empty:
    from folium.plugins import MarkerCluster

    mc = MarkerCluster(name="Estaciones").add_to(m)
    for _, row in gdf_filtered.iterrows():
        folium.CircleMarker(
            location=[row["latitude"], row["longitude"]],
            radius=4,
            color="blue",
            fill=True,
            fill_color="cyan",
            fill_opacity=0.8,
            popup=row.get(Config.STATION_NAME_COL, "Estación"),
        ).add_to(mc)

    folium.LayerControl().add_to(m)
    st_folium(m, height=500, use_container_width=True)

except Exception as e:
    st.error(f"Error cargando mapa: {e}")

with col_info:
    st.info(
        """
        **Capas:**  

        1. **Fondo:** CartoDB (Ligero).  

        2. **Nubes:** Infrarrojo GOES-16.
        """
    )
# --- TAB 2: ANIMACIÓN (GIF NOAA - Descarga Segura) ---

```

```

with tab_anim:
    st.markdown("#### 🎥 Animación GeoColor (Sector Norte de Suramérica)")

    # URL Oficial NOAA (Northern South America)
    url_gif = "https://cdn.star.nesdis.noaa.gov/GOES16/ABI/SECTOR/nsa/GEOCOLOR/GOES16-NSA-GEOCOLOR-1000x1000.gif"

    with st.spinner("Descargando animación de la NOAA..."):
        gif_data = fetch_secure_content(url_gif)

    if gif_data:
        st.image(
            gif_data,
            caption="Animación GeoColor (Tiempo Real)",
            width=700,
        )
    else:
        st.error("⚠️ No se pudo descargar la animación automáticamente.")
        st.markdown(
            f"[Haga clic aquí para verla directamente en la NOAA]({url_gif})"
        )

def display_advanced_maps_tab(df_long, gdf_stations, **kwargs):
    """
    Versión Corrección Final:
    - Solución KeyError 'df_raw' (blindaje de memoria).
    - Solución Mapas Vacíos (Fallback inteligente si el recorte falla).
    - Restauración total de Popups, FDC y Textos.
    """

    import plotly.graph_objects as go
    import plotly.express as px
    from scipy.interpolate import Rbf, griddata
    import numpy as np
    from streamlit_folium import st_folium
    import folium
    from folium.plugins import LocateControl
    import geopandas as gpd
    import matplotlib
    import matplotlib.pyplot as plt
    from matplotlib import path as mpath
    from shapely.geometry import LineString
    import tempfile
    import os
    import shutil

```

```

matplotlib.use('Agg')

# Recuperar datos
# 1. Recuperar Coberturas (Si no vienen en kwargs, intentar cargar por defecto)
gdf_coberturas = kwargs.get("gdf_coberturas", None)
if gdf_coberturas is None:
    try:
        # Ruta por defecto en tu carpeta data
        path_cob = "data/coberturas_antioquia.geojson" # Ajusta si tu archivo tiene otro nombre
        if os.path.exists(path_cob):
            gdf_coberturas = gpd.read_file(path_cob)
    except:
        pass # Si falla, se quedará como None y no mostrará esos mapas

df_trends_global = kwargs.get("df_trends", None)

# Configuración Títulos
selected_months = kwargs.get("selected_months", [])
titulo_meses = ""
if selected_months and len(selected_months) < 12:
    nombres_meses = ["Ene", "Feb", "Mar", "Abr", "May", "Jun", "Jul", "Ago", "Sep", "Oct", "Nov", "Dic"]
    meses_str = ", ".join([nombres_meses[m - 1] for m in selected_months])
    titulo_meses = f" {meses_str}""

st.subheader(f" ● Superficies de Interpolación{titulo_meses} y Análisis Hidrológico")

mode = st.radio("Modo de Análisis:", ["Regional (Comparación)", "Por Cuenca (Detallado)"],
horizontal=True)
user_loc = kwargs.get("user_loc", None)

# --- HELPERS ---
# --- HELPER: INTERPOLACIÓN ROBUSTA (CON EXTRAPOLACIÓN) ---
def run_interp(df_puntos, metodo, bounds_box):
    try:
        # 1. Extraer límites
        minx, maxx, miny, maxy = bounds_box

        # 2. Generar grilla con mayor resolución (1500x150)
        # Usamos números complejos (150j) para incluir el punto final exacto
        gx, gy = np.mgrid[minx:maxx:150j, miny:maxy:150j]

        # 3. Preparar datos
        df_unique = df_puntos.drop_duplicates(subset=["longitude", "latitude"])
        pts = df_unique[["longitude", "latitude"]].values
        vals = df_unique[Config.PRECIPITATION_COL].values
    
```

```

if len(pts) < 3: return None, None, None

# 4. Interpolación Principal
if "Kriging" in metodo:
    try:
        # Rbf (Thin Plate Spline) suele extrapolar bien suavemente
        rbf = Rbf(pts[:, 0], pts[:, 1], vals, function="thin_plate")
        gz = rbf(gx, gy)
    except:
        gz = griddata(pts, vals, (gx, gy), method="linear")
else:
    # IDW / Lineal genera NaNs afuera.
    gz = griddata(pts, vals, (gx, gy), method="linear")

# 5. RELLENO DE BORDES (EXTRAPOLACIÓN) - EL SECRETO
# Buscamos dónde quedó vacío (NaN) y lo llenamos con el vecino más cercano
mask_nan = np.isnan(gz)
if np.any(mask_nan):
    gz_nearest = griddata(pts, vals, (gx, gy), method="nearest")
    gz[mask_nan] = gz_nearest[mask_nan]

return gx, gy, gz

except Exception as e:
    print(f"Error interpolación: {e}")
    return None, None, None

def calcular_promedios_reales(df_datos):
    if df_datos.empty: return pd.DataFrame()
    conteo = df_datos[df_datos[Config.PRECIPITATION_COL] >= 0].groupby([Config.STATION_NAME_COL,
Config.YEAR_COL]).size()
    anos_validos = conteo[conteo >= 5].index
    df_filtrado = df_datos.set_index([Config.STATION_NAME_COL,
Config.YEAR_COL]).loc[anos_validos].reset_index()
    suma_anual = df_filtrado.groupby([Config.STATION_NAME_COL,
Config.YEAR_COL])[Config.PRECIPITATION_COL].sum().reset_index()
    return
    suma_anual.groupby(Config.STATION_NAME_COL)[Config.PRECIPITATION_COL].mean().reset_index()

def generate_isohyets_gdf(gx, gy, gz, levels=10, crs="EPSG:4326"):
    try:
        fig, ax = plt.subplots()
        contour = ax.contour(gx, gy, gz, levels=levels)
        plt.close(fig)
        lines, values = [], []
        for collection in contour.collections:

```

```

z_val = 0
try: z_val = collection.level
except: pass
for path in collection.get_paths():
    if len(path.vertices) >= 2:
        lines.append(LineString(path.vertices))
        values.append(z_val)
if not lines: return None
return gpd.GeoDataFrame({"valor": values, "geometry": lines}, crs=crs)
except: return None

def create_zipped_shapefile(gdf, filename_base):
    with tempfile.TemporaryDirectory() as tmpdir:
        path = os.path.join(tmpdir, f"{filename_base}.shp")
        gdf.to_file(path, driver="ESRI Shapefile")
        base_zip = os.path.join(tmpdir, filename_base)
        shutil.make_archive(base_zip, 'zip', tmpdir)
        with open(f"{base_zip}.zip", "rb") as f: return f.read()

# --- HELPER MÁSCARA (MEJORADO PARA EVITAR MAPAS VACÍOS) ---
# --- HELPER OPTIMIZADO: MÁSCARA RÁPIDA (CON CORRECCIÓN DE GIRO) ---
def mask_grid_with_geometries(gx, gy, grid_values, gdf_mask):
    if gdf_mask is None or gdf_mask.empty or grid_values is None:
        return np.zeros_like(grid_values) if grid_values is not None else None
    try:
        from rasterio import features
        from rasterio.transform import from_bounds

        # 1. Asegurar proyección Lat/Lon
        if gdf_mask.crs is None: gdf_mask.set_crs("EPSG:3116", inplace=True)
        if gdf_mask.crs.to_string() != "EPSG:4326":
            gdf_mask = gdf_mask.to_crs("EPSG:4326")

        # 2. Configurar dimensiones
        # OJO: grid_values viene de np.mgrid que es (X, Y)
        # Rasterio genera imágenes (Filas=Y, Columnas=X)
        rows, cols = gx.shape # Aquí rows es X, cols es Y en la lógica de mgrid

        minx, maxx = gx.min(), gx.max()
        miny, maxy = gy.min(), gy.max()

        # 3. Crear transformación
        # IMPORTANTE: Intercambiamos filas/cols en from_bounds para engañar a rasterio
        # y que genere la matriz en la orientación que coincide con mgrid (X, Y)
        transform = from_bounds(minx, miny, maxx, maxy, rows, cols)
    
```

```

# 4. Rasterizar
shapes = [(geom, 1) for geom in gdf_mask.geometry if not geom.is_empty]

if not shapes: return np.zeros_like(grid_values)

mask_arr = features.rasterize(
    shapes=shapes,
    out_shape=(rows, cols), # Dimensiones invertidas
    transform=transform,
    fill=0,
    default_value=1,
    dtype='uint8'
)

# 5. CORRECCIÓN DE GIRO (TRANSPOSE)
# Como mgrid es (X, Y) y rasterio suele ser (Y, X), a veces quedan rotados.
# Esta línea asegura que la máscara encaje si quedó rotada.
if mask_arr.shape != grid_values.shape:
    mask_arr = mask_arr.T
else:
    # Si las dimensiones son cuadradas (150x150), el shape no nos avisa.
    # Aplicamos la transpuesta porque sabemos que mgrid vs imagen siempre invierte ejes.
    mask_arr = mask_arr.T

# 6. Aplicar máscara
result = grid_values.copy()
# Donde no hay máscara, ponemos NaN (transparente)
result[mask_arr == 0] = np.nan

return result

except Exception as e:
    print(f"Error máscara rápida: {e}")
    return np.zeros_like(grid_values)

# =====
# MODO 1: REGIONAL (COMPARACIÓN) - CON DESCARGAS
# =====
if mode == "Regional (Comparación)":
    st.markdown("#### vs Comparación de Periodos Climáticos")
    st.info("Visualice cambios en el patrón de lluvias entre dos periodos.")

c1, c2 = st.columns(2)
with c1:
    st.markdown("##### Periodo 1 (Referencia)")

```

```

r1 = st.slider("Rango P1:", 1980, 2024, (1990, 2000), key="r1")
m1 = st.selectbox("Método P1:", ["Kriging (RBF)", "IDW (Lineal)", "Spline"], key="m1")
with c2:
    st.markdown("##### Periodo 2 (Reciente)")
    r2 = st.slider("Rango P2:", 1980, 2024, (2010, 2020), key="r2")
    m2 = st.selectbox("Método P2:", ["Kriging (RBF)", "IDW (Lineal)", "Spline"], key="m2")

# Botón de cálculo con PERSISTENCIA
if st.button("👉 Generar Comparación"):
    st.session_state["regional_done"] = True
    st.session_state["reg_params"] = {"r1": r1, "m1": m1, "r2": r2, "m2": m2}

if st.session_state.get("regional_done"):
    p = st.session_state["reg_params"]

# Función interna plot_panel corregida y con descargas
def plot_panel(rng, meth, col, tag, u_loc):
    mask = (df_long[Config.YEAR_COL] >= rng[0]) & (df_long[Config.YEAR_COL] <= rng[1])
    df_sub = df_long[mask]
    df_avg = calcular_promedios_reales(df_sub)

    if df_avg.empty:
        col.warning(f"Sin datos válidos para {rng}")
        return

    if Config.STATION_NAME_COL not in df_avg.columns:
        df_avg = df_avg.reset_index()

    df_m = pd.merge(df_avg, gdf_stations, on=Config.STATION_NAME_COL).dropna(subset=["latitude", "longitude"])

    if len(df_m) > 2:
        bounds = [
            df_m.longitude.min() - 0.1,
            df_m.longitude.max() + 0.1,
            df_m.latitude.min() - 0.1,
            df_m.latitude.max() + 0.1,
        ]
        gx, gy, gz = run_interp(df_m, meth, bounds)

    if gz is not None:
        # 1. Mapa Plotly (Isoyetas Visuales)
        fig = go.Figure()
        go.Contour(
            z=gz.T,
            x=gx[:, 0],

```

```

        y=gy[0,:],
        colorscale="Viridis",
        colorbar=dict(title="mm/año", len=0.5),
        contours=dict(start=0, end=5000, size=200),
    )
)

# Puntos Estaciones
fig.add_trace(
    go.Scatter(
        x=df_m.longitude,
        y=df_m.latitude,
        mode="markers",
        marker=dict(color="black", size=7, line=dict(width=1, color="white")),
        text=df_m.apply(lambda x: f"<b>{x[Config.STATION_NAME_COL]}</b><br>Ppt:<br>{x[Config.PRECIPITATION_COL]:.0f} mm", axis=1),
        hoverinfo="text",
        showlegend=False,
    )
)

# Capa Usuario
if u_loc:
    fig.add_trace(
        go.Scatter(
            x=[u_loc[1]],
            y=[u_loc[0]],
            mode="markers+text",
            marker=dict(color="red", size=12, symbol="star"),
            text=["📍 TÚ"],
            textposition="top center",
        )
    )

fig.update_layout(
    title=f"Ppt Media Anual ({rng[0]}-{rng[1]}",
    margin=dict(l=0, r=0, b=0, t=40),
    height=400,
)
col.plotly_chart(fig, use_container_width=True)

# 2. Generación de Vectores para Descarga
gdf_iso = generate_isohyets_gdf(gx, gy, gz, levels=12, crs=gdf_stations.crs)

# 3. Zona de Descargas
if gdf_iso is not None:

```

```

with col.expander("💾 Descargar Capas (GIS)"):
    # GeoJSON (Rápido)
    col.download_button(
        label="🌐 Descargar GeoJSON",
        data=gdf_iso.to_json(),
        file_name=f"isoyetas_{tag}_{rng[0]}_{rng[1]}.geojson",
        mime="application/json"
    )

    # Shapefile (Manual para evitar bloqueos)
    if col.button(f"📦 Generar Shapefile ({tag})", key=f"btn_shp_{tag}"):
        try:
            zip_shp = create_zipped_shapefile(gdf_iso, f"isoyetas_{tag}")
            col.download_button(
                label="ZIP Descargar ZIP (.shp)",
                data=zip_shp,
                file_name=f"isoyetas_{tag}_{rng[0]}_{rng[1]}.zip",
                mime="application/zip"
            )
        except Exception as e:
            col.error(f"Error generando ZIP: {e}")

# 4. Mapa Interactivo (Folium) con Popups
with col.expander(f"📍 Ver Mapa Interactivo Detallado ({tag})", expanded=True):
    col.write("Mapa navegable con detalles por estación.")
    center_lat = (bounds[2] + bounds[3]) / 2
    center_lon = (bounds[0] + bounds[1]) / 2
    m = folium.Map(
        location=[center_lat, center_lon],
        zoom_start=8,
        tiles="CartoDB positron",
    )

# Añadimos isoyetas al folium también para referencia
if gdf_iso is not None:
    folium.GeoJson(
        gdf_iso,
        name="Isoyetas",
        style_function=lambda x: {'color': '#2c3e50', 'weight': 1, 'dashArray': '5, 5'}
    ).add_to(m)

for _, row in df_m.iterrows():
    nombre = row[Config.STATION_NAME_COL]
    lluvia = row[Config.PRECIPITATION_COL]
    html = f"<b>{nombre}</b><br>{lluvia:.0f} mm"

```

```

        folium.CircleMarker(
            [row["latitude"], row["longitude"]],
            radius=6,
            color="blue",
            fill=True,
            fill_color="cyan",
            fill_opacity=0.9,
            tooltip=html,
        ).add_to(m)

LocateControl(auto_start=False).add_to(m)
st_folium(
    m,
    height=350,
    use_container_width=True,
    key=f"folium_comp_{tag}",
)
)

# Renderizar paneles
plot_panel(p["r1"], p["m1"], c1, "A", user_loc)
plot_panel(p["r2"], p["m2"], c2, "B", user_loc)

# =====
# MODO 2: CUENCA (COMPLETO E INTEGRADO)
# =====
else:
    gdf_subcuenca = kw_args.get("gdf_subcuenca")
    if gdf_subcuenca is None or gdf_subcuenca.empty:
        st.warning("⚠️ No se ha cargado la capa de Cuencas.")
        return

    col_name = next((c for c in gdf_subcuenca.columns if "nombre" in c.lower() or "cuenca" in c.lower()), None)
    gdf_subcuenca.columns[0]
    default_cuenca = st.session_state.get("last_sel_cuenca", [])
    avail_opts = sorted(gdf_subcuenca[col_name].unique().astype(str))
    valid_defaults = [x for x in default_cuenca if x in avail_opts]
    sel_cuenca = st.multiselect("Seleccionar Cuenca(s):", avail_opts, default=valid_defaults)

    with st.expander("⚙️ Configuración Avanzada", expanded=False):
        buffer_km = st.slider("Radio búsqueda (km):", 0, 50, 15, step=5)

    if sel_cuenca:
        c_p1, c_p2 = st.columns(2)
        rng_c = c_p1.slider("Periodo:", 1980, 2025, (2000, 2020))
        meth_c = c_p2.selectbox("Método Interpolación:", ["Kriging (RBF)", "IDW"])

```

```

# --- LÓGICA DE CÁLCULO ---
if st.button("⚡ Analizar Cuenca"):
    st.session_state["last_sel_cuencas"] = sel_cuencas
    if "basin_res" in st.session_state: del st.session_state["basin_res"]

    with st.spinner("Procesando hidrología, IVC y tendencias..."):
        # 1. Geometría
        sub = gdf_subcuencas[gdf_subcuencas[col_name].isin(sel_cuencas)]
        try:
            geom_union = sub.geometry.unary_union
            gdf_union = gpd.GeoDataFrame({"geometry": [geom_union]}, crs=gdf_subcuencas.crs)
        except:
            gdf_union = gpd.GeoDataFrame({"geometry": [sub.geometry.iloc[0]]}, crs=gdf_subcuencas.crs)
            geom_union = sub.geometry.iloc[0]

        gdf_vis = gdf_union.copy()
        gdf_vis["geometry"] = gdf_vis.geometry.simplify(0.005)

        buf = geom_union.buffer(buffer_km / 111.32)
        gdf_buf = gpd.GeoDataFrame({"geometry": [buf]}, crs=gdf_stations.crs)
        stns = gpd.sjoin(gdf_stations, gdf_buf, predicate="intersects")

        if not stns.empty:
            # 2. Datos
            mask = (df_long[Config.STATION_NAME_COL].isin(stns[Config.STATION_NAME_COL].unique()))
& \
            (df_long[Config.YEAR_COL] >= rng_c[0]) & (df_long[Config.YEAR_COL] <= rng_c[1])
            df_raw = df_long[mask].copy()
            df_avg = calcular_promedios_reales(df_raw)

            if Config.STATION_NAME_COL not in df_avg.columns: df_avg = df_avg.reset_index()
            df_int = pd.merge(df_avg, gdf_stations,
on=Config.STATION_NAME_COL).dropna(subset=["latitude", "longitude"])
            if Config.ALTITUDE_COL not in df_int.columns: df_int[Config.ALTITUDE_COL] = 1500
            gdf_pts = gpd.GeoDataFrame(df_int, geometry=gpd.points_from_xy(df_int.longitude,
df_int.latitude), crs=gdf_stations.crs)

            if len(df_int) >= 3:
                # 3. Malla Base e Interpolación (CORREGIDO)
                # Usamos los límites del buffer para asegurar cobertura total
                minx, miny, maxx, maxy = gdf_buf.total_bounds

                # A. Interpolación de PRECIPITACIÓN (gz)
                # Llamamos al helper run_interp que ya incluye Extrapolación (relleno de bordes)
                gx, gy, gz = run_interp(df_int, meth_c, [minx, maxx, miny, maxy])

```

```

# B. Interpolación de ALTITUD (gz_alt)
# Necesaria para el cálculo de IVC. La hacemos sobre la misma malla gx, gy.
gz_alt = None
if gx is not None:
    # Definimos los puntos y valores aquí para evitar errores de variables no definidas
    pts = df_int[["longitude", "latitude"]].values
    vals_alt = df_int[Config.ALTITUDE_COL].values

    # 1. Interpolación Lineal (precisa adentro)
    gz_alt = griddata(pts, vals_alt, (gx, gy), method='linear')

    # 2. Extrapolación Nearest (para llenar los bordes vacíos NaN)
    mask_nan_alt = np.isnan(gz_alt)
    if np.any(mask_nan_alt):
        gz_alt_near = griddata(pts, vals_alt, (gx, gy), method='nearest')
        gz_alt[mask_nan_alt] = gz_alt_near[mask_nan_alt]

# B. IVC (Física y Normalización)
gz_t = np.maximum(28 - (0.006 * gz_alt), 0)
l_t = 300 + (25 * gz_t) + (0.05 * gz_t**3)
with np.errstate(divide='ignore', invalid='ignore'):
    gz_etr = gz / np.sqrt(0.9 + (gz / l_t)**2)
    gz_etr = np.minimum(gz_etr, gz)
    gz_esd = gz - gz_etr

    t_max = np.nanmax(gz_t)
    gz_it = 100 * (gz_t / t_max) if t_max > 0 else gz_t * 0
    esd_max = np.nanmax(gz_esd)
    if esd_max <= 0: esd_max = 1
    gz_iesd = 100 * ((esd_max - gz_esd) / esd_max)
    gz_ivc = (gz_it + gz_iesd) / 2

# C. Tendencias (In-Situ Backup)
gz_iv_var = None
df_slopes = df_trends_global # Intentar usar el global

# Si no hay global, calcular localmente
if df_slopes is None and len(df_raw) > 0:
    try:
        import scipy.stats
        slopes = []
        for s in df_int[Config.STATION_NAME_COL].unique():
            d = df_raw[df_raw[Config.STATION_NAME_COL] ==
s].groupby(Config.YEAR_COL)[Config.PRECIPITATION_COL].sum()
            if len(d) > 5:

```

```

        sl, _, _, _, _ = scipy.stats.linregress(d.index, d.values)
        slopes.append({Config.STATION_NAME_COL: s, 'slope': sl})
    if slopes: df_slopes = pd.DataFrame(slopes)
except: pass

if df_slopes is not None:
    try:
        df_tm = pd.merge(df_int, df_slopes, on=Config.STATION_NAME_COL, how="left")
        if 'slope' in df_tm.columns:
            gz_slope = griddata(pts, df_tm['slope'].fillna(0).values, (gx, gy), method='linear')
            gz_tr = np.clip(50 - (gz_slope * 25), 0, 100)
            gz_iv_var = (gz_iesd + gz_tr) / 2
    except: pass

# D. Recorte (Masking) para Cultivos e Incendios (CÓDIGO SIMPLIFICADO)
# Inicializamos con ceros por defecto
gz_cult = np.zeros_like(gz_ivc)
gz_inc = np.zeros_like(gz_ivc)

if gdf_coberturas is not None:
    try:
        # Identificar columna de texto
        txt_cols = gdf_coberturas.select_dtypes(include=['object']).columns
        if len(txt_cols) > 0:
            c = txt_cols[0]
            # Filtros flexibles (case-insensitive)
            gdf_a = gdf_coberturas[gdf_coberturas[c].astype(str).str.contains("Cult|Past|Agri",
case=False, na=False)]
            gdf_b = gdf_coberturas[gdf_coberturas[c].astype(str).str.contains("Bosq|For|Selv",
case=False, na=False)]

        # Aplicar la nueva función de máscara corregida
        if not gdf_a.empty:
            # gz_cult tendrá valores donde hay cultivo, y NaN donde no
            gz_cult = mask_grid_with_geometries(gx, gy, gz_ivc, gdf_a)

        if not gdf_b.empty:
            # gz_inc tendrá valores donde hay bosque, y NaN donde no
            gz_inc = mask_grid_with_geometries(gx, gy, gz_ivc, gdf_b)

    except Exception as e:
        print(f"Error en filtrado de coberturas: {e}")

# 5. Hidrología Completa (ORDEN CORREGIDO)
# A. Primero generamos isoyetas (para que exista gdf_iso)
gdf_iso = generate_isohyets_gdf(gx, gy, gz, levels=12, crs=gdf_stations.crs)

```

```

# B. Calculamos precipitación media
ppt_med = np.nanmean(gz) if gz is not None else 0

# C. Morfometría
try: morph = analysis.calculate_morphometry(gdf_union)
except: morph = {"area_km2": 0, "alt_prom_m": 1500}
area_km2 = morph.get("area_km2", 100)

# D. Balance de Turc y Caudales
tm = max(0, 28 - 0.006 * morph.get("alt_prom_m", 1500))
try: _, q_mm = analysis.calculate_water_balance_turc(ppt_med, tm)
except: q_mm = 0

vol_hm3 = (q_mm * area_km2) / 1000
q_m3s = (vol_hm3 * 1_000_000) / 31536000

# --- INTERVENCIÓN 2: CÁLCULO DE CURVAS (CORREGIDO) ---

# A. CURVA HIPSOMÉTRICA (Usando DEM Real)
hyp_data = None
try:
    # Pasamos la geometría Y la ruta del archivo DEM definida en Config
    hyp_data = analysis.calculate_hypsometric_curve(
        gdf_union,
        dem_path=getattr(Config, "DEM_FILE_PATH", None) # Usa None si no está en Config
    )
except Exception as e:
    print(f"Error hipsometría: {e}")

# B. CURVA DE DURACIÓN DE CAUDALES (FDC)
fdc_data = None
try:
    # 1. Serie de Precipitación Mensual de la Cuenca
    ppt_series = df_raw.groupby(Config.DATE_COL)[Config.PRECIPITATION_COL].mean()

    # 2. Coeficiente de Escorrentía (Estimado simple)
    # Si hay mucha lluvia (>2000mm) asumimos mayor escorrentía (0.5), si no 0.3
    c_est = 0.5 if ppt_med > 2000 else 0.3

    # 3. Calcular usando la función de analysis.py
    fdc_data = analysis.calculate_duration_curve(
        ppt_series,
        runoff_coeff=c_est,
        area_km2=area_km2
    )

```

```

except Exception as e:
    print(f"Error FDC: {e}")

# E. Índices Climáticos
idx_c = {}
try:
    idx_c = analysis.calculate_climatic_indices(
        df_raw.groupby(Config.DATE_COL)[Config.PRECIPITATION_COL].mean(),
        morph.get("alt_prom_m", 1500)
    )
except: pass

# GUARDADO FINAL EN SESSION STATE
st.session_state["basin_res"] = {
    "ready": True, "names": ".join(sel_cuencas), "bounds": [minx, maxx, miny, maxy],
    "gz": gz, "gx": gx, "gy": gy, "gz_ivc": gz_ivc, "gz_iv_var": gz_iv_var,
    "gz_cult": gz_cult, "gz_inc": gz_inc,
    "gdf_union": gdf_union, "gdf_vis": gdf_vis, "gdf_pts": gdf_pts, "gdf_buf": gdf_buf,
    "gdf_iso": gdf_iso,
    "df_int": df_int, "df_raw": df_raw,
    "bal": {"P": ppt_med, "Q": q_mm, "Q_m3s": q_m3s, "Vol": vol_hm3},
    "morph": morph,
    "idx": idx_c,
    "fdc": fdc_data,    # <--- Guardamos el resultado limpio
    "hyp": hyp_data    # <--- Guardamos el resultado limpio
}
else: st.error("Insuficientes estaciones (<3).")
else: st.error("Sin estaciones cercanas.")

# --- VISUALIZACIÓN ---
res = st.session_state.get("basin_res")
if res and res.get("ready"):
    st.markdown(f"##### 📊 Resultados: {res['names']}")

opts = ["Precipitación (Isoyetas)", "IVC (Vulnerabilidad Climática)", "Vulnerabilidad Variación Clima (ESD + Tendencias)"]
if gdf_coberturas is not None:
    opts += ["Vulnerabilidad Cultivos (IVC + Agric)", "Vulnerabilidad Incendios (IVC + Bosque)"]

sel = st.selectbox("Seleccionar Mapa Temático:", opts)

z, tit, colors = res["gz"], "Precipitación (mm)", "Blues"
zmin, zmax = 0, np.nanmax(z)

if "IVC" in sel:
    colors = "RdYIGn_r"

```

```

zmin, zmax = 0, 100
if sel == "IVC (Vulnerabilidad Climática)":
    z, tit = res["gz_ivc"], "IVC Global"
elif "Cultivos" in sel:
    z, tit = res.get("gz_cult", res["gz_ivc"]), "Amenaza en Cultivos (Recortado)"
elif "Incendios" in sel:
    z, tit = res.get("gz_inc", res["gz_ivc"]), "Amenaza Incendios (Recortado)"
elif "Variación" in sel:
    z, tit, colors = res.get("gz_iv_var", res["gz_ivc"]), "Vulnerabilidad Variación (Tendencia)",
"RdYlGn_r"
zmin, zmax = 0, 100

# Mapa Principal Plotly
# --- CORRECCIÓN 1: BLINDAJE CONTRA MAPAS VACÍOS ---

# Verificamos si z tiene datos antes de intentar graficar
if z is not None:
    # Mapa Principal Plotly
    try:
        # Nota: z.T es la transpuesta, necesaria porque Plotly interpreta x/y distinto a numpy
        fig = go.Figure(go.Contour(
            z=z.T,
            x=res["gx"][:,0],
            y=res["gy"][0,:],
            colorscale=colors,
            zmin=zmin,
            zmax=zmax,
            contours=dict(start=zmin, end=zmax, size=(zmax-zmin)/15 if zmax>zmin else 1),
            colorbar=dict(title=tit)
        ))
    except Exception as e:
        print(f"No se pudo dibujar el contorno de la cuenca: {e}")

    # Agregar contorno de la cuenca (Si existe)
    try:
        # Manejo robusto de geometrías (Polygon vs MultiPolygon)
        geoms = res["gdf_vis"].geometry
        if not geoms.empty:
            geom_list = geoms.iloc[0].geoms if geoms.iloc[0].geom_type == 'MultiPolygon' else
[geoms.iloc[0]]
            for g in geom_list:
                xs, ys = g.exterior.xy
                fig.add_trace(go.Scatter(x=list(xs), y=list(ys), mode="lines", line=dict(color="black",
width=2), name="Cuenca"))
    except Exception as e:
        print(f"No se pudo dibujar el contorno de la cuenca: {e}")

    # Agregar puntos de estaciones

```

```

fig.add_trace(go.Scatter(
    x=res["gdf_pts"].geometry.x,
    y=res["gdf_pts"].geometry.y,
    mode="markers",
    marker=dict(color="black", size=4),
    name="Estaciones"
))

fig.update_layout(title=tit, height=500, margin=dict(l=20, r=20, t=40, b=20))
st.plotly_chart(fig, use_container_width=True)

except Exception as e:
    st.error(f"Error al renderizar el gráfico: {e}")

else:
    # Si z es None, mostramos una advertencia amigable en lugar de crashear
    st.warning(f"⚠️ No se pudieron generar datos para: **{sel}**.")
    st.info("Posible causa: La capa de cobertura (cultivos/bosques) no se superpone con la zona seleccionada o hay un error de proyección.")

if "IVC" in sel or "Variación" in sel:
    with st.expander("💡 Interpretación del IVC (Semáforo de Riesgo)", expanded=True):
        st.markdown("""
            El **Índice de Vulnerabilidad Climática (IVC)** identifica zonas críticas por condiciones biofísicas:
            * ⚡ Rojo (80-100): Vulnerabilidad Crítica.** Coincidencia de altas temperaturas y bajo balance hídrico (déficit).
            * 🟠 Naranja (60-80): Vulnerabilidad Alta.**
            * 🟡 Amarillo (40-60): Vulnerabilidad Media.**
            * 🟢 Verde (0-40): Vulnerabilidad Baja.** Zonas con superávit hídrico y temperaturas moderadas.
        """)
        # Descargas
        col_d1, col_d2 = st.columns(2)
        with col_d1:
            if st.button("📍 Generar GeoJSON"):
                gdf_out = generate_isohyets_gdf(res["gx"], res["gy"], z, levels=15)
                if gdf_out is not None: col_d1.download_button("Descargar", gdf_out.to_json(),
                    "mapa.geojson")

        # Métricas y Balance (Restaurado)
        st.markdown("---")
        st.subheader("💧 Balance Hídrico y Morfometría")
        m, b = res["morph"], res["bal"]

```

```

c1, c2, c3, c4 = st.columns(4)
c1.metric("Área", f"{m.get('area_km2', 0):.1f} km2")
c2.metric("Altitud Media", f"{m.get('alt_prom_m', 0):.0f} m")
c3.metric("Ppt Media", f"{b.get('P', 0):.0f} mm")
c4.metric("Caudal (Q)", f"{{(b.get('Q', 0)*m.get('area_km2', 0)/1000 * 1e6 / 31536000):.2f} m3/s}")

with st.expander("   Detalle Metodológico: Balance de Turc"):
    st.markdown("""
        **Fórmula de Turc (1954):** Estima la escorrentía anual ( $Q$ ) en función de la Precipitación ( $P$ ) y Temperatura ( $T$ ).
        
$$E = \frac{P}{\sqrt{0.9 + \frac{P^2}{L(T)^2}}} \quad \text{donde } L(T) = 300 + 25T + 0.05T^3$$

        El caudal específico se deriva como  $Q = P - E$ . Es ideal para estimar oferta hídrica media a largo plazo.
    """)
# ÍNDICES, FDC Y HIPSOMETRÍA COMPLETOS

# A. ÍNDICES CLIMÁTICOS (Con Interpretación Numérica)
st.markdown("---")
st.subheader("   Índices Climáticos")
idx = res["idx"]

c_idx1, c_idx2 = st.columns(2)
with c_idx1:
    st.metric("Arídez (Martonne)", f"{idx.get('martonne_val', 0):.1f}",
              delta=idx.get("martonne_class", ""))
    with c_idx2:
        st.metric("Erosividad (Fournier)", f"{idx.get('fournier_val', 0):.1f}",
                  delta=idx.get("fournier_class", ""))
with st.expander("   Interpretación y Rangos Numéricos", expanded=False):
    st.markdown("""
        **1. Índice de Arídez de Martonne ( $I_M$ ):**
        
$$I_M = \frac{P}{T + 10}$$

        * **0 - 10:** Desértico / Árido 
        * **10 - 20:** Semiárido 
        * **20 - 30:** Mediterráneo 
        * **30 - 60:** Húmedo 
        * **> 60:** Perhúmedo 

        **2. Índice de Fournier ( $I_F$ ):**
        
$$I_F = \sum p_i^2 / P$$

        * **< 60:** Erosividad Muy Baja 
        * **60 - 90:** Moderada 
    """)

```

```

* **90 - 120:** Alta 
* **> 120:** Muy Alta (Riesgo Erosión) 
""")
```

B. CURVA DE DURACIÓN DE CAUDALES (FDC) - Restaurada

```

if res.get("fdc"):
    st.markdown("---")
    st.subheader("📈 Curva de Duración de Caudales (FDC)")

# Recuperar datos
fdc_data = res["fdc"]
df_fdc = fdc_data.get("data") if isinstance(fdc_data, dict) else fdc_data

if df_fdc is not None and not df_fdc.empty:
    col_f1, col_f2 = st.columns([3, 1])
    with col_f1:
        # Gráfico de Área
        fig_f = go.Figure()
        fig_f.add_trace(go.Scatter(
            x=df_fdc["Probabilidad Excedencia (%)"],
            y=df_fdc["Caudal (m³/s)"],
            fill='tozeroY',
            mode='lines',
            line=dict(color='#3498db')
        ))
        fig_f.update_layout(
            xaxis_title="Probabilidad Excedencia (%)",
            yaxis_title="Caudal (m³/s)",
            margin=dict(l=20, r=20, t=20, b=20),
            height=350
        )
        st.plotly_chart(fig_f, use_container_width=True)

    with col_f2:
        st.markdown("**Ecuación Ajustada:**")
        if isinstance(fdc_data, dict) and "equation" in fdc_data:
            eq_latex = fdc_data["equation"].replace("P", "P_{exc}")
            st.latex(eq_latex)
            st.caption(f"$R^2 = {fdc_data.get('r_squared', 0):.4f}$")
        else:
            st.info("Ecuación no disponible")

with st.expander("💡 Metodología e Interpretación FDC"):
    st.markdown("")
```

Definición: La curva FDC representa la relación entre la magnitud del caudal y la frecuencia con la que es superado.

Metodología:

1. Se genera la serie de caudales medios diarios/mensuales a partir del balance hídrico ($Q = P - ETR$$).
2. Se ordenan los datos de mayor a menor.
3. Se calcula la probabilidad: $P(\%) = \frac{m}{n+1} \times 100\%$.

Interpretación:

* **Q95 (Caudal Ecológico):** Caudal superado el 95% del tiempo. Indica la oferta hídrica en estiaje.

* **Pendiente:** Una curva con mucha pendiente indica una cuenca con respuesta rápida y poca regulación (acuíferos pobres).

""")

```
# C. CURVA HIPSOMÉTRICA - Restaurada
if hasattr(analysis, "calculate_hypsometric_curve"):
    try:
        hyp = analysis.calculate_hypsometric_curve(res["gdf_cuenca"])
        if hyp:
            st.markdown("---")
            st.subheader("
```

```

st.latex(eq_clean)

with st.expander("  Interpretación Geomorfológica"):

    st.markdown("""
        **Concepto:** Muestra la distribución del área de la cuenca en función de la altura.

        **Interpretación (Integral Hipsométrica):**
        * **Curva Convexa (Integral > 0.6):** Cuenca en fase de **Juventud**. Gran potencial erosivo, laderas inestables.
        * **Curva en 'S' (Integral 0.35 - 0.60):** Cuenca en fase de **Madurez**. Equilibrio entre erosión y sedimentación.
        * **Curva Cónica (Integral < 0.35):** Cuenca en fase de **Vejez**. Predomina la sedimentación, relieve suaves.
    """)
    except Exception as e:
        pass # Silenciar errores no críticos aquí

# Mapa de Contexto (Restaurado con Popups)
st.markdown("---")
st.subheader("  Mapa de Contexto")
if "bounds" in res:
    m_ctx = folium.Map([(res["bounds"][2]+res["bounds"][3])/2,
(res["bounds"][0]+res["bounds"][1])/2], zoom_start=10, tiles="CartoDB positron")
    if res.get("gdf_buf") is not None:
        folium.GeoJson(res["gdf_buf"], style_function=lambda x: {"color": "gray", "dashArray": "5,5",
"fill": False}).add_to(m_ctx)
    if "gdf_vis" in res:
        folium.GeoJson(res["gdf_vis"], style_function=lambda x: {"color": "blue", "weight": 2,
"fillOpacity": 0.1}).add_to(m_ctx)

# --- POPUPS DETALLADOS Y SEGUROS ---
# 1. Pre-cálculo seguro de años de registro (Evita KeyError)
df_raw_safe = res.get("df_raw", pd.DataFrame())
if not df_raw_safe.empty:
    years_per_station =
df_raw_safe.groupby(Config.STATION_NAME_COL)[Config.YEAR_COL].nunique()
else:
    years_per_station = {}

# 2. Iteración sobre estaciones interpoladas
for _, r in res["df_int"].iterrows():
    nombre = r[Config.STATION_NAME_COL]
    ppt_val = r[Config.PRECIPITATION_COL]

# 3. Recuperación inteligente de Municipio

```

```

# Convertimos índices a string y buscamos 'muni' o 'ciud' sin importar mayúsculas
cols_r = [str(c) for c in r.index.tolist()]
col_muni = next((c for c in cols_r if "muni" in c.lower() or "municipio" in c.lower()), None)
muni_val = r[col_muni] if col_muni else "N/A"

# 4. Recuperación de Años y Altitud
n_anos = years_per_station.get(nombre, 0) if isinstance(years_per_station, dict) else
years_per_station.get(nombre, 0)
alt_val = r.get(Config.ALTITUDE_COL, 0)

# 5. Construcción del HTML
html_content = f"""
<div style='font-family:sans-serif; font-size:12px; min-width:150px'>
    <h5 style='margin:0; color:#2c3e50; border-bottom:1px solid #ccc; padding-
bottom:3px'>{nombre}</h5>
    <div style='margin-top:5px'>
         <b>Mun:</b> {muni_val}<br>
         <b>Alt:</b> {alt_val:.0f} msnm<br>
         <b>Ppt:</b> {ppt_val:.0f} mm<br>
         <b>Reg:</b> {n_anos} años
    </div>
</div>
"""

# 6. Creación del Popup y Marcador
iframe = folium.IFrame(html_content, width=220, height=130)
popup = folium.Popup(iframe, max_width=220)

folium.CircleMarker(
    [r.latitude, r.longitude],
    radius=5,
    color="darkred",
    fill=True,
    fill_color="#e74c3c",
    fill_opacity=0.9,
    popup=popup
).add_to(m_ctx)

st_folium(m_ctx, height=450, width="100%")

# --- INTERVENCIÓN 3: VISUALIZACIÓN CORREGIDA (CON PROTECCIÓN DE ERROR) ---

# 1. Intentamos recuperar los resultados
res = st.session_state.get("basin_res")

```

```
# 2. VERIFICACIÓN CRÍTICA: Solo intentamos dibujar si 'res' tiene datos  
if res is not None:
```

```
    st.markdown("---")  
    st.subheader("📊 Análisis Hidrológico Detallado")  
  
    # Recuperamos los resultados del diccionario de forma segura  
    hyp = res.get("hyp")  
    fdc = res.get("fdc")  
  
    col_curvas_1, col_curvas_2 = st.columns(2)  
  
    # --- A. VISUALIZACIÓN CURVA HIPSOMÉTRICA ---  
    with col_curvas_1:  
        st.markdown("** 📈 Curva Hipsométrica**")  
        if hyp:  
            import plotly.graph_objects as go # Aseguramos importación  
  
            fig_h = go.Figure()  
            fig_h.add_trace(go.Scatter(  
                x=hyp["area_percent"],  
                y=hyp["elevations"],  
                fill='tozeroY',  
                mode='lines',  
                line=dict(color='#2E86C1'),  
                name='Terreno'  
            ))  
            fig_h.update_layout(  
                xaxis_title="% Área Acumulada",  
                yaxis_title="Elevación (msnm)",  
                height=300,  
                margin=dict(l=0,r=0,t=30,b=0),  
                hovermode="x unified"  
            )  
            st.plotly_chart(fig_h, use_container_width=True)  
  
            st.caption(f"📐 **Modelo:** ${hyp.get('equation', 'N/A')}")  
            if hyp.get("source") == "Simulado":  
                st.caption("⚠ *Datos simulados (DEM no detectado)*")  
            else:  
                st.warning("⚠ Datos de elevación no disponibles.")  
  
    # --- B. VISUALIZACIÓN CURVA FDC ---  
    with col_curvas_2:  
        st.markdown("** 💧 Curva de Duración de Caudales (FDC)**")
```

```

if fdc and fdc.get("data") is not None:
    df_fdc = fdc["data"]

    fig_f = go.Figure()
    fig_f.add_trace(go.Scatter(
        x=df_fdc["Probabilidad Excedencia (%)"],
        y=df_fdc["Caudal (m³/s)"],
        mode='lines',
        line=dict(color='#27AE60', width=2),
        name='Caudal'
    ))
    fig_f.update_layout(
        xaxis_title="% Tiempo excedencia",
        yaxis_title="Caudal (m³/s)",
        yaxis_type="log",
        height=300,
        margin=dict(l=0,r=0,t=30,b=0)
    )
    st.plotly_chart(fig_f, use_container_width=True)

try:
    q_vals = df_fdc["Caudal (m³/s)"].values
    q95 = np.percentile(q_vals, 5)
    st.caption(f" 💧 **Caudal Ecológico (Q95):** {q95:.2f} m³/s")
    st.caption(f" 📈 **R² Ajuste:** {fdc.get('r_squared', 0):.2f}")
except: pass
else:
    st.info("⚠️ Faltan datos de lluvia para generar la curva FDC.")

else:
    # Si res es None (aún no se ha analizado nada), mostramos esto:
    st.info("👉 Seleccione una cuenca y haga clic en '⚡ Analizar Cuenca' para ver los detalles.")

```

```

# PESTAÑA DE PRONÓSTICO CLIMÁTICO (INDICES + GENERADOR)
# -----
def display_climate_forecast_tab(**kwargs):
    st.subheader("🌐 Pronóstico Climático & Fenómenos Globales")

    # Recuperamos los datos históricos pasados desde app.py
    df_enso = kwargs.get("df_enso")

    # Definimos las 4 pestañas solicitadas
    tab_hist, tab_iri_plumas, tab_iri_probs, tab_gen = st.tabs([
        [

```

```

        "📅 Historia Índices (ONI/SOI/IOD)",
        "🌐 Pronóstico Oficial (IRI)",
        "📊 Probabilidad Multimodelo",
        "⚙️ Generador Prophet",
    ]
)

# =====
# CARGA DE DATOS IRI (Comunes para tabs 2 y 3)
# =====
# Cargar datos desde archivos locales
json_plumas = fetch_iridata("enso_plumes.json")
json_probs = fetch_iridata("enso_cpc_prob.json") # Usamos CPC Probabilities

# --- CAJA INFORMATIVA (Extendida y Mejorada) ---
with st.expander():

    "ℹ️ Acerca de los Pronósticos IRI/CPC (Columbia University)", expanded=False
):
    st.markdown(
        """
Este módulo utiliza datos del **International Research Institute for Climate and Society (IRI)**.  

Los datos se actualizan mensualmente (aprox. el día 19) y representan el estándar global.

```

1. Definición:

El Pronóstico ENSO del IRI recopila predicciones de más de 20 instituciones científicas (NASA, NOAA, JMA, ECMWF, etc.).

2. Metodología:

Se basa en la región **Niño 3.4** (Pacífico Ecuatorial Central) y combina:

- * **🤖 Modelos Dinámicos:** Simulaciones físicas (ej. NCEP CFSv2). Mejores a largo plazo.
- * **📈 Modelos Estadísticos:** Proyecciones matemáticas. Eficientes a corto plazo.

3. Interpretación:

- * **📝 Pluma (Spaghetti):** Muestra la incertidumbre.
- * **línea Negra:** Promedio (Consenso).
- * **Umbráles:** **El Niño** ($\geq +0.5^{\circ}\text{C}$), **La Niña** ($\leq -0.5^{\circ}\text{C}$).
- * **📊 Probabilidades:** Porcentaje de certeza de cada evento por trimestre.

4. Impacto en Colombia:

- * **🔥 El Niño:** Sequías, altas temperaturas, menos lluvias.
- * **💧 La Niña:** Lluvias intensas, inundaciones, deslizamientos.

*** Definiciones:

Anomalías: Variaciones respecto de un valor medio u otro valor de referencia estadístico

Bimodal: Tener dos picos (o modos)

Boyante: A medida que el aire se calienta, se expande y se vuelve menos denso que el aire que está encima de él.

Convección: La transferencia de energía al mover la molécula calentada de un lugar a otro – tambien el ascenso del aire caliente que forma nubes cumuliformes y da lugar a precipitaciones

El Niño: "El niño niño", en referencia al nacimiento de Jesucristo.

Infrarrojo: Longitud de onda de la radiación más larga que la luz visible y asociada al "calor" emitido por un cuerpo.

Moche: Una civilización precolombina en el norte de Perú que existió desde aproximadamente el año 100 al 800 d.C.

Datos proxy del paleoclima: Información climática anterior a la invención de los instrumentos de monitoreo atmosférico

y derivada de indicadores químicos y biológicos conocidos en capas de hielo glacial, corales, sedimentos del fondo marino, anillos de árboles, etc.

precolombino: Se refiere a América del Sur y Central en el período anterior a la influencia europea.

gradiente de presión: El cambio en la presión del aire a lo largo de una distancia. Un gradiente más fuerte produce un flujo de aire más rápido de alta a baja presión.

Presión barométrica a nivel del mar: La fuerza ejercida por la atmósfera al nivel del mar (cero metros de altura) medida con un barómetro.

Estandarizado: Ajustado para que los valores de muestras con diferentes propiedades se puedan comparar entre sí

Expansión térmica: Calentar un fluido o gas no contenido aumenta su volumen al intentar mantener una presión constante.

En el océano, dado que solo la superficie es ilimitada, la expansión eleva el nivel del mar.

Termoclina: Zona bajo la superficie del océano donde el agua superficial se transforma en agua profunda, produciéndose una marcada disminución de su temperatura.

Tanto en la capa superficial (o mixta) como en las aguas profundas, la temperatura se mantiene relativamente constante con la profundidad.

Dentro de la termoclina, la temperatura del agua disminuye rápidamente de la superficial a la profunda.

Vientos alisios: Los vientos en los trópicos generalmente soplan de este a oeste (vientos del este).

Se llaman así porque su consistencia facilita la navegación y el comercio transoceánico.

Fuentes de información primaria: <https://www.ncei.noaa.gov/access/monitoring/enso/sst>

)

```
# =====
```

```
# PESTAÑA 1: HISTORIA
```

```
# =====
```

```
with tab_hist:
```

```
    st.markdown("#### 📈 Índices Climáticos Históricos")
```

```
    if df_enso is not None and not df_enso.empty:
```

```
        c1, _ = st.columns([1, 3])
```

```
        idx_sel = c1.selectbox(
```

```
            "Seleccione Índice:",
```

```
            [Config.ENSO_ONI_COL, Config.SOI_COL, Config.IOD_COL],
```

```
)
```

```

if idx_sel in df_enso.columns:
    d = df_enso.dropna(subset=[idx_sel, Config.DATE_COL]).sort_values(
        Config.DATE_COL
    )
    if idx_sel == Config.ENSO_ONI_COL:
        st.plotly_chart(
            create_enso_chart(d),
            use_container_width=True,
            key="chart_oni_hist",
        )
    else:
        fig_simple = px.line(
            d,
            x=Config.DATE_COL,
            y=idx_sel,
            title=f"Evolución Histórica: {idx_sel}",
        )
        fig_simple.add_hline(
            y=0, line_width=1, line_color="black", opacity=0.5
        )
        st.plotly_chart(
            fig_simple,
            use_container_width=True,
            key=f"chart_{idx_sel}_hist",
        )
    else:
        st.warning(
            f"La columna '{idx_sel}' no se encuentra en la base de datos."
        )
else:
    st.warning("No hay datos históricos cargados (df_enso).")

# =====
# PESTAÑA 2: PRONÓSTICO OFICIAL (PLUMAS)
# =====
with tab_iri_plumas:
    if json_plumas:
        # Mensaje de Fecha
        try:
            last_year = json_plumas["years"][-1]["year"]
            last_month_idx = json_plumas["years"][-1]["months"][-1]["month"]
            meses = [
                "Ene",
                "Feb",
                "Mar",
            ]

```

```

    "Abr",
    "May",
    "Jun",
    "Jul",
    "Ago",
    "Sep",
    "Oct",
    "Nov",
    "Dic",
]
st.info(
    f" 📈 Pronóstico de Plumas actualizado a: **{meses[last_month_idx]} {last_year}**"
)
except:
    st.info(" 📈 Pronóstico Mensual Oficial (Plumas)")

st.markdown("#### 🎨 Modelos de Predicción (Plumas)")
data_plume = process_iri_plume(json_plumas)

if data_plume:
    fig_plume = go.Figure()

    # Colección de valores para calcular promedio
    all_values = []

    # Variables para controlar la leyenda (que aparezca solo una vez por tipo)
    show_dyn_legend = True
    show_stat_legend = True

    for model in data_plume["models"]:
        is_dyn = model["type"] == "Dynamical"
        color = (
            "rgba(100, 149, 237, 0.6)"
            if is_dyn
            else "rgba(255, 165, 0, 0.6)"
        ) # Azul/Naranja

        # Nombre genérico para la leyenda
        legend_group = (
            "Modelos Dinámicos" if is_dyn else "Modelos Estadísticos"
        )

        # Control de visualización en leyenda (solo el primero de cada grupo)
        show_in_legend = False
        if is_dyn and show_dyn_legend:

```

```

show_in_legend = True
show_dyn_legend = False
elif not is_dyn and show_stat_legend:
    show_in_legend = True
    show_stat_legend = False

# Guardar valores para promedio
vals = model["values"][: len(data_plume["seasons"])]
all_values.append(vals)

fig_plume.add_trace(
    go.Scatter(
        x=data_plume["seasons"],
        y=model["values"],
        mode="lines",
        name=legend_group, # Nombre agrupado para la leyenda
        legendgroup=legend_group, # Agrupar interactividad
        showlegend=show_in_legend,
        line=dict(color=color, width=1.5),
        opacity=0.7,
        hovertemplate=f"<b>{model['name']}</b><br>%{{y:.2f}} °C<extra></extra>", # Nombre real
en hover
)
)
)

# --- CÁLCULO DE PROMEDIO MULTIMODELO ---
try:
    max_len = len(data_plume["seasons"])
    clean_matrix = []
    for v in all_values:
        row = [float(x) if x is not None else np.nan for x in v]
        if len(row) < max_len:
            row += [np.nan] * (max_len - len(row))
        clean_matrix.append(row)

    avg_vals = np.nanmean(np.array(clean_matrix), axis=0)

    fig_plume.add_trace(
        go.Scatter(
            x=data_plume["seasons"],
            y=avg_vals,
            mode="lines+markers",
            name="PROMEDIO MULTIMODELO",
            line=dict(color="black", width=4),
            marker=dict(size=6, color="black"),
            showlegend=True,

```

```

        )
    )
except Exception as e:
    st.warning(f"Nota: No se pudo calcular el promedio ({e})")

# Umbrales
fig_plume.add_hline(
    y=0.5,
    line_dash="dash",
    line_color="red",
    annotation_text="El Niño (+0.5)",
)
fig_plume.add_hline(
    y=-0.5,
    line_dash="dash",
    line_color="blue",
    annotation_text="La Niña (-0.5"),
)

fig_plume.update_layout(
    title="Anomalía SST Niño 3.4 (Spaghetti Plot)",
    height=550,
    xaxis_title="Trimestres Móviles",
    yaxis_title="Anomalía SST (°C)",
    hovermode="x unified",
    legend=dict(
        orientation="h", yanchor="bottom", y=1.02, xanchor="right", x=1
    ),
)
st.plotly_chart(
    fig_plume, use_container_width=True, key="chart_iri_plume"
)
else:
    st.warning("Error al procesar la estructura del archivo de plumas.")
else:
    st.error("⚠️ No se encontró el archivo `enso_plumes.json` en `data/iri/`.")

# =====
# PESTAÑA 3: PROBABILIDAD MULTIMODELO
# =====
with tab_iri_probs:
    if json_probs:
        # Mensaje de Fecha para Probabilidades
        try:
            last_year = json_probs["years"][-1]["year"]
            last_month_idx = json_probs["years"][-1]["months"][-1]["month"]

```

```

meses = [
    "Ene",
    "Feb",
    "Mar",
    "Abr",
    "May",
    "Jun",
    "Jul",
    "Ago",
    "Sep",
    "Oct",
    "Nov",
    "Dic",
]
st.info(
    f" 📈 Pronóstico de Probabilidades (Consenso CPC/IRI) actualizado a: **{meses[last_month_idx]}**
{last_year}**"
)
except:
    pass

st.markdown("#### 🎨 Probabilidad de Eventos (El Niño/La Niña/Neutral)")
df_probs = process_iri_probabilities(json_probs)

if df_probs is not None and not df_probs.empty:
    try:
        # Normalización de columnas
        df_probs.columns = [str(c).strip() for c in df_probs.columns]

        # Identificar columna de tiempo
        col_tiempo = None
        for nombre in ["Trimestre", "Season", "season", "SEASON"]:
            if nombre in df_probs.columns:
                col_tiempo = nombre
                break

        if not col_tiempo and len(df_probs.columns) > 0:
            col_tiempo = df_probs.columns[0]

        if col_tiempo:
            if col_tiempo != "Trimestre":
                df_probs.rename(
                    columns={col_tiempo: "Trimestre"}, inplace=True
                )
    
```

```

# Melt seguro
# Buscamos columnas de eventos (ignorando mayúsculas/minúsculas)
cols_val = [c for c in df_probs.columns if c != "Trimestre"]

df_melt = df_probs.melt(
    id_vars="Trimestre",
    value_vars=cols_val,
    var_name="Evento",
    value_name="Probabilidad",
)

# Normalización para colores
df_melt["Evento_Norm"] = (
    df_melt["Evento"]
    .astype(str)
    .str.lower()
    .str.replace(" ", "")
)

# Mapeo de colores
color_map = {
    "elnino": "#FF4B4B",
    "el niño": "#FF4B4B",
    "lanina": "#1C83E1",
    "la niña": "#1C83E1",
    "neutral": "#808495",
}

def get_color(evt_norm):
    for key, color in color_map.items():
        if key in evt_norm:
            return color
    return "gray"

df_melt["Color"] = df_melt["Evento_Norm"].apply(get_color)

fig_probs = px.bar(
    df_melt,
    x="Trimestre",
    y="Probabilidad",
    color="Evento",
    color_discrete_map={
        evt: get_color(evt.lower().replace(" ", ""))
        for evt in df_melt["Evento"].unique()
    },
    text="Probabilidad",
)

```

```

        barmode="group",
    )
fig_probs.update_traces(
    texttemplate="%{text:.0f}", textposition="outside"
)
fig_probs.update_layout(
    height=500,
    yaxis=dict(range=[0, 105]),
    xaxis_title="Trimestre Pronosticado",
    legend=dict(
        orientation="h",
        yanchor="bottom",
        y=1.02,
        xanchor="right",
        x=1,
    ),
)
st.plotly_chart(
    fig_probs, use_container_width=True, key="chart_iri_probs"
)
else:
    st.error("No se pudo identificar la columna de tiempo.")
except Exception as e:
    st.error(f"Error generando gráfico: {e}")
else:
    st.warning("DataFrame de probabilidades vacío.")
else:
    st.error("⚠️ No se encontró el archivo `enso_cpc_prob.json` en `data/iri/`.")

# =====
# PESTAÑA 4: PROPHET
# =====

with tab_gen:
    st.markdown("#### 📈 Generador Prophet (Proyección Estadística Local)")
    indices = {}
    if df_enso is not None:
        cols_map = {
            Config.ENSO_ONI_COL: "ONI",
            Config.SOI_COL: "SOI",
            Config.IOD_COL: "IOD",
        }
        for col, name in cols_map.items():
            if col in df_enso.columns:
                indices[name] = (
                    df_enso[[Config.DATE_COL, col]]

```

```

    .rename(columns={Config.DATE_COL: "ds", col: "y"})
    .dropna()
)

if indices:
    c_sel, c_hor = st.columns(2)
    sel_idx = c_sel.selectbox("Índice a proyectar:", list(indices.keys()))
    hor = c_hor.slider("Meses a futuro:", 6, 60, 24)

    if st.button("Generar Proyección Prophet"):
        try:
            with st.spinner(f"Entrenando modelo para {sel_idx}..."):
                m = Prophet()
                m.fit(indices[sel_idx])
                fut = m.make_future_dataframe(periods=hor, freq="MS")
                fc = m.predict(fut)

                fig = px.line(
                    fc,
                    x="ds",
                    y="yhat",
                    title=f"Proyección {sel_idx} (Prophet)",
                )
                fig.add_trace(
                    go.Scatter(
                        x=fc["ds"],
                        y=fc["yhat_upper"],
                        mode="lines",
                        line=dict(width=0),
                        showlegend=False,
                        hoverinfo="skip",
                    )
                )
                fig.add_trace(
                    go.Scatter(
                        x=fc["ds"],
                        y=fc["yhat_lower"],
                        mode="lines",
                        line=dict(width=0),
                        fill="tonexty",
                        fillcolor="rgba(68, 68, 68, 0.1)",
                        showlegend=False,
                        hoverinfo="skip",
                    )
                )

```

```

        st.plotly_chart(
            fig, use_container_width=True, key="chart_prophet"
        )
    except Exception as e:
        st.error(f"Error en Prophet: {e}")
    else:
        st.warning("No hay datos suficientes para generar proyecciones.")

# -----



def display_trends_and_forecast_tab(**kwargs):
    st.subheader("📈 Tendencias y Pronósticos (Series de Tiempo)")

    # Recuperar datos
    df_monthly = kwargs.get("df_monthly_filtered")
    stations = kwargs.get("stations_for_analysis")
    df_enso = kwargs.get("df_enso")

    if not stations or df_monthly is None or df_monthly.empty:
        st.warning("Seleccione estaciones en el panel lateral.")
        return

    # 1. SELECTOR GLOBAL DE SERIE
    st.markdown("##### Configuración de la Serie de Tiempo")
    mode_fc = st.radio(
        "Modo de Análisis:",
        ["Estación Individual", "Serie Regional (Promedio)"],
        horizontal=True,
        key="fc_mode_selector",
    )

    ts_clean = None
    station_name_title = ""

    try:
        if mode_fc == "Estación Individual":
            selected_station = st.selectbox(
                "Seleccionar Estación:", stations, key="trend_st"
            )
        if selected_station:
            station_data = (
                df_monthly[df_monthly[Config.STATION_NAME_COL] == selected_station]
                .sort_values(Config.DATE_COL)
                .set_index(Config.DATE_COL)
            )

```

```

        )
        full_idx = pd.date_range(
            start=station_data.index.min(),
            end=station_data.index.max(),
            freq="MS",
        )
        ts_clean = (
            station_data[Config.PRECIPITATION_COL]
            .reindex(full_idx)
            .interpolate(method="time")
            .dropna()
        )
        station_name_title = selected_station
    else:
        station_name_title = "Serie Regional (Promedio)"
        reg_data = df_monthly.groupby(Config.DATE_COL)[
            Config.PRECIPITATION_COL
        ].mean()
        full_idx = pd.date_range(
            start=reg_data.index.min(), end=reg_data.index.max(), freq="MS"
        )
        ts_clean = reg_data.reindex(full_idx).interpolate(method="time").dropna()

    if ts_clean is None or len(ts_clean) < 24:
        st.error(f"Datos insuficientes (<24 meses) para {station_name_title}.")
        return

except Exception as e:
    st.error(f"Error preparando los datos: {e}")
    return

# --- PREPARACIÓN DE REGRESORES EXTERNOS ---
avail_regs = []
regressors_df = None

if df_enso is not None and not df_enso.empty:
    potential_regs = [
        c
        for c in df_enso.columns
        if c in [Config.ENSO_ONI_COL, Config.SOI_COL, Config.IOD_COL]
    ]
    avail_regs = potential_regs
    if avail_regs:
        temp_enso = df_enso.copy()
        if temp_enso[Config.DATE_COL].dtype == "object":
            temp_enso[Config.DATE_COL] = pd.to_datetime(temp_enso[Config.DATE_COL])

```

```

regressors_df = (
    temp_enso.set_index(Config.DATE_COL)[avail_regs]
    .resample("MS")
    .mean()
    .interpolate(method="time")
)

# 2. PESTAÑAS (Mapa de Riesgo MOVIDO a Clima Futuro)
tabs = st.tabs(
    [
        "📊 Tendencia Mann-Kendall",
        "🔍 Descomposición",
        "⌚ Autocorrelación",
        "🧠 SARIMA",
        "🔮 Prophet",
        "⚖️ Comparación Modelos",
    ]
)

```

--- TAB 1: TENDENCIA MANN-KENDALL (INDEPENDIENTE Y RESTAURADA) ---

```

with tabs[0]:
    st.markdown("#### Análisis de Tendencia no Paramétrica (Mann-Kendall)")
    st.caption(f"Evaluando serie: **{station_name_title}**")

    try:
        # Mann-Kendall Test Original
        res = mk.original_test(ts_clean)

        # Métricas Clave
        c1, c2, c3 = st.columns(3)

        # Interpretación visual de la tendencia
        trend_icon = "—" if res.trend == "increasing" else "↗️"
        if res.trend == "increasing":
            trend_icon = "↗️ (Aumento)"
        elif res.trend == "decreasing":
            trend_icon = "↘️ (Disminución)"

        c1.metric("Dirección Tendencia", trend_icon)
        c2.metric("Pendiente (Sen)", f"{res.slope:.3f} mm/mes")

        # Interpretación de Significancia
        is_significant = res.p < 0.05
        sig_text = (
            "Significativo (Confianza > 95%)"
        )
    
```

```

        if is_significant
        else "No Significativo"
    )
c3.metric(
    "Significancia Estadística",
    sig_text,
    delta=f"p-value: {res.p:.4f}",
    delta_color="normal" if is_significant else "off",
)
# Gráfico Visual
df_plot = ts_clean.reset_index()
df_plot.columns = ["Fecha", "Precipitación"]

# Línea de tendencia calculada (y = mx + b)
# Aproximación visual usando índices numéricos para la pendiente
x_nums = np.arange(len(df_plot))
y_trend = res.slope * x_nums + res.intercept

fig = go.Figure()
fig.add_trace(
    go.Scatter(
        x=df_plot["Fecha"],
        y=df_plot["Precipitación"],
        mode="lines",
        name="Serie Histórica",
        line=dict(color="gray", width=1),
    )
)
fig.add_trace(
    go.Scatter(
        x=df_plot["Fecha"],
        y=y_trend,
        mode="lines",
        name="Tendencia de Sen",
        line=dict(color="red", width=3, dash="dash"),
    )
)
fig.update_layout(
    title="Ajuste de Tendencia (Theil-Sen)", hovermode="x unified"
)
st.plotly_chart(fig)

with st.expander("Ver detalles estadísticos completos"):
    st.write(res)

```

```

except Exception as e:
    st.error(f"No se pudo calcular la tendencia: {e}")

# --- TAB 2: DESCOMPOSICIÓN ---
with tabs[1]:
    try:
        decomp = seasonal_decompose(ts_clean, model="additive", period=12)
        fig = go.Figure()
        fig.add_trace(
            go.Scatter(x=ts_clean.index, y=decomp.trend, name="Tendencia (Ciclo)")
        )
        fig.add_trace(
            go.Scatter(x=ts_clean.index, y=decomp.seasonal, name="Estacionalidad")
        )
        fig.add_trace(
            go.Scatter(
                x=ts_clean.index, y=decomp.resid, name="Residuo", mode="markers"
            )
        )
        fig.update_layout(title="Descomposición Estacional (Aditiva)", height=500)
        st.plotly_chart(fig)
    except:
        st.warning("Error en descomposición (datos insuficientes o discontinuos).")

# --- TAB 3: AUTOCORRELACIÓN ---
with tabs[2]:
    try:
        from statsmodels.tsa.stattools import acf, pacf

        nlags = min(24, len(ts_clean) // 2 - 1)
        lag_acf = acf(ts_clean, nlags=nlags)
        lag_pacf = pacf(ts_clean, nlags=nlags)
        c1, c2 = st.columns(2)
        c1.plotly_chart(
            px.bar(x=range(len(lag_acf)), y=lag_acf, title="ACF (Autocorrelación)")
        )
        c2.plotly_chart(
            px.bar(x=range(len(lag_pacf)), y=lag_pacf, title="PACF (Parcial)")
        )
    except:
        pass

# --- TAB 4: SARIMA ---
with tabs[3]:
    st.markdown("#### Pronóstico SARIMA")

```

```

sel_regs = st.multiselect(
    "Usar Regresor Externo (ONI/SOI/IOD):", avail_regs, key="sarima_regs_sel"
)

final_reg_df = None
if sel_regs and regressors_df is not None:
    final_reg_df = (
        regressors_df[sel_regs]
        .reindex(ts_clean.index)
        .fillna(method="ffill")
        .fillna(method="bfill")
    )

horizon = st.slider("Horizonte (Meses):", 12, 48, 12, key="h_sarima")

if st.button("Calcular SARIMA"):
    from modules.forecasting import generate_sarima_forecast

    with st.spinner("Calculando SARIMA..."):
        try:
            ts_in = ts_clean.reset_index()
            t_size = max(1, min(12, int(len(ts_clean) * 0.2)))
            _, fc, ci, met, _ = generate_sarima_forecast(
                ts_in,
                order=(1, 1, 1),
                seasonal_order=(1, 1, 1, 12),
                horizon=horizon,
                test_size=t_size,
                regressors=final_reg_df,
            )
            st.success(f"Modelo Ajustado. RMSE: {met['RMSE']:.1f}")
        except Exception as e:
            st.error(f"Error: {e}")

    fig = go.Figure()
    fig.add_trace(
        go.Scatter(x=ts_clean.index, y=ts_clean, name="Histórico")
    )
    fig.add_trace(
        go.Scatter(
            x=fc.index, y=fc, name="Pronóstico", line=dict(color="red")
        )
    )
    if not ci.empty:
        fig.add_trace(
            go.Scatter(
                x=pd.concat([
                    pd.Series(ci.index), pd.Series(ci.index)[::-1]
                ])
            )
        )

```

```

        ),
        y=pd.concat([ci.iloc[:, 0], ci.iloc[:, 1][:-1]]),
        fill="toself",
        fillcolor="rgba(255,0,0,0.1)",
        line=dict(color="rgba(255,255,255,0)"),
        name="Confianza 95%",
    )
)
st.plotly_chart(fig)
st.session_state["sarima_res"] = fc
except Exception as e:
    st.error(f"Error SARIMA: {e}")

# --- TAB 5: PROPHET ---
with tabs[4]:
    st.markdown("#### Pronóstico Prophet")
    sel_regs_p = st.multiselect(
        "Usar Regresor Externo (ONI/SOI/IOD):", avail_regs, key="prophet_regs_sel"
    )

    final_reg_p = None
    horizon_p = st.slider("Horizonte (Meses):", 12, 48, 12, key="h_prophet")

    if sel_regs_p and regressors_df is not None:
        try:
            last_date = ts_clean.index.max()
            future_dates = pd.date_range(
                start=regressors_df.index.min(),
                periods=len(regressors_df) + horizon_p + 12,
                freq="MS",
            )
            extended_regs = (
                regressors_df[sel_regs_p]
                .reindex(future_dates)
                .fillna(method="ffill")
                .fillna(method="bfill")
            )
            final_reg_p = extended_regs.reset_index().rename(
                columns={"index": "ds", Config.DATE_COL: "ds"}
            )
        except:
            if "ds" not in final_reg_p.columns and "date" in final_reg_p.columns:
                final_reg_p.rename(columns={"date": "ds"}, inplace=True)
            elif "ds" not in final_reg_p.columns:
                final_reg_p.rename(
                    columns={final_reg_p.columns[0]: "ds"}, inplace=True
                )

```

```

except Exception as e:
    st.warning(f"No se pudieron preparar regresores: {e}")
    final_reg_p = None

if st.button("Calcular Prophet"):
    from modules.forecasting import generate_prophet_forecast

    with st.spinner("Calculando Prophet..."):
        try:
            ts_in = ts_clean.reset_index()
            ts_in.columns = ["ds", "y"]
            t_size = max(1, min(12, int(len(ts_clean) * 0.2)))
            _, fc, met = generate_prophet_forecast(
                ts_in, horizon_p, test_size=t_size, regressors=final_reg_p
            )
            st.success(f"Modelo Ajustado. RMSE: {met['RMSE']:.1f}")

            fig = go.Figure()
            fig.add_trace(
                go.Scatter(x=ts_clean.index, y=ts_clean, name="Histórico")
            )
            fig.add_trace(
                go.Scatter(
                    x=fc["ds"],
                    y=fc["yhat"],
                    name="Pronóstico",
                    line=dict(color="green"),
                )
            )
            fig.add_trace(
                go.Scatter(
                    x=pd.concat([fc["ds"], fc["ds"][:-1]]),
                    y=pd.concat([fc["yhat_upper"], fc["yhat_lower"][:-1]]),
                    fill="toself",
                    fillcolor="rgba(0,255,0,0.1)",
                    line=dict(color="rgba(255,255,255,0)"),
                    name="Confianza",
                )
            )
            st.plotly_chart(fig)
            st.session_state["prophet_res"] = fc[["ds", "yhat"]].set_index(
                "ds"
            )["yhat"]
        except Exception as e:
            st.error(f"Error Prophet: {e}")

```

```

# --- TAB 6: COMPARACIÓN ---
with tabs[5]:
    s, p = st.session_state.get("sarima_res"), st.session_state.get("prophet_res")
    if s is not None and p is not None:
        fig = go.Figure()
        fig.add_trace(
            go.Scatter(x=s.index, y=s, name="SARIMA", line=dict(color="red"))
        )
        fig.add_trace(
            go.Scatter(x=p.index, y=p, name="Prophet", line=dict(color="green"))
        )
        fig.update_layout(title="Comparativa de Modelos")
        st.plotly_chart(fig)
    else:
        st.info("Ejecute ambos modelos para comparar.")

def display_anomalies_tab(
    df_long, df_monthly_filtered, stations_for_analysis, **kwargs
):
    st.subheader("⚠️ Análisis de Anomalías de Precipitación")

    df_enso = kwargs.get("df_enso")

    if df_monthly_filtered is None or df_monthly_filtered.empty:
        st.warning("No hay datos de precipitación filtrados.")
        return

    # 1. CONFIGURACIÓN
    st.markdown("#### Configuración del Análisis")
    col_conf1, col_conf2 = st.columns([1, 2])

    with col_conf1:
        reference_method = st.radio(
            "Calcular anomalía con respecto a:",
            [
                "El promedio de todo el período",
                "Una Normal Climatológica (período base fijo)",
            ],
            key="anomaly_ref_method",
        )

        start_base, end_base = None, None

        if reference_method == "Una Normal Climatológica (período base fijo)":
            with col_conf2:

```

```

all_years = sorted(df_long[Config.YEAR_COL].unique())
if not all_years:
    st.error("No hay datos anuales disponibles.")
    return

min_y, max_y = all_years[0], all_years[-1]

def_start = 1991 if 1991 in all_years else min_y
def_end = 2020 if 2020 in all_years else max_y

c_start, c_end = st.columns(2)
start_base = c_start.selectbox(
    "Año Inicio Período Base:", all_years, index=all_years.index(def_start)
)
end_base = c_end.selectbox(
    "Año Fin Período Base:", all_years, index=all_years.index(def_end)
)

if start_base > end_base:
    st.error("El año de inicio debe ser menor al año de fin.")
    return

# 2. CÁLCULO
with st.spinner("Calculando anomalías..."):

    # A. Definir datos de referencia
    if reference_method == "Una Normal Climatológica (período base fijo)":
        mask_base = (df_long[Config.YEAR_COL] >= start_base) & (
            df_long[Config.YEAR_COL] <= end_base
        )
        df_reference = df_long[mask_base]
        ref_text = f"Normal {start_base}-{end_base}"
    else:
        df_reference = df_long
        ref_text = "Promedio Histórico Total"

    # B. Serie regional mensual (promedio de estaciones seleccionadas)
    df_regional = (
        df_monthly_filtered.groupby(Config.DATE_COL)[Config.PRECIPITATION_COL]
        .mean()
        .reset_index()
    )
    df_regional[Config.MONTH_COL] = df_regional[Config.DATE_COL].dt.month

    # C. Climatología regional
    stations_list = df_monthly_filtered[Config.STATION_NAME_COL].unique()
    df_ref_stations = df_reference[

```

```

        df_reference[Config.STATION_NAME_COL].isin(stations_list)
    ]
climatology = (
    df_ref_stations.groupby(Config.MONTH_COL)[Config.PRECIPITATION_COL]
    .mean()
    .reset_index()
)
climatology.rename(
    columns={Config.PRECIPITATION_COL: "clim_mean"}, inplace=True
)

# D. Unir y Restar
df_anom = pd.merge(df_regional, climatology, on=Config.MONTH_COL, how="left")
df_anom["anomalia"] = df_anom[Config.PRECIPITATION_COL] - df_anom["clim_mean"]

df_anom["color"] = np.where(df_anom["anomalia"] >= 0, "blue", "red")

# 3. VISUALIZACIÓN
tab_ts, tab_enso, tab_table = st.tabs(
    ["Gráfico de Anomalías", "Anomalías por Fase ENSO", "Tabla de Eventos Extremos"]
)

# --- A. SERIE TEMPORAL ---
with tab_ts:
    st.markdown(f"##### Anomalías Mensuales (Ref: {ref_text})")
    fig = go.Figure()
    fig.add_trace(
        go.Bar(
            x=df_anom[Config.DATE_COL],
            y=df_anom["anomalia"],
            marker_color=df_anom["color"],
            name="Anomalía",
        )
    )
    fig.update_layout(
        yaxis_title="Anomalía (mm)",
        xaxis_title="Fecha",
        height=500,
        showlegend=False,
    )
    fig.add_hline(y=0, line_color="black", line_width=1)
    st.plotly_chart(fig)

csv = df_anom.to_csv(index=False).encode("utf-8")
st.download_button(
    "⬇️ Descargar Anomalías (CSV)", csv, "anomalias.csv", "text/csv"
)

```

```

    )

# --- B. DISTRIBUCIÓN POR FASE ENSO ---
with tab_enso:
    st.subheader("Distribución por Fase Climática")
    if df_enso is None or df_enso.empty:
        st.warning("No hay datos ENSO.")
    else:
        c_idx, _ = st.columns([1, 2])
        idx_name = c_idx.selectbox("Índice:", ["ONI (El Niño)", "SOI", "IOD"])
        idx_col_map = {
            "ONI (El Niño)": Config.ENSO_ONI_COL,
            "SOI": Config.SOI_COL,
            "IOD": Config.IOD_COL,
        }
        target_idx_col = idx_col_map[idx_name]

        if target_idx_col in df_enso.columns:
            enso_clean = df_enso.copy()
            # Parseo seguro de fechas
            if enso_clean[Config.DATE_COL].dtype == "object":
                enso_clean[Config.DATE_COL] = enso_clean[Config.DATE_COL].apply(
                    parse_spanish_date
                )
            else:
                enso_clean[Config.DATE_COL] = pd.to_datetime(
                    enso_clean[Config.DATE_COL], errors="coerce"
                )

            df_merged = pd.merge(
                df_anom,
                enso_clean[[Config.DATE_COL, target_idx_col]],
                on=Config.DATE_COL,
                how="inner",
            )

            if not df_merged.empty:
                if idx_name == "ONI (El Niño)":
                    conds = [
                        df_merged[target_idx_col] >= 0.5,
                        df_merged[target_idx_col] <= -0.5,
                    ]
                    choices = ["El Niño", "La Niña"]
                    colors = {
                        "El Niño": "#d62728",
                        "La Niña": "#1f77b4",
                    }

```

```

        "Neutral": "lightgrey",
    }
elif idx_name == "SOI":
    conds = [
        df_merged[target_idx_col] <= -7,
        df_merged[target_idx_col] >= 7,
    ]
    choices = ["El Niño", "La Niña"]
    colors = {
        "El Niño": "#d62728",
        "La Niña": "#1f77b4",
        "Neutral": "lightgrey",
    }
else:
    conds = [
        df_merged[target_idx_col] >= 0.4,
        df_merged[target_idx_col] <= -0.4,
    ]
    choices = ["Positivo", "Negativo"]
    colors = {
        "Positivo": "#d62728",
        "Negativo": "#1f77b4",
        "Neutral": "lightgrey",
    }

df_merged["Fase"] = np.select(conds, choices, default="Neutral")

fig_enso = px.box(
    df_merged,
    x="Fase",
    y="anomalia",
    color="Fase",
    color_discrete_map=colors,
    points="all",
    title=f"Anomalías según Fase {idx_name}",
    category_orders={"Fase": choices + ["Neutral"]},
)
fig_enso.update_layout(
    height=600, showlegend=False, yaxis_title="Anomalía (mm)"
)
fig_enso.add_hline(
    y=0, line_width=1, line_color="black", line_dash="dot"
)
st.plotly_chart(fig_enso, use_container_width=True)
else:
    st.warning("No hay datos coincidentes.")

```

```

else:
    st.error(f"Columna {target_idx_col} no encontrada.")

# --- C. TABLA DE EXTREMOS (CORREGIDA) ---
with tab_table:
    st.subheader("Eventos Extremos")

    # CORRECCIÓN: Usar variables de Config en lugar de strings fijos
    cols_to_select = [
        Config.DATE_COL,
        Config.PRECIPITATION_COL,
        "clim_mean",
        "anomalia",
    ]
    cols_rename = ["Fecha", "Ppt Real", "Ppt Normal", "Diferencia"]

    c1, c2 = st.columns(2)
    with c1:
        st.markdown(*** ⚫ Top 10 Meses Más Secos***)
        driest = df_anom.nsmallest(10, "anomalia")[cols_to_select]
        driest.columns = cols_rename
        driest["Fecha"] = driest["Fecha"].dt.strftime("%Y-%m")
        st.dataframe(
            driest.style.format(
                "{:.1f}", subset=["Ppt Real", "Ppt Normal", "Diferencia"]
            ),
        )

    with c2:
        st.markdown(*** ⚪ Top 10 Meses Más Húmedos***)
        wettest = df_anom.nlargest(10, "anomalia")[cols_to_select]
        wettest.columns = cols_rename
        wettest["Fecha"] = wettest["Fecha"].dt.strftime("%Y-%m")
        st.dataframe(
            wettest.style.format(
                "{:.1f}", subset=["Ppt Real", "Ppt Normal", "Diferencia"]
            ),
        )

# FUNCIÓN ESTADÍSTICAS (REVISADA Y MEJORADA)
# =====
def display_stats_tab(df_long, df_anual_melted, gdf_stations, **kwargs):
    st.subheader("📊 Estadísticas Hidrológicas Detalladas")

```

```

# Validación de datos
if df_long is None or df_long.empty:
    st.warning("No hay datos mensuales disponibles para calcular estadísticas.")
    return

# Definición de Pestañas Internas
# Agregamos la pestaña "Síntesis (Récords)" que creamos antes
tab_desc, tab_matriz, tab_sintesis = st.tabs(
    [
        "📋 Resumen Descriptivo",
        "🗓️ Matriz de Disponibilidad",
        "🏆 Síntesis de Récords",
    ]
)

# --- PESTAÑA 1: RESUMEN DESCRIPTIVO ---
with tab_desc:
    st.markdown("##### Estadísticas Descriptivas por Estación (Mensual)")

    # Agrupar y calcular estadísticas básicas
    stats_df = df_long.groupby(Config.STATION_NAME_COL)[
        Config.PRECIPITATION_COL
    ].describe()

    # Añadir suma total histórica (útil para ver volumen total registrado)
    sum_total = df_long.groupby(Config.STATION_NAME_COL)[
        Config.PRECIPITATION_COL
    ].sum()
    stats_df["Total Histórico (mm)"] = sum_total

    # Formatear y mostrar
    st.dataframe(stats_df.style.format("{:.1f}"))

    # Botón de descarga
    st.download_button(
        "📥 Descargar Estadísticas (CSV)",
        stats_df.to_csv().encode("utf-8"),
        "estadisticas_precipitacion.csv",
        "text/csv",
    )

# --- PESTAÑA 2: MATRIZ DE DISPONIBILIDAD ---
with tab_matriz:
    st.markdown("##### Disponibilidad de Datos (Mapa de Calor)")
    st.info(

```

```

    "Muestra la densidad de registros por mes. Color más oscuro = Más datos."
)

try:
    # --- CORRECCIÓN MATRIZ ---
    # Copiamos para no afectar el original
    df_matrix = df_long.copy()

    # Forzamos la creación de una columna 'date' compatible con Pandas
    # Asumiendo que Config.YEAR_COL y Config.MONTH_COL son tus columnas de año y mes
    df_matrix["date"] = pd.to_datetime(
        dict(
            year=df_matrix[Config.YEAR_COL],
            month=df_matrix[Config.MONTH_COL],
            day=1,
        )
    )

    matrix = df_matrix.pivot_table(
        index=df_matrix["date"].dt.year,
        columns=df_matrix["date"].dt.month,
        values=Config.PRECIPITATION_COL,
        aggfunc="count",
    ).fillna(0)

    # Mapa de calor semáforo
    fig_matrix = px.imshow(
        matrix,
        labels=dict(x="Mes", y="Año", color="N° Registros"),
        x=[
            "Ene",
            "Feb",
            "Mar",
            "Abr",
            "May",
            "Jun",
            "Jul",
            "Ago",
            "Sep",
            "Oct",
            "Nov",
            "Dic",
        ],
        title="Matriz de Densidad de Datos (Semáforo)",
        color_continuous_scale="RdYIGn", # Rojo a Verde
        aspect="auto",
    )

```

```

        )
fig_matrix.update_layout(height=600)
st.plotly_chart(fig_matrix, use_container_width=True)

except Exception as e:
    st.warning(f"No se pudo generar la matriz: {e}")

# --- PESTAÑA 3: SÍNTESIS (NUEVA) ---
with tab_sintesis:
    # Llamamos a la función que creamos en el paso anterior
    # Asegúrate de que esta función exista en el mismo archivo o esté importada
    display_statistics_summary_tab(df_long, df_anual_melted, gdf_stations)

def display_correlation_tab(**kwargs):
    st.subheader("🔗 Análisis de Correlación")

    # Recuperar datos
    df_monthly = kwargs.get("df_monthly_filtered")
    df_enso = kwargs.get("df_enso")

    # Validaciones
    if df_monthly is None or df_monthly.empty:
        st.warning("Faltan datos de precipitación para el análisis.")
        return

    # Crear pestañas
    tab1, tab2 = st.tabs(["Fenómenos Globales (ENSO)", "Matriz entre Estaciones"])

    # -----
    # PESTAÑA 1: RELACIÓN LLUVIA REGIONAL VS ENSO (ONI)
    # -----
    with tab1:
        if df_enso is None or df_enso.empty:
            st.warning("No se han cargado datos del índice ENSO.")
        else:
            st.markdown(
                "##### Correlación: Índice Oceánico El Niño (ONI) vs. Precipitación"
            )
            st.info(
                "Analiza cómo la temperatura superficial del mar afecta la lluvia en la zona seleccionada."
            )

    try:
        # 1. Preparar copias de datos para no alterar los originales
        ppt_data = df_monthly.copy()

```

```

enso_data = df_enso.copy()

# 2. Asegurar formato de fecha en Precipitación
ppt_data[Config.DATE_COL] = pd.to_datetime(
    ppt_data[Config.DATE_COL], errors="coerce"
)

# 3. Asegurar formato de fecha en ENSO (Manejo de 'ene-70', etc.)
# Usamos la función auxiliar parse_spanish_date si existe, o lógica inline
if enso_data[Config.DATE_COL].dtype == "object":
    # Intento de conversión directa primero
    enso_data[Config.DATE_COL] = pd.to_datetime(
        enso_data[Config.DATE_COL], errors="coerce"
    )

# Si falló (quedaron NaTs), intentamos el parseo manual de español
if enso_data[Config.DATE_COL].isnull().any():

    def manual_spanish_parse(x):
        if isinstance(x, str):
            x = x.lower().strip()
            trans = {
                "ene": "Jan",
                "feb": "Feb",
                "mar": "Mar",
                "abr": "Apr",
                "may": "May",
                "jun": "Jun",
                "jul": "Jul",
                "ago": "Aug",
                "sep": "Sep",
                "oct": "Oct",
                "nov": "Nov",
                "dic": "Dec",
            }
            for es, en in trans.items():
                if es in x:
                    x = x.replace(es, en)
                    break
            try:
                return pd.to_datetime(x, format="%b-%y")
            except:
                return pd.NaT
        return x

    # Recargar columna original para parsear

```

```

enso_original = df_enso.copy()
enso_data[Config.DATE_COL] = enso_original[
    Config.DATE_COL
].apply(manual_spanish_parse)

# 4. Limpiar fechas nulas en ambos lados
ppt_data = ppt_data.dropna(subset=[Config.DATE_COL])
enso_data = enso_data.dropna(subset=[Config.DATE_COL])

# 5. Calcular Promedio Regional de Lluvia (una sola serie de tiempo)
regional_ppt = (
    ppt_data.groupby(Config.DATE_COL)[Config.PRECIPITATION_COL]
    .mean()
    .reset_index()
)

# 6. Unir las dos series por fecha
merged = pd.merge(
    regional_ppt, enso_data, on=Config.DATE_COL, how="inner"
)

if len(merged) > 12:
    c1, c2 = st.columns([2, 1])

    # Gráfico de Dispersión
    with c1:
        if Config.ENSO_ONI_COL in merged.columns:
            fig = px.scatter(
                merged,
                x=Config.ENSO_ONI_COL,
                y=Config.PRECIPITATION_COL,
                trendline="ols",
                title="Dispersión: ONI vs Lluvia Regional",
                labels={
                    Config.ENSO_ONI_COL: "Anomalía ONI (°C)",
                    Config.PRECIPITATION_COL: "Lluvia Mensual Promedio (mm)",
                },
                opacity=0.6,
            )
            st.plotly_chart(fig)
        else:
            st.warning(
                f"No se encontró la columna '{Config.ENSO_ONI_COL}' en los datos ENSO."
            )

    # Métricas Estadísticas

```

```

with c2:
    if Config.ENSO_ONI_COL in merged.columns:
        corr = merged[Config.ENSO_ONI_COL].corr(
            merged[Config.PRECIPITATION_COL]
        )
        st.markdown("##### Estadísticas")
        st.metric("Correlación (Pearson)", f"{corr:.2f}")

        if abs(corr) > 0.5:
            st.success("Existe una **fuerte** correlación.")
        elif abs(corr) > 0.3:
            st.info("Existe una correlación **moderada**.")
        else:
            st.warning("La correlación es **débil** o inexistente.")

        st.caption(f"Basado en {len(merged)} meses coincidentes.")
    else:
        st.warning(
            "No hay suficientes datos coincidentes en el tiempo entre la Lluvia y el ENSO para calcular la correlación."
        )

except Exception as e:
    st.error(f"Error en el cálculo de correlación ENSO: {e}")

# -----
# PESTAÑA 2: MATRIZ DE CORRELACIÓN ENTRE ESTACIONES
# -----
with tab2:
    st.markdown("##### Matriz de Correlación de Precipitación entre Estaciones")
    st.info(
        "Muestra qué tan similar es el comportamiento de la lluvia entre las diferentes estaciones seleccionadas. (1.0 = Idéntico, 0.0 = Sin relación)."
    )

try:
    # 1. Pivotear datos: Fechas en filas, Estaciones en columnas
    # Esto crea una tabla donde cada columna es una estación
    df_pivot = df_monthly.pivot_table(
        index=Config.DATE_COL,
        columns=Config.STATION_NAME_COL,
        values=Config.PRECIPITATION_COL,
    )

    # Validar que haya suficientes datos
    if df_pivot.shape[1] < 2:

```

```

st.warning(
    "Se necesitan al menos 2 estaciones seleccionadas para calcular una matriz de correlación."
)
else:
    # 2. Calcular Matriz de Correlación (Pearson)
    corr_matrix = df_pivot.corr()

    # 3. Heatmap Interactivo
    fig_corr = px.imshow(
        corr_matrix,
        text_auto=".2f",
        aspect="auto",
        color_continuous_scale="RdBu", # Rojo a Azul
        zmin=-1,
        zmax=1,
        title="Mapa de Calor de Correlaciones",
    )
    fig_corr.update_layout(height=700)
    st.plotly_chart(fig_corr, use_container_width=True)

    # 4. Botón de Descarga (CSV)
    csv_corr = corr_matrix.to_csv().encode("utf-8")
    st.download_button(
        label="📥 Descargar Matriz de Correlación (CSV)",
        data=csv_corr,
        file_name="matriz_correlacion_estaciones.csv",
        mime="text/csv",
        key="dl_corr_matrix",
    )

except Exception as e:
    st.error(f"Error generando la matriz de correlación: {e}")

def display_enso_tab(**kwargs):
    st.subheader("🌙 Fenómeno ENSO (El Niño - Oscilación del Sur)")

    # Recuperamos el DataFrame histórico que viene de la base de datos
    df_enso = kwargs.get("df_enso")

    # CREAMOS LAS PESTAÑAS
    # 1. Pronóstico Oficial (Nuevo, datos del IRI)
    # 2. Histórico ONI (Tu gráfico original que funciona bien)
    tab_iri, tab_historico = st.tabs([
        "⭐ Pronóstico Oficial (IRI/CPC)", "📋 Histórico ONI"
    ])

```

```

    )

# -----
# PESTAÑA 1: PRONÓSTICO IRI (NUEVO - DATOS LOCALES)
# -----
with tab_iri:
    st.info(
        " 📈 Datos oficiales del IRI (International Research Institute for Climate and Society) - Columbia University. Actualización Mensual."
    )

# Cargar datos desde archivos locales
json_plumas = fetch_iri_data("enso_plumes.json")
json_probs = fetch_iri_data("enso_iri_prob.json")

if json_plumas and json_probs:
    col1, col2 = st.columns(2)

    # A. Gráfico de Plumas
    with col1:
        st.markdown("#### 🌡️ Modelos de Predicción (Plumas)")
        data_plume = process_iri_plume(json_plumas)

        if data_plume:
            fig_plume = go.Figure()
            seasons = data_plume["seasons"]

            for model in data_plume["models"]:
                is_dynamic = model["type"] == "Dynamical"
                color = (
                    "rgba(100, 149, 237, 0.6)"
                    if is_dynamic
                    else "rgba(255, 165, 0, 0.6)"
                )
                name_prefix = "Dinámico" if is_dynamic else "Estadístico"

                fig_plume.add_trace(
                    go.Scatter(
                        x=seasons,
                        y=model["values"],
                        mode="lines",
                        name=f"{name_prefix}: {model['name']}",
                        line=dict(color=color, width=1.5),
                        opacity=0.7,
                        showlegend=False,
                    )
                )

    col2.subheader("B. Tabla de Probabilidades")
    st.dataframe(data_probs)

```

```

        hovertemplate=f"<b>{model['name']}</b><br>%{{y:.2f}} °C<extra></extra>",
    )
)

fig_plume.add_hline(
    y=0.5,
    line_dash="dash",
    line_color="red",
    annotation_text="El Niño",
)
fig_plume.add_hline(
    y=-0.5,
    line_dash="dash",
    line_color="blue",
    annotation_text="La Niña",
)
fig_plume.add_hline(y=0, line_color="black", opacity=0.3)

fig_plume.update_layout(
    yaxis_title="Anomalía SST (°C)",
    xaxis_title="Trimestres",
    height=450,
    margin=dict(l=40, r=40, t=40, b=40),
    hovermode="x unified",
)
st.plotly_chart(fig_plume, use_container_width=True)
else:
    st.warning("Error procesando plumas.")

# B. Gráfico de Probabilidades
with col2:
    st.markdown("#### 🌈 Probabilidad Multimodelo")
    df_probs = process_iri_probabilities(json_probs)

    if df_probs is not None and not df_probs.empty:
        df_melt = df_probs.melt(
            id_vars="Trimestre",
            var_name="Evento",
            value_name="Probabilidad",
        )
        color_map = {
            "El Niño": "#FF4B4B",
            "La Niña": "#1C83E1",
            "Neutral": "#808495",
        }

```

```

fig_probs = px.bar(
    df_melt,
    x="Trimestre",
    y="Probabilidad",
    color="Evento",
    color_discrete_map=color_map,
    text="Probabilidad",
    barmode="group",
)
fig_probs.update_traces(
    texttemplate="%{text:.0f}%", textposition="outside"
)
fig_probs.update_layout(
    yaxis_title="Probabilidad (%)",
    yaxis=dict(range=[0, 105]),
    height=450,
    legend=dict(
        orientation="h",
        yanchor="bottom",
        y=1.02,
        xanchor="right",
        x=1,
    ),
)
st.plotly_chart(fig_probs, use_container_width=True)
else:
    st.warning("Error procesando probabilidades.")
else:
    st.error(
        "⚠️ No se encontraron los archivos JSON en `data/iri/`. Verifica que los hayas subido."
    )

# -----
# PESTAÑA 2: HISTÓRICO ONI (USANDO TU FUNCIÓN ORIGINAL)
# -----
with tab_historico:
    st.markdown("#### 📈 Índice Oceánico del Niño (ONI) - Histórico")

if df_enso is not None and not df_enso.empty:
    # Limpieza básica de fechas para asegurar que el gráfico funcione
    data = df_enso.copy()

    # Intento de conversión de fechas seguro
    if data[Config.DATE_COL].dtype == "object":
        try:

```

```

# Intentamos usar pd.to_datetime directo primero
data[Config.DATE_COL] = pd.to_datetime(
    data[Config.DATE_COL], errors="coerce"
)
except:
    pass

data = data.dropna(subset=[Config.DATE_COL])

if Config.ENSO_ONI_COL in data.columns:
    # AQUÍ LLAMAMOS A TU FUNCIÓN PRESERVADA
    fig_oni = create_enso_chart(data)
    st.plotly_chart(fig_oni, use_container_width=True)
else:
    st.warning(
        f"No se encontró la columna '{Config.ENSO_ONI_COL}' en los datos."
    )
else:
    st.info("No hay datos históricos cargados.")

def display_life_zones_tab(
    df_long, gdf_stations, gdf_subcuencas=None, user_loc=None, **kwargs
):
    user_loc = kwargs.get("user_loc", user_loc)

    st.subheader("🌿 Zonas de Vida (Sistema Holdridge)")

    # --- SECCIÓN EDUCATIVA ---
    with st.expander("📚 Conceptos, Metodología e Importancia (Sistema Holdridge)"):
        st.markdown(
            """
<div style="font-size: 13px; line-height: 1.4;">
    <p><strong>Metodología:</strong> Clasificación ecológica basada en el cruce de Temperatura (estimada por Altura) y Precipitación anual.</p>

```

Pisos Altitudinales: (Altura vs Temperatura)

1. PISO NIVAL (> 4500 msnm, <-1.5C): 1. Nieves perpetuas y roca desnuda.
2. PISO ALPINO / SUPERPÁRAMO (3800 - 4500 msnm, >-1.5C): Tundra pluvial o húmeda. Vegetación escasa, transición a nieve.
3. PISO SUBALPINO / PÁRAMO (3000 - 3800 msnm, 1.5-3C): Ecosistema estratégico. Baja temperatura, ET reducida, excedentes de agua.
4. PISO MONTANO (2000 - 3000 msnm, 3-6C): Bosques de niebla y alto andinos. [13, 14, 15]

5. PISO MONTANO BAJO (1000 - 2000 msnm , 6-12C): Alta biodiversidad, temperaturas moderadas y precipitaciones significativas.

5. PISO PREMONTANO (1000 - 2000 msnm , 12-24C): Zona cafetera típica.

6. PISO TROPICAL (BASAL) (h < 1000 msnm , T > 24C).

Provincias de Humedad:

A. SECO: (ET>ppt), Deficit hidrico, stress hidrico

B. HUMEDO: (ppt > 1,2 ET), equilibrio o excedente hidrico

c. MUY HUMEDO: (ppt > 2 ET), exceso hidrico

C. Pluvial: Exceso extremo de lluvia (Chocó).

Clases:

Nival: 1; Tundra pluvial (tp-A): 2; Tundra húmeda (th-A): 3; Tundra seca (ts-A): 4; Páramo pluvial subalpino (pp-SA): 5; Páramo muy húmedo subalpino (pmh-SA): 6;

Páramo seco subalpino (ps-SA): 7; Bosque pluvial Montano (bp-M): 8; Bosque muy húmedo Montano (bmh-M): 9; Bosque húmedo Montano (bh-M): 10; Bosque seco Montano (bs-M): 11;

Monte espinoso Montano (me-M): 12; Bosque pluvial Premontano (bp-PM): 13, Bosque muy húmedo Premontano (bmh-PM): 14; Bosque húmedo Premontano (bh-PM): 15

Bosque seco Premontano (bs-PM): 16; Monte espinoso Premontano (me-PM): 17; Bosque pluvial Tropical (bp-T): 18; Bosque muy húmedo Tropical (bmh-T): 19;

Bosque húmedo Tropical (bh-T): 20; Bosque seco Tropical (bs-T): 21; Monte espinoso Tropical (me-T): 22; Zona Desconocida: 0

```
</div>
""",
unsafe_allow_html=True,
)

tab_raster, tab_puntos, tab_vector = st.tabs(
    ["<img alt='Raster icon' style='vertical-align: middle; height: 1em;"/> Mapa Raster", "📍 Puntos (Estaciones)", "⬇️ Descarga Vectorial"]
)

# --- PESTAÑA 1: MAPA RASTER ---
with tab_raster:
    col1, col2 = st.columns(2)
    with col1:
        res_option = st.select_slider(
            "Resolución:",
            options=["Baja (Rápido)", "Media", "Alta (Lento)"],
            value="Baja (Rápido)",
        )
        downscale = (
            8 if "Baja" in res_option else (4 if "Media" in res_option else 1)
        )
    with col2:
```

```

use_mask = st.checkbox("Recortar por Cuenca Seleccionada", value=True)

basin_geom = None
if use_mask:
    # --- CORRECCIÓN DE PRIORIDAD DE MÁSCARA ---
    # 1. Prioridad ALTA: Verificar si hay una cuenca específica en memoria (desde Mapas Avanzados)
    res_basin = st.session_state.get("basin_res")
    if res_basin and res_basin.get("ready"):
        basin_geom = res_basin.get("gdf_cuenca", res_basin.get("gdf_union"))
        st.success(
            f" ✅ Máscara activa: {res_basin.get('names', 'Cuenca Específica')}"
        )
    else:
        st.warning(
            " 🚨 No se detectó ninguna geometría para recortar. Se usará el mapa completo."
        )

if st.button("Generar Mapa de Zonas de Vida"):
    dem_path = getattr(Config, "DEM_FILE_PATH", "data/static/dem_antioquia.tif")
    ppt_path = getattr(
        Config, "PRECIP_RASTER_PATH", "data/static/ppt_anual_media.tif"
    )
    if not os.path.exists(dem_path) or not os.path.exists(ppt_path):
        st.error(f" ❌ Faltan archivos raster en: {dem_path} o {ppt_path}")
    else:
        with st.spinner("Generando mapa clasificado..."):
            try:
                lz_arr, profile, dynamic_legend, color_map = (
                    lz.generate_life_zone_map(
                        dem_path,
                        ppt_path,
                        mask_geometry=basin_geom,
                        downscale_factor=downscale,
                    )
            )

```

```

if lz_arr is not None:
    st.session_state.lz_raster_result = lz_arr
    st.session_state.lz_profile = profile
    st.session_state.lz_names = dynamic_legend
    st.session_state.lz_colors = color_map

# VISUALIZACIÓN
h, w = lz_arr.shape
transform = profile["transform"]
dx, dy = transform.a, transform.e
x0, y0 = transform.c, transform.f

xs = np.linspace(x0, x0 + dx * w, w)
ys = np.linspace(y0, y0 + dy * h, h)
xx, yy = np.meshgrid(xs, ys)

lat_flat = yy.flatten()
lon_flat = xx.flatten()
z_flat = lz_arr.flatten()
mask = z_flat > 0

if not np.any(mask):
    st.warning(
        "El mapa se generó pero está vacío (quizás la máscara no coincide con el área del DEM)."
    )
else:
    lat_clean = lat_flat[mask]
    lon_clean = lon_flat[mask]
    z_clean = z_flat[mask]

    center_lat = np.mean(lat_clean)
    center_lon = np.mean(lon_clean)

    # Área
    meters_deg = 111132.0
    px_area_ha = (
        abs(dx * meters_deg * cos(radians(center_lat)))
        * abs(dy * meters_deg)
    ) / 10000.0

    colors_hex = [
        color_map.get(v, "#808080") for v in z_clean
    ]
    hover_text = [
        f"{dynamic_legend.get(v, 'ID '+str(v))}"
        for v in z_clean
    ]

```

```

        ]

fig = go.Figure(
    go.Scattermapbox(
        lat=lat_clean,
        lon=lon_clean,
        mode="markers",
        marker=go.scattermapbox.Marker(
            size=8 if downscale > 4 else 5,
            color=colors_hex,
            opacity=0.75,
        ),
        text=hover_text,
        hovertemplate="%{text}<extra></extra>",
    )
)

if user_loc:
    fig.add_trace(
        go.Scattermapbox(
            lat=[user_loc[0]],
            lon=[user_loc[1]],
            mode="markers+text",
            marker=go.scattermapbox.Marker(
                size=15, color="black", symbol="star"
            ),
            text=["📍 TÚ ESTÁS AQUÍ"],
            textposition="top center",
        )
    )

fig.update_layout(
    mapbox_style="carto-positron",
    mapbox=dict(
        center=dict(lat=center_lat, lon=center_lon),
        zoom=9,
    ),
    height=600,
    margin={"r": 0, "t": 0, "l": 0, "b": 0},
    showlegend=False,
)
st.plotly_chart(fig)

# Tabla
unique, counts = np.unique(z_clean, return_counts=True)
data = [

```

```

    {
        "Zona": dynamic_legend.get(v, str(v)),
        "Ha": c * px_area_ha,
        "%": c / counts.sum() * 100,
    }
    for v, c in zip(unique, counts)
]
st.dataframe(
    pd.DataFrame(data)
    .sort_values("%", ascending=False)
    .style.format({"Ha": "{:.1f}", "%": "{:.1f}%"}),
)

```

tiff = lz.get_raster_bytes(lz_arr, profile)

if tiff:

```

        st.download_button(
            "⬇️ Descargar TIFF",
            tiff,
            "zonas_vida.tif",
            "image/tiff",
        )

```

except Exception as e:

```

        st.error(f"Error visualizando: {e}")

```

--- PESTAÑA 2: PUNTOS (TU CÓDIGO ORIGINAL) ---

with tab_puntos:

```

        df_anual = kwargs.get("df_anual_melted")
        if df_anual is None or gdf_stations is None:
            st.warning("Datos insuficientes para el análisis de estaciones.")
        else:
            try:
                # Usamos las funciones del backend para consistencia
                ppt_media = (
                    df_anual.groupby(Config.STATION_NAME_COL)[Config.PRECIPITATION_COL]
                    .mean()
                    .reset_index()
                )
                if Config.STATION_NAME_COL not in ppt_media.columns:
                    ppt_media = ppt_media.reset_index()

```

merged = pd.merge(

```

        ppt_media,
        gdf_stations[
            [
                Config.STATION_NAME_COL,

```

```

        Config.ALTITUDE_COL,
        "latitude",
        "longitude",
    ],
],
on=Config.STATION_NAME_COL,
)

def get_zone_data(row):
    z_id = lz.classify_life_zone_alt_ppt(
        row[Config.ALTITUDE_COL], row[Config.PRECIPITATION_COL]
    )
    return pd.Series(
        [
            lz.holdridge_int_to_name_simplified.get(
                z_id, "Desconocido"
            ),
            lz.holdridge_colors.get(z_id, "#808080"),
        ]
    )

merged[["Zona de Vida", "Color"]] = merged.apply(get_zone_data, axis=1)

fig_map = px.scatter_mapbox(
    merged,
    lat="latitude",
    lon="longitude",
    color="Zona de Vida",
    size=Config.PRECIPITATION_COL,
    hover_name=Config.STATION_NAME_COL,
    zoom=8,
    mapbox_style="carto-positron",
    title="Clasificación en Estaciones",
)
if user_loc:
    fig_map.add_trace(
        go.Scattermapbox(
            lat=[user_loc[0]],
            lon=[user_loc[1]],
            mode="markers+text",
            marker=go.scattermapbox.Marker(
                size=12, color="black", symbol="star"
            ),
            text=["📍 TÚ"],
            textposition="top center",
        )
    )

```

```

        )

    st.plotly_chart(fig_map, use_container_width=True)
    st.dataframe(
        merged[
            [
                Config.STATION_NAME_COL,
                "Zona de Vida",
                Config.PRECIPITATION_COL,
                Config.ALTITUDE_COL,
            ]
        ],
    )
except Exception as e:
    st.error(f"Error en puntos: {e}")

# --- PESTAÑA 3: VECTORIAL (NUEVA FUNCIONALIDAD) ---
with tab_vector:
    st.info(
        "🛠 Herramienta para convertir el mapa raster generado a polígonos (GeoJSON) para uso en SIG."
    )

if st.session_state.lz_raster_result is None:
    st.warning("⚠️ Primero debes generar el mapa en la pestaña 'Mapa Raster'.")
else:
    if st.button("Generar Polígonos (Vectorizar)"):
        with st.spinner("Convirtiendo píxeles a vectores..."):
            gdf_vec = lz.vectorize_raster_to_gdf(
                st.session_state.lz_raster_result,
                st.session_state.lz_profile["transform"],
                st.session_state.lz_profile["crs"],
            )

        if not gdf_vec.empty:
            st.success(
                f"Vectorización completada: {len(gdf_vec)} polígonos."
            )
            st.dataframe(gdf_vec.drop(columns="geometry").head())

        geojson_data = gdf_vec.to_json()
        st.download_button(
            label="⬇️ Descargar GeoJSON",
            data=geojson_data,
            file_name="zonas_vida_vectorial.geojson",
            mime="application/json",
        )

```

```

        )
else:
    st.error("No se generaron polígonos válidos.")

def display_drought_analysis_tab(df_long, gdf_stations, **kwargs):
    """
    Módulo de Extremos: Incluye Análisis Temporal (Series) y Espacial (Vulnerabilidad IVC).
    """

    import plotly.graph_objects as go
    import pandas as pd
    import numpy as np
    from scipy import stats
    from scipy.interpolate import griddata
    from modules.config import Config
    import matplotlib
    import matplotlib.pyplot as plt
    from shapely.geometry import LineString
    import geopandas as gpd
    import tempfile
    import os
    import shutil

    # Configuración backend para evitar errores de hilos en servidor
    matplotlib.use('Agg')

    # --- HELPERS INTERNOS PARA DESCARGAS EN ESTE MÓDULO ---
    def vectorizar_grid(gx, gy, gz, levels=10, crs="EPSG:4326"):
        """Convierte la matriz numpy actual en líneas vectoriales para descarga."""
        try:
            fig, ax = plt.subplots()
            contour = ax.contour(gx, gy, gz, levels=levels)
            plt.close(fig)
            lines, values = [], []
            for collection in contour.collections:
                z_val = 0
                try: z_val = collection.level
                except: pass
                for path in collection.get_paths():
                    if len(path.vertices) >= 2:
                        lines.append(LineString(path.vertices))
                        values.append(z_val)
            if not lines: return None
            return gpd.GeoDataFrame({"valor": values, "geometry": lines}, crs=crs)
        except: return None

```

```

st.subheader("⚡ Análisis de Extremos y Vulnerabilidad Climática")
st.info("Evaluación integral: Series temporales de extremos y Mapas de Vulnerabilidad Climática (IVC).")

stations_filtered = st.kwargs.get("stations_for_analysis", [])
if df_long is None or df_long.empty or not stations_filtered:
    st.warning("No hay datos o estaciones seleccionadas.")
    return

# Tabs Principales
tabs = st.tabs([
    "📊 Índices (SPI/SPEI)",
    "📊 Frecuencia (Gumbel)",
    "🌡️ Umbrales",
    "🔥 Vulnerabilidad (IVC)",
])
options = ["Serie Regional (Promedio)"] + sorted(stations_filtered)

# =====
# CONFIGURACIÓN COMÚN PARA ANÁLISIS TEMPORAL (Tabs 0, 1, 2)
# =====
with st.expander("📍 Configuración de Estación (Para SPI, Gumbel y Umbrales)", expanded=False):
    selected_station = st.selectbox("Seleccionar Estación:", options, key="extremes_station_sel")

# Preparación de datos temporal
if selected_station == "Serie Regional (Promedio)":
    df_subset = df_long[df_long[Config.STATION_NAME_COL].isin(stations_filtered)]
    df_station = df_subset.groupby(Config.DATE_COL)[Config.PRECIPITATION_COL].mean().reset_index()
    alt = 1500
else:
    df_station = df_long[df_long[Config.STATION_NAME_COL] == selected_station].copy()
    try:
        alt = gdf_stations[gdf_stations[Config.STATION_NAME_COL] ==
                           selected_station].iloc[0][Config.ALTITUDE_COL]
    except:
        alt = 1500

df_station = df_station.sort_values(by=Config.DATE_COL).set_index(Config.DATE_COL)
ts_ppt = df_station[Config.PRECIPITATION_COL].resample("MS").sum()

# --- TAB 1: SPI / SPEI ---
with tabs[0]:
    c1, c2 = st.columns(2)
    idx_type = c1.radio("Índice:", ["SPI (Lluvia)", "SPEI (Balance)"], horizontal=True)
    scale = c2.selectbox("Escala (Meses):", [1, 3, 6, 12, 24], index=2)
    try:

```

```

series_idx = None
if "SPI" in idx_type:
    from modules.analysis import calculate_spi
    series_idx = calculate_spi(ts_ppt, window=scale)
else:
    from modules.analysis import calculate_spei
    t_series = pd.Series([28 - (0.006 * float(alt))] * len(ts_ppt), index=ts_ppt.index)
    series_idx = calculate_spei(ts_ppt, t_series, window=scale)

if series_idx is not None and not series_idx.dropna().empty:
    df_vis = pd.DataFrame({"Val": series_idx})
    df_vis["Color"] = np.where(df_vis["Val"] >= 0, "blue", "red")
    fig = go.Figure()
    fig.add_trace(go.Bar(x=df_vis.index, y=df_vis["Val"], marker_color=df_vis["Color"],
name=idx_type))
    fig.add_hline(y=-1.5, line_dash="dash", line_color="red")
    fig.update_layout(title=f"Evolución {idx_type}-{scale} ({selected_station})", height=400)
    st.plotly_chart(fig, use_container_width=True)
else:
    st.warning("Datos insuficientes.")
except Exception as e: st.error(f"Error: {e}")

# --- TAB 2: FRECUENCIA (GUMBEL) ---
with tabs[1]:
    from modules.analysis import calculate_return_periods
    df_g = df_station.reset_index()
    df_g[Config.STATION_NAME_COL] = selected_station
    df_g[Config.YEAR_COL] = df_g[Config.DATE_COL].dt.year
    res_df, debug_data = calculate_return_periods(df_g, selected_station)
    if res_df is not None:
        c1, c2 = st.columns([1, 2])
        with c1: st.dataframe(res_df.style.format({"Ppt Máxima Esperada (mm)": "{:.1f}"}))
        with c2:
            tr = np.linspace(1.01, 100, 100)
            ppt_plot = stats.gumbel_r.ppf(1 - (1/tr), *debug_data["params"])
            fig = go.Figure()
            fig.add_trace(go.Scatter(x=tr, y=ppt_plot, name="Gumbel", line=dict(color="red")))
            fig.update_layout(xaxis_title="Período Retorno", yaxis_title="Ppt MÁX (mm)", xaxis_type="log",
height=400)
            st.plotly_chart(fig, use_container_width=True)
    else: st.warning("Datos insuficientes (min 10 años).")

# --- TAB 3: UMBRALES ---
with tabs[2]:
    c1, c2 = st.columns(2)
    p_l = c1.slider("Percentil Bajo:", 1, 20, 10)

```

```

p_h = c2.slider("Percentil Alto:", 80, 99, 90)
df_station["Mes"] = df_station.index.month
clim = df_station.groupby("Mes")[Config.PRECIPITATION_COL].quantile([p_l/100, 0.5,
p_h/100]).unstack()
clim.columns = ["low", "median", "high"]
fig = go.Figure()
months = ["Ene", "Feb", "Mar", "Abr", "May", "Jun", "Jul", "Ago", "Sep", "Oct", "Nov", "Dic"]
fig.add_trace(go.Scatter(x=months, y=clim["high"], name=f"P{p_h}", line=dict(color="blue")))
fig.add_trace(go.Scatter(x=months, y=clim["median"], name="Mediana", line=dict(color="green",
dash="dot")))
fig.add_trace(go.Scatter(x=months, y=clim["low"], name=f"P{p_l}", line=dict(color="red")))
st.plotly_chart(fig, use_container_width=True)

# =====
# TAB 4: VULNERABILIDAD CLIMÁTICA (IVC) - ACTUALIZADO Y PERSISTENTE
# =====
with tabs[3]:
    # 1. ENCABEZADO Y PRONÓSTICO ENSO
    st.markdown("#### 📈 Índice de Vulnerabilidad a la Variabilidad Climática (IVC)")

    # Caja de Pronóstico ENSO (Solicitud #5)
    st.warning("""
        🚨 **Pronóstico ENSO (Próximos 6 Meses):**  

        Según el último reporte del IRI/CPC, existe una **Probabilidad del 60% de condiciones de La Niña**  

        hacia el final del año,  

        lo que incrementaría el riesgo de excesos hídricos en la región Andina. Se recomienda monitorear los  

        boletines oficiales del IDEAM.
    """)
    # 2. METODOLOGÍA DESPLEGABLE (Solicitud #2 y #4)
    with st.expander("💡 Ver Metodología Detallada y Ecuaciones", expanded=False):
        st.markdown("""
            **Premisa:** El desabastecimiento hídrico se asocia a zonas cálidas y secas. El exceso, a zonas frías y  

            húmedas.
        """)

```

Para construir el índice adimensional **IVC (0-100)**:

1. **Parametrización de Temperatura (\$IT\$):**

$$IT = 100 \times \left(\frac{T_{max}}{T_{max}} \right)$$

Donde \$T\$ es la temperatura estimada (\$28 - 0.006 \cdot Altitud\$).

2. **Parametrización de Escorrentía (\$IESD\$):**

Se usa el balance de Turc para hallar la Escorrentía Superficial Directa (\$ESD = P - ETR\$).

$$IESD = 100 \times \left(\frac{ESD_{max} - ESD}{ESD_{max}} \right)$$

Nota: Esta fórmula invierte la escala (Menor agua = Mayor valor de índice).

```

3. **Índice Final ($IVC$):**
$$ IVC = \frac{IT + IESD}{2} $$

**Interpretación:**
* ● **Rojo (80-100):** Vulnerabilidad Crítica (Alta T, Baja ESD).
* ● **Verde (0-40):** Vulnerabilidad Baja (Baja T, Alta ESD).
"""

# 3. CONTROLES
c_ctrl1, c_ctrl2 = st.columns(2)
year_range_ivc = c_ctrl1.slider("Periodo Climático:", 1980, 2025, (2000, 2020), key="ivc_slider")
res_grid = c_ctrl2.select_slider("Resolución:", options=["Baja", "Media", "Alta"], value="Media")
grid_density = 50j if res_grid == "Baja" else 80j if res_grid == "Media" else 100j

# 4. LÓGICA DE CÁLCULO CON PERSISTENCIA (Solicitud #1 - Arreglo del reinicio)
if st.button("⚡ Calcular Mapa de Vulnerabilidad (IVC)"):
    with st.spinner("Realizando álgebra de mapas..."):

        # A. Preparar Datos
        mask = (df_long[Config.YEAR_COL] >= year_range_ivc[0]) & (df_long[Config.YEAR_COL] <=
year_range_ivc[1])
        df_filtered = df_long[mask]
        df_p =
        df_filtered.groupby(Config.STATION_NAME_COL)[Config.PRECIPITATION_COL].mean().reset_index()
        df_map = pd.merge(df_p, gdf_stations, on=Config.STATION_NAME_COL).dropna(subset=["latitude",
"longitude"])
        if Config.ALTITUDE_COL not in df_map.columns: df_map[Config.ALTITUDE_COL] = 1500

        if len(df_map) < 4:
            st.error("Se requieren al menos 4 estaciones.")
        else:
            # B. Interpolación y Álgebra
            points = df_map[["longitude", "latitude"]].values
            minx, miny = df_map.longitude.min(), df_map.latitude.min()
            maxx, maxy = df_map.longitude.max(), df_map.latitude.max()
            gx, gy = np.mgrid[minx:maxx:grid_density, miny:maxy:grid_density]

            grid_p = griddata(points, df_map[Config.PRECIPITATION_COL].values, (gx, gy), method='linear')
            grid_alt = griddata(points, df_map[Config.ALTITUDE_COL].values, (gx, gy), method='linear')

            # Variables Físicas
            grid_t = np.maximum(28 - (0.006 * grid_alt), 0)

            # Turc
            l_t = 300 + (25 * grid_t) + (0.05 * grid_t**3)

```

```

with np.errstate(divide='ignore', invalid='ignore'):
    grid_etr = grid_p / np.sqrt(0.9 + (grid_p / l_t)**2)
    grid_etr = np.minimum(grid_etr, grid_p)
    grid_esd = grid_p - grid_etr

    # Índices Normalizados
    t_max = np.nanmax(grid_t)
    grid_it = 100 * (grid_t / t_max)

    esd_max = np.nanmax(grid_esd) if np.nanmax(grid_esd) > 0 else 1
    grid_iesd = 100 * ((esd_max - grid_esd) / esd_max)

    grid_ivc = (grid_it + grid_iesd) / 2

    # GUARDAR EN SESSION STATE (EL SECRETO)
    st.session_state['ivc_results'] = {
        'ready': True,
        'gx': gx, 'gy': gy,
        'grid_ivc': grid_ivc,
        'grid_it': grid_it,
        'grid_iesd': grid_iesd,
        'grid_esd': grid_esd, # Para ver valor real
        'grid_p': grid_p, # Para ver valor real
        'grid_t': grid_t, # Para ver valor real
        'df_pts': df_map
    }

# 5. VISUALIZACIÓN DESDE MEMORIA (Solicitud #1 y #3)
if st.session_state.get('ivc_results', {}).get('ready'):
    res = st.session_state['ivc_results']

    # Selector de Capa
    layer = st.radio("Capa a visualizar:",
                     ["IVC (Vulnerabilidad Final)", "IT (Índice Temperatura)", "IESD (Índice Déficit)", "Variables Reales (P, T, Q)"],
                     horizontal=True)

    # Lógica de visualización
    z_data, title, colors, zmin, zmax = None, "", "", 0, 100

    if layer == "IVC (Vulnerabilidad Final)":
        z_data, title, colors = res['grid_ivc'], "Índice de Vulnerabilidad (IVC)", "RdYIGn_r"
    elif layer == "IT (Índice Temperatura)":
        z_data, title, colors = res['grid_it'], "Índice de Temperatura (IT)", "OrRd"
    elif layer == "IESD (Índice Déficit)":
        z_data, title, colors = res['grid_iesd'], "Índice de Déficit de Escorrentía (IESD)", "YlOrRd"

```

```

else:
    # Sub-selector para variables reales
    sub_layer = st.selectbox("Seleccionar Variable Física:", ["Precipitación (mm)", "Temperatura (°C)",
    "Escorrentía (mm)"])

    if "Precipitación" in sub_layer:
        z_data, title, colors = res['grid_p'], "Precipitación Media (mm)", "Blues"
        zmax = np.nanmax(res['grid_p'])

    elif "Temperatura" in sub_layer:
        z_data, title, colors = res['grid_t'], "Temperatura Media (°C)", "Thermal"
        zmax = np.nanmax(res['grid_t'])

    else:
        z_data, title, colors = res['grid_esd'], "Escorrentía Superficial (mm)", "Teal"
        zmax = np.nanmax(res['grid_esd'])

# Estadísticas Min/Max (Solicitud #3)
st.markdown(f"**Estadísticas de la capa: {title}**")
c_min, c_max = st.columns(2)
c_min.metric("Mínimo", f"{np.nanmin(z_data):.1f}")
c_max.metric("Máximo", f"{np.nanmax(z_data):.1f}")

# Mapa
fig_map = go.Figure(data=go.Contour(
    z=z_data.T, x=res['gx'][:, 0], y=res['gy'][0, :],
    colorscale=colors, colorbar=dict(title="Valor"),
    contours=dict(start=zmin, end=zmax, size=(zmax-zmin)/15 if zmax>zmin else 1),
    zmin=zmin, zmax=zmax
))
fig_map.add_trace(go.Scatter(
    x=res['df_pts'].longitude, y=res['df_pts'].latitude, mode='markers',
    marker=dict(color='black', size=4), name='Estaciones'
))
fig_map.update_layout(title=title, height=550, margin=dict(l=20, r=20, t=40, b=20))
st.plotly_chart(fig_map, use_container_width=True)

# Descarga del Mapa (Solicitud #3)
if st.button(f"⬇️ Preparar Descarga de {layer}"):
    gdf_iso = vectorizar_grid(res['gx'], res['gy'], z_data, levels=15)
    if gdf_iso is not None:
        json_data = gdf_iso.to_json()
        st.download_button(
            label=f"⬇️ Descargar GeoJSON ({layer})",
            data=json_data,
            file_name=f"mapa_{layer.split()[0].lower()}.geojson",
            mime="application/json"
        )

```

```

else:
    st.warning("No se pudo vectorizar esta capa para descarga.")

# FUNCIÓN CLIMA FUTURO (MAPA RIESGO MEJORADO + SIMULADOR)
# =====
def display_climate_scenarios_tab(**kwargs):
    st.subheader("📍 Clima Futuro y Vulnerabilidad (CMIP6 / Riesgo)")

    # Recuperamos datos
    df_anual = kwargs.get("df_anual_melted")
    gdf_stations = kwargs.get("gdf_stations")

    # Intentamos recuperar la cuenca para recorte y SU NOMBRE
    basin_geom = None
    basin_name = "Regional (Todas las Estaciones)" # Nombre por defecto

    res_basin = st.session_state.get("basin_res")
    if res_basin and res_basin.get("ready"):
        basin_geom = res_basin.get("gdf_cuenca", res_basin.get("gdf_union"))
        # Intentamos obtener el nombre si existe en el diccionario
        if "names" in res_basin:
            basin_name = f"Cuenca: {res_basin['names']}"
        elif "name" in res_basin:
            basin_name = f"Cuenca: {res_basin['name']}"

    tab_risk, tab_cmip6 = st.tabs(
        [
            "🗺️ Mapa de Riesgo (Tendencias Históricas)",
            "🌐 Simulador de Cambio Climático (CMIP6)",
        ]
    )

    # --- TAB 1: MAPA DE RIESGO (MEJORADO VISUALMENTE) ---
    with tab_risk:
        st.markdown("#### Vulnerabilidad Hídrica: Tendencias de Precipitación")
        st.caption(f"**Zona de Análisis:** {basin_name}" # Mostramos el nombre aquí

        with st.expander("ℹ️ Acerca de este Mapa de Riesgo", expanded=False):
            st.markdown(
                """
                Este mapa muestra la **tendencia espacial histórica** de la lluvia interpolando la pendiente de Sen (Mann-Kendall).
                * **Objetivo:** Identificar zonas que se están secando (Vulnerables) o humedeciendo.
                * **Interpretación:***
                """
            )

```



```

if len(trend_data) >= 4:
    df_trend = pd.DataFrame(trend_data)

    # Interpolación
    grid_res = 200j
    grid_x, grid_y = np.mgrid[
        df_trend.lon.min() - 0.1 : df_trend.lon.max() + 0.1 : grid_res,
        df_trend.lat.min() - 0.1 : df_trend.lat.max() + 0.1 : grid_res,
    ]

from scipy.interpolate import griddata

grid_z = griddata(
    df_trend[["lon", "lat"]].values,
    df_trend["slope"].values,
    (grid_x, grid_y),
    method="cubic",
)

# Máscara Geométrica (Recorte)
if use_mask and basin_geom is not None:
    try:
        from shapely.geometry import Point
        from shapely.prepared import prep

        poly = (
            basin_geom.unary_union
            if hasattr(basin_geom, "unary_union")
            else basin_geom
        )
        prep_poly = prep(poly)

        flat_x = grid_x.flatten()
        flat_y = grid_y.flatten()
        flat_z = grid_z.flatten()

        # Optimización: Solo verificar puntos que no son NaN (ahorra tiempo si griddata ya puso
        # NaNs afuera del convex hull)
        mask_array = np.isnan(flat_z) # True donde ya es NaN

        # Verificamos los puntos válidos
        valid_indices = np.where(~mask_array)[0]
        for idx in valid_indices:
            if not prep_poly.contains(
                Point(flat_x[idx], flat_y[idx])
            )

```

```

    ):
        flat_z[idx] = np.nan

    grid_z = flat_z.reshape(grid_x.shape)
except Exception as e:
    st.warning(f"No se pudo recortar visualmente: {e}")

fig = go.Figure()

# Mapa de Calor / Contornos
fig.add_trace(
    go.Contour(
        z=grid_z.T,
        x=grid_x[:, 0],
        y=grid_y[0, :],
        colorscale="RdBu",
        colorbar=dict(
            title="Tendencia (mm/año)",
            titleside="right",
            thickness=15,
            len=0.7, # Hacemos la barra un poco más corta para que no choque
        ),
        zmid=0,
        opacity=0.8,
        contours=dict(showlines=False),
        connectgaps=False,
    )
)

# Puntos de Estaciones (MEJORADOS VISUALMENTE)
# Usamos color de relleno amarillo pálido para resaltar sobre azul/rojo
# Borde negro siempre visible
# Grosor de borde indica significancia
df_trend["line_width"] = df_trend["p"].apply(
    lambda x: 2 if x < 0.05 else 1
)

fig.add_trace(
    go.Scatter(
        x=df_trend.lon,
        y=df_trend.lat,
        mode="markers",
        text=df_trend.apply(
            lambda r: f"<b>{r['name']}</b><br>Mun: {r['municipio']}<br>Pendiente: {r['slope']:.2f}<br>Sig: {'Sí' if r['p']<0.05 else 'No'}",
            axis=1,
        )
    )
)

```

```

),
marker=dict(
    size=10,
    color="#FFFFE0", # LightYellow (Resalta sobre oscuros)
    line=dict(
        width=df_trend["line_width"],
        color="black", # Borde negro para contraste
    ),
),
name="Estaciones",
)
)

# Borde de la Cuenca
if basin_geom is not None:
    try:
        poly = (
            basin_geom.unary_union
            if hasattr(basin_geom, "unary_union")
            else basin_geom
        )
        if poly.geom_type == "Polygon":
            x, y = poly.exterior.xy
            fig.add_trace(
                go.Scatter(
                    x=list(x),
                    y=list(y),
                    mode="lines",
                    line=dict(color="black", width=2),
                    name="Límite Cuenca",
                )
            )
        elif poly.geom_type == "MultiPolygon":
            for i, p in enumerate(poly.geoms):
                x, y = p.exterior.xy
                fig.add_trace(
                    go.Scatter(
                        x=list(x),
                        y=list(y),
                        mode="lines",
                        line=dict(color="black", width=2),
                        showlegend=(i == 0),
                        name="Límite Cuenca",
                    )
                )
    except:

```

```

    pass

# Configuración de Layout (LEYENDA AJUSTADA)
fig.update_layout(
    title=f"Tendencia Espacial de Precipitación<br><sup>{basin_name}</sup>", # Título con
    subtítulo de cuenca
    xaxis_title="Longitud",
    yaxis_title="Latitud",
    height=650, # Un poco más alto
    yaxis=dict(scaleanchor="x", scaleratio=1),
    legend=dict(
        orientation="h",
        yanchor="top",
        y=-0.1, # Movemos la leyenda DEBAJO del gráfico
        xanchor="center",
        x=0.5,
    ),
    margin=dict(
        l=20, r=20, t=60, b=80
    ), # Más margen abajo para la leyenda
)
st.plotly_chart(fig)

c_d1, c_d2 = st.columns(2)
with c_d1:
    geojson = df_trend.to_json(orient="records")
    st.download_button(
        "⬇️ Descargar Puntos (JSON)",
        geojson,
        "tendencias_puntos.json",
        "application/json",
    )
with c_d2:
    flat_x = grid_x.flatten()
    flat_y = grid_y.flatten()
    flat_z = grid_z.flatten()
    df_grid = pd.DataFrame(
        {"lon": flat_x, "lat": flat_y, "tendencia": flat_z}
    ).dropna()
    csv_grid = df_grid.to_csv(index=False).encode("utf-8")
    st.download_button(
        "⬇️ Descargar Grilla (CSV)",
        csv_grid,
        "tendencias_grilla.csv",
        "text/csv",
    )

```

```

        )
    else:
        st.warning("Datos insuficientes para interpolar.")

# --- TAB 2: SIMULADOR CMIP6 (MANTENIDO IGUAL) ---
with tab_cmip6:
    # (El código del simulador se mantiene idéntico al bloque anterior que ya funcionaba)
    st.subheader("Simulador de Cambio Climático (Escenarios CMIP6)")
    st.info(
        "Proyección de anomalías climatológicas para la región Andina (Horizonte 2040-2060)."
    )

# 1. Caja Informativa
with st.expander(
    "💡 Conceptos Clave: Escenarios SSP y Modelos CMIP6 (IPCC AR6)",
    expanded=False,
):
    st.markdown(
        """
        **🔍 Anatomía del Código: {Escenario} = {SSP(X)} - {Y.Y}**  

        Combina la **Trayectoria Social (SSP 1-5)** con el **Forzamiento Radiativo (W/m²)** al 2100.

        **📈 Escenarios "Tier 1" (Proyecciones):**  

        * **SSP1-2.6 (Sostenibilidad):** "Ruta Verde". Emisiones cero neto a 2050. Escenario optimista (<2°C).  

        * **SSP2-4.5 (Camino Medio):** Tendencias actuales. Progreso desigual. Escenario de planificación "realista" (~2.7°C).  

        * **SSP3-7.0 (Rivalidad Regional):** Nacionalismo y baja cooperación. Muy peligroso (~3.6°C a 4°C).  

        * **SSP5-8.5 (Desarrollo Fósil):** "La Autopista". Crecimiento rápido basado en carbón/petróleo. El peor caso (>4.4°C).
        """
    )
    st.info("Use **SSP2-4.5** para planificación estándar. Use **SSP5-8.5** solo para **pruebas de estrés** en infraestructura crítica (validar resiliencia ante eventos extremos inéditos).")
    st.info("")

scenarios_db = {
    "SSP1-2.6 (Sostenibilidad)": {
        "temp": 1.6,
        "ppt_anual": 5.2,
        "desc": "Escenario optimista...",
    },
    "SSP2-4.5 (Camino Medio)": {

```

```

        "temp": 2.1,
        "ppt_anual": -2.5,
        "desc": "Escenario intermedio...",
    },
    "SSP3-7.0 (Rivalidad Regional)": {
        "temp": 2.8,
        "ppt_anual": -8.4,
        "desc": "Escenario pesimista...",
    },
    "SSP5-8.5 (Desarrollo Fósil)": {
        "temp": 3.4,
        "ppt_anual": -12.1,
        "desc": "Peor escenario...",
    },
}
}

st.markdown("##### 🛠️ Ajuste Manual de Escenarios (Simulación)")
c_sim1, c_sim2 = st.columns(2)
with c_sim1:
    delta_temp = st.slider(
        "Aumento de Temperatura (°C):",
        0.0,
        5.0,
        1.5,
        0.1,
        help="Simular aumento de temperatura.",
    )
with c_sim2:
    delta_ppt = st.slider(
        "Cambio en Precipitación (%):",
        -30,
        30,
        -5,
        1,
        help="Simular cambio porcentual.",
    )

if st.button("🚀 Simular Escenario Futuro"):
    et_increase = delta_temp * 3
    water_balance_change = delta_ppt - et_increase
    st.metric(
        "Impacto Estimado en Balance Hídrico",
        f"{water_balance_change:.1f}%",
        delta="Déficit Hídrico" if water_balance_change < 0 else "Excedente",
        delta_color="inverse",
    )

```

```

        )
        st.caption(f"Nota: Aumento de ET estimado: {et_increase:.1f}%.")
    st.divider()

    st.markdown("##### 📊 Comparativa de Escenarios Oficiales vs. Simulación")
    c_sel, c_sort = st.columns([2, 1])
    with c_sel:
        selected_scenarios = st.multiselect(
            "Seleccionar Escenarios:",
            list(scenarios_db.keys()),
            default=list(scenarios_db.keys()),
        )
    with c_sort:
        sort_order = st.selectbox(
            "Ordenar Gráfico:",
            ["Ascendente ⬆️", "Descendente ⬇️", "Nombre Escenario"],
        )

    if selected_scenarios:
        plot_data = []
        for sc in selected_scenarios:
            row = scenarios_db[sc]
            plot_data.append(
                {
                    "Escenario": sc,
                    "Anomalía Temperatura (°C)": row["temp"],
                    "Anomalía Precipitación (%)": row["ppt_anual"],
                    "Tipo": "Oficial",
                }
            )
        plot_data.append(
            {
                "Escenario": "Mi Simulación (Manual)",
                "Anomalía Temperatura (°C)": delta_temp,
                "Anomalía Precipitación (%)": delta_ppt,
                "Tipo": "Usuario",
            }
        )
        df_sim = pd.DataFrame(plot_data)

        if "Ascendente" in sort_order:
            df_sim = df_sim.sort_values(

```

```

        "Anomalía Precipitación (%)", ascending=True
    )
elif "Descendente" in sort_order:
    df_sim = df_sim.sort_values(
        "Anomalía Precipitación (%)", ascending=False
    )
else:
    df_sim = df_sim.sort_values("Escenario")

c_g1, c_g2 = st.columns(2)
with c_g1:
    fig_ppt = px.bar(
        df_sim,
        y="Escenario",
        x="Anomalía Precipitación (%)",
        color="Anomalía Precipitación (%)",
        title="Anomalía Precipitación (%)",
        color_continuous_scale="RdBu",
        text_auto=".1f",
        orientation="h",
    )
    fig_ppt.add_vline(x=0, line_width=1, line_color="black")
    st.plotly_chart(fig_ppt, use_container_width=True)

with c_g2:
    fig_temp = px.bar(
        df_sim,
        y="Escenario",
        x="Anomalía Temperatura (°C)",
        color="Anomalía Temperatura (°C)",
        title="Aumento Temperatura (°C)",
        color_continuous_scale="YlOrRd",
        text_auto=".1f",
        orientation="h",
    )
    st.plotly_chart(fig_temp, use_container_width=True)

st.markdown("##### 📊 Detalles de Escenarios")
st.dataframe(
    df_sim[
        [
            "Escenario",
            "Anomalía Precipitación (%)",
            "Anomalía Temperatura (°C)",
            "Tipo",
        ]
    ],

```

```

        )
else:
    st.warning("Seleccione escenarios para comparar.")

def display_station_table_tab(**kwargs):
    st.subheader("📋 Tabla Detallada de Datos")

    # Podemos mostrar los datos mensuales o anuales
    df_monthly = kwargs.get("df_monthly_filtered")

    if df_monthly is not None and not df_monthly.empty:
        st.write(f"Mostrando {len(df_monthly)} registros filtrados.")

        # Formatear fecha para que se vea bonita
        df_show = df_monthly.copy()
        df_show["Fecha"] = df_show[Config.DATE_COL].dt.strftime("%Y-%m-%d")

        # Selección de columnas limpias
        cols = ["Fecha", Config.STATION_NAME_COL, Config.PRECIPITATION_COL]
        st.dataframe(df_show[cols])

        # Botón de descarga
        csv = df_show[cols].to_csv(index=False).encode("utf-8")
        st.download_button(
            "⬇️ Descargar CSV",
            csv,
            "datos_precipitacion.csv",
            "text/csv",
            key="download-csv",
        )
    else:
        st.warning("No hay datos para mostrar.")

# LAND_COVER (Coberturas)
def display_land_cover_analysis_tab(df_long, gdf_stations, **kwargs):
    st.subheader("🌿 Análisis de Cobertura del Suelo y Escenarios")

    # 1. Configuración
    Config = None
    try:
        from modules.config import Config as Cfg
        Config = Cfg
    except:
        pass

```

```

raster_path = "data/Cob25m_WGS84.tif"
if Config and hasattr(Config, "LAND_COVER_RASTER_PATH"):
    raster_path = Config.LAND_COVER_RASTER_PATH

# 2. Control de Vista
res_basin = st.session_state.get("basin_res")
has_basin_data = res_basin and res_basin.get("ready")

col_ctrl, col_info = st.columns([1, 2])
with col_ctrl:
    idx = 1 if has_basin_data else 0
    view_mode = st.radio("📍 Modo Visualización:", ["Regional", "Cuenca"], index=idx, horizontal=True)

gdf_mask = None
basin_name = "Regional (Antioquia)"
ppt_anual = 2000
area_cuenca_km2 = None

if view_mode == "Cuenca":
    if has_basin_data:
        gdf_mask = res_basin.get("gdf_cuenca", res_basin.get("gdf_union"))
        basin_name = res_basin.get("names", "Cuenca Actual")
        bal = res_basin.get("bal", {})
        ppt_anual = bal.get("P", 2000)
        morph = res_basin.get("morph", {})
        area_cuenca_km2 = morph.get("area_km2", 0)
        with col_info:
            st.success(f"Analizando: **{basin_name}**")
    else:
        st.warning("⚠️ No hay cuenca delimitada. Cambiando a modo Regional.")
        view_mode = "Regional"

# 3. Procesamiento
try:
    import modules.land_cover as lc

    # Procesar Raster (lc.process_land_cover_raster ya maneja proyecciones internamente gracias a nuestro
fix anterior)
    scale = 10 if view_mode == "Regional" else 1
    data, transform, crs, nodata = lc.process_land_cover_raster(
        raster_path, gdf_mask=gdf_mask, scale_factor=scale
    )

    if data is None:

```

```

st.error("Error cargando mapa. Verifica el archivo raster o la superposición con la cuenca.")
return

# Cálculo Estadístico
df_res, area_total_km2 = lc.calculate_land_cover_stats(
    data, transform, crs, nodata, manual_area_km2=area_cuenca_km2
)

# 4. Visualización
tab_map, tab_stat, tab_sim = st.tabs(["Mapa Interactivo", "Tabla & Gráficos", "Simulador SCS-CN"])

with tab_map:
    c_tools, c_map = st.columns([1, 4])
    with c_tools:
        st.markdown("##### Opciones")
        use_hover = st.checkbox("Activar Hover", value=False, help="Muestra nombres al pasar el mouse.")
        show_legend = st.checkbox("Leyenda", value=True)

    tiff_bytes = lc.get_tiff_bytes(data, transform, crs, nodata)
    if tiff_bytes:
        st.download_button("Bajar Mapa (TIFF)", tiff_bytes, "cobertura.tif", "image/tiff")

    with c_map:
        from rasterio.transform import array_bounds
        from pyproj import Transformer
        import folium
        from folium import plugins
        from streamlit_folium import st_folium

        # Bounds
        h, w = data.shape
        minx, miny, maxx, maxy = array_bounds(h, w, transform)

        # Transformar bounds a Lat/Lon para centrar el mapa
        transformer = Transformer.from_crs(crs, "EPSG:4326", always_xy=True)
        lon_min, lat_min = transformer.transform(minx, miny)
        lon_max, lat_max = transformer.transform(maxx, maxy)
        bounds = [[lat_min, lon_min], [lat_max, lon_max]]
        center = [(lat_min+lat_max)/2, (lon_min+lon_max)/2]

        # --- CREACIÓN DEL MAPA ---
        m = folium.Map(location=center, zoom_start=12 if view_mode=="Cuenca" else 8, tiles="CartoDB positron")

```

```

plugins.Fullscreen(
    position='topright', title='Pantalla completa',
    title_cancel='Salir', force_separate_button=True
).add_to(m)

# --- LEYENDA HTML DINÁMICA ---
if show_legend and not df_res.empty:
    legend_html = lc.generate_legend_html() # Usamos la del módulo si existe, o construimos
manual
    # Si prefieres la manual que tenías, mantenla, pero aquí uso una lógica simplificada
    if not hasattr(lc, 'generate_legend_html'):
        # Fallback a tu lógica manual si la función no está en lc
        pass
    else:
        m.get_root().html.add_child(folium.Element(lc.generate_legend_html()))

# CAPA 1: IMAGEN (Raster)
img_url = lc.get_raster_img_b64(data, nodata)
if img_url:
    folium.raster_layers.ImageOverlay(
        image=img_url, bounds=bounds, opacity=0.75, name="Cobertura"
    ).add_to(m)

# CAPA 2: INTERACTIVA (Vectorial)
if use_hover:
    with st.spinner("Generando capa interactiva..."):
        scale_vec = 50 if view_mode == "Regional" else 1
        # Re-procesar para hover si es regional (downsampling)
        if view_mode == "Regional":
            d_hov, t_hov, _, _ = lc.process_land_cover_raster(raster_path, gdf_mask=None,
scale_factor=scale_vec)
            gdf_vec = lc.vectorize_raster_optimized(d_hov, t_hov, crs, nodata)
        else:
            gdf_vec = lc.vectorize_raster_optimized(data, transform, crs, nodata)

        if not gdf_vec.empty:
            folium.GeoJson(
                gdf_vec,
                style_function=lambda x: {'fillColor': '#ffffff', 'color': 'none', 'fillOpacity': 0},
                tooltip=folium.GeoJsonTooltip(fields=['Cobertura'], aliases=['Tipo']),
                name="Hover Info"
            ).add_to(m)

# --- CORRECCIÓN CRÍTICA: PROYECCIÓN DE LA MÁSCARA ---
if view_mode == "Cuenca" and gdf_mask is not None:

```

```

try:
    # Asegurar que la máscara esté en Lat/Lon para que Folium la muestre
    gdf_mask_viz = gdf_mask.to_crs(epsg=4326) if gdf_mask.crs.to_string() != "EPSG:4326" else
gdf_mask
    folium.GeoJson(
        gdf_mask_viz,
        style_function=lambda x: {'color': 'black', 'fill': False, 'weight': 2},
        name="Límite Cuenca"
    ).add_to(m)
except Exception as e:
    print(f"Error proyectando máscara: {e}")

# --- INTERVENCIÓN 2: CAPAS DE VULNERABILIDAD (Con búsqueda segura) ---

# Intentar buscar las capas en kwargs o session_state
gdf_inc = kwargs.get('gdf_amenaza_incendios', st.session_state.get('gdf_amenaza_incendios'))
gdf_agr = kwargs.get('gdf_aptitud_agricola', st.session_state.get('gdf_aptitud_agricola'))

# Capa Incendios
if gdf_inc is not None and not gdf_inc.empty:
    try:
        gdf_inc_viz = gdf_inc.to_crs(epsg=4326) # Reproyectar siempre por seguridad
        folium.GeoJson(
            data=gdf_inc_viz,
            name='Amenaza Incendios',
            style_function=lambda x: {
                'fillColor': '#e74c3c' if x['properties'].get('riesgo') == 'Alto' else '#f1c40f',
                'color': 'black', 'weight': 0.5, 'fillOpacity': 0.6
            },
            tooltip="Riesgo Incendio: " + folium.features.GeoJsonTooltip(fields=['riesgo'])
        ).add_to(m)
    except Exception as e: print(f"Error capa incendios: {e}")

# Capa Agrícola
if gdf_agr is not None and not gdf_agr.empty:
    try:
        gdf_agr_viz = gdf_agr.to_crs(epsg=4326) # Reproyectar siempre por seguridad
        folium.GeoJson(
            data=gdf_agr_viz,
            name='Aptitud Agrícola',
            show=False,
            style_function=lambda x: {
                'fillColor': '#2ecc71', 'color': 'black', 'weight': 0.5, 'fillOpacity': 0.5
            }
        ).add_to(m)
    except Exception as e: print(f"Error capa agrícola: {e}")

```

```

folium.LayerControl().add_to(m)
st_folium(m, height=600, use_container_width=True, key="map_lc_final")

with tab_stat:
    c1, c2 = st.columns([1, 1])
    with c1:
        st.dataframe(df_res[["ID", "Cobertura", "Área (km²)", "%"]].style.format({"Área (km²)": "{:.2f}", "%": "{:.1f}"}))
    csv = df_res.to_csv(index=False).encode('utf-8')
    st.download_button("⬇️ Descargar CSV", csv, "stats_coberturas.csv", "text/csv")
    with c2:
        import plotly.express as px
        fig = px.pie(df_res, values="Área (km²)", names="Cobertura", color="Cobertura",
                      color_discrete_map={r["Cobertura"]: r["Color"] for _, r in df_res.iterrows()}, hole=0.4)
        st.plotly_chart(fig)

with tab_sim:
    if view_mode == "Cuenca":
        st.info("Simula cambios de uso del suelo.")
        with st.expander("⚙️ Configuración CN", expanded=False):
            cc = st.columns(5)
            cn_cfg = {
                'bosque': cc[0].number_input("Bosque", value=55),
                'pasto': cc[1].number_input("Pasto", value=75),
                'cultivo': cc[2].number_input("Cultivo", value=85),
                'urbano': cc[3].number_input("Urbano", value=95),
                'suelo': cc[4].number_input("Suelo", value=90)
            }
            st.write("Defina el Escenario Futuro (%):")
            sl = st.columns(5)
            inputs = [sl[0].slider("% Bosque", 0, 100, 40), sl[1].slider("% Pasto", 0, 100, 30),
                      sl[2].slider("% Cultivo", 0, 100, 20), sl[3].slider("% Urbano", 0, 100, 5),
                      sl[4].slider("% Suelo", 0, 100, 5)]
            if abs(sum(inputs) - 100) < 0.1:
                if st.button("✍️ Calcular Escenario"):
                    import plotly.graph_objects as go
                    cn_act = lc.calculate_weighted_cn(df_res, cn_cfg)
                    cn_fut = (inputs[0]*cn_cfg['bosque'] + inputs[1]*cn_cfg['pasto'] +
                              inputs[2]*cn_cfg['cultivo'] + inputs[3]*cn_cfg['urbano'] +
                              inputs[4]*cn_cfg['suelo']) / 100
                    q_act = lc.calculate_scs_runoff(cn_act, ppt_anual)

```

```

q_fut = lc.calculate_scs_runoff(cn_fut, ppt_anual)

vol_act = (q_act * area_total_km2) / 1000
vol_fut = (q_fut * area_total_km2) / 1000

c_res = st.columns(3)
c_res[0].metric("CN Escenario", f"{cn_fut:.1f}", delta=f"{cn_fut-cn_act:.1f}",
delta_color="inverse")
c_res[1].metric("Escorrentía Q", f"{q_fut:.0f} mm", delta=f"{q_fut-q_act:.0f} mm",
delta_color="inverse")
c_res[2].metric("Volumen Total", f"{vol_fut:.2f} Mm³", delta=f"{vol_fut-vol_act:.2f} Mm³")

fig_sim = go.Figure(data=[
    go.Bar(name="Actual", x=["Escorrentía"], y=[q_act], marker_color="#1f77b4",
text=f"{q_act:.0f}", textposition="auto"),
    go.Bar(name="Futuro", x=["Escorrentía"], y=[q_fut], marker_color="#2ca02c",
text=f"{q_fut:.0f}", textposition="auto"),
])
st.plotly_chart(fig_sim, use_container_width=True)
else:
    st.warning("La suma debe ser 100%.")
else:
    st.info("⚠ Requiere modo Cuenca.")

except Exception as e:
    st.error(f"Error en módulo de coberturas: {e}")

# PESTAÑA: CORRECCIÓN DE SESGO (VERSIÓN BLINDADA)
# -----
def display_bias_correction_tab(df_long, gdf_stations, gdf_filtered, **kwargs):
    """
    Módulo de validación y corrección de sesgo (Estaciones vs Satélite ERA5).
    Versión optimizada para series temporales mensuales.
    """

    st.subheader("📊 Validación Mensual (Estaciones vs. Satélite)")

    # --- DOCUMENTACIÓN Y AYUDA (NUEVO BLOQUE) ---
    with st.expander("💡 Guía Técnica: Fuentes, Metodología e Interpretación", expanded=False):
        st.markdown(
            """
            #### 1. ¿Qué hace este módulo?
            """
        )

```

Este módulo permite comparar la **precipitación observada** (medida por pluviómetros en tierra) con la **precipitación estimada** por modelos satelitales/reanálisis (ERA5-Land) para evaluar la precisión de estos últimos en la región Andina.

2. Fuentes de Datos

- * **Estaciones (Observado):** Datos hidrometeorológicos reales cargados en el sistema (IDEAM/Particulares).
- * **Satélite (Estimado):** [ERA5-Land](<https://cds.climate.copernicus.eu/>), un reanálisis climático global de alta resolución (~9km) producido por el ECMWF.
- * *Ventaja:* Cobertura global continua y datos desde 1950.
- * *Desventaja:* Tiende a subestimar lluvias extremas en topografía compleja (montañas) debido a su resolución espacial.

3. Metodología de Procesamiento

1. **Agregación Temporal:** Se transforman los datos diarios a **acumulados mensuales** exactos.
2. **Emparejamiento Espacial (Nearest Neighbor):** Para cada estación en tierra, el sistema busca el *píxel (celda) más cercano* del modelo satelital utilizando un algoritmo *KD-Tree*.
 - * *Radio de búsqueda:* Máximo 0.1 grados (~11 km). Si no hay datos satelitales cerca, la estación se descarta.
3. **Cálculo de Diferencia:** 'Dif = Obs - Sat'.
 - * Valores positivos indican que la estación midió más lluvia que el satélite (Subestimación del modelo).
 - * Valores negativos indican lo contrario.

4. Interpretación de Gráficos

- * **📅 Series Temporales:** Permite ver si el satélite "sigue el ritmo" de la estación (captura las temporadas de lluvias y sequías) aunque los montos no sean exactos.
- * **gMaps:** Muestra la ubicación real de las estaciones sobre el fondo interpolado del satélite. Útil para identificar zonas donde el modelo falla sistemáticamente.
- * **🔍 Correlación:** Un R^2 cercano a 1 indica que el satélite es un buen predictor. Si los puntos están muy dispersos, el uso de datos satelitales debe hacerse con precaución (Bias Correction requerido).

)

```
st.info(
```

```
    "Comparación de series temporales mensuales: Lluvia Observada vs. ERA5-Land."
```

```
)
```

```
# 1. Selección de Estaciones
```

```
target_gdf = (  
    gdf_filtered  
    if gdf_filtered is not None and not gdf_filtered.empty  
    else gdf_stations  
)
```

```

if df_long.empty or target_gdf is None or target_gdf.empty:
    st.warning("Faltan datos para realizar el análisis.")
    return

# 2. Controles de UI
c1, c2 = st.columns([2, 1])
with c1:
    # Obtener rango de años disponibles EN LOS DATOS OBSERVADOS
    years = sorted(df_long[Config.YEAR_COL].unique())
    if not years:
        st.error("El dataset no contiene información de años.")
        return

    min_y, max_y = int(min(years)), int(max(years))
    # Slider con valores por defecto inteligentes
    default_start = max(min_y, max_y - 5)
    start_year, end_year = st.slider(
        "Período de Análisis:", min_y, max_y, (default_start, max_y), key="bias_rng"
    )
with c2:
    st.write("") # Espaciador para alineación vertical
    calc_btn = st.button(
        "🚀 Calcular Series", type="primary"
    )

# 3. Lógica de Cálculo (Solo si se presiona el botón)
if calc_btn:
    # Importaciones locales
    import geopandas as gpd # Necesario para exportar GeoJSON
    from scipy.interpolate import griddata
    from scipy.spatial import cKDTree

    from modules.openmeteo_api import get_historical_monthly_series

    # --- PASO 1: PROCESAR DATOS OBSERVADOS ---
    with st.spinner("1/3. Procesando datos de estaciones (Agregación Mensual)..."):
        # Filtrar datos
        mask = (
            (df_long[Config.YEAR_COL] >= start_year)
            & (df_long[Config.YEAR_COL] <= end_year)
            & (
                df_long[Config.STATION_NAME_COL].isin(
                    target_gdf[Config.STATION_NAME_COL]
                )
            )
        )

```

```

df_subset = df_long[mask].copy()

if df_subset.empty:
    st.error(
        "No se encontraron datos observados en el periodo seleccionado."
    )
    return

# Construir fecha robusta
try:
    cols_data = {"year": df_subset[Config.YEAR_COL], "day": 1}
    if (
        hasattr(Config, "MONTH_COL")
        and Config.MONTH_COL in df_subset.columns
    ):
        cols_data["month"] = df_subset[Config.MONTH_COL]
    elif "MONTH" in df_subset.columns:
        cols_data["month"] = df_subset["MONTH"]
    elif "MES" in df_subset.columns:
        cols_data["month"] = df_subset["MES"]
    else:
        pass

    df_subset["date"] = pd.to_datetime(cols_data)
except Exception:
    date_col = next(
        (
            col
            for col in df_subset.columns
            if "date" in col.lower() or "fecha" in col.lower()
        ),
        None,
    )
    if date_col:
        df_subset["date"] = pd.to_datetime(df_subset[date_col])
    else:
        st.error(
            "Error crítico: No se pudo construir la fecha. Verifique columnas Año/Mes."
        )
        return

# Normalizar fecha
df_subset["date"] = df_subset["date"].dt.to_period("M").dt.to_timestamp()

# Agrupar: Suma total por mes y estación
df_obs = (

```

```

df_subset.groupby([Config.STATION_NAME_COL, "date"])[
    Config.PRECIPITATION_COL
]
.sum()
.reset_index()
)

# --- PASO 2: DESCARGA SATELITAL (ACTUALIZADO) ---
with st.spinner("2/3. Descargando series satelitales (ERA5-Land)..."):
    # Obtener coordenadas únicas
    unique_locs = target_gdf[
        [Config.STATION_NAME_COL, "latitude", "longitude"]
    ].drop_duplicates(Config.STATION_NAME_COL)
    lats = unique_locs["latitude"].tolist()
    lons = unique_locs["longitude"].tolist()

    # Llamada a la función robusta
    df_sat = get_historical_monthly_series(
        lats, lons, f"{start_year}-01-01", f"{end_year}-12-31"
    )

if df_sat.empty:
    st.error(
        "📡 La API satelital no retornó datos. Puede ser un error de conexión o timeout."
    )
    st.info(
        "Intenta reducir el rango de años o el número de estaciones seleccionadas."
    )
    return

# --- PASO 3: EMPAREJAMIENTO ---
with st.spinner("3/3. Cruzando información espacial..."):
    obs_coords = np.column_stack(
        (unique_locs["latitude"], unique_locs["longitude"])
    )
    sat_unique = df_sat[["latitude", "longitude"]].drop_duplicates()
    sat_coords = np.column_stack(
        (sat_unique["latitude"], sat_unique["longitude"])
    )

    tree = cKDTree(sat_coords)
    dists, idxs = tree.query(obs_coords)

    map_data = []
    for i, station_name in enumerate(unique_locs[Config.STATION_NAME_COL]):
        if dists[i] < 0.1:

```

```

map_data.append(
    {
        Config.STATION_NAME_COL: station_name,
        "sat_lat": sat_coords[idxs[i]][0],
        "sat_lon": sat_coords[idxs[i]][1],
        "dist_deg": dists[i],
    }
)
)

df_map = pd.DataFrame(map_data)
if df_map.empty:
    st.error("No se encontraron coincidencias espaciales.")
    return

# MERGE 1: Obs + Map
df_merged = pd.merge(df_obs, df_map, on=Config.STATION_NAME_COL)
# MERGE 1b: Agregar coordenadas REALES
df_merged = pd.merge(
    df_merged, unique_locs, on=Config.STATION_NAME_COL, how="left"
)

# MERGE 2: + Satélite
df_final = pd.merge(
    df_merged,
    df_sat.rename(columns={"latitude": "sat_lat", "longitude": "sat_lon"}),
    on=["date", "sat_lat", "sat_lon"],
    how="inner",
)
df_final["diff_mm"] = (
    df_final[Config.PRECIPITATION_COL] - df_final["ppt_sat"]
)

st.success("✅ Análisis completado exitosamente.")

# --- VISUALIZACIÓN ---
tab_series, tab_mapa, tab_datos = st.tabs(
    ["📈 Series Temporales", "🌐 Mapa Promedio", "📋 Datos & Descargas"]
)

# TAB 1: SERIES
with tab_series:
    c_sel, _ = st.columns([1, 2])
    with c_sel:
        estaciones_disp = sorted(df_final[Config.STATION_NAME_COL].unique())

```

```

sel_st = st.selectbox(
    "Seleccionar Visualización:",
    ["Promedio Regional"] + estaciones_disp,
)

if sel_st == "Promedio Regional":
    plot_df = (
        df_final.groupby("date")[[Config.PRECIPITATION_COL, "ppt_sat"]]
        .mean()
        .reset_index()
    )
    title_plot = "Promedio Regional (Todas las Estaciones)"
else:
    plot_df = df_final[df_final[Config.STATION_NAME_COL] == sel_st]
    title_plot = f"Estación: {sel_st}"

fig = go.Figure()
fig.add_trace(
    go.Scatter(
        x=plot_df["date"],
        y=plot_df[Config.PRECIPITATION_COL],
        name="Observado (Real)",
        mode="lines+markers",
    )
)
fig.add_trace(
    go.Scatter(
        x=plot_df["date"],
        y=plot_df["ppt_sat"],
        name="Satélite (ERA5)",
        mode="lines+markers",
        line=dict(dash="dash"),
    )
)
fig.update_layout(title=title_plot, hovermode="x unified")
st.plotly_chart(fig)

# TAB 2: MAPA
with tab_mapa:
    st.markdown("**Comparativa Espacial (Promedio del Periodo)**")
    # Agregamos por ubicación REAL y SATELITAL
    map_agg = (
        df_final.groupby(
            [
                Config.STATION_NAME_COL,
                "latitude",
            ]
        )
        .mean()
        .reset_index()
    )

```

```

        "longitude",
        "sat_lat",
        "sat_lon",
    ]
)[["ppt_sat", Config.PRECIPITATION_COL]]
.mean()
.reset_index()
)

# -- GENERACIÓN DE TEXTO PARA POPUP (HOVER) --
map_agg["hover_text"] = map_agg.apply(
    lambda row: f"<b>{row[Config.STATION_NAME_COL]}</b><br>💧 Obs:  

{row[Config.PRECIPITATION_COL]:.1f} mm<br>📡 Sat: {row['ppt_sat']:.1f} mm",
    axis=1,
)
try:
    # Interpolación Satélite (Fondo)
    grid_x, grid_y = np.mgrid[
        map_agg["sat_lon"].min() : map_agg["sat_lon"].max() : 100j,
        map_agg["sat_lat"].min() : map_agg["sat_lat"].max() : 100j,
    ]
    grid_z = griddata(
        (map_agg["sat_lon"], map_agg["sat_lat"]),
        map_agg["ppt_sat"],
        (grid_x, grid_y),
        method="cubic",
    )
    fig_map = go.Figure()
    fig_map.add_trace(
        go.Contour(
            z=grid_z.T,
            x=grid_x[:, 0],
            y=grid_y[0, :],
            colorscale="Blues",
            opacity=0.6,
            showscale=False,
            name="Satélite (Fondo)",
        )
    )
    # Puntos Reales con HOVER PERSONALIZADO
    fig_map.add_trace(
        go.Scatter(
            x=map_agg["longitude"],

```

```

y=map_agg["latitude"],
mode="markers",
marker=dict(
    size=10,
    color=map_agg[Config.PRECIPITATION_COL],
    colorscale="RdBu",
    showscale=True,
    line=dict(width=1, color="black"),
),
text=map_agg["hover_text"], # Usamos la columna formateada
hoverinfo="text", # Forzamos a mostrar solo el texto
name="Estaciones",
)
)
fig_map.update_layout(
title="Fondo: Satélite | Puntos: Estaciones (Posición Real)",
height=500,
)
st.plotly_chart(fig_map, use_container_width=True)
except Exception as e:
    st.warning(f"No se pudo interpolar: {e}")
    st.map(map_agg)

# TAB 3: DATOS Y GEOJSON
with tab_datos:
    st.markdown("### Datos Tabulares")
    st.dataframe(
        df_final[
            [
                Config.STATION_NAME_COL,
                "date",
                Config.PRECIPITATION_COL,
                "ppt_sat",
                "diff_mm",
            ]
        ].sort_values(by=[Config.STATION_NAME_COL, "date"]),
    )

    c_csv, c_geo = st.columns(2)

    # 1. Descarga CSV
    with c_csv:
        csv = df_final.to_csv(index=False).encode("utf-8")
        st.download_button(
            "⬇️ Descargar Series (CSV)",
            csv,

```

```

        "validacion_mensual_satelite.csv",
        "text/csv",
    )

# 2. Descarga GEOJSON (Promedios Espaciales)
with c_geo:
    # Convertir el DataFrame agregado (map_agg) a GeoDataFrame
    # map_agg ya tiene el promedio por estación calculado en el bloque anterior (Tab 2)
    gdf_export = gpd.GeoDataFrame(
        map_agg,
        geometry=gpd.points_from_xy(
            map_agg.longitude, map_agg.latitude
        ),
        crs="EPSG:4326",
    )
    geojson_data = gdf_export.to_json()
    st.download_button(
        "🌐 Descargar Mapa Promedio (GeoJSON)",
        data=geojson_data,
        file_name="estaciones_promedio_satelite.geojson",
        mime="application/geo+json",
    )

def display_statistics_summary_tab(df_monthly, df_anual, gdf_stations, **kwargs):
    """
    Tablero de resumen estadístico de alto nivel: Récords, extremos y promedios.
    """

    st.markdown("### 🏆 Síntesis Estadística de Precipitación")
    st.info(
        "Resumen de valores extremos históricos y promedios climatológicos de la red seleccionada."
    )

    if df_monthly is None or df_monthly.empty or df_anual is None or df_anual.empty:
        st.warning("No hay suficientes datos para calcular estadísticas.")
        return

    # --- 1. PREPARACIÓN DE DATOS ---
    # Aseguramos columnas auxiliares
    if "Municipio" not in df_anual.columns and gdf_stations is not None:
        # Merge para traer municipio y cuenca si no están
        cols_to_merge = [Config.STATION_NAME_COL, Config.MUNICIPALITY_COL]
        if "Cuenca" in gdf_stations.columns:
            cols_to_merge.append("Cuenca")

```

```

# Limpieza de duplicados en gdf antes del merge
gdf_clean = gdf_stations[cols_to_merge].drop_duplicates(Config.STATION_NAME_COL)

df_anual = pd.merge(df_anual, gdf_clean, on=Config.STATION_NAME_COL, how="left")
df_monthly = pd.merge(
    df_monthly, gdf_clean, on=Config.STATION_NAME_COL, how="left"
)

# Rellenar nulos de texto
df_anual[Config.MUNICIPALITY_COL] = df_anual[Config.MUNICIPALITY_COL].fillna(
    "Desconocido"
)
df_monthly[Config.MUNICIPALITY_COL] = df_monthly[Config.MUNICIPALITY_COL].fillna(
    "Desconocido"
)

col_cuenca = "Cuenca" if "Cuenca" in df_anual.columns else None
if col_cuenca:
    df_anual[col_cuenca] = df_anual[col_cuenca].fillna("N/A")
    df_monthly[col_cuenca] = df_monthly[col_cuenca].fillna("N/A")

# --- 2. CÁLCULO DE RÉCORDS ANUALES ---
# Máximo Anual
idx_max_anual = df_anual[Config.PRECIPITATION_COL].idxmax()
row_max_anual = df_anual.loc[idx_max_anual]

# Mínimo Anual (evitando ceros si se desea, o absoluto)
# Filtramos ceros si se considera error, o los dejamos si son reales. Asumimos > 0 para "año seco real" vs
# "sin datos"
df_anual_pos = df_anual[df_anual[Config.PRECIPITATION_COL] > 0]
if not df_anual_pos.empty:
    idx_min_anual = df_anual_pos[Config.PRECIPITATION_COL].idxmin()
    row_min_anual = df_anual_pos.loc[idx_min_anual]
else:
    row_min_anual = row_max_anual # Fallback

# --- 3. CÁLCULO DE RÉCORDS MENSUALES ---
idx_max_men = df_monthly[Config.PRECIPITATION_COL].idxmax()
row_max_men = df_monthly.loc[idx_max_men]

# Mínimo Mensual > 0 (el 0 es común, buscamos el mínimo llovido)
df_men_pos = df_monthly[df_monthly[Config.PRECIPITATION_COL] > 0]
if not df_men_pos.empty:
    idx_min_men = df_men_pos[Config.PRECIPITATION_COL].idxmin()
    row_min_men = df_men_pos.loc[idx_min_men]
else:

```

```

row_min_men = row_max_men

# --- 4. PROMEDIOS REGIONALES ---
# Año más lluvioso (Promedio de todas las estaciones ese año)
regional_anual = df_anual.groupby(Config.YEAR_COL)[Config.PRECIPITATION_COL].mean()
year_max_reg = regional_anual.idxmax()
val_max_reg = regional_anual.max()

year_min_reg = regional_anual.idxmin()
val_min_reg = regional_anual.min()

# Mes Climatológico más lluvioso
regional_mensual = df_monthly.groupby(Config.MONTH_COL)[
    Config.PRECIPITATION_COL
].mean()
mes_max_reg_idx = regional_mensual.idxmax()
val_mes_max_reg = regional_mensual.max()
meses_dict = {
    1: "Ene",
    2: "Feb",
    3: "Mar",
    4: "Abr",
    5: "May",
    6: "Jun",
    7: "Jul",
    8: "Ago",
    9: "Sep",
    10: "Oct",
    11: "Nov",
    12: "Dic",
}
mes_max_name = meses_dict.get(mes_max_reg_idx, str(mes_max_reg_idx))

# --- 5. TENDENCIAS (Si hay datos suficientes) ---
# Calculamos Mann-Kendall rápido para todas las estaciones
trend_results = []
import pymannkendall as mk

stations = df_anual[Config.STATION_NAME_COL].unique()
for stn in stations:
    sub = df_anual[df_anual[Config.STATION_NAME_COL] == stn]
    if len(sub) >= 10:
        try:
            res = mk.original_test(sub[Config.PRECIPITATION_COL])
            trend_results.append({"Estacion": stn, "Slope": res.slope})
        except:

```

```

    pass

df_trends = pd.DataFrame(trend_results)
if not df_trends.empty:
    max_trend = df_trends.loc[df_trends["Slope"].idxmax()]
    min_trend = df_trends.loc[df_trends["Slope"].idxmin()]
    regional_trend = df_trends["Slope"].mean()
else:
    max_trend = {"Estacion": "N/A", "Slope": 0}
    min_trend = {"Estacion": "N/A", "Slope": 0}
    regional_trend = 0

# --- 6. ALTITUD ---
if gdf_stations is not None and Config.ALTITUDE_COL in gdf_stations.columns:
    # Filtrar solo las que tienen datos
    gdf_valid = gdf_stations[gdf_stations[Config.STATION_NAME_COL].isin(stations)]
    max_alt = gdf_valid.loc[gdf_valid[Config.ALTITUDE_COL].idxmax()]
    min_alt = gdf_valid.loc[gdf_valid[Config.ALTITUDE_COL].idxmin()]
else:
    max_alt = {"Estacion": "N/A", Config.ALTITUDE_COL: 0}
    min_alt = {"Estacion": "N/A", Config.ALTITUDE_COL: 0}

# =====
# RENDERIZADO VISUAL (TARJETAS)
# =====

# Estilos CSS para tarjetas
st.markdown(
    """
<style>
div.metric-card {
    background-color: #f9f9f9;
    border: 1px solid #e0e0e0;
    padding: 15px;
    border-radius: 10px;
    box-shadow: 2px 2px 5px rgba(0,0,0,0.05);
    margin-bottom: 10px;
}
h5.card-title { color: #1f77b4; margin-bottom: 0.5rem; font-size: 1.1rem; }
div.big-val { font-size: 1.8rem; font-weight: bold; color: #333; }
div.sub-val { font-size: 0.9rem; color: #666; margin-top: 5px; }
span.label { font-weight: bold; color: #444; }
</style>
""",
    unsafe_allow_html=True,
)

```

```

def card(title, val, unit, stn, loc_info, date_info, icon="降水 "):
    # Función helper para renderizar tarjeta HTML
    cuenca_str = (
        f"<br><span class='label'>Cuenca:</span> {loc_info.get(col_cuenca, 'N/A')}"
        if col_cuenca
        else ""
    )
    return st.markdown(
        f"""
<div class="metric-card">
    <h5 class="card-title">{icon} {title}</h5>
    <div class="big-val">{val:.1f} {unit}</div>
    <div class="sub-val">
        <span class="label">Estación:</span> {stn}<br>
        <span class="label">Ubicación:</span> {loc_info.get(Config.MUNICIPALITY_COL, 'N/A')}
{cuenca_str}<br>
        <span class="label">Fecha:</span> {date_info}
    </div>
</div>
""",
        unsafe_allow_html=True,
    )

# --- FILA 1: RÉCORDS ANUALES ---
st.markdown("#### 📈 Récords Históricos Anuales")
c1, c2 = st.columns(2)
with c1:
    card(
        "Máxima Precipitación Anual",
        row_max_anual[Config.PRECIPITATION_COL],
        "mm",
        row_max_anual[Config.STATION_NAME_COL],
        row_max_anual,
        row_max_anual[Config.YEAR_COL],
        "降水",
    )
with c2:
    card(
        "Mínima Precipitación Anual",
        row_min_anual[Config.PRECIPITATION_COL],
        "mm",
        row_min_anual[Config.STATION_NAME_COL],
        row_min_anual,
        row_min_anual[Config.YEAR_COL],
    )

```

```

        "☔",
    )

# --- FILA 2: RÉCORDS MENSUALES ---
st.markdown("#### 📆 Récords Históricos Mensuales")
c3, c4 = st.columns(2)
with c3:
    # Formatear fecha mensual
    try:
        m_date = f"{meses_dict[row_max_men[Config.MONTH_COL]]} - {row_max_men[Config.YEAR_COL]}"
    except:
        m_date = str(row_max_men[Config.YEAR_COL])
    card(
        "Máxima Lluvia Mensual",
        row_max_men[Config.PRECIPITATION_COL],
        "mm",
        row_max_men[Config.STATION_NAME_COL],
        row_max_men,
        m_date,
        "🌧",
    )
with c4:
    try:
        m_date_min = f"{meses_dict[row_min_men[Config.MONTH_COL]]} - {row_min_men[Config.YEAR_COL]}"
    except:
        m_date_min = str(row_min_men[Config.YEAR_COL])
    card(
        "Mínima Lluvia Mensual (>0)",
        row_min_men[Config.PRECIPITATION_COL],
        "mm",
        row_min_men[Config.STATION_NAME_COL],
        row_min_men,
        m_date_min,
        "☀️",
    )
st.divider()

# --- FILA 3: COMPORTAMIENTO REGIONAL ---
st.markdown("#### 🌎 Comportamiento Regional y Tendencias")

# Métricas Regionales
m1, m2, m3, m4 = st.columns(4)
m1.metric(

```

```

        "Año Más Lluvioso (Promedio)", f"{{year_max_reg}}", f"{{val_max_reg:.0f}} mm/año"
    )
m2.metric(
    "Año Menos Lluvioso (Promedio)",
    f"{{year_min_reg}}",
    f"{{val_min_reg:.0f}} mm/año",
    delta_color="inverse",
)
m3.metric(
    "Mes Más Lluvioso (Climatología)",
    f"{{mes_max_name}}",
    f"{{val_mes_max_reg:.0f}} mm/mes",
)
m4.metric(
    "Tendencia Regional Promedio",
    f"{{regional_trend:+.2f}} mm/año",
    delta="Aumento" if regional_trend > 0 else "Disminución",
)

# --- FILA 4: EXTREMOS GEOGRÁFICOS Y TENDENCIAS ---
c5, c6 = st.columns(2)

with c5:
    st.markdown("** 🏓 Extremos Altitudinales**")
    st.dataframe(
        pd.DataFrame(
            [
                {
                    "Tipo": "Mayor Altitud",
                    "Estación": max_alt[Config.STATION_NAME_COL],
                    "Altitud": f"{{max_alt[Config.ALTITUDE_COL]:.0f}} msnm",
                },
                {
                    "Tipo": "Menor Altitud",
                    "Estación": min_alt[Config.STATION_NAME_COL],
                    "Altitud": f"{{min_alt[Config.ALTITUDE_COL]:.0f}} msnm",
                },
            ],
        ),
        hide_index=True,
    )

with c6:
    st.markdown("** 💼 Extremos de Tendencia (Mann-Kendall)**")
    st.dataframe(

```

```

pd.DataFrame(
    [
        {
            "Tipo": "Mayor Aumento",
            "Estación": max_trend["Estacion"],
            "Pendiente": f"{max_trend['Slope']:.2f} mm/año",
        },
        {
            "Tipo": "Mayor Disminución",
            "Estación": min_trend["Estacion"],
            "Pendiente": f"{min_trend['Slope']:.2f} mm/año",
        },
    ],
),
hide_index=True,
)

# --- FUNCIÓN AUXILIAR: RESUMEN DE FILTROS ---
def display_current_filters(stations_sel, regions_sel, munis_sel, year_range, interpolacion, df_data,
gdf_filtered=None, **kwargs):
    """
    Muestra resumen de filtros.
    """

    # 1. SOLUCIÓN ESPACIO: Un contenedor invisible
    st.markdown("<div style='margin-top: 20px;'></div>", unsafe_allow_html=True)

    with st.expander("🔍 Resumen de Configuración (Clic para ocultar/mostrar)", expanded=True):
        col1, col2, col3, col4 = st.columns(4)
        with col1: st.metric("📅 Años", f"{year_range[0]} - {year_range[1]}")
        with col2: st.metric("📍 Estaciones", f"{len(stations_sel)}")
        with col3: st.metric("🕒 Interpolación", interpolacion)
        with col4:
            count = len(df_data) if df_data is not None else 0
            st.metric("📝 Registros", f"{count:,}")

        st.markdown("---")
        c_geo1, c_geo2 = st.columns(2)

        with c_geo1:
            if regions_sel: reg_txt = ", ".join(regions_sel)
            else: reg_txt = "Todas (Global)"
            st.markdown(f"*** 📌 Región: ** {reg_txt} **")

        with c_geo2:
            txt_munis = "Todos los disponibles"

```

```
lista_nombres = []
if munis_sel: lista_nombres = munis_sel
elif gdf_filtered is not None and not gdf_filtered.empty:
    col_muni = next((c for c in gdf_filtered.columns if "muni" in c.lower() or "ciud" in c.lower()), None)
    if col_muni: lista_nombres = sorted(gdf_filtered[col_muni].astype(str).unique().tolist())

if lista_nombres:
    if len(lista_nombres) > 3:
        muestras = ", ".join(lista_nombres[:3])
        restantes = len(lista_nombres) - 3
        txt_munis = f"{muestras} y {restantes} más..."
    else: txt_munis = ", ".join(lista_nombres)
    if not munis_sel: txt_munis = f"(Incluye: {txt_munis})"

st.markdown(f"**🌐 Municipios:** {txt_munis}")
```