Nixtla Forecasting Framework - Complete Documentation

Author: Generated for Deep Forecasting Course **Date:** 2025 **Modules:** `df_statsforecast.py`, `df_mlforecast.py`, `df_neuralforecast.py`

Table of Contents

1. Overview
2. Forecasting Paradigms
3. Nixtla Implementation Details
4. API Documentation
5. Integration Guide for Web App
6. Usage Examples
7. Performance Comparison

Overview

This documentation covers three production-ready Python modules for time series forecasting using Nixtla's ecosystem:

- **StatsForecast** (`df_statsforecast.py`): Statistical models (ARIMA, ETS, baselines)
- **MLForecast** (`df_mlforecast.py`): Machine learning models (XGBoost, LightGBM, RandomForest, CatBoost)
- **NeuralForecast** (`df_neuralforecast.py`): Deep learning models (MLP, RNN, LSTM, NBEATS, NHITS, TCN)

Each module implements three forecasting strategies:

1. **One-step ahead forecasting** (h=1 iterative)
2. **Multi-step recursive forecasting** (iterative predictions with error propagation)
3. **Multi-output direct forecasting** (all predictions simultaneously)

Forecasting Paradigms

1. One-Step Ahead Forecasting

Definition: Predict only the next single time step (h=1) for each timestamp in the test set.

Process:

```
For each timestamp t in test set:
  1. Train on actuals up to t-1
  2. Predict ŷ(t) using h=1
```

```
  3. Prediction does NOT feed into next step
  4. Proceed iteratively
```

Characteristics:

- **Most accurate** for immediate next step predictions
- **"Optimistic backtesting"** - uses future actual values
- **Not real deployment** - requires continuous retraining
- **Computationally expensive** - refits model for each prediction

Use Cases:

- Establishing upper-bound performance
- Understanding model's single-step capability
- Evaluation when you can update your data frequently

Error Propagation: None - each prediction uses actual historical data

---

2. Multi-Step Recursive Forecasting

Definition: Predict multiple steps ahead by iteratively feeding predictions back as inputs.

Process:

```
For forecasting horizon h:
  1. Predict step 1 using actual history
  2. Use prediction from step 1 as input for step 2
  3. Use prediction from step 2 as input for step 3
  4. Repeat until horizon h is reached
```

Characteristics:

- **Real deployment simulation** - mimics production scenarios
- **Error accumulation** - errors compound over longer horizons
- **Computationally efficient** - train once, predict many
- **Default behavior** for most forecasting libraries

Use Cases:

- Production forecasting where future actuals are unknown
- Realistic performance assessment
- When computational resources are limited

Error Propagation: High - errors accumulate at each step as predictions are reused

---

3. Multi-Output Direct Forecasting

Definition: Predict all future time steps simultaneously in a single forward pass.

Process:

```
For forecasting horizon h:
  1. Train model(s) to predict all h steps at once
  2. Generate forecasts [ŷ(t+1), ŷ(t+2), ..., ŷ(t+h)] simultaneously
  3. No iterative feedback of predictions
```

Characteristics:

- **Better accuracy** than recursive for multi-step
- **No error accumulation** - predictions are independent
- **Computationally expensive** - requires more training
- **Requires sufficient data** to learn multi-step patterns

Implementation Approaches:

A. Multiple Models (Direct Strategy - MLForecast):

- Train H separate models, one per forecast step
- Each model specializes in predicting a specific horizon
- Implemented via `max_horizon` parameter in MLForecast

B. Single Model (True Multi-Output - NeuralForecast):

- Train one model that outputs h values simultaneously
- Neural network architecture designed for multi-output
- Default behavior for NBEATS, NHITS, MLP, TCN

Use Cases:

- When maximum accuracy is critical
- Sufficient computational resources available
- Medium to long forecast horizons
- Models: ML and Neural (NOT statistical models like ARIMA/ETS)

Error Propagation: Minimal - forecasts generated simultaneously, no iterative feedback

---

Nixtla Implementation Details

How Each Forecasting Paradigm is Implemented

StatsForecast (Statistical Models)

Strategy	Support	Implementation	Notes
One-Step	✅ Yes	Manual iteration with `h=1`	Refit model for each prediction
Multi-Step	✅ Yes	Default `sf.forecast(h=H)`	Recursive forecasting built-in
Multi-Output	❌ No	Raises `NotImplementedError`	ARIMA/ETS cannot do multi-output

Why Multi-Output is NOT supported:

- Statistical models (ARIMA/ETS) are inherently recursive
- They model temporal dependencies sequentially
- Cannot generate all future values simultaneously
- Use ML or Neural models for multi-output forecasting

Example Implementation:

```
# One-Step Ahead (Manual Loop)
for i in range(len(test)):
    current_train = data[:train_size + i]
    model = AutoARIMA(season_length=12)
    sf = StatsForecast(models=[model], freq='MS')
    forecast = sf.forecast(df=current_train, h=1)
    predictions.append(forecast['AutoARIMA'].values[0])

# Multi-Step Recursive (Default Behavior)
model = AutoARIMA(season_length=12)
sf = StatsForecast(models=[model], freq='MS')
forecast = sf.forecast(df=train, h=12)  # Recursive forecasting

# Multi-Output (Not Supported)
# Raises NotImplementedError - use ML/Neural models instead
```

MLForecast (Machine Learning Models)

Strategy	Support	Implementation	Notes
One-Step	✅ Yes	Manual iteration with `h=1`	Refit model for each prediction
Multi-Step	✅ Yes	Default `mlf.predict(h)`	Recursive by default
Multi-Output	✅ Yes	`mlf.fit(max_horizon=H)`	Direct strategy - one model per step

Key Features:

- **Lag features:** Automatically creates windowed features
- **Target transforms:** Supports differencing, scaling, etc.
- **Flexible models:** XGBoost, LightGBM, RandomForest, CatBoost, Linear

Implementation Details:

```
# One-Step Ahead (Manual Loop)
for i in range(len(test)):
    current_train = data[:train_size + i]
    mlf = MLForecast(models=[XGBRegressor()], freq='MS', lags=[1, 12])
    mlf.fit(df=current_train)
    forecast = mlf.predict(h=1)
```

```
    predictions.append(forecast['XGBRegressor'].values[0])

# Multi-Step Recursive (Default Behavior)
mlf = MLForecast(models=[XGBRegressor()], freq='MS', lags=[1, 12])
mlf.fit(df=train)
forecast = mlf.predict(h=12)  # Recursive: uses own predictions

# Multi-Output Direct (max_horizon Parameter)
mlf = MLForecast(models=[XGBRegressor()], freq='MS', lags=[1, 12])
mlf.fit(df=train, max_horizon=12)  # Train 12 separate models
forecast = mlf.predict(h=12)  # Each model predicts its specific step
```

Recursive vs Direct:

- **Recursive** (`predict(h)`): Fast, but error accumulation
- **Direct** (`fit(max_horizon=H)`): Slower training, better accuracy

NeuralForecast (Deep Learning Models)

Strategy	Support	Implementation	Notes
One-Step	✅ Yes	Manual iteration with `h=1`	Refit model for each prediction
Multi-Step Recursive	✅ Partial	RNN/LSTM/GRU with `recurrent=True`	Only for recurrent models
Multi-Output	✅ Yes	Default for most models	NBEATS, NHITS, MLP, TCN

Model-Specific Behavior:

A. Recurrent Models (RNN, LSTM, GRU):

- Support both recursive (`recurrent=True`) and multi-output (`recurrent=False`)
- Recursive: uses own predictions iteratively
- Multi-output: predicts all steps simultaneously

B. Multi-Output Models (NBEATS, NHITS, MLP, TCN):

- Only support multi-output direct forecasting
- Cannot do recursive forecasting
- Default behavior: predict all h steps at once

Implementation Details:

```
# One-Step Ahead (Manual Loop)
for i in range(len(test)):
    current_train = data[:train_size + i]
    model = NHITS(h=1, input_size=12, max_steps=100)
    nf = NeuralForecast(models=[model], freq='MS')
```

```
    nf.fit(df=current_train)
    forecast = nf.predict(df=current_train)
    predictions.append(forecast['NHITS'].values[0])

# Multi-Step Recursive (RNN/LSTM/GRU only)
model = LSTM(h=12, input_size=12, recurrent=True, max_steps=100)
nf = NeuralForecast(models=[model], freq='MS')
nf.fit(df=train)
forecast = nf.predict(df=train)  # Recursive forecasting

# Multi-Output Direct (Default for most models)
model = NBEATS(h=12, input_size=24, max_steps=100)
nf = NeuralForecast(models=[model], freq='MS')
nf.fit(df=train)
forecast = nf.predict(df=train)  # All 12 steps at once
```

Key Parameters:

- `h`: Forecast horizon (number of steps ahead)
- `input_size`: Length of input window (number of past observations)
- `recurrent`: Whether to use recursive forecasting (RNN/LSTM/GRU only)
- `max_steps`: Number of training epochs

API Documentation

df_statsforecast.py

Class: `StatsforecastForecaster`

Production-ready forecaster for statistical models.

Constructor:

```
StatsforecastForecaster(
    model_type: str,              # 'arima', 'auto_arima', 'auto_ets', 
'naive', 'seasonal_naive', 'rw_drift'
    freq: str = 'MS',             # Frequency: 'MS' (month), 'D' (day), 
'H' (hour)
    season_length: int = 12,      # Seasonal period (e.g., 12 for monthly 
data)
    **model_params                # Model-specific parameters
)
```

Methods:

`one_step_forecast(train_df, test_df, target_col='y', date_col='ds', unique_id='series_1')`

One-step ahead forecasting with iterative refitting.

Parameters:

- `train_df` (DataFrame): Training data
- `test_df` (DataFrame): Test data
- `target_col` (str): Name of target column (default: 'y')
- `date_col` (str): Name of date column (default: 'ds')
- `unique_id` (str): Series identifier (default: 'series_1')

Returns:

- `dict` with keys:
  - `'forecasts'`: DataFrame with columns `[unique_id, ds, y_true, y_pred]`
  - `'metrics'`: Dict with `{'mae': float, 'rmse': float, 'mape': float}`

Example:

```
forecaster = StatsforecastForecaster(
    model_type='auto_arima',
    freq='MS',
    season_length=12,
    seasonal=True
)

results = forecaster.one_step_forecast(train_df, test_df)
print(f"MAE: {results['metrics']['mae']:.2f}")
print(results['forecasts'].head())
```

Error Cases:

- Raises `ValueError` if `model_type` is invalid
- Returns `NaN` for MAPE if target contains zeros

`multi_step_forecast(train_df, horizon, target_col='y', date_col='ds', unique_id='series_1', test_df=None)`

Multi-step recursive forecasting using default StatsForecast behavior.

Parameters:

- `train_df` (DataFrame): Training data
- `horizon` (int): Forecast horizon
- `target_col` (str): Name of target column
- `date_col` (str): Name of date column
- `unique_id` (str): Series identifier
- `test_df` (DataFrame, optional): Test data for metrics

Returns:

- `dict` with keys:

- `'forecasts'`: DataFrame with predictions
- `'metrics'`: Dict with MAE/RMSE/MAPE (if test_df provided, else None)

Example:

```
forecaster = StatsforecastForecaster(
    model_type='auto_ets',
    freq='MS',
    season_length=12,
    model='ZZZ'  # Auto ETS model selection
)

results = forecaster.multi_step_forecast(
    train_df,
    horizon=12,
    test_df=test_df
)
print(f"MAE: {results['metrics']['mae']:.2f}")
```

`multi_output_forecast(train_df, horizon, ...)`

NOT SUPPORTED - Raises `NotImplementedError`.

ARIMA/ETS models cannot perform multi-output forecasting. Use ML or Neural models instead.

Example:

```
forecaster = StatsforecastForecaster(model_type='arima', freq='MS')

try:
    forecaster.multi_output_forecast(train_df, horizon=12)
except NotImplementedError as e:
    print(e)  # "Multi-output forecasting is NOT supported for statistical
models..."
```

---

df_mlforecast.py

Class: `MLForecastForecaster`

Production-ready forecaster for machine learning models.

Constructor:

```
MLForecastForecaster(
    model_type: str,                     # 'xgboost', 'lightgbm',
'random_forest', 'catboost', 'linear'
```

```
    freq: str = 'MS',                    # Frequency
    lags: List[int] = None,              # Lag features (default: [1,
12])
    lag_transforms: Dict = None,         # Lag transformations (optional)
    date_features: List[str] = None,     # Date features (default: [])
    target_transforms: List = None,      # Target transforms (default:
[])
    **model_params                       # Model-specific parameters
)
```

Methods:

`one_step_forecast(train_df, test_df, ...)`

One-step ahead forecasting with iterative refitting.

Parameters: Same as StatsForecast

Returns: Same as StatsForecast

Example:

```
from mlforecast.target_transforms import Differences

forecaster = MLForecastForecaster(
    model_type='xgboost',
    freq='MS',
    lags=[1, 12],
    target_transforms=[Differences([1])],  # First differencing
    n_estimators=200,
    max_depth=6,
    learning_rate=0.1
)

results = forecaster.one_step_forecast(train_df, test_df)
```

Error Cases:

- Raises `ImportError` if XGBoost/LightGBM/CatBoost not installed
- Requires at least `max(lags)` observations in training data

`multi_step_forecast(train_df, horizon, ...)`

Multi-step recursive forecasting (default MLForecast behavior).

Example:

```python
forecaster = MLForecastForecaster(
    model_type='lightgbm',
    freq='MS',
    lags=[1, 2, 3, 12],
    target_transforms=[Differences([1, 12])],  # Detrend and deseasonalize
    n_estimators=100
)

results = forecaster.multi_step_forecast(train_df, horizon=12,
test_df=test_df)
```

`multi_output_forecast(train_df, horizon, ...)`

Multi-output direct forecasting via `max_horizon` parameter.

How it works:

- Trains H separate models (one per forecast step)
- Each model specializes in predicting a specific horizon
- No error accumulation between steps

Example:

```python
forecaster = MLForecastForecaster(
    model_type='random_forest',
    freq='MS',
    lags=[1, 12],
    n_estimators=100
)

# Trains 12 models: one for h=1, one for h=2, ..., one for h=12
results = forecaster.multi_output_forecast(train_df, horizon=12,
test_df=test_df)
print(f"Multi-output MAE: {results['metrics']['mae']:.2f}")
```

Trade-offs:

- **Advantages:** Better accuracy, no error accumulation
- **Disadvantages:** Longer training time (H models vs 1 model)

---

df_neuralforecast.py

Class: `NeuralForecastForecaster`

Production-ready forecaster for neural network models.

Constructor:

```
NeuralForecastForecaster(
    model_type: str,            # 'mlp', 'rnn', 'lstm', 'gru', 'nbeats',
'nhits', 'tcn'
    freq: str = 'MS',           # Frequency
    input_size: int = 12,       # Length of input window
    horizon: int = 1,           # Forecast horizon
    **model_params              # Model-specific parameters
)
```

Methods:

`one_step_forecast(train_df, test_df, ...)`

One-step ahead forecasting with iterative refitting.

Example:

```
forecaster = NeuralForecastForecaster(
    model_type='mlp',
    freq='MS',
    input_size=12,
    horizon=1,
    hidden_size=32,
    num_layers=2,
    max_steps=100,         # Training epochs
    scaler_type='robust',
    random_seed=42
)

results = forecaster.one_step_forecast(train_df, test_df)
```

Note: This refits the neural network for EACH prediction, which is computationally expensive but provides most accurate one-step forecasts.

`multi_step_forecast(train_df, horizon, ..., use_recurrent=False)`

Multi-step forecasting with option for recursive or direct.

Parameters:

- `use_recurrent` (bool): Use recursive forecasting (only for RNN/LSTM/GRU)

Example (Recursive - RNN/LSTM/GRU):

```
forecaster = NeuralForecastForecaster(
    model_type='lstm',
    freq='MS',
```

```
    input_size=12,
    horizon=12,
    encoder_hidden_size=16,
    max_steps=300
)

# Recursive forecasting: uses own predictions iteratively
results = forecaster.multi_step_forecast(
    train_df,
    horizon=12,
    test_df=test_df,
    use_recurrent=True  # Enable recursive mode
)
```

Example (Direct - All Models):

```
forecaster = NeuralForecastForecaster(
    model_type='nhits',
    freq='MS',
    input_size=24,
    horizon=12,
    max_steps=200
)

# Multi-output direct: predicts all steps simultaneously
results = forecaster.multi_step_forecast(
    train_df,
    horizon=12,
    test_df=test_df,
    use_recurrent=False  # Direct forecasting (default)
)
```

`multi_output_forecast(train_df, horizon, ...)`

Multi-output direct forecasting (default for most neural models).

Example:

```
forecaster = NeuralForecastForecaster(
    model_type='nbeats',
    freq='MS',
    input_size=24,
    horizon=12,
    stack_types=['trend', 'seasonality'],
    max_steps=200
)
```

```python
results = forecaster.multi_output_forecast(train_df, horizon=12,
test_df=test_df)
```

Supported Models:

- ✅ NBEATS, NHITS, MLP, TCN (default multi-output)
- ✅ RNN, LSTM, GRU (with `recurrent=False`)

Integration Guide for Web App

Application Architecture

```
Full-Stack Forecasting Web App
│
├── Frontend (React/Vue/Streamlit)
│   ├── File Upload (CSV)
│   ├── Configuration Panel
│   │   ├── Target Selection
│   │   ├── Feature Selection
│   │   ├── Horizon (H)
│   │   ├── Forecasting Mode (one-step, multi-step, multi-output)
│   │   ├── Model Family (econometrics, ML, DL)
│   │   ├── Specific Model
│   │   └── Train/Test Split %
│   └── Results Display
│       ├── Prediction Tables
│       ├── Metrics (MAPE, MAE, RMSE)
│       └── Comparison Charts
│
└── Backend (FastAPI/Flask)
    ├── API Endpoints
    │   ├── /upload (POST)
    │   ├── /configure (POST)
    │   ├── /forecast (POST)
    │   └── /results (GET)
    │
    ├── Core Modules
    │   ├── df_statsforecast.py
    │   ├── df_mlforecast.py
    │   └── df_neuralforecast.py
    │
    └── Utilities
        ├── data_validation.py
        ├── preprocessing.py
        └── visualization.py
```

Backend API Implementation

Example using FastAPI

File Structure:

```
forecasting_app/
│
├── api/
│   ├── __init__.py
│   ├── main.py              # FastAPI app
│   └── endpoints.py         # API endpoints
│
├── core/
│   ├── __init__.py
│   ├── df_statsforecast.py
│   ├── df_mlforecast.py
│   └── df_neuralforecast.py
│
├── utils/
│   ├── __init__.py
│   ├── data_handler.py      # Data loading/validation
│   ├── preprocessing.py     # Data preprocessing
│   └── visualization.py     # Chart generation
│
└── requirements.txt
```

main.py:

```python
from fastapi import FastAPI, File, UploadFile, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
import pandas as pd
from typing import Literal, Optional, List

from core.df_statsforecast import StatsforecastForecaster
from core.df_mlforecast import MLForecastForecaster
from core.df_neuralforecast import NeuralForecastForecaster

app = FastAPI(title="Nixtla Forecasting API")

# CORS middleware for frontend communication
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
)

# Request models
class ForecastRequest(BaseModel):
    target_col: str
```

```
    features: Optional[List[str]] = None
    horizon: int
    mode: Literal['one_step', 'multi_step', 'multi_output']
    model_family: Literal['econometrics', 'ml', 'dl']
    model_name: str
    train_split: float = 0.8

# Global data store (use Redis/database in production)
data_store = {}

@app.post("/upload")
async def upload_data(file: UploadFile = File(...)):
    """Upload CSV file and return data preview."""
    try:
        df = pd.read_csv(file.file)
        data_id = str(hash(file.filename))
        data_store[data_id] = df

        return {
            "data_id": data_id,
            "columns": df.columns.tolist(),
            "shape": df.shape,
            "preview": df.head().to_dict()
        }
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))

@app.post("/forecast")
async def create_forecast(data_id: str, request: ForecastRequest):
    """Generate forecasts based on configuration."""

    if data_id not in data_store:
        raise HTTPException(status_code=404, detail="Data not found")

    df = data_store[data_id]

    # Train/test split
    split_idx = int(len(df) * request.train_split)
    train_df = df.iloc[:split_idx]
    test_df = df.iloc[split_idx:]

    # Select forecaster based on model family
    if request.model_family == 'econometrics':
        forecaster = StatsforecastForecaster(
            model_type=request.model_name,
            freq='MS',
            season_length=12
        )
    elif request.model_family == 'ml':
        forecaster = MLForecastForecaster(
            model_type=request.model_name,
            freq='MS',
            lags=[1, 12]
        )
```

```python
    elif request.model_family == 'dl':
        forecaster = NeuralForecastForecaster(
            model_type=request.model_name,
            freq='MS',
            input_size=12,
            horizon=request.horizon
        )

    # Generate forecast based on mode
    try:
        if request.mode == 'one_step':
            results = forecaster.one_step_forecast(
                train_df,
                test_df,
                target_col=request.target_col
            )
        elif request.mode == 'multi_step':
            results = forecaster.multi_step_forecast(
                train_df,
                horizon=request.horizon,
                target_col=request.target_col,
                test_df=test_df
            )
        elif request.mode == 'multi_output':
            if request.model_family == 'econometrics':
                raise ValueError(
                    "Multi-output not supported for statistical models"
                )
            results = forecaster.multi_output_forecast(
                train_df,
                horizon=request.horizon,
                target_col=request.target_col,
                test_df=test_df
            )

        return {
            "forecasts": results['forecasts'].to_dict(orient='records'),
            "metrics": results['metrics']
        }

    except NotImplementedError as e:
        raise HTTPException(status_code=400, detail=str(e))
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Frontend Integration

Example using React

Forecast Configuration Component:

```
import React, { useState } from 'react';
import axios from 'axios';

function ForecastConfig({ dataId, columns }) {
  const [config, setConfig] = useState({
    target_col: '',
    horizon: 12,
    mode: 'multi_step',
    model_family: 'ml',
    model_name: 'xgboost',
    train_split: 0.8
  });

  const [results, setResults] = useState(null);
  const [loading, setLoading] = useState(false);

  const modelOptions = {
    econometrics: ['auto_arima', 'auto_ets', 'naive', 'seasonal_naive'],
    ml: ['xgboost', 'lightgbm', 'random_forest', 'catboost'],
    dl: ['mlp', 'lstm', 'nbeats', 'nhits']
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    setLoading(true);

    try {
      const response = await axios.post(
        `http://localhost:8000/forecast?data_id=${dataId}`,
        config
      );
      setResults(response.data);
    } catch (error) {
      alert(error.response?.data?.detail || 'Forecast failed');
    } finally {
      setLoading(false);
    }
  };

  return (
    <div className="forecast-config">
      <h2>Forecast Configuration</h2>
      <form onSubmit={handleSubmit}>
        {/* Target Selection */}
        <div>
          <label>Target Column:</label>
          <select
            value={config.target_col}
```

```
            onChange={(e) => setConfig({...config, target_col:
e.target.value})}
          >
            <option value="">Select...</option>
            {columns.map(col => <option key={col} value={col}>{col}
</option>)}
          </select>
        </div>

        {/* Horizon */}
        <div>
          <label>Forecast Horizon:</label>
          <input
            type="number"
            min="1"
            value={config.horizon}
            onChange={(e) => setConfig({...config, horizon:
parseInt(e.target.value)})}
          />
        </div>

        {/* Forecasting Mode */}
        <div>
          <label>Forecasting Mode:</label>
          <select
            value={config.mode}
            onChange={(e) => setConfig({...config, mode: e.target.value})}
          >
            <option value="one_step">One-Step Ahead</option>
            <option value="multi_step">Multi-Step Recursive</option>
            <option value="multi_output">Multi-Output Direct</option>
          </select>
        </div>

        {/* Model Family */}
        <div>
          <label>Model Family:</label>
          <select
            value={config.model_family}
            onChange={(e) => setConfig({
              ...config,
              model_family: e.target.value,
              model_name: modelOptions[e.target.value][0]
            })}
          >
            <option value="econometrics">Econometric (ARIMA/ETS)</option>
            <option value="ml">Machine Learning</option>
            <option value="dl">Deep Learning</option>
          </select>
        </div>

        {/* Specific Model */}
        <div>
          <label>Model:</label>
```

```
          <select
            value={config.model_name}
            onChange={(e) => setConfig({...config, model_name:
e.target.value})}
          >
            {modelOptions[config.model_family].map(model => (
              <option key={model} value={model}>{model.toUpperCase()}
</option>
            ))}
          </select>
        </div>

        {/* Train/Test Split */}
        <div>
          <label>Train Split: {(config.train_split * 100).toFixed(0)}%
</label>
          <input
            type="range"
            min="0.5"
            max="0.9"
            step="0.05"
            value={config.train_split}
            onChange={(e) => setConfig({...config, train_split:
parseFloat(e.target.value)})}
          />
        </div>

        <button type="submit" disabled={loading || !config.target_col}>
          {loading ? 'Forecasting...' : 'Generate Forecast'}
        </button>
      </form>

      {/* Results Display */}
      {results && (
        <div className="results">
          <h3>Forecast Results</h3>
          <div className="metrics">
            <p>MAE: {results.metrics.mae.toFixed(2)}</p>
            <p>RMSE: {results.metrics.rmse.toFixed(2)}</p>
            <p>MAPE: {results.metrics.mape.toFixed(2)}%</p>
          </div>
          {/* Add charts and tables here */}
        </div>
      )}
    </div>
  );
}

export default ForecastConfig;
```

Deployment on Render.com

1. Project Structure:

```
forecasting_app/
├── Dockerfile
├── requirements.txt
├── render.yaml
├── api/
│   └── main.py
├── core/
│   ├── df_statsforecast.py
│   ├── df_mlforecast.py
│   └── df_neuralforecast.py
└── frontend/
    └── (React build files)
```

2. Dockerfile:

```
FROM python:3.11-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    g++ \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements and install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Expose port
EXPOSE 8000

# Run application
CMD ["uvicorn", "api.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

3. requirements.txt:

```
fastapi==0.104.1
uvicorn[standard]==0.24.0
pandas==2.1.3
numpy==1.26.2
statsforecast==1.6.0
mlforecast==0.10.0
```

```
neuralforecast==1.6.4
utilsforecast==0.0.27
xgboost==2.0.2
lightgbm==4.1.0
catboost==1.2.2
scikit-learn==1.3.2
pydantic==2.5.0
python-multipart==0.0.6
```

4. render.yaml:

```yaml
services:
  - type: web
    name: forecasting-api
    env: python
    buildCommand: pip install -r requirements.txt
    startCommand: uvicorn api.main:app --host 0.0.0.0 --port $PORT
    envVars:
      - key: PYTHON_VERSION
        value: 3.11.0
```

Usage Examples

Complete Workflow Example

```python
import pandas as pd
from core.df_statsforecast import StatsforecastForecaster
from core.df_mlforecast import MLForecastForecaster
from core.df_neuralforecast import NeuralForecastForecaster

# Load data
data = pd.read_csv('airline_passengers.csv')
data['ds'] = pd.to_datetime(data['Month'])
data = data.rename(columns={'Passengers': 'y'})

# Train/test split
train_size = int(len(data) * 0.8)
train = data.iloc[:train_size]
test = data.iloc[train_size:]

print(f"Train: {len(train)}, Test: {len(test)}")

# ============================================
# 1. ECONOMETRIC MODELS (StatsForecast)
# ============================================

print("\n" + "="*80)
print("ECONOMETRIC MODELS")
```

```
print("="*80)

# One-Step Ahead
stats_forecaster = StatsforecastForecaster(
    model_type='auto_arima',
    freq='MS',
    season_length=12
)
stats_one = stats_forecaster.one_step_forecast(train, test)
print(f"\nAutoARIMA One-Step MAE: {stats_one['metrics']['mae']:.2f}")

# Multi-Step Recursive
stats_multi = stats_forecaster.multi_step_forecast(
    train,
    horizon=len(test),
    test_df=test
)
print(f"AutoARIMA Multi-Step MAE: {stats_multi['metrics']['mae']:.2f}")

# Multi-Output (Not Supported)
try:
    stats_forecaster.multi_output_forecast(train, horizon=len(test))
except NotImplementedError as e:
    print(f"\nExpected: {str(e)[:50]}...")

# ============================================
# 2. MACHINE LEARNING MODELS (MLForecast)
# ============================================

print("\n" + "="*80)
print("MACHINE LEARNING MODELS")
print("="*80)

from mlforecast.target_transforms import Differences

ml_forecaster = MLForecastForecaster(
    model_type='xgboost',
    freq='MS',
    lags=[1, 12],
    target_transforms=[Differences([1])],
    n_estimators=100
)

# One-Step
ml_one = ml_forecaster.one_step_forecast(train, test)
print(f"\nXGBoost One-Step MAE: {ml_one['metrics']['mae']:.2f}")

# Multi-Step Recursive
ml_multi = ml_forecaster.multi_step_forecast(
    train,
    horizon=len(test),
    test_df=test
)
print(f"XGBoost Multi-Step (Recursive) MAE: {ml_multi['metrics']
```

```
['mae']:.2f}")

# Multi-Output Direct
ml_multiout = ml_forecaster.multi_output_forecast(
    train,
    horizon=len(test),
    test_df=test
)
print(f"XGBoost Multi-Output (Direct) MAE: {ml_multiout['metrics']
['mae']:.2f}")

# ==============================================
# 3. DEEP LEARNING MODELS (NeuralForecast)
# ==============================================

print("\n" + "="*80)
print("DEEP LEARNING MODELS")
print("="*80)

dl_forecaster = NeuralForecastForecaster(
    model_type='nbeats',
    freq='MS',
    input_size=24,
    horizon=len(test),
    max_steps=100
)

# Multi-Output (Default for NBEATS)
dl_multiout = dl_forecaster.multi_output_forecast(
    train,
    horizon=len(test),
    test_df=test
)
print(f"\nNBEATS Multi-Output MAE: {dl_multiout['metrics']['mae']:.2f}")

# LSTM with Recursive
lstm_forecaster = NeuralForecastForecaster(
    model_type='lstm',
    freq='MS',
    input_size=12,
    horizon=len(test),
    max_steps=100
)

lstm_recursive = lstm_forecaster.multi_step_forecast(
    train,
    horizon=len(test),
    test_df=test,
    use_recurrent=True
)
print(f"LSTM Recursive MAE: {lstm_recursive['metrics']['mae']:.2f}")

# ==============================================
# COMPARISON SUMMARY
```

```python
# ==========================================

print("\n" + "="*80)
print("COMPARISON SUMMARY")
print("="*80)

results_summary = pd.DataFrame({
    'Model': [
        'AutoARIMA (One-Step)',
        'AutoARIMA (Multi-Step)',
        'XGBoost (One-Step)',
        'XGBoost (Multi-Step Recursive)',
        'XGBoost (Multi-Output Direct)',
        'NBEATS (Multi-Output)',
        'LSTM (Recursive)'
    ],
    'MAE': [
        stats_one['metrics']['mae'],
        stats_multi['metrics']['mae'],
        ml_one['metrics']['mae'],
        ml_multi['metrics']['mae'],
        ml_multiout['metrics']['mae'],
        dl_multiout['metrics']['mae'],
        lstm_recursive['metrics']['mae']
    ]
})

print(results_summary.sort_values('MAE'))
```

Performance Comparison

Expected Performance Characteristics

Strategy	Accuracy	Computational Cost	Real Deployment	Error Propagation
One-Step	⭐⭐⭐⭐⭐ Highest	🔴 Very High (refit each step)	❌ Not realistic	✅ None
Multi-Step Recursive	⭐⭐⭐ Moderate	🟢 Low (train once)	✅ Realistic	🔴 High (accumulates)
Multi-Output Direct	⭐⭐⭐⭐ High	🟡 Medium-High	✅ Realistic	🟢 Minimal

Model Family Comparison

Model Family	One-Step	Multi-Step	Multi-Output	Training Speed	Accuracy

Model Family	One-Step	Multi-Step	Multi-Output	Training Speed	Accuracy
Statistical	✅ Yes	✅ Yes (recursive)	❌ No	⚡ Fast	⭐⭐⭐ Good
ML	✅ Yes	✅ Yes (recursive & direct)	✅ Yes (max_horizon)	⚡⚡ Medium	⭐⭐⭐⭐ Very Good
Neural	✅ Yes	✅ Partial (RNN/LSTM/GRU)	✅ Yes (default)	🐌 Slow	⭐⭐⭐⭐⭐ Excellent

Conclusion

This documentation provides complete coverage of the Nixtla forecasting ecosystem implementation. The three modules (`df_statsforecast.py`, `df_mlforecast.py`, `df_neuralforecast.py`) are production-ready and can be integrated directly into a web application for deployment on platforms like Render.com.

Key Takeaways:

1. **Three Forecasting Modes:** Each module implements one-step, multi-step, and multi-output forecasting with consistent APIs
2. **Model-Specific Support:** Statistical models don't support multi-output; ML and Neural models support all modes
3. **Trade-offs:** One-step is most accurate but not realistic; multi-step recursive is efficient but error-prone; multi-output balances accuracy and realism
4. **Production Ready:** Full typing, error handling, documentation, and examples included
5. **Web Integration:** Complete API and frontend integration guide provided

For questions or issues, refer to:

- StatsForecast: https://nixtlaverse.nixtla.io/statsforecast
- MLForecast: https://nixtlaverse.nixtla.io/mlforecast
- NeuralForecast: https://nixtlaverse.nixtla.io/neuralforecast