```python
# modules/forecasting.py

import numpy as np
import pandas as pd
import pmdarima as pm
import streamlit as st
from prophet import Prophet
from sklearn.metrics import mean_absolute_error, mean_squared_error
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.stattools import acf, pacf

from modules.config import Config


def evaluate_forecast(y_true, y_pred):
    """Calcula métricas de error."""
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    mae = mean_absolute_error(y_true, y_pred)
    return {"RMSE": rmse, "MAE": mae}


def get_decomposition_results(series, period=12, model="additive"):
    """Descompone la serie de tiempo."""
    # Asegurar integridad
    series_clean = series.asfreq("MS").interpolate(method="time").dropna()
    if len(series_clean) < period * 2:
        raise ValueError("Serie muy corta para descomposición estacional.")
    return seasonal_decompose(series_clean, model=model, period=period)


def create_acf_chart(series, max_lag):
    import plotly.graph_objects as go

    try:
        acf_vals = acf(series, nlags=max_lag)
        fig = go.Figure(data=[go.Bar(x=list(range(len(acf_vals))), y=acf_vals)])
        fig.update_layout(
            title="Autocorrelación (ACF)",
            xaxis_title="Lag",
            yaxis_title="ACF",
            height=350,
        )
        return fig
    except:
        return go.Figure()
```

```
def create_pacf_chart(series, max_lag):
    import plotly.graph_objects as go

    try:
        pacf_vals = pacf(series, nlags=max_lag)
        fig = go.Figure(data=[go.Bar(x=list(range(len(pacf_vals))), y=pacf_vals)])
        fig.update_layout(
            title="Autocorrelación Parcial (PACF)",
            xaxis_title="Lag",
            yaxis_title="PACF",
            height=350,
        )
        return fig
    except:
        return go.Figure()


@st.cache_data(show_spinner=False)
def auto_arima_search(ts_data, test_size):
    train = ts_data[:-test_size]
    model = pm.auto_arima(
        train,
        start_p=0,
        start_q=0,
        m=12,
        seasonal=True,
        trace=False,
        error_action="ignore",
    )
    return model.order, model.seasonal_order


@st.cache_data
def generate_sarima_forecast(
    ts_data_raw, order, seasonal_order, horizon, test_size=12, regressors=None
):
    """
    Pronóstico SARIMA con soporte para regresores exógenos.
    BLINDADO: Maneja índices y cruces de datos para evitar errores de límites.
    """
    # 1. Preparar Serie Principal
    df = ts_data_raw.copy()
    # Asegurar nombre de columna de fecha
    if Config.DATE_COL not in df.columns:
```

```
    # Intentar encontrar la columna de fecha
    cols = df.columns
    # Si hay una columna llamada 'index' o similar que parece fecha
    if "index" in cols:
        df = df.rename(columns={"index": Config.DATE_COL})
    elif "ds" in cols:
        df = df.rename(columns={"ds": Config.DATE_COL})

# Convertir a datetime y setear indice
df[Config.DATE_COL] = pd.to_datetime(df[Config.DATE_COL])
df = df.set_index(Config.DATE_COL).sort_index()

# Extraer serie y limpiar
ts = df[Config.PRECIPITATION_COL].asfreq("MS").interpolate(method="time").dropna()

if len(ts) < 24:
    raise ValueError("Serie de tiempo insuficiente tras limpieza (<24 meses).")

# 2. Preparar Regresores (Exógenos)
exog_train = None
exog_test = None
exog_future = None
exog_full = None

if regressors is not None and not regressors.empty:
    try:
        # Asegurar índice fecha en regresores
        reg_clean = regressors.copy()
        if Config.DATE_COL in reg_clean.columns:
            reg_clean = reg_clean.set_index(Config.DATE_COL)
        elif "ds" in reg_clean.columns:
            reg_clean = reg_clean.set_index("ds")

        reg_clean = (
            reg_clean.sort_index()
            .asfreq("MS")
            .interpolate(method="time")
            .bfill()
            .ffill()
        )

        # Alinear con la serie histórica (intersección)
        # Esto es clave: solo usamos los tiempos donde existen ambos datos
        common_idx = ts.index.intersection(reg_clean.index)
        if (
            len(common_idx) < len(ts) * 0.8
```

```
        ):  # Si perdemos muchos datos, mejor no usar regresor
            pass
        else:
            exog_full = reg_clean.loc[ts.index]

            # Preparar futuro
            last_date = ts.index[-1]
            future_dates = pd.date_range(
                start=last_date + pd.DateOffset(months=1),
                periods=horizon,
                freq="MS",
            )
            # Verificar si tenemos regresores para el futuro
            if future_dates.isin(reg_clean.index).all():
                exog_future = reg_clean.loc[future_dates]
            else:
                # Si falta futuro, no podemos usar regresores para pronosticar
                exog_full = None
    except:
        exog_full = None  # Fallback seguro

# 3. Split
# Asegurar que test_size no rompa la serie
real_test_size = min(test_size, int(len(ts) * 0.2))

train = ts.iloc[:-real_test_size]
test = ts.iloc[-real_test_size:]

if exog_full is not None:
    exog_train = exog_full.iloc[:-real_test_size]
    exog_test = exog_full.iloc[-real_test_size:]

# 4. Modelo (Train)
model = SARIMAX(
    train,
    order=order,
    seasonal_order=seasonal_order,
    exog=exog_train,
    enforce_stationarity=False,
    enforce_invertibility=False,
)
res = model.fit(disp=False)

# Evaluar
pred = res.get_forecast(steps=len(test), exog=exog_test)
metrics = evaluate_forecast(test, pred.predicted_mean)
```

```
    # 5. Modelo Final (Full)
    full_model = SARIMAX(
        ts,
        order=order,
        seasonal_order=seasonal_order,
        exog=exog_full,
        enforce_stationarity=False,
        enforce_invertibility=False,
    )
    full_res = full_model.fit(disp=False)

    # Pronóstico Futuro
    # Si tenemos regresores futuros, los usamos. Si no, pronóstico simple.
    final_forecast = full_res.get_forecast(steps=horizon, exog=exog_future)

    fc_val = final_forecast.predicted_mean
    ci = final_forecast.conf_int()

    return ts, fc_val, ci, metrics, fc_val.reset_index()


@st.cache_data
def generate_prophet_forecast(ts_data_raw, horizon, test_size=12, regressors=None):
    """Pronóstico Prophet con regresores."""
    # Preparar
    df = ts_data_raw.rename(
        columns={Config.DATE_COL: "ds", Config.PRECIPITATION_COL: "y"}
    )
    df["ds"] = pd.to_datetime(df["ds"])
    df = df.sort_values("ds").dropna(subset=["y"])

    if len(df) < 24:
        raise ValueError("Datos insuficientes para Prophet.")

    # Regresores
    reg_cols = []
    if regressors is not None and not regressors.empty:
        r = regressors.copy()
        if "ds" not in r.columns and Config.DATE_COL in r.columns:
            r = r.rename(columns={Config.DATE_COL: "ds"})

        # Merge
        df = pd.merge(df, r, on="ds", how="left")
        reg_cols = [c for c in r.columns if c != "ds"]
        # Llenar huecos
```

```
    for c in reg_cols:
        df[c] = df[c].interpolate(method="linear").bfill().ffill()
    # Limpiar filas que sigan con nulos en regresores
    df = df.dropna()

# Split
real_test = min(test_size, int(len(df) * 0.2))
train = df.iloc[:-real_test]
test = df.iloc[-real_test:]

# Modelo Eval
m = Prophet()
for c in reg_cols:
    m.add_regressor(c)
m.fit(train)

future_test = test.drop(columns=["y"])
fc_test = m.predict(future_test)
metrics = evaluate_forecast(test["y"], fc_test["yhat"])

# Modelo Final
m_full = Prophet()
for c in reg_cols:
    m_full.add_regressor(c)
m_full.fit(df)

future = m_full.make_future_dataframe(periods=horizon, freq="MS")
# Pegar regresores futuros
if reg_cols:
    future = pd.merge(future, r, on="ds", how="left")
    for c in reg_cols:
        future[c] = future[c].interpolate(method="linear").bfill().ffill()
    future = (
        future.dropna()
    )  # Si no hay futuro en regresores, cortamos el pronóstico

forecast = m_full.predict(future)
return m_full, forecast, metrics
```