NESTED Returns-Based PWFCV (H-45, OPTIMIZED)

Advanced Time Series Forecasting with Validation

Implemented López de Prado purged walk-forward cross-validation (Rolling Window with overlap - Nested CV):

- Nested CV: Hierarchical validation with 7-10 outer folds, testing generalization across multiple validation sets
- Independent fold scaling: Each CV fold uses separate StandardScaler instances trained only on fold-specific training data
- Unseen validation approach: Validation sets never used in training, ensuring true out-of-sample evaluation
- Temporal barriers: Strict chronological ordering prevents future information leakage into past predictions

RETURN-BASED MODELING:

- Converted prices to percentage returns eliminating non-stationary trends while preserving directional information
- Horizon-adaptive feature selection: Dynamic feature set optimization per forecast period (5d: 45 features, 20d: 48, 45d: 52) maximizing information while minimizing overfitting
- 72 engineered features:
  - Technical indicators: SMA (5,10,20,50,200), EMA (12,26), RSI (14), MACD, Bollinger Bands, ATR, momentum, rate of change
  - Macro regime signals: VIX (volatility), interest rates, economic calendar events, market breadth indicators
  - Statistical features: Rolling volatility, trend strength, autocorrelation, regime detection

LSTM ARCHITECTURE & OPTIMIZATION:

- Two-layer LSTM with dropout regularization
- Input sequences: 180-day lookback windows with 51 features per timestep
- Output: 45-day multi-step ahead forecasts (single shot prediction)
- Early stopping: Monitor validation loss with 3-epoch patience, restoring best weights preventing overtraining

CUSTOM LOSS FUNCTIONS EVALUATED:

- MSE Loss (Selected): Standard mean squared error on returns, balancing magnitude and direction prediction
- Weighted Directional Loss: Custom loss emphasizing direction correctness with penalties (alpha=10/50, beta=1/5)
- Clipped Weighted Loss: Hybrid approach with gradient clipping preventing extreme predictions (alpha=20, beta=0.5, penalty=2.0)

HYPERPARAMETER OPTIMIZATION:

• Latin Hypercube Sampling: Space-filling design ensuring even coverage of hyperparameter space (10 trials per CV method)
• Search space: LSTM units (32-128), dropout (0.1-0.5), learning rate (1e-5 to 1e-2), batch size (8-64), epochs (10-50)

CONSERVATIVE RISK-AWARE MODELING:

• The model **optimized for direction prediction**, not magnitude prediction.
• Magnitude ratio: 0.45 (predicts 45% of actual price moves)
• Direction Accuracy: 80.77%
• Interpretation: Model intentionally underestimates volatility for safer position sizing in trading applications

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras import Input
import tensorflow as tf
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

from tensorflow.keras import backend as K
import ta

# Disable oneDNN optimizations to avoid potential minor numerical differences caused by floating-point round-off errors.
import os
os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0'
```

In [2]:
```
################## 1. Data Preparation Function  ####################
def prepare_returns_features_OPT(data_df):
    """
    Prepare features for returns-based modeling
    Leverages existing _pct() columns and adds minimal transformations
    """
    df = data_df.copy()

    # 1. Rename Close_pct() to target
    df = df.rename(columns={'Close_pct()': 'target'})

    # 2. Transform OBV (cumulative → changes)
    df['OBV_base_dif'] = df['OBV_base'].diff()
    df['OBV_futures_dif'] = df['OBV_futures'].diff()

    # 3. Single list of columns to drop
```

```
    columns_to_drop = [
        'Volume', 'S&P_fut_close', 'S&P_fut_vol',
        'GOLD_fut_close', 'DXY_indx_close', 'VIX_indx_close',
        'DGS10', 'GDP_value', 'CPIAUCSL_value',
        'PAYEMS_value', 'UNRATE_value',
        'OBV_base', 'OBV_futures',
        'Open', 'High', 'Low'
    ]

    # Keep Close for price reconstruction
    close_prices = df['Close'].copy()

    # Drop raw levels
    features_df = df.drop(columns=['Close'] + columns_to_drop, errors='ignore')

    # 4. Move 'target' to end
    cols = [col for col in features_df.columns if col != 'target'] + ['target']
    features_df = features_df[cols]

    # 5. Drop NaN from diff() operations
    features_df.dropna(inplace=True)
    close_prices = close_prices.loc[features_df.index]

    print(f"   Data prepared: {len(features_df.columns) - 1} features, {len(features_df)} samples")

    # Verify 'target' position
    print(f"   Target at position: {features_df.columns.get_loc('target')} (last)")

    return features_df, close_prices
################################### 1.OPT ######################################################
```

In [3]:

```
################################### 2.Scaling Function ########################################
def scale_fun_returns_OPT(train_window, validation_window):
    """
    Scaling for returns-based features
    Uses StandardScaler for features, StandardScaler for the target
    """
    # Scaler
    feature_scaler = StandardScaler()
    target_scaler = StandardScaler()

    # Separate features and target
    train_features = train_window.drop(columns=['target']).values  # Direct to numpy
    train_target = train_window[['target']].values

    val_features = validation_window.drop(columns=['target']).values
    val_target = validation_window[['target']].values

    # Scale
```

```
    train_features_scaled = feature_scaler.fit_transform(train_features)
    train_target_scaled = target_scaler.fit_transform(train_target)

    val_features_scaled = feature_scaler.transform(val_features)
    val_target_scaled = target_scaler.transform(val_target)

    # Return as numpy arrays (faster for LSTM)
    train_scaled = np.hstack([train_features_scaled, train_target_scaled])
    val_scaled = np.hstack([val_features_scaled, val_target_scaled])

    return train_scaled, val_scaled, target_scaler, feature_scaler

################################## 2.OPT#############################################################
```

In [4]:

```
################################## 3.Sequence Creation ######################################
def create_sequences_returns_OPT(train_data, val_data, parameters):
    """
    Create LSTM sequences for returns-based modeling
    Target is 'target' (returns) instead of 'Close' (prices)
    train_data is a data set of each fold using for training
    val_data is a data set of each fold using for validation (y_val only)
    """

    time_step = parameters['time_step']
    horizon = parameters['forecast_horizon']
    #target_col = 'target'

    # Calculate number of sequences
    # Reserve last time_step days for unseen X_val
    # This ensures validation uses truly novel data
    # Formula: len - time_step(X) - horizon(y) - time_step(reserve)
    n_sequences = len(train_data) - time_step - horizon - time_step

    if n_sequences <= 0:
        raise ValueError(f"Not enough data for sequences. Need {time_step + horizon + time_step} samples.")

    # PRE-ALLOCATE arrays
    n_features = train_data.shape[1]
    X_train = np.zeros((n_sequences, time_step, n_features), dtype=np.float32)
    y_train = np.zeros((n_sequences, horizon), dtype=np.float32)

    # Target column index (must be the last column)
    target_idx = n_features - 1

    # Vectorized sequence creation
    for i in range(n_sequences):
        X_train[i] = train_data[i : i + time_step]
        y_train[i] = train_data[i + time_step : i + time_step + horizon, target_idx]
```

```
    # Validation sequences
    # Use Last never seen time_step days
    X_val = train_data[-time_step:].reshape(1, time_step, n_features)
    y_val = val_data[:horizon, target_idx].reshape(1, horizon)

    return X_train, y_train, X_val, y_val
################################ 3. OPT ##########################################################
```

In [5]:

```
############# 4. Split Function (Rolling Window CV with Overlap - Nested) ##################
def split_returns_step_size(data_df, parameters, step_size_type='nested', split_prop=0.8, overlap_strategy='adaptive'):
    """
    Split function for returns-based modeling
    Nested CV: Training Window slides on Step Size
    """

    train_window_len = parameters['train_window']
    time_step = parameters['time_step']
    horizon = parameters['forecast_horizon']
    embargo_prop = parameters['embargo_prop']

    # Fold parameters
    validation_len = horizon
    purge_len = time_step + horizon - 1
    embargo_len = round(purge_len * embargo_prop, ndigits=None) + 1
    fold_len = train_window_len + purge_len + validation_len + embargo_len

    # Calculate step size (Rolling Window CV with Overlap)
    step_size = calculate_step_size_rolling(
        horizon=horizon,
        embargo_len=embargo_len,
        train_window_len=train_window_len,
        purge_len=purge_len,
        overlap_strategy=overlap_strategy
    )
#-------------------------------------------------------------------------------------------#

    data = data_df.copy()
    total_len = len(data)

    # Split
    train_len = int(total_len * split_prop)
    test_len = total_len - train_len

    print(f'Length of the TRAIN DATA SET is: {train_len}')
    print(f'Length of the TEST DATA SET is: {test_len}')

    TRAIN_data_set = data.iloc[:train_len]
    TEST_data_set = data.iloc[train_len:]
```

```
    # Calculate number of folds
    available_length = train_len - fold_len
    n_folds = max(1, available_length // step_size + 1)

    print(f"\nRolling Window CV Configuration:")
    print(f"  Overlap strategy: {overlap_strategy}")

    print(f"Fold length = {fold_len}")
    print(f"Train window = {train_window_len}")
    print(f"Purge = {purge_len}")
    print(f"Validation = {validation_len}")
    print(f"Embargo = {embargo_len}")
    print(f"Step size = {step_size}")
    print(f'The number of folds: {n_folds}')
    print(f"Time step: {time_step}")
    print(f"Forecast horizon: {horizon}")
#============================================================================================

    # Create folds
    folds = []

    for i in range(n_folds):
        fold_start = i * step_size
        fold_end = fold_start + fold_len

        # Check if we have enough data
        if fold_start + fold_len > train_len:
            print(f"\n  Stopping at fold {i}: Not enough data")
            break

        # Training window
        train_start = fold_start
        train_end = train_start + train_window_len
        train_data = data.iloc[train_start:train_end]

        # Purge
        purge_start = train_end
        purge_end = purge_start + purge_len

        # Validation
        val_start = purge_end
        val_end = val_start + validation_len

        # Safety Check
        if val_end > train_len:
            print(f"\n Stopping at fold {i}: Validation exceeds available data")
            break

        val_data = data.iloc[val_start:val_end]
```

```
        # Embargo
        embargo_start = val_end
        embargo_end = embargo_start + embargo_len

        # Scale each fold using returns-specific scaler
        train_scaled, val_scaled, target_scaler, features_scaler = scale_fun_returns_OPT(
            train_data, val_data
        )

        folds.append({
            'fold': i,
            'train_data': train_data,
            'train_data_scaled': train_scaled,
            'validation_data': val_data,
            'validation_data_scaled': val_scaled,
            'target_scaler': target_scaler,
            'features_scaler': features_scaler,
            'train_indices': (train_start, train_end),
            'validation_indices': (val_start, val_end),
            'purge_period': (purge_start, purge_end),
            'embargo_period': (embargo_start, embargo_end)
        })

    for fold in folds:
        print(f"\nFold {fold['fold']}:")
        print(f"  Train indices: {fold['train_indices']}")
        print(f"  Purge: {fold['purge_period']}")
        print(f"  Val indices: {fold['validation_indices']}")
        print(f"  Embargo: {fold['embargo_period']}")

#-------------------------------------------------------------------------------------------#
    # Verify no validation leakage
    print(f"\n{'='*70}")
    print("VALIDATION LEAKAGE CHECK:")
    print(f"{'='*70}")
    all_safe = True
    for i in range(len(folds) - 1):
        val_end_i = folds[i]['validation_indices'][1]
        embargo_end_i = folds[i]['embargo_period'][1]
        val_start_next = folds[i+1]['validation_indices'][0]

        if val_end_i > val_start_next:
            print(f" Fold {i} → {i+1}: LEAKAGE! Val overlap detected")
            all_safe = False
        else:
            gap = val_start_next - embargo_end_i
            print(f"✓ Fold {i} → {i+1}: Safe (gap after embargo = {gap} days)")
```

```
    if all_safe:
        print(f"\n✓ All {len(folds)} folds are safe from validation leakage!")
    print(f"{'='*70}\n")
#-------------------------------------------------------------------------------------#

    return folds, TRAIN_data_set, TEST_data_set
################################ 4. ###############################################################
```

In [6]:

```
################################ 5.PWFCV - LSTM Function ###################################
def run_pwfcv_lstm_returns_OPT(data_df, parameters, n_samples=10):
    """
    Run PWFCV for returns-based modeling
    """
    # Split and create folds
    folds, _, _ = split_returns_step_size(data_df, parameters)      # Contains  def scale_fun_returns_OPT()

    early_stop = EarlyStopping(
        monitor='val_loss',          # Monitors validation
        patience=3,
        restore_best_weights=True,
        verbose=0
    )

    time_step = parameters['time_step']
    horizon = parameters['forecast_horizon']

    best_hp = None
    best_loss = np.inf

    #-------------------------------------------
    # Option 1: Generate Random HP combinations"
    #for trial in range(n_samples):
    #hp = generate_random_hyperparams_OPT()
    #-------------------------------------------
    #-------------------------------------------
    # Option 2: Latin Hypercube HP Sampling (LHS)
    hp_list = generate_hyperparams_LHS(n_samples=n_samples, seed=42)
    for trial, hp in enumerate(hp_list):
    #-------------------------------------------

        print(f"\nTrial {trial+1}/{n_samples}")  # Shows progress
        print(f"Evaluating hyperparameters: {hp}")

        fold_losses = []
        # -------------------------
        # PWFCV LOOP OVER FOLDS
        # -------------------------
        for fold in folds:
            train_df = fold['train_data_scaled']
```

```
        val_df = fold['validation_data_scaled']

        # ------------------------------------------------------------
        # Create LSTM sequences ( using 'target' as prediction)
        # ------------------------------------------------------------
        X_train, y_train, X_val, y_val = create_sequences_returns_OPT(
            train_df,
            val_df,
            parameters
        )

        num_features = X_train.shape[-1]

        # --------------------------
        # Build & train model
        # --------------------------
        model = build_lstm_model_OPT(
            time_step,
            num_features,
            horizon,
            lstm_units=int(hp['lstm_units']),
            dropout_rate=float(hp['dropout']),
            learning_rate=float(hp['lr']),
            loss_type=hp['loss_type']   # may be 'conservative', 'aggressive', 'clip', 'mse'
        )  # Contains  def directional_mse_loss_weighted_OPT() and def directional_mse_loss_clip_weighted()

        history = model.fit(
            X_train, y_train,
            epochs=int(hp['epochs']),
            batch_size=int(hp['batch_size']),
            validation_data=(X_val, y_val),
            callbacks = [early_stop],
            verbose=0,
            shuffle=False
        )

        # --------------------------
        # Collect validation loss
        # --------------------------
        best_val_loss = min(history.history['val_loss'])
        fold_losses.append(best_val_loss)

        # Memory cleanup
        del model
        K.clear_session()

    mean_loss = np.mean(fold_losses)

    print(f"Mean PWFCV loss: {mean_loss:.6f}")
```

```
        if mean_loss < best_loss:
            print(f"  New best!")
            best_loss = mean_loss
            best_hp = hp

    print(f"\n{'='*70}")
    print(f"Best hyperparameters: {best_hp}")
    print(f"Best PWFCV loss: {best_loss:.6f}")
    print(f"{'='*50}")

    return best_hp, folds
##################################### 5. ######################################################
```

In [7]:

```
##################################### 6.1. Retrain Function ##################################
def retrain_final_model_returns(train_df, parameters, best_hp,
                                use_early_stopping=True):
    """
    Retrain final model on full training set (returns-based)
    Args:
        train_df: Full training data (80% of dataset)
        parameters: Model parameters dict
        best_hp: Best hyperparameters from PWFCV
        use_early_stopping: If True, split data for early stopping (default: True)

    Returns:
        model: Trained Keras model
        target_scaler: StandardScaler for targets
        features_scaler: StandardScaler for features

    """
    time_step = parameters['time_step']
    horizon = parameters['forecast_horizon']

    # Scale full training data
    # NOTE: scale_fun_returns_OPT requires both train and val data
    # We use first 'horizon' rows as dummy validation (not actually used)
    full_train_scaled, _, target_scaler, features_scaler = scale_fun_returns_OPT(
        train_df,
        train_df.iloc[:horizon]  # Dummy val for function signature
    )

    # Create sequences
    # NOTE: create_sequences_returns_OPT requires both train and val data
    # We use first 'horizon' rows as dummy validation (not actually used)
    X_train, y_train, _, _ = create_sequences_returns_OPT(
        full_train_scaled,
        full_train_scaled[:horizon],
        parameters
```

```
    )

    num_features = X_train.shape[-1]

    # Build model
    model = build_lstm_model_OPT(
        time_step,
        num_features,
        horizon,
        lstm_units=int(best_hp['lstm_units']),
        dropout_rate=float(best_hp['dropout']),
        learning_rate=float(best_hp['lr']),
        loss_type=best_hp['loss_type']   # 'conservative', 'aggressive', 'clip', 'mse'
    )  # Contains  def directional_mse_loss_weighted_OPT() and def directional_mse_loss_clip_weighted()

    # Print training info
    print("\n" + "="*50)
    print("TRAINING FINAL MODEL")
    print("="*70)
    print(f"Total training samples: {len(X_train)}")
    print(f"Features: {num_features}")
    print(f"Time step: {time_step}, Horizon: {horizon}")
    print(f"\nBest Hyperparameters:")
    for key, value in best_hp.items():
        if key == 'lr':
            print(f"  {key}: {value:.2e}")
        elif key == 'dropout':
            print(f"  {key}: {value:.3f}")
        else:
            print(f"  {key}: {value}")
    print("="*50)
#-------------------------------------------------------------------------------#
    # Train with optional early stopping
    if use_early_stopping:
        # Split 90/10 for early stopping validation
        split_idx = int(len(X_train) * 0.9)
        X_train_fit = X_train[:split_idx]
        y_train_fit = y_train[:split_idx]
        X_val_es = X_train[split_idx:]
        y_val_es = y_train[split_idx:]

        print(f"\nUsing early stopping:")
        print(f"  Training samples: {len(X_train_fit)}")
        print(f"  Validation samples: {len(X_val_es)}")
        print("="*70)

        early_stop = EarlyStopping(
            monitor='val_loss',
            patience=3,
```

```
            restore_best_weights=True,
            verbose=1
        )

        history = model.fit(
            X_train_fit, y_train_fit,
            epochs=int(best_hp['epochs']),
            batch_size=int(best_hp['batch_size']),
            validation_data=(X_val_es, y_val_es),
            callbacks=[early_stop],
            verbose=1  # Progress bar instead of verbose=2
        )

        actual_epochs = len(history.history['loss'])
        print(f"\nTraining stopped at epoch {actual_epochs}/{best_hp['epochs']}")
        print(f"Best validation loss: {min(history.history['val_loss']):.6f}")

#-------------------------------------------------------------------------#
    # Train on all data without early stopping
    else:
        print(f"\nTraining on ALL data (no early stopping)")
        print(f"  Training samples: {len(X_train)}")
        print("  WARNING: No early stopping - may overfit!")
        print("="*70)

        history = model.fit(
            X_train, y_train,
            epochs=int(best_hp['epochs']),
            batch_size=int(best_hp['batch_size']),
            verbose=1
        )

        print(f"\nCompleted all {best_hp['epochs']} epochs")
        print(f"Final training loss: {history.history['loss'][-1]:.6f}")
        print("="*70)

#-------------------------------------------------------------------------#

    print("FINAL MODEL TRAINING COMPLETE")
    print("="*70 + "\n")

    return model, target_scaler, features_scaler
######################### 6.1. ##############################################################
```

In [36]:

```
################### Calculate Step_size Function (For Nested CV only) ######################
def calculate_step_size_rolling(horizon, embargo_len, train_window_len,
                                purge_len, overlap_strategy='adaptive'):
    """
    Calculate step size for ROLLING WINDOW CV WITH OVERLAP (Nested)
```

```
    This function is ONLY for rolling window approach where:
    - Training windows overlap
    - Validation windows NEVER overlap (no leakage)
    - Step size is between min_safe and max_overlap
    Args:
        horizon: forecast horizon
        embargo_len: embargo period
        train_window_len: training window length
        purge_len: purge period length
        overlap_strategy: 'max_folds', 'adaptive', 'min_overlap'
    Returns:
        step_size: in range [min_safe_step, max_overlap_step]
    """
    validation_len = horizon
    fold_len = train_window_len + purge_len + validation_len + embargo_len

    # ============================================================
    # MINIMUM: No validation leakage
    # ============================================================
    min_safe_step = validation_len + embargo_len

    # ============================================================
    # MAXIMUM: Still maintains rolling window with SOME overlap
    # ============================================================
    # Option 1: Leave at least half the minimum gap
    max_overlap_step_v1 = fold_len - (min_safe_step // 2)

    # Option 2: Leave at least the embargo length as overlap
    max_overlap_step_v2 = fold_len - embargo_len

    # Option 3: Leave at least validation length as overlap
    max_overlap_step_v3 = fold_len - validation_len

    # RECOMMENDED: Option 2 (keeps at least embargo as buffer)
    max_overlap_step = max_overlap_step_v2

    # Ensure max > min (safety check)
    if max_overlap_step <= min_safe_step:
        max_overlap_step = int(min_safe_step * 1.5)

    # ============================================================
    # Calculate step size based on strategy
    # ============================================================
    if overlap_strategy == 'max_folds':
        # Maximum folds: Use minimum safe step
        step_size = min_safe_step
        overlap_description = "Maximum (most folds)"

    elif overlap_strategy == 'min_overlap':
```

```
    # Minimum overlap: Use maximum step while keeping rolling approach
    step_size = max_overlap_step
    overlap_description = "Minimum (fewest folds, still overlapping)"

elif overlap_strategy == 'adaptive':
    # Adaptive based on horizon
    if horizon <= 7:
        # Short horizon: Need more folds for stability
        # Use 40% of range from min
        step_size = int(min_safe_step + (max_overlap_step - min_safe_step) * 0.4)
    elif horizon <= 20:
        # Medium horizon: Moderate overlap
        # Use 60% of range from min
        step_size = int(min_safe_step + (max_overlap_step - min_safe_step) * 0.6)
    elif horizon <= 30:
        # Medium-long horizon
        # Use 75% of range from min
        step_size = int(min_safe_step + (max_overlap_step - min_safe_step) * 0.75)
    else:
        # Long horizon: Closer to minimum overlap
        # Use 20% of range from min
        step_size = int(min_safe_step + (max_overlap_step - min_safe_step) * 0.2)
    overlap_description = f"Adaptive for {horizon}-day horizon"

else:
    raise ValueError(f"Unknown overlap_strategy: {overlap_strategy}")

# ============================================================
# Safety checks
# ============================================================
step_size = max(min_safe_step, min(step_size, max_overlap_step))

# Calculate metrics
train_overlap = max(0, train_window_len - step_size)
train_overlap_pct = (train_overlap / train_window_len * 100) if train_window_len > 0 else 0

print(f"\n{'='*70}")
print(f"ROLLING WINDOW CV - STEP SIZE CALCULATION")
print(f"{'='*70}")
print(f"Strategy: {overlap_strategy} ({overlap_description})")
print(f"\nHorizon: {horizon} days")
print(f"Validation length: {validation_len} days")
print(f"Embargo length: {embargo_len} days")
print(f"Purge length: {purge_len} days")
print(f"Train window: {train_window_len} days")
print(f"Total fold length: {fold_len} days")
print(f"\n{'-'*70}")
print(f"STEP SIZE RANGE (for Rolling Window with Overlap):")
print(f"  Min safe step (max folds):  {min_safe_step:4d} days")
```

```
    print(f"  Max overlap step (min folds): {max_overlap_step:4d} days")
    print(f"  Selected step size:           {step_size:4d} days")
    print(f"\n{'−'*70}")
    print(f"RESULTING OVERLAP:")
    print(f"  Training window overlap: {train_overlap} days ({train_overlap_pct:.1f}%)")
    print(f"  Validation overlap: 0 days (0%) ✓ No leakage")
    print(f"{'='*70}\n")

    return step_size
################################## Calculate Step_size Function ##############################
```

In [9]:

```
################################### 6.2. Retrain Function ################################
                ###################### Helper functions ####################
def scale_train_only_returns(train_df):
    """
    Scale training data only (for final model training)
    This is a dedicated function that doesn't require dummy validation data.
    Args:
        train_df: Training DataFrame with features + target
    Returns:
        train_scaled: Scaled training data (numpy array)
        target_scaler: Fitted StandardScaler for target
        feature_scaler: Fitted StandardScaler for features
    """

    feature_scaler = StandardScaler()
    target_scaler = StandardScaler()

    # Separate features and target
    train_features = train_df.drop(columns=['target']).values
    train_target = train_df[['target']].values

    # Fit and transform
    train_features_scaled = feature_scaler.fit_transform(train_features)
    train_target_scaled = target_scaler.fit_transform(train_target)

    # Combine
    train_scaled = np.hstack([train_features_scaled, train_target_scaled])

    return train_scaled, target_scaler, feature_scaler


def create_sequences_train_only_returns(train_data, parameters):
    """
    Create LSTM sequences for final training only
    This is a dedicated function that doesn't require dummy validation data.
    Reserves last time_step days for potential future use (consistent with CV).
    Args:
        train_data: Scaled training data (numpy array)
```

```
        parameters: Dict with time_step and forecast_horizon

    Returns:
        X_train: Training input sequences
        y_train: Training target sequences
    """

    time_step = parameters['time_step']
    horizon = parameters['forecast_horizon']

    # Reserve last time_step days (consistent with CV approach)
    n_sequences = len(train_data) - time_step - horizon - time_step

    if n_sequences <= 0:
        raise ValueError(
            f"Not enough data. Need at least {time_step + horizon + time_step} samples, "
            f"but got {len(train_data)}"
        )

    # Pre-allocate arrays
    n_features = train_data.shape[1]
    X_train = np.zeros((n_sequences, time_step, n_features), dtype=np.float32)
    y_train = np.zeros((n_sequences, horizon), dtype=np.float32)

    target_idx = n_features - 1

    # Create sequences
    for i in range(n_sequences):
        X_train[i] = train_data[i : i + time_step]
        y_train[i] = train_data[i + time_step : i + time_step + horizon, target_idx]

    return X_train, y_train

            ######################################################################
               ###################### Main function ######################

def retrain_final_model_returns_CLEAN(train_df, parameters, best_hp,
                                      use_early_stopping=True):
    """
    Clean version using dedicated helper functions
    This version doesn't use dummy validation data - cleaner and more explicit
    Args:
        train_df: Full training data (80% of dataset)
        parameters: Model parameters dict
        best_hp: Best hyperparameters from PWFCV
        use_early_stopping: If True, split data for early stopping (default: True)

    Returns:
        model: Trained Keras model
```

```
    target_scaler: StandardScaler for targets
    features_scaler: StandardScaler for features
"""

time_step = parameters['time_step']
horizon = parameters['forecast_horizon']

# Scale training data (dedicated function - no dummy validation)
train_scaled, target_scaler, features_scaler = scale_train_only_returns(train_df)

# Create sequences (dedicated function - no dummy validation)
X_train, y_train = create_sequences_train_only_returns(train_scaled, parameters)

num_features = X_train.shape[-1]

# Build model with best hyperparameters
model = build_lstm_model_OPT(
    time_step,
    num_features,
    horizon,
    lstm_units=int(best_hp['lstm_units']),
    dropout_rate=float(best_hp['dropout']),
    learning_rate=float(best_hp['lr']),
    loss_type=best_hp['loss_type']   # 'conservative', 'aggressive', 'clip', 'mse'
)

# Print training info
print("\n" + "="*70)
print("TRAINING FINAL MODEL (CLEAN VERSION)")
print("="*70)
print(f"Total training samples: {len(X_train)}")
print(f"Features: {num_features}")
print(f"Time step: {time_step}, Horizon: {horizon}")
print(f"\nBest Hyperparameters:")
for key, value in best_hp.items():
    if key == 'lr':
        print(f"  {key}: {value:.2e}")
    elif key == 'dropout':
        print(f"  {key}: {value:.3f}")
    else:
        print(f"  {key}: {value}")
print("="*70)

# Train with optional early stopping
if use_early_stopping:
    # Split 90/10 for early stopping validation
    split_idx = int(len(X_train) * 0.9)
    X_train_fit = X_train[:split_idx]
    y_train_fit = y_train[:split_idx]
```

```
        X_val_es = X_train[split_idx:]
        y_val_es = y_train[split_idx:]

        print(f"\nUsing early stopping:")
        print(f"  Training samples: {len(X_train_fit)}")
        print(f"  Validation samples: {len(X_val_es)}")
        print("="*70)

        early_stop = EarlyStopping(
            monitor='val_loss',
            patience=3,
            restore_best_weights=True,
            verbose=1
        )

        history = model.fit(
            X_train_fit, y_train_fit,
            epochs=int(best_hp['epochs']),
            batch_size=int(best_hp['batch_size']),
            validation_data=(X_val_es, y_val_es),
            callbacks=[early_stop],
            verbose=1
        )

        actual_epochs = len(history.history['loss'])
        print(f"\nTraining stopped at epoch {actual_epochs}/{best_hp['epochs']}")
        print(f"Best validation loss: {min(history.history['val_loss']):.6f}")

    else:
        # Train on all data
        print(f"\nTraining on ALL data (no early stopping)")
        print(f"  Training samples: {len(X_train)}")
        print("  WARNING: No early stopping - may overfit!")
        print("="*70)

        history = model.fit(
            X_train, y_train,
            epochs=int(best_hp['epochs']),
            batch_size=int(best_hp['batch_size']),
            verbose=1
        )

        print(f"\nCompleted all {best_hp['epochs']} epochs")
        print(f"Final training loss: {history.history['loss'][-1]:.6f}")

    print("="*70)
    print("FINAL MODEL TRAINING COMPLETE")
    print("="*70 + "\n")
```

```
    return model, target_scaler, features_scaler

######################### 6.2. ##################################################################
```

In [10]:

```
######################### 7.Evaluation on Test Returns ######################################
#-------------------------------------
# Add Helper Function
#-------------------------------------
def reconstruct_prices(returns, start_price):
    """
    Reconstruct prices from returns

    Args:
        returns: Array of returns (as decimals, e.g., 0.02 = 2%)
        start_price: Starting price

    Returns:
        Array of prices
    """
    prices = []
    current_price = start_price

    for ret in returns:
        next_price = current_price * (1 + ret)
        prices.append(next_price)
        current_price = next_price

    return np.array(prices)

#-----------------------------------
# Add Visualization
#-----------------------------------
def plot_first_predictionon_test_set(y_true_prices, y_pred_prices, save_path=None):
    '''Plot true vs predicted prices'''

    days = np.arange(len(y_true_prices))

    plt.figure(figsize=(12, 6))
    plt.plot(days, y_true_prices, 'b-', label='True', linewidth=2)
    plt.plot(days, y_pred_prices, 'r--', label='Predicted', linewidth=2)
    plt.xlabel('Days')
    plt.ylabel('Price ($)')
    plt.title(f'{len(y_true_prices)}-Day Price Forecast')
    plt.legend()
    plt.grid(True, alpha=0.3)

    if save_path:
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.show()
```

```python
#-------------------------------------------------------------------------------#

def evaluate_on_test_returns(model, train_df, test_df, train_close_prices, test_close_prices, parameters):
    """
    Evaluate model and reconstruct prices from predicted returns
    INPUTS:
    -------
      model: Trained Keras model
      train_df: Training features (4783, 54)
      test_df: Test features (1196, 54)
      train_close_prices: Training prices (for reconstruction)
      test_close_prices: Test prices (for verification)
      parameters: Dict with time_step, horizon


    OUTPUTS:
    --------
      y_true_returns: True returns (horizon, )
      y_pred_returns: Predicted returns (horizon, )
      y_true_prices: True prices (horizon, )
      y_pred_prices: Predicted prices (horizon, )
      target_scaler: For future use
    """
    time_step = parameters['time_step']
    horizon = parameters['forecast_horizon']

    # Scale data
    train_scaled, test_scaled, target_scaler, features_scaler = scale_fun_returns_OPT(
        train_df, test_df
    )

    # Build test sequence
    X_test = train_scaled[-time_step:].reshape(1, time_step, -1)

    # Get true returns (scaled)
    y_test_true_scaled = test_scaled[:horizon, -1]  # Last column = target

    # Predict returns (scaled)
    y_test_pred_scaled = model.predict(X_test, verbose=0)[0]

    # Inverse transform to get actual returns
    y_true_returns = target_scaler.inverse_transform(
        y_test_true_scaled.reshape(-1, 1)
    ).flatten()

    y_pred_returns = target_scaler.inverse_transform(
        y_test_pred_scaled.reshape(-1, 1)
    ).flatten()
```

```python
# Reconstruct prices from returns
last_train_price = train_close_prices.iloc[-1]

# True prices
y_true_prices = reconstruct_prices(y_true_returns, last_train_price)
# Predicted prices
y_pred_prices = reconstruct_prices(y_pred_returns, last_train_price)

# Display results
print("\n" + "="*70)
print("RETURNS (Scaled):")
print(f"TRUE: {y_test_true_scaled}")
print(f"PRED: {y_test_pred_scaled}")

print("\n" + "="*70)
print("RETURNS (Original %):")
print(f"TRUE: {y_true_returns * 100}%")
print(f"PRED: {y_pred_returns * 100}%")

print("\n" + "="*70)
print("PRICES (Reconstructed $):")
print(f"TRUE: {y_true_prices}")
print(f"PRED: {y_pred_prices}")
print("="*70)


# Display results
print("\\n" + "="*50)
print("FIRST PREDICTION (45-day forecast)")
print("="*70)
print("\\nRETURNS (Original %):")
print(f"TRUE: Mean={np.mean(y_true_returns)*100:.2f}%, Std={np.std(y_true_returns)*100:.2f}%")
print(f"PRED: Mean={np.mean(y_pred_returns)*100:.2f}%, Std={np.std(y_pred_returns)*100:.2f}%")

print("\\n" + "-"*50)
print("PRICES (Reconstructed $):")
print(f"Start: ${last_train_price:.2f}")
print(f"True End: ${y_true_prices[-1]:.2f}")
print(f"Pred End: ${y_pred_prices[-1]:.2f}")
print("\\n" + "-"*50)

# Calculate metrics on prices
mae = np.mean(np.abs(y_true_prices - y_pred_prices))
mape = np.mean(np.abs((y_true_prices - y_pred_prices) / y_true_prices)) * 100
rmse = np.sqrt(np.mean((y_true_prices - y_pred_prices) ** 2))

print(f"\nMetrics (Price Scale):")
print(f"MAE:  ${mae:.2f}")
print(f"MAPE: {mape:.2f}%")
```

```
    print(f"RMSE: ${rmse:.2f}")

    # Direction accuracy
#--------------------------------------------------------------------------------
    print(f"\\n{'='*50}")

    true_direction = y_true_prices[-1] > y_true_prices[0]  # Overall H-days
    pred_direction = y_pred_prices[-1] > y_pred_prices[0]

    # Also add cumulative return:
    true_cumulative = (y_true_prices[-1] / y_true_prices[0]) - 1
    pred_cumulative = (y_pred_prices[-1] / y_pred_prices[0]) - 1

    print(f"\nCumulative Return (45 days):")
    print(f"  True: {true_cumulative*100:+.2f}%")
    print(f"  Pred: {pred_cumulative*100:+.2f}%")
    print(f"  Direction: {'✓ CORRECT' if np.sign(true_cumulative) == np.sign(pred_cumulative) else '✗ INCORRECT'}")

#--------------------------------------------------------------------------------
    # Visualization
    plot_first_predictionon_test_set(y_true_prices, y_pred_prices, save_path=None)

    return y_true_returns, y_pred_returns, y_true_prices, y_pred_prices, target_scaler
###################### 7. new ###########################################################
```

In [11]:

```
###################### 8. Rolling Evaluation for Full Test Set ##############################
def evaluate_rolling_test_returns_OPT(model, train_df, test_df, train_close_prices, test_close_prices, parameters):
    """
    Rolling window evaluation on test set with price reconstruction
    NPUTS:
    -------
      model: Trained final model
      train_df: Training features (4783, 54)
      test_df: Test features (1196, 54)
      train_close_prices: Training prices
      test_close_prices: Test prices
      parameters: Dict with time_step, horizon


    OUTPUTS:
    --------
      actuals_returns: (n_predictions, horizon) - true returns
      predictions_returns: (n_predictions, horizon) - predicted returns
      actuals_prices: (n_predictions, horizon) - true prices
      predictions_prices: (n_predictions, horizon) - predicted prices
      target_scaler: For reference
    """
    time_step = parameters['time_step']
    horizon = parameters['forecast_horizon']
```

```python
# Scale data
train_scaled, test_scaled, target_scaler, _ = scale_fun_returns_OPT(train_df, test_df)

# Combine for rolling window
combined = np.vstack([train_scaled, test_scaled])
combined_prices = pd.concat([train_close_prices, test_close_prices], axis=0).values

train_len = len(train_scaled)
n_predictions = (len(test_scaled) - horizon) // horizon + 1

# PRE-ALLOCATE arrays for all predictions
all_predictions = np.zeros((n_predictions, horizon))
all_actuals = np.zeros((n_predictions, horizon))
last_prices = np.zeros(n_predictions)

print(f"\nRunning rolling evaluation on test set...")
print(f"Making {n_predictions} predictions ({horizon}-day each)")

# Collect all predictions
pred_idx = 0
for i in range(0, len(test_scaled) - horizon + 1, horizon):
    window_start = train_len + i - time_step
    window_end = train_len + i

    if window_start < 0:
        continue

    # Extract and predict
    X_test = combined[window_start:window_end].reshape(1, time_step, -1)
    y_pred_scaled = model.predict(X_test, verbose=0)[0]
    y_true_scaled = test_scaled[i:i+horizon, -1]  # Last column is target

    all_predictions[pred_idx] = y_pred_scaled
    all_actuals[pred_idx] = y_true_scaled
    start_price_idx = window_end - 1
    last_prices[pred_idx] = combined_prices[start_price_idx]
    pred_idx += 1

    #----------------------------------------------------
    # Progress indicator
    if pred_idx % 5 == 0 or pred_idx == n_predictions:
        print(f"  Completed {pred_idx}/{n_predictions} predictions...")
    #----------------------------------------------------

# Trim to actual predictions made
all_predictions = all_predictions[:pred_idx]
all_actuals = all_actuals[:pred_idx]
last_prices = last_prices[:pred_idx]
```

```
    # Batch inverse transform
    actuals_returns = target_scaler.inverse_transform(
        all_actuals.reshape(-1, 1)
    ).reshape(-1, horizon)

    predictions_returns = target_scaler.inverse_transform(
        all_predictions.reshape(-1, 1)
    ).reshape(-1, horizon)

    # VECTORIZED price reconstruction (no loops!)
    # For each prediction window, compound returns into prices
    actuals_prices = np.zeros_like(actuals_returns)
    predictions_prices = np.zeros_like(predictions_returns)

    for i in range(pred_idx):
        # Cumulative product of (1 + return)
        actual_multipliers = np.cumprod(1 + actuals_returns[i])
        pred_multipliers = np.cumprod(1 + predictions_returns[i])

        actuals_prices[i] = last_prices[i] * actual_multipliers
        predictions_prices[i] = last_prices[i] * pred_multipliers

    print(f"Completed {pred_idx} predictions")

    # Calculate metrics
    actuals_flat = actuals_prices.flatten()
    predictions_flat = predictions_prices.flatten()

    mae = np.mean(np.abs(actuals_flat - predictions_flat))
    mape = np.mean(np.abs((actuals_flat - predictions_flat) / actuals_flat)) * 100
    rmse = np.sqrt(np.mean((actuals_flat - predictions_flat) ** 2))

#-----------------------------------------------------------------------------------------------------
# Direction accuracy
    direction_correct = 0
    for i in range(pred_idx):
        true_direction = actuals_prices[i][-1] > actuals_prices[i][0]
        pred_direction = predictions_prices[i][-1] > predictions_prices[i][0]
        if true_direction == pred_direction:
            direction_correct += 1

    direction_accuracy = direction_correct / pred_idx

    # ADDED: Per-window statistics
    per_window_mae = np.mean(np.abs(actuals_prices - predictions_prices), axis=1)

    # Print results
    print(f"\\n{'='*70}")
```

```
    print(f"ROLLING EVALUATION RESULTS ({pred_idx} predictions)")
    print(f"{'='*70}")
    print(f"\\nTest Set Metrics:")
    print(f"  Direction Accuracy: {direction_accuracy*100:.2f}% " +
          f"({direction_correct}/{pred_idx} correct)")
    print(f"  MAE:  ${mae:.2f}")
    print(f"  MAPE: {mape:.2f}%")
    print(f"  RMSE: ${rmse:.2f}")

    print(f"\\nPer-Window MAE Statistics:")
    print(f"  Best:   ${np.min(per_window_mae):.2f}")
    print(f"  Worst:  ${np.max(per_window_mae):.2f}")
    print(f"  Median: ${np.median(per_window_mae):.2f}")
    print(f"  Std:    ${np.std(per_window_mae):.2f}")
    print(f"{'='*70}\\n")
#--------------------------------------------------------------------------------------------------

    return actuals_returns, predictions_returns, actuals_prices, predictions_prices, target_scaler
############################## 8. ################################################################
```

In [12]:

```
########################## 9(Scalable).CORR Direction Visualization ######################
def plot_direction_analysis_CORR(actuals, predictions, horizon=None):
    """
    Visual analysis of directional accuracy for H-day horizon
    Args:
        actuals_prices: (n_windows, horizon) array of actual prices in $
        predictions_prices: (n_windows, horizon) array of predicted prices in $
        horizon: Forecast horizon (auto-detected if None)
    """

    if horizon is None:
        horizon = actuals.shape[1]  # From data shape

    # Transform to original prices
    actuals_flat = actuals.flatten()
    predictions_flat = predictions.flatten()

    actuals_original = target_scaler.inverse_transform(
        actuals_flat.reshape(-1, 1)
    ).flatten()

    predictions_original = target_scaler.inverse_transform(
        predictions_flat.reshape(-1, 1)
    ).flatten()

    n_windows = len(actuals)
    actuals_prices = actuals_original.reshape(n_windows, horizon)
    predicted_prices = predictions_original.reshape(n_windows, horizon)
```

```python
# Calculate changes using horizon-1
actual_changes = actuals_prices[:, horizon-1] - actuals_prices[:, 0]
pred_changes = predicted_prices[:, horizon-1] - predicted_prices[:, 0]

# Plot setup
fig, axes = plt.subplots(2, 3, figsize=(18, 10))

# ========================================
# Subplot 1: Scatter of changes
# ========================================
axes[0, 0].scatter(actual_changes, pred_changes, alpha=0.5)

max_change = max(abs(actual_changes.max()), abs(actual_changes.min()),
                 abs(pred_changes.max()), abs(pred_changes.min()))
limit = max_change * 1.1

axes[0, 0].plot([-limit, limit], [-limit, limit], 'r--', label='Perfect Prediction')
axes[0, 0].axhline(0, color='gray', linestyle='--', alpha=0.3)
axes[0, 0].axvline(0, color='gray', linestyle='--', alpha=0.3)
axes[0, 0].set_xlabel(f'Actual Price Change ($) - {horizon} Days')
axes[0, 0].set_ylabel(f'Predicted Price Change ($) - {horizon} Days')
axes[0, 0].set_title(f'Predicted vs Actual Changes ({horizon}-Day)')
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)

# ========================================
# Subplot 2: Histogram of changes
# ========================================
axes[0, 1].hist(actual_changes, bins=50, alpha=0.5, label='Actual', color='blue')
axes[0, 1].hist(pred_changes, bins=50, alpha=0.5, label='Predicted', color='orange')
axes[0, 1].axvline(0, color='gray', linestyle='--', alpha=0.3)
axes[0, 1].set_xlabel(f'Price Change ($) - {horizon} Days')
axes[0, 1].set_ylabel('Frequency')
axes[0, 1].set_title(f'Distribution of Price Changes ({horizon}-Day)')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

# ========================================
# Subplot 3: Rolling direction accuracy
# ========================================
actual_direction = actual_changes > 0
pred_direction = pred_changes > 0
direction_correct = (actual_direction == pred_direction)

window_size = max(horizon // 3, 10)  # Dynamic window size
if len(direction_correct) > window_size:
    rolling_accuracy = np.convolve(
        direction_correct.astype(float),
        np.ones(window_size)/window_size,
```

```
        mode='valid'
    ) * 100
else:
    rolling_accuracy = [np.mean(direction_correct) * 100]

axes[0, 2].plot(rolling_accuracy)
axes[0, 2].axhline(50, color='red', linestyle='--', label='Random (50%)')
axes[0, 2].set_xlabel('Window')
axes[0, 2].set_ylabel('Direction Accuracy (%)')
axes[0, 2].set_title(f'Rolling Direction Accuracy - {horizon} Day (window={window_size})')
axes[0, 2].legend()
axes[0, 2].grid(True, alpha=0.3)
axes[0, 2].set_ylim([0, 100])

# ========================================
# Subplot 4: Confusion matrix
# ========================================
# Use horizon-1
actual_direction_overall = actuals_prices[:, horizon-1] > actuals_prices[:, 0]
pred_direction_overall = predicted_prices[:, horizon-1] > predicted_prices[:, 0]

tp = ((actual_direction_overall) & (pred_direction_overall)).sum()
tn = ((~actual_direction_overall) & (~pred_direction_overall)).sum()
fp = ((~actual_direction_overall) & (pred_direction_overall)).sum()
fn = ((actual_direction_overall) & (~pred_direction_overall)).sum()

confusion = np.array([[tn, fp], [fn, tp]])

im = axes[1, 0].imshow(confusion, cmap='Blues')
axes[1, 0].set_xticks([0, 1])
axes[1, 0].set_yticks([0, 1])
axes[1, 0].set_xticklabels(['Pred DOWN', 'Pred UP'])
axes[1, 0].set_yticklabels(['Actual DOWN', 'Actual UP'])
axes[1, 0].set_title(f'Confusion Matrix ({horizon}-Day Direction)')

for i in range(2):
    for j in range(2):
        text = axes[1, 0].text(j, i, confusion[i, j],
                               ha="center", va="center", color="black", fontsize=16)

plt.colorbar(im, ax=axes[1, 0])

# ========================================
# Subplot 5 - Dynamic weekly accuracy
# ========================================
weekly_accuracies = []
week_labels = []

# Generate checkpoints every 5 days up to horizon
```

```
    week_checkpoints = list(range(4, horizon, 5))  # [4, 9, 14, ...]

    for i, day_idx in enumerate(week_checkpoints[:10]):  # Max 10 weeks for readability
        if day_idx >= horizon:
            break

        actual_dir = actuals_prices[:, day_idx] > actuals_prices[:, 0]
        pred_dir = predicted_prices[:, day_idx] > predicted_prices[:, 0]
        accuracy = np.mean(actual_dir == pred_dir) * 100
        weekly_accuracies.append(accuracy)
        week_labels.append(f'Week {i+1}\n(Day {day_idx+1})')

    if weekly_accuracies:
        axes[1, 1].bar(range(len(weekly_accuracies)), weekly_accuracies,
                       color='steelblue', alpha=0.7)
        axes[1, 1].axhline(50, color='red', linestyle='--', label='Random (50%)')
        axes[1, 1].set_xlabel('Week')
        axes[1, 1].set_ylabel('Direction Accuracy (%)')
        axes[1, 1].set_title('Weekly Direction Accuracy')
        axes[1, 1].set_xticks(range(len(weekly_accuracies)))
        axes[1, 1].set_xticklabels(week_labels, fontsize=9)
        axes[1, 1].legend()
        axes[1, 1].grid(True, alpha=0.3, axis='y')
        axes[1, 1].set_ylim([0, 100])

    # ========================================
    # Subplot 6 - Magnitude comparison
    # ========================================
    correct_idx = (actual_direction == pred_direction)

    if correct_idx.sum() > 0:
        avg_actual = np.mean(np.abs(actual_changes[correct_idx]))
        avg_pred = np.mean(np.abs(pred_changes[correct_idx]))

        axes[1, 2].bar(['Actual', 'Predicted'], [avg_actual, avg_pred],
                       color=['blue', 'orange'], alpha=0.7)
        axes[1, 2].set_ylabel('Average Magnitude ($)')
        axes[1, 2].set_title(f'Avg {horizon}-Day Change Magnitude\n(When Direction Correct)')
        axes[1, 2].grid(True, alpha=0.3, axis='y')

        ratio = avg_pred / avg_actual if avg_actual > 0 else 0
        axes[1, 2].text(0.5, max(avg_actual, avg_pred) * 0.9,
                        f'Ratio: {ratio:.2f}',
                        ha='center', fontsize=12,
                        bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))

    plt.tight_layout()
    plt.show()
################################### 9.CORR (Scalable) #######################################
```

In [13]:
```
####################### 10 (Scalable) Weighted Directional Loss ############################
#@tf.function  # Compile to graph for faster execution
def directional_mse_loss_weighted_OPT(alpha=20.0, beta=3.0):
    """
    Penalizes directional errors MORE when the true move is large
    This makes the model prioritize getting big moves right
    Args:
        alpha: Base directional penalty weight
        beta: Magnitude scaling factor (higher = more weight on large moves)
    """
    def loss(y_true, y_pred):
        # Standard MSE
        mse = K.mean(K.square(y_true - y_pred))

        # Cumulative returns
        true_cumulative = K.sum(y_true, axis=1, keepdims=True)
        pred_cumulative = K.sum(y_pred, axis=1, keepdims=True)

        # Directions
        true_direction = K.sign(true_cumulative)
        pred_direction = K.sign(pred_cumulative)

        # Direction match
        direction_match = K.cast(
            K.equal(true_direction, pred_direction),
            dtype='float32'
        )

        # Weight by magnitude of true move
        magnitude_weight = K.abs(true_cumulative) * beta

        # Weighted directional loss
        weighted_direction_loss = K.mean(
            (1.0 - direction_match) * (1.0 + magnitude_weight)
        )

        total_loss = mse + alpha * weighted_direction_loss

        return total_loss
    return loss
###################################### 10. (Scalable) ########################################
```

In [14]:
```
####################### 10.1 (Scalable) Clip Weighted Directional Loss ######################
def directional_mse_loss_clip_weighted(alpha=10.0, beta=1.0, max_penalty=5.0):
    """
    Add clipping to prevent huge losses
    """
    def loss(y_true, y_pred):
```

```
        mse = K.mean(K.square(y_true - y_pred))

        true_cumulative = K.sum(y_true, axis=1, keepdims=True)
        pred_cumulative = K.sum(y_pred, axis=1, keepdims=True)

        true_direction = K.sign(true_cumulative)
        pred_direction = K.sign(pred_cumulative)

        direction_match = K.cast(
            K.equal(true_direction, pred_direction),
            K.floatx()
        )

        # CLIP magnitude weight to prevent explosion
        magnitude_weight = K.clip(K.abs(true_cumulative) * beta, 0.0, max_penalty)

        weighted_direction_loss = K.mean(
            (1.0 - direction_match) * (1.0 + magnitude_weight)
        )

        # CLIP total directional component
        total_loss = mse + K.clip(alpha * weighted_direction_loss, 0.0, max_penalty)

        return total_loss
    return loss
############################## 10.1 ############################################################
```

In [15]:

```
######################## Check correlation bitween continuous features ######################
def correlation_fun(data, features_no_close=None):
    threshold = 0.85
    data_df = data.copy()
    if features_no_close:
        correlation_matrix = data_df[features_no_close].corr()
    else:
        correlation_matrix = data_df.corr()

    # Identify highly correlated pairs
    high_corr_pairs = []
    for i in range(len(correlation_matrix.columns)):
        for j in range(i+1, len(correlation_matrix.columns)):
            if abs(correlation_matrix.iloc[i, j]) > threshold:
                high_corr_pairs.append({
                    'feature1': correlation_matrix.columns[i],
                    'feature2': correlation_matrix.columns[j],
                    'correlation': correlation_matrix.iloc[i, j]
                })

    print("High correlation pairs:")
    for pair in high_corr_pairs:
```

```
        print(f"{pair['feature1']} <-> {pair['feature2']}: {pair['correlation']:.3f}")

##############################################################################################
```

In [17]:

```
##############################################################################################
def generate_random_hyperparams_OPT(seed=None):
    """
    Generate random hyperparameter combinations
    Uses random search across predefined ranges
    Args:
        seed: Random seed for reproducibility

    Returns:
        dict: Hyperparameter configuration
    """
    if seed is not None:
        np.random.seed(seed)

    # LatinHypercube-inspired sampling
    lstm_units_options = [16, 32, 64, 128]
    dropout_options = np.linspace(0.1, 0.4, 10)
    lr_options = np.logspace(-5, -3, 20)  # log scale for learning rate
    epochs_options = range(10, 35, 2)
    batch_size_options = [16, 32, 64]
    loss_type_options = ['mse', 'clip']   # 'conservative' (weighted), 'aggressive' (weighted)


    hp = {
        'lstm_units': np.random.choice(lstm_units_options),
        'dropout': np.random.choice(dropout_options),
        'lr': np.random.choice(lr_options),
        'epochs': np.random.choice(epochs_options),
        'batch_size': np.random.choice(batch_size_options),
        'loss_type': np.random.choice(loss_type_options)
    }

    return hp
##############################################################################################
```

In [40]:

```
#################  Latin Hypercube  ##############################################
from scipy.stats import qmc
def generate_hyperparams_LHS(n_samples=10, seed=None):
    """
    Generate hyperparameters using Latin Hypercube Sampling.
    LHS ensures better coverage of the search space compared to random sampling.
    Each parameter is divided into n_samples bins, and exactly one sample is
    drawn from each bin (with random permutation across parameters).

    Args:
```

```
    n_samples: Number of hyperparameter sets to generate
    seed: Random seed for reproducibility

Returns:
    list of dicts: n_samples hyperparameter configurations
"""

# Set seed for reproducibility
if seed is not None:
    np.random.seed(seed)

# Create LHS sampler (5 dimensions = 5 hyperparameters)
sampler = qmc.LatinHypercube(d=5, seed=seed)

# Generate samples in [0, 1]^5 hypercube
samples = sampler.random(n=n_samples)

# Define parameter ranges
lstm_units_options = [32, 64]
dropout_range = [0.1, 0.4]
lr_range_log = [-4, -3]  # log10 scale
epochs_range = [10, 34]
batch_size_options = [16, 32, 64]
#loss_type_options = ['mse']   # 'conservative' (weighted), 'aggressive' (weighted),  'clip' (weighted)

# Initialize list to store all hyperparameter sets
hp_list = []

for i in range(n_samples):
    # Extract sample for this trial
    sample = samples[i]

    # 1. LSTM units (discrete: map to 4 options)
    lstm_idx = int(sample[0] * len(lstm_units_options))
    lstm_idx = min(lstm_idx, len(lstm_units_options) - 1)  # Safety
    lstm_units = lstm_units_options[lstm_idx]

    # 2. Dropout (continuous: scale to [0.1, 0.4])
    dropout = dropout_range[0] + sample[1] * (dropout_range[1] - dropout_range[0])

    # 3. Learning rate (log scale: map to [1e-5, 1e-3])
    lr_log = lr_range_log[0] + sample[2] * (lr_range_log[1] - lr_range_log[0])
    lr = 10 ** lr_log

    # 4. Epochs (discrete: map to [10, 34])
    epochs = int(epochs_range[0] + sample[3] * (epochs_range[1] - epochs_range[0] + 1))
    epochs = min(epochs, epochs_range[1])  # Safety

    # 5. Batch size (discrete: map to 3 options)
```

```
        batch_idx = int(sample[4] * len(batch_size_options))
        batch_idx = min(batch_idx, len(batch_size_options) - 1)  # Safety
        batch_size = batch_size_options[batch_idx]

        #loss_type = np.random.choice(loss_type_options)
        loss_type = 'mse'   #'loss_type': 'mse'  # Based on testing

        # Create hyperparameter dict
        hp = {
            'lstm_units': int(lstm_units),
            'dropout': float(dropout),
            'lr': float(lr),
            'epochs': int(epochs),
            'batch_size': int(batch_size),
            'loss_type': loss_type
        }

        hp_list.append(hp)

    return hp_list
#################### Latin Hypercube ########################################
```

In [19]:

```
################### Build LSTM Model Function (accepts hyperparameters:) (Scalable) ########
def build_lstm_model_OPT(time_step, num_features, horizon,
                         lstm_units=64, dropout_rate=0.2, learning_rate=1e-3, loss_type='mse'):
    """
      Build LSTM model for multi-step returns forecasting
      Architecture: LSTM → LSTM → Dense
      Features: Dropout, gradient clipping, custom loss options
      Args:
          time_step: Lookback window length
          num_features: Number of input features
          horizon: Forecast horizon length
          lstm_units: Units per LSTM layer (default: 64)
          dropout_rate: Dropout rate (default: 0.2)
          learning_rate: Adam learning rate (default: 1e-3)
          loss_type: Loss function type
              - 'mse': Standard MSE  (default)
              - 'conservative': Directional loss (a=10, b=1)
              - 'aggressive': Directional loss (a=50, b=5)
              - 'clip': Clipped directional loss

      Returns:
          Compiled Keras Sequential model
    """

    # ========================================
    # Use directional loss
    # ========================================
```

```
    model = Sequential()
    model.add(Input(shape=(time_step, num_features)))
    model.add(LSTM(lstm_units, return_sequences=True, dropout=dropout_rate))
    model.add(LSTM(lstm_units, return_sequences=False, dropout=dropout_rate))
    model.add(Dense(horizon))  # This automatically handles horizon
    optimizer = Adam(
        learning_rate=learning_rate,
        clipnorm=1.0                      # Gradient clipping. Prevent exploding gradients
    )

    # Select loss function
    if loss_type == 'conservative':
        custom_loss = directional_mse_loss_weighted_OPT(
            alpha=10.0, beta=1.0
        )
    elif loss_type == 'aggressive':
        custom_loss = directional_mse_loss_weighted_OPT(
            alpha=50.0, beta=5.0
        )
    elif loss_type == 'mse':
        custom_loss = 'mse'

    elif loss_type == 'clip':
        custom_loss = directional_mse_loss_clip_weighted(
            alpha=10.0, beta=1.0, max_penalty=3.0
        )

    else:
        raise ValueError(f"Unknown loss_type: {loss_type}")

    model.compile(
        loss=custom_loss,
        optimizer=optimizer
    )

    return model
################################### (Scalable) ############################################
```

In [20]:

```
########################## Analyze_direction_accuracy ####################################
def analyze_direction_accuracy_CORRECTED(actuals_prices, predictions_prices,
                                         horizon=None):
    """
    Analyze directional accuracy using ALREADY RECONSTRUCTED prices

    Args:
        actuals_prices: (n_windows, horizon) array of actual prices in $
        predictions_prices: (n_windows, horizon) array of predicted prices in $
        horizon: forecast horizon (inferred from data if None)
```

```
Returns:
    actual_changes, pred_changes, statistics_dict
"""

if horizon is None:
    horizon = actuals_prices.shape[1]

n_windows = len(actuals_prices)

print("="*70)
print(f"DIRECTIONAL ACCURACY ANALYSIS ({horizon}-DAY HORIZON)")
print("="*70)

# ========================================
# 1. Overall direction (Day 1 → Day H)
# ========================================
print(f"\n1. OVERALL DIRECTION (Day 1 → Day {horizon}):")
print("-" * 70)

# Calculate actual price changes
actual_changes = actuals_prices[:, -1] - actuals_prices[:, 0]
pred_changes = predictions_prices[:, -1] - predictions_prices[:, 0]

# Direction based on changes
actual_direction_overall = actual_changes > 0
pred_direction_overall = pred_changes > 0

direction_match_overall = (actual_direction_overall == pred_direction_overall)
accuracy_overall = np.mean(direction_match_overall) * 100

print(f"Accuracy: {accuracy_overall:.2f}%")
print(f"Correct: {direction_match_overall.sum()}/{len(direction_match_overall)}")

# Print sample changes for verification
print(f"\nSample changes (first 3 windows):")
for i in range(min(3, n_windows)):
    print(f"  Window {i}: Actual=${actual_changes[i]:.2f}, "
          f"Pred=${pred_changes[i]:.2f}, "
          f"Match={'✓' if direction_match_overall[i] else '✗'}")

# ... rest of analysis (weekly, day-to-day, etc.) ...
# ========================================
# 2. Weekly directions (every 5 days)
# ========================================
print("\n2. WEEKLY DIRECTIONS (5-day intervals):")
print("-" * 70)

# Generate week checkpoints dynamically
```

```python
week_checkpoints = list(range(4, horizon, 5))  # [4, 9, 14, 19, 24, ...]
weekly_accuracies = []

for i, day_idx in enumerate(week_checkpoints):
    if day_idx >= horizon:  # ← Safety check
        break

    actual_dir = actuals_prices[:, day_idx] > actuals_prices[:, 0]
    pred_dir = predictions_prices[:, day_idx] > predictions_prices[:, 0]
    match = (actual_dir == pred_dir)
    acc = np.mean(match) * 100
    weekly_accuracies.append(acc)

    week_num = i + 1
    print(f"Week {week_num} (Day {day_idx+1}): {acc:.2f}%")

if weekly_accuracies:
    print(f"\nAverage weekly accuracy: {np.mean(weekly_accuracies):.2f}%")

# ========================================
# 3. Day-by-day directions (all transitions)
# ========================================
print("\n3. DAY-TO-DAY DIRECTIONS (summary by week):")
print("-" * 70)

# Generate week ranges dynamically
week_ranges = [(i*5, min((i+1)*5, horizon)) for i in range((horizon + 4) // 5)]

for week_num, (start, end) in enumerate(week_ranges, 1):
    if start >= horizon - 1:
        break

    week_accs = []
    for i in range(start, min(end-1, horizon-1)):
        actual_dir = actuals_prices[:, i+1] > actuals_prices[:, i]
        pred_dir = predictions_prices[:, i+1] > predictions_prices[:, i]
        match = (actual_dir == pred_dir)
        acc = np.mean(match) * 100
        week_accs.append(acc)

    if week_accs:
        avg_week_acc = np.mean(week_accs)
        print(f"Week {week_num} avg (days {start+1}-{min(end, horizon)}): {avg_week_acc:.2f}%")

# ========================================
# 4. Magnitude Analysis (H-day cumulative)
# ========================================
print(f"\n4. MAGNITUDE ANALYSIS ({horizon}-Day Cumulative):")
print("-" * 70)
```

```python
# Use horizon-1 instead of hard-coded 44
actual_changes = actuals_prices[:, horizon-1] - actuals_prices[:, 0]
pred_changes = predictions_prices[:, horizon-1] - predictions_prices[:, 0]

correct_idx = direction_match_overall

if correct_idx.sum() > 0:
    avg_actual_change = np.mean(np.abs(actual_changes[correct_idx]))
    avg_pred_change = np.mean(np.abs(pred_changes[correct_idx]))

    print(f"Avg actual {horizon}-day change (when correct):    ${avg_actual_change:.2f}")
    print(f"Avg predicted {horizon}-day change (when correct): ${avg_pred_change:.2f}")

    if avg_actual_change > 0:
        print(f"Ratio (pred/actual): {avg_pred_change/avg_actual_change:.2f}")

# ========================================
# 5. Detailed Breakdown
# ========================================
print(f"\n5. DETAILED BREAKDOWN (Overall {horizon}-Day Direction):")
print("-" * 70)

tp = ((actual_direction_overall) & (pred_direction_overall)).sum()
tn = ((~actual_direction_overall) & (~pred_direction_overall)).sum()
fp = ((~actual_direction_overall) & (pred_direction_overall)).sum()
fn = ((actual_direction_overall) & (~pred_direction_overall)).sum()

print(f"True Positives (predicted UP, was UP):     {tp}")
print(f"True Negatives (predicted DOWN, was DOWN): {tn}")
print(f"False Positives (predicted UP, was DOWN):  {fp}")
print(f"False Negatives (predicted DOWN, was UP):  {fn}")

if (tp + fp) > 0:
    precision = tp / (tp + fp)
    print(f"\nPrecision (UP predictions): {precision:.2%}")

if (tp + fn) > 0:
    recall = tp / (tp + fn)
    print(f"Recall (catching UP moves): {recall:.2%}")

# ========================================
# 6. Direction by Time Period
# ========================================
print("\n6. DIRECTIONAL ACCURACY OVER TIME:")
print("-" * 70)

quarter_size = n_windows // 4
```

```
    for q in range(4):
        start_idx = q * quarter_size
        end_idx = (q + 1) * quarter_size if q < 3 else n_windows

        q_matches = direction_match_overall[start_idx:end_idx]
        q_accuracy = np.mean(q_matches) * 100

        print(f"Quarter {q+1} (windows {start_idx:3d}-{end_idx:3d}): {q_accuracy:5.2f}%")

    # ========================================
    # SUMMARY
    # ========================================
    print("\n" + "="*70)
    print("SUMMARY:")
    print("="*70)
    print(f"Overall {horizon}-Day Direction Accuracy: {accuracy_overall:.2f}%")

    if accuracy_overall > 65:
        print(f" EXCELLENT - Model captures {horizon}-day trends very well!")
    elif accuracy_overall > 60:
        print(f" GOOD - Model has solid {horizon}-day directional edge")
    elif accuracy_overall > 55:
        print(f" MODERATE - Better than random, shows promise")
    else:
        print(f" POOR - Not significantly better than random")

    return actual_changes, pred_changes, {
        'accuracy': accuracy_overall,
        'weekly_accuracies': weekly_accuracies,
        'tp': tp, 'tn': tn, 'fp': fp, 'fn': fn,
        'correct_predictions': direction_match_overall
    }
```

In [21]:

```
################################ Add OHLC featres ########################################
def add_ohlc_features_CORR(data_df):
    """
    Adds 13 candlestick/OHLC-based features to capture intraday dynamics
    """
    df = data_df.copy()
    # ========================================
    # 1. Basic OHLC returns (stationary)
    # ========================================
    df['Open_return'] = df['Open'].pct_change()
    df['High_return'] = df['High'].pct_change()
    df['Low_return'] = df['Low'].pct_change()
    # Close_return already exists

    # ========================================
    # 2. Gap Analysis (overnight moves)
```

```
    # ========================================
    df['gap'] = (df['Open'] - df['Close'].shift(1)) / df['Close'].shift(1)
    # Gap down filled: gap > 0 and price recovered
    gap_down_filled = (df['gap'] > 0) & (df['Low'] < df['Open'])
    # Gap up filled: gap < 0 and price declined
    gap_up_filled = (df['gap'] < 0) & (df['High'] > df['Open'])
    # Either gap filled
    df['gap_filled'] = (gap_down_filled | gap_up_filled).astype(int)

    # ========================================
    # 3. Intraday Range (volatility proxy)
    # ========================================
    df['daily_range'] = (df['High'] - df['Low']) / df['Close']
    df['range_pct_change'] = df['daily_range'].pct_change()

    # ========================================
    # 4. Close Position in Range (strength)
    # ========================================
    # 0 = closed at low, 1 = closed at high
    df['close_position'] = (df['Close'] - df['Low']) / (df['High'] - df['Low'] + 1e-10)  # Avoid div by 0

    # ========================================
    # 5. Body and Wicks (candlestick analysis)
    # ========================================
    df['body'] = (df['Close'] - df['Open']) / df['Close']  # Positive = bullish candle
    df['upper_wick'] = (df['High'] - df[['Open', 'Close']].max(axis=1)) / df['Close']
    df['lower_wick'] = (df[['Open', 'Close']].min(axis=1) - df['Low']) / df['Close']

    # ========================================
    # 6. OHLC Relative Relationships
    # ========================================
    df['high_close_ratio'] = df['High'] / df['Close']  # How much higher was intraday peak?
    df['low_close_ratio'] = df['Low'] / df['Close']    # How much lower was intraday trough?

    # ========================================
    # Final cleanup
    # ========================================
    df = df.dropna()

    return df
##############################################################################################
```

In [22]:

```
############################## Add Macro Regime Featurres ###################################
def add_macro_regime_features_CORR(data_df, vix_threshold=20,
                                        curve_normal=0.5,
                                        curve_steep=1.5,
                                        rolling_window=60,
                                        min_periods=30):
    """
```

```
    Add regime based on FUNDAMENTAL indicators
    Not technical indicators!
    Returns: DataFrame with 4 additional columns

     Missing values are forward-filled then filled with 0
    """

    # Input validation
    required_cols = ['DGS10', 'VIX_indx_close', 'T10Y2Y', 'gold_vix']
    missing = [col for col in required_cols if col not in data_df.columns]
    if missing:
        raise ValueError(f"Missing required columns: {missing}")

    df = data_df.copy()

    # Economic regime (based on fundamental features)
    # 1. Interest rate regime(with NaN handling)
    rate_rolling_median = (
        df['DGS10']
        .rolling(rolling_window, min_periods=min_periods)
        .median()
        .shift(1)  # Avoid data leakage
     #   .fillna(method='bfill')  # Fill initial NaN
    )
    df['rate_regime'] = (df['DGS10'] > rate_rolling_median).astype(int)

    # 2. Volatility regime (VIX-based simple threshold)
    # Low vol - 0; High vol - 1
    df['vol_regime'] = (df['VIX_indx_close'] > vix_threshold).astype(int)

    # 3. Yield curve regime(categorical)
    df['curve_regime'] = 0  # Flat/inverted
    df.loc[df['T10Y2Y'] > curve_normal, 'curve_regime'] = 1  # Normal
    df.loc[df['T10Y2Y'] > curve_steep, 'curve_regime'] = 2  # Steep

    # 4. Gold/Dollar regime (risk on/off with NaN handling)
    risk_rolling_median = (
        df['gold_vix']
        .rolling(rolling_window, min_periods=min_periods)
        .median()
        .shift(1)  # Avoid data leakage
        #.fillna(method='bfill')  # Fill initial NaN
    )
    df['risk_regime'] = (df['gold_vix'] > risk_rolling_median).astype(int)
    df['risk_regime'] = (df['gold_vix'] > df['gold_vix'].rolling(60).median().shift(1)).astype(int)

    return df
##############################################################################################
```

In [23]:

```
##################################### Add TA featres ###################################
def add_technical_momentum_features_CORR(data_df):
    """
    Add technical indicators for short-term (5-15 day) predictions
    adds 19 TA features, loses ~50 rows
    """
    df = data_df.copy()

    # ========================================
    # 1. RSI (momentum)
    # ========================================
    from ta.momentum import RSIIndicator
    df['RSI_14'] = RSIIndicator(df['Close'], window=14).rsi()
    df['RSI_7'] = RSIIndicator(df['Close'], window=7).rsi()

    # ADDED: Normalize RSI to [-1, 1] range for better scaling
    df['RSI_14_norm'] = (df['RSI_14'] - 50) / 50  # -1 (oversold) to +1 (overbought)
    df['RSI_7_norm'] = (df['RSI_7'] - 50) / 50

    # Drop raw RSI (keep normalized versions)
    df = df.drop(columns=['RSI_14', 'RSI_7'])

    # ========================================
    # 2. MACD (trend)
    # ========================================
    from ta.trend import MACD
    macd = MACD(df['Close'])
    df['MACD'] = macd.macd()
    df['MACD_signal'] = macd.macd_signal()
    df['MACD_diff'] = macd.macd_diff()

    # ADDED: Normalize MACD by price (make it scale-invariant)
    df['MACD_norm'] = df['MACD'] / df['Close']
    df['MACD_signal_norm'] = df['MACD_signal'] / df['Close']
    df['MACD_diff_norm'] = df['MACD_diff'] / df['Close']

    # Drop raw MACD (keep normalized versions)
    df = df.drop(columns=['MACD', 'MACD_signal', 'MACD_diff'])

    # ========================================
    # 3. Bollinger Bands (volatility + mean reversion)
    # ========================================
    from ta.volatility import BollingerBands
    bb = BollingerBands(df['Close'])
    df['BB_width'] = (bb.bollinger_hband() - bb.bollinger_lband()) / df['Close']
    df['BB_position'] = (df['Close'] - bb.bollinger_lband()) / (bb.bollinger_hband() - bb.bollinger_lband())

    # BB squeeze indicator (low volatility before breakout)
    df['BB_squeeze'] = (df['BB_width'] < df['BB_width'].rolling(20).mean()).astype(int)
```

```python
# ========================================
# 4. EMA - SMA (Moving Averages)
# ========================================
# Short-term: EMA (minimal data loss)
df['EMA_10'] = df['Close'].ewm(span=10, adjust=False).mean()
df['EMA_20'] = df['Close'].ewm(span=20, adjust=False).mean()

# Long-term: SMA (trend following)
df['SMA_20'] = df['Close'].rolling(20).mean()
df['SMA_50'] = df['Close'].rolling(50).mean()

# Derived features (relative to price)
df['price_vs_EMA10'] = (df['Close'] - df['EMA_10']) / df['Close']
df['price_vs_EMA20'] = (df['Close'] - df['EMA_20']) / df['Close']
df['price_vs_SMA20'] = (df['Close'] - df['SMA_20']) / df['Close']
df['price_vs_SMA50'] = (df['Close'] - df['SMA_50']) / df['Close']

# Crossover signals (binary)
df['EMA_cross'] = (df['EMA_10'] > df['EMA_20']).astype(int)
df['MA_cross'] = (df['SMA_20'] > df['SMA_50']).astype(int)

# Normalize trend_strength by price
df['trend_strength_ema'] = (df['EMA_20'] - df['SMA_50']) / df['Close']

# Drop raw MA columns (save memory)
df = df.drop(columns=['EMA_10', 'EMA_20', 'SMA_20', 'SMA_50'])

# ========================================
# 5. Volume momentum
# ========================================
df['volume_SMA_20'] = df['Volume'].rolling(20).mean()
df['volume_ratio'] = df['Volume'] / (df['volume_SMA_20'] + 1e-10)  # ← ADDED: Avoid div by 0

# ADDED: Volume trend (is volume increasing or decreasing?)
df['volume_trend'] = df['volume_SMA_20'].pct_change(5)

# Drop raw volume SMA
df = df.drop(columns=['volume_SMA_20'])

# ========================================
# 6. Additional Momentum Indicators
# ========================================

# ADDED: Rate of Change (momentum)
df['ROC_10'] = ((df['Close'] - df['Close'].shift(10)) / df['Close'].shift(10))
df['ROC_20'] = ((df['Close'] - df['Close'].shift(20)) / df['Close'].shift(20))

# ADDED: ADX (trend strength)
```

```
    from ta.trend import ADXIndicator
    adx = ADXIndicator(df['High'], df['Low'], df['Close'], window=14)
    df['ADX'] = adx.adx()
    df['ADX_norm'] = df['ADX'] / 100  # Normalize to [0, 1]

    # Drop raw ADX
    df = df.drop(columns=['ADX'])
    # ========================================
    # Final cleanup
    # ========================================
    df = df.dropna()

    print(f"   Added {df.shape[1] - data_df.shape[1]} technical features")
    print(f"   Remaining data: {len(df)} rows (lost {len(data_df) - len(df)} rows)")

    return df
##############################################################################################
```

In [24]:

```
###################### Market Regime Features (10 new) ###################################
def add_market_regime_features_CORR(data_df, lookback_short=20, lookback_long=60):
    """
    Add market regime features:
    - Trend: Bullish/Bearish/Flat
    - Volatility: High/Low
    - Combined regime classification

    Parameters:
    lookback_short : int
        Short-term lookback for trend (default: 20 days)
    lookback_long : int
        Long-term lookback for trend (default: 60 days)
    Adds 10 regime features, loses ~272 rows
    """

    df = data_df.copy()

    print("\n" + "="*70)
    print("ADDING MARKET REGIME FEATURES")
    print("="*70)

    # ========================================
    # 1. TREND REGIME (Bullish/Bearish/Flat)
    # ========================================

    # Calculate SMAs
    df['SMA_short'] = df['Close'].rolling(lookback_short).mean()
    df['SMA_long'] = df['Close'].rolling(lookback_long).mean()

    # Trend strength (normalized by price)
```

```python
df['trend_strength_sma'] = (df['SMA_short'] - df['SMA_long']) / df['Close']

# Classify trend
trend_threshold = 0.02  # 2% difference = clear trend
df['regime_trend'] = 0  # Flat
df.loc[df['trend_strength_sma'] > trend_threshold, 'regime_trend'] = 1   # Bullish
df.loc[df['trend_strength_sma'] < -trend_threshold, 'regime_trend'] = -1 # Bearish

# Trend momentum (is trend accelerating?)
df['trend_momentum'] = df['trend_strength_sma'].diff(5)

# ============================================
# 2. VOLATILITY REGIME (High/Low)
# ============================================

# Historical volatility (20-day rolling std of returns, annualized)
df['volatility_20d'] = df['Close'].pct_change().rolling(20).std() * np.sqrt(252)

# Volatility percentile (where is current vol vs last year?)
# Calculate percentile using expanding window
df['vol_rank'] = df['volatility_20d'].rolling(252).rank(pct=True)
df['vol_percentile'] = df['vol_rank']  # Last value is the percentile

# Classify volatility
df['regime_volatility'] = 0  # Low vol
df.loc[df['vol_percentile'] > 0.7, 'regime_volatility'] = 1  # High vol
df.loc[df['vol_percentile'] > 0.9, 'regime_volatility'] = 2  # Extreme vol

# ============================================
# 3. COMBINED MARKET REGIME
# ============================================

# Create single regime feature (9 possible states)
# Encoding: regime_combined = (regime_trend + 1) * 3 + regime_volatility
# Results in: 0-8 representing all combinations
df['regime_combined'] = (df['regime_trend'] + 1) * 3 + df['regime_volatility']

# Regime labels for interpretation
regime_names = {
    0: 'Bearish_LowVol',    1: 'Bearish_HighVol',    2: 'Bearish_ExtremeVol',
    3: 'Flat_LowVol',       4: 'Flat_HighVol',       5: 'Flat_ExtremeVol',
    6: 'Bullish_LowVol',    7: 'Bullish_HighVol',    8: 'Bullish_ExtremeVol'
}


# ============================================
# CLEANUP
# ============================================
```

```
    # Drop temporary SMA columns
    df = df.drop(columns=['SMA_short', 'SMA_long'], errors='ignore')

    # Drop NaN rows from rolling calculations
    df = df.dropna()

    # ========================================
    # 4. ADDITIONAL REGIME FEATURES
    # ========================================

    # Regime persistence (how many days in current regime?)
    df['regime_duration'] = (
        df['regime_combined'] != df['regime_combined'].shift(1)
    ).cumsum()
    df['regime_duration'] = df.groupby('regime_duration').cumcount() + 1

    # Regime transition indicator (just changed?)
    df['regime_transition'] = (
        df['regime_combined'] != df['regime_combined'].shift(1)
    ).astype(int)

    # ========================================
    # 5. VIX-BASED REGIME (if available)
    # ========================================

    if 'VIX_indx_close' in df.columns:
        # VIX regime thresholds
        df['regime_vix'] = 0  # Low fear
        df.loc[df['VIX_indx_close'] > 20, 'regime_vix'] = 1  # Elevated
        df.loc[df['VIX_indx_close'] > 30, 'regime_vix'] = 2  # High fear
        df.loc[df['VIX_indx_close'] > 40, 'regime_vix'] = 3  # Panic

    # ========================================
    # SUMMARY
    # ========================================

    print(f"\n Added regime features")
    print(f" Data shape: {df.shape}")
    print(f"\n Regime Distribution:")

    for code, count in df['regime_combined'].value_counts().sort_index().items():
        regime_name = regime_names.get(code, 'Unknown')
        pct = count / len(df) * 100
        print(f"  {code}: {regime_name:25s} - {count:4d} days ({pct:5.1f}%)")

    print("="*70)

    return df
################################################################################################
```

In [25]:
```python
############################## Select Features By Horizon ###################################
def select_features_by_horizon(data_df, horizon=None):
    """
    Select optimal features based on prediction horizon
    Features ordered by cross-category importance
    """

    if horizon is None:
        horizon = parameters['forecast_horizon']

    # ========================================
    # Short-term features (horizon <= 7)
    # Ordered by: immediate impact + cross-category usefulness
    # ========================================
    short_term = [
        # Top 5 (useful for ALL horizons - momentum & regime changes)
        'regime_transition',      # #1 - Immediate regime shifts
        'regime_vix',            # #2 - Fear gauge (quick mover)
        'VIX_indx_close_pct()',  # #3 - VIX returns
        'gap',                   # #4 - Overnight gaps
        'volume_ratio',          # #5 - Volume spikes

        # Next tier (strong short-term signals)
        'trend_momentum',        # Trend acceleration
        'RSI_7_norm',           # Short-term overbought/oversold
        'MACD_diff_norm',       # MACD momentum
        'gap_filled',           # Gap behavior
        'BB_squeeze',           # Volatility compression

        # Technical indicators (pure short-term)
        'RSI_14_norm',
        'MACD_norm',
        #'MACD_signal_norm',     # droped
        'ROC_10',
        'body',
        'upper_wick',
        'lower_wick',
        'close_position',
        'BB_position',
        'BB_width',
        'price_vs_EMA10',
        'EMA_cross',
        'volume_trend',
        'vol_regime',
        'daily_range',
        'range_pct_change',
        'ADX_norm',
    ]
```

```python
# ========================================
# Medium-term features (horizon <= 20)
# Ordered by: trend stability + cross-category usefulness
# ========================================
medium_term = [
    # Top 5 (useful for ALL horizons - persistent trends)
    'regime_trend',          # #1 - Current trend state
    'regime_combined',       # #2 - Overall market regime
    'trend_strength_ema',    # #3 - Trend magnitude (from regime features)
    'price_vs_SMA20',        # #4 - Price vs 20-day trend
    'MA_cross',              # #5 - Moving average crossover

    # Next tier (medium-term dynamics)
    'regime_volatility',     # Current vol regime
    'regime_duration',       # Regime persistence
    'volatility_20d',        # 20-day volatility
    'price_vs_EMA20',
    'ROC_20',
    'Open_return',
    'High_return',
    'Low_return',

    # Market internals
    'high_close_ratio',
    'low_close_ratio',
    'OBV_base_dif',
    'OBV_futures_dif',

    # Related markets
    'GOLD_fut_close_pct()',
    'DXY_indx_close_pct()',
    'S&P_fut_close_pct()',
    'S&P_fut_vol_pct()',
    'Volume_pct()',
    'front_spot',
    'gold_vix',
    'fut_to_vix_ratio',
    'risk_regime',
]

# ========================================
# Long-term features (horizon > 20)
# Ordered by: macro stability + cross-category usefulness
# ========================================
long_term = [
    # Top 5 (useful for ALL horizons - macro context)
    'vol_percentile',         # #1 - Long-term vol context (252-day)
    'T10Y2Y_yield_curve',    # #2 - Yield curve (recession signal)
    'rate_regime',           # #3 - Interest rate environment
```

```
    'curve_regime',          # #4 - Yield curve regime
    'term_spread',           # #5 - Credit spreads

    # Next tier (fundamental trends)
    'trend_strength_sma',    # - Trend magnitude long (from regime features)
    #'price_vs_SMA50',        # droped
    'DGS10_pct()',
    'T10Y2Y',
    'T10Y2Y_diff',

    # Economic indicators
    'GDP_value_pct()',
    'CPIAUCSL_value_pct()',
    'PAYEMS_value_pct()',
    'UNRATE_value_diff',

    # Calendar effects
    'is_GDP_release_day',
    'is_CPIAUCSL_release_day',
    'is_UNRATE_release_day',
    'period_CPI',
    'period_PAYMES_UNRATE',
    'period_GDP',
]

# ========================================
# Feature Selection Logic
# ========================================

# Always include target
selected = ['target']

if horizon <= 7:
    # Short-term model: All short + top medium + top long
    selected += short_term
    selected += medium_term[:10]  # Top 10 medium (includes regime features)
    selected += long_term[:5]     # Top 5 long (macro context)

elif horizon <= 20:
    # Medium-term model: Top short + all medium + top long
    selected += short_term[:10]   # Top 10 short (includes regime transitions)
    selected += medium_term       # All medium features
    selected += long_term[:12]    # Top 12 long (extended macro)

else:  # horizon > 20
    # Long-term model: Top short + all medium + all long
    selected += short_term[:5]    # Top 5 short (regime signals)
    selected += medium_term       # All medium (regime context)
    selected += long_term         # All long-term features
```

```
    # Filter to only columns that exist in dataframe
    selected = [col for col in selected if col in data_df.columns]

    # Check for missing columns
    missing = [col for col in selected if col not in data_df.columns
           and col != 'target']
    if missing:
        print(f"WARNING: {len(missing)} features not found:")
        for feat in missing[:5]:
            print(f"  - {feat}")

#---------------------------------- me ----------------------------------------------------
    # 4. Move 'target' to end
    selected_cols = [col for col in selected if col != 'target'] + ['target']

#---------------------------------- me ----------------------------------------------------


    print(f"   Selected {len(selected_cols)} features ('target' included) for {horizon}-day horizon")
    print(f"   Short-term: {sum(1 for f in selected_cols if f in short_term)}")
    print(f"   Medium-term: {sum(1 for f in selected_cols if f in medium_term)}")
    print(f"   Long-term: {sum(1 for f in selected_cols if f in long_term)}")

    return data_df[selected_cols]
##############################################################################################
```

In [26]:
```
################ Check if any NaN exists in the entire DataFrame ##########################
def check_nan(df):
    nan_mask = df.isna()
    has_nan = nan_mask.any().any()
    print(f"Does the DataFrame contain any NaN values? {has_nan}")
    if has_nan:
        nan_counts = nan_mask.sum()
        print(f"Count of NaN values per column:\n{nan_counts[nan_counts > 0]}")
##############################################################################################
```

In [27]:
```
################################ Get Optimal Time Step ####################################
def get_optimal_timestep(horizon):
    """
    Dynamic time step based on horizon
    Uses 4-6x multiplier for optimal pattern recognition
    """
    if horizon <= 5:
        return max(30, horizon * 6)  # 6x for short-term
    elif horizon <= 20:
        return max(60, horizon * 4)  # 4x for medium-term
    else:  # horizon > 20
        return max(90, horizon * 4)  # 4x for long-term
```

```
##############################################################################################
```

PIPELINE

In [28]:
```
# ==========================================
# PIPELINE EXECUTION
# ==========================================
# 1. Load and prepare data (tatal 44 features)
data_data = pd.read_csv('../data_df_features.csv', index_col="Date", parse_dates=True)
data_data_cols = data_data.columns.tolist()
```

In [29]:
```
# Fundamentals regime (add 4 more features)
data_df_MACRO = add_macro_regime_features_CORR(data_data)
# Add OHLC features (13 features added)
data_df_ohlc = add_ohlc_features_CORR(data_df_MACRO)
# Add TA features (19 features added)
data_df_ta = add_technical_momentum_features_CORR(data_df_ohlc)
# Add Market Regime features (10 features added)
data_df_mr = add_market_regime_features_CORR(data_df_ta)
# Copy
data_df = data_df_mr.copy()
# Check in any NaN
check_nan(data_df)
```

```
  Added 20 technical features
  Remaining data: 6252 rows (lost 49 rows)

======================================================================
ADDING MARKET REGIME FEATURES
======================================================================

 Added regime features
 Data shape: (5981, 92)

 Regime Distribution:
  0: Bearish_LowVol            -  163 days (  2.7%)
  1: Bearish_HighVol           -  314 days (  5.2%)
  2: Bearish_ExtremeVol        -  430 days (  7.2%)
  3: Flat_LowVol               - 2286 days ( 38.2%)
  4: Flat_HighVol              -  577 days (  9.6%)
  5: Flat_ExtremeVol           -  292 days (  4.9%)
  6: Bullish_LowVol            - 1764 days ( 29.5%)
  7: Bullish_HighVol           -  127 days (  2.1%)
  8: Bullish_ExtremeVol        -   28 days (  0.5%)
======================================================================
Does the DataFrame contain any NaN values? False
```

In [30]:
```
# 2. Prepare returns-based features
features, close_prices = prepare_returns_features_OPT(data_df)
```

```
  Data prepared: 76 features, 5980 samples
  Target at position: 76 (last)
```

In [32]:
```
correlation_fun(features.drop(columns=['target']))
```

```
High correlation pairs:
T10Y2Y <-> curve_regime: 0.934
S&P_fut_close_pct() <-> body: 0.931
PAYEMS_value_pct() <-> UNRATE_value_diff: -0.946
PAYEMS_value_pct() <-> macro_pressure: 0.946
UNRATE_value_diff <-> macro_pressure: -1.000
is_PAYEMS_release_day <-> is_UNRATE_release_day: 1.000
vol_regime <-> regime_vix: 0.864
RSI_14_norm <-> RSI_7_norm: 0.941
RSI_14_norm <-> BB_position: 0.912
RSI_7_norm <-> BB_position: 0.947
MACD_norm <-> MACD_signal_norm: 0.955
MACD_norm <-> price_vs_SMA50: 0.962
MACD_norm <-> trend_strength_ema: 0.930
MACD_norm <-> ROC_20: 0.863
MACD_signal_norm <-> price_vs_SMA50: 0.897
MACD_signal_norm <-> trend_strength_ema: 0.985
MACD_signal_norm <-> trend_strength_sma: 0.921
MACD_diff_norm <-> ROC_10: 0.851
price_vs_EMA10 <-> price_vs_EMA20: 0.944
price_vs_EMA10 <-> price_vs_SMA20: 0.930
price_vs_EMA20 <-> price_vs_SMA20: 0.984
price_vs_EMA20 <-> price_vs_SMA50: 0.882
price_vs_EMA20 <-> ROC_10: 0.871
price_vs_SMA20 <-> ROC_10: 0.903
price_vs_SMA50 <-> trend_strength_ema: 0.902
price_vs_SMA50 <-> ROC_20: 0.855
trend_strength_ema <-> trend_strength_sma: 0.949
regime_trend <-> regime_combined: 0.938
vol_rank <-> vol_percentile: 1.000
OBV_base_dif <-> OBV_futures_dif: 0.851
```

In [33]:
```
# Drop high correlated columns
features_total = features.drop(columns=['macro_pressure', 'is_PAYEMS_release_day', 'vol_rank', 'MACD_signal_norm', 'price_vs_SMA50'])
```

In [34]:
```
print(f"\nFeatures prepared:")
print("   Original data shape:", data_df.shape)
print("   Features+target data shape:", features_total.shape)
print(f"   \nColumns: {list(features_total.columns)}")
```

```
Features prepared:
   Original data shape: (5981, 92)
   Features+target data shape: (5980, 72)

Columns: ['T10Y2Y', 'Volume_pct()', 'S&P_fut_close_pct()', 'S&P_fut_vol_pct()', 'GOLD_fut_close_pct()', 'DXY_indx_close_pct()', 'VIX_indx_c
lose_pct()', 'DGS10_pct()', 'T10Y2Y_diff', 'GDP_value_pct()', 'CPIAUCSL_value_pct()', 'PAYEMS_value_pct()', 'UNRATE_value_diff', 'front_spo
t', 'gold_vix', 'is_GDP_release_day', 'is_CPIAUCSL_release_day', 'is_UNRATE_release_day', 'T10Y2Y_yield_curve', 'fut_to_vix_ratio', 'term_s
pread', 'period_CPI', 'period_PAYMES_UNRATE', 'period_GDP', 'rate_regime', 'vol_regime', 'curve_regime', 'risk_regime', 'Open_return', 'Hig
h_return', 'Low_return', 'gap', 'gap_filled', 'daily_range', 'range_pct_change', 'close_position', 'body', 'upper_wick', 'lower_wick', 'hig
h_close_ratio', 'low_close_ratio', 'RSI_14_norm', 'RSI_7_norm', 'MACD_norm', 'MACD_diff_norm', 'BB_width', 'BB_position', 'BB_squeeze', 'pr
ice_vs_EMA10', 'price_vs_EMA20', 'price_vs_SMA20', 'EMA_cross', 'MA_cross', 'trend_strength_ema', 'volume_ratio', 'volume_trend', 'ROC_10',
'ROC_20', 'ADX_norm', 'trend_strength_sma', 'regime_trend', 'trend_momentum', 'volatility_20d', 'vol_percentile', 'regime_volatility', 'reg
ime_combined', 'regime_duration', 'regime_transition', 'regime_vix', 'OBV_base_dif', 'OBV_futures_dif', 'target']
```

In [37]:
```python
# 3. Set parameters for Nested CV
forecast_horizon = 45  # 20, 45

parameters = {
    'forecast_horizon': forecast_horizon,
    'time_step': get_optimal_timestep(forecast_horizon),
    'train_window': 1500,
    'embargo_prop': 0.10,
    # step_size_type: 'nested'
}
```

In [38]:
```python
# Select features by horizon
features_selected = select_features_by_horizon(features_total)
features_df = features_selected.copy()
features_df_cols = features_df.columns.tolist()
```

```
  Selected 51 features ('target' included) for 45-day horizon
  Short-term: 5
  Medium-term: 26
  Long-term: 19
```

In [41]:
```python
# 5. Run PWFCV
best_hp, folds = run_pwfcv_lstm_returns_OPT(
    features_df,
    parameters,
    n_samples=10     # For hp
)
```

```
======================================================================
ROLLING WINDOW CV - STEP SIZE CALCULATION
======================================================================
Strategy: adaptive (Adaptive for 45-day horizon)

Horizon: 45 days
Validation length: 45 days
Embargo length: 23 days
Purge length: 224 days
Train window: 1500 days
Total fold length: 1792 days

______________________________________________________________________
STEP SIZE RANGE (for Rolling Window with Overlap):
  Min safe step (max folds):    68 days
  Max overlap step (min folds): 1769 days
  Selected step size:           408 days

______________________________________________________________________
RESULTING OVERLAP:
  Training window overlap: 1092 days (72.8%)
  Validation overlap: 0 days (0%) ✓ No leakage
======================================================================

Length of the TRAIN DATA SET is: 4784
Length of the TEST DATA SET is: 1196

Rolling Window CV Configuration:
  Overlap strategy: adaptive
Fold length = 1792
Train window = 1500
Purge = 224
Validation = 45
Embargo = 23
Step size = 408
The number of folds: 8
Time step: 180
Forecast horizon: 45

Fold 0:
  Train indices: (0, 1500)
  Purge: (1500, 1724)
  Val indices: (1724, 1769)
  Embargo: (1769, 1792)

Fold 1:
  Train indices: (408, 1908)
  Purge: (1908, 2132)
  Val indices: (2132, 2177)
```

```
  Embargo: (2177, 2200)

Fold 2:
  Train indices: (816, 2316)
  Purge: (2316, 2540)
  Val indices: (2540, 2585)
  Embargo: (2585, 2608)

Fold 3:
  Train indices: (1224, 2724)
  Purge: (2724, 2948)
  Val indices: (2948, 2993)
  Embargo: (2993, 3016)

Fold 4:
  Train indices: (1632, 3132)
  Purge: (3132, 3356)
  Val indices: (3356, 3401)
  Embargo: (3401, 3424)

Fold 5:
  Train indices: (2040, 3540)
  Purge: (3540, 3764)
  Val indices: (3764, 3809)
  Embargo: (3809, 3832)

Fold 6:
  Train indices: (2448, 3948)
  Purge: (3948, 4172)
  Val indices: (4172, 4217)
  Embargo: (4217, 4240)

Fold 7:
  Train indices: (2856, 4356)
  Purge: (4356, 4580)
  Val indices: (4580, 4625)
  Embargo: (4625, 4648)

======================================================================
VALIDATION LEAKAGE CHECK:
======================================================================
✓ Fold 0 → 1: Safe (gap after embargo = 340 days)
✓ Fold 1 → 2: Safe (gap after embargo = 340 days)
✓ Fold 2 → 3: Safe (gap after embargo = 340 days)
✓ Fold 3 → 4: Safe (gap after embargo = 340 days)
✓ Fold 4 → 5: Safe (gap after embargo = 340 days)
✓ Fold 5 → 6: Safe (gap after embargo = 340 days)
✓ Fold 6 → 7: Safe (gap after embargo = 340 days)
```

```
✓ All 8 folds are safe from validation leakage!
======================================================================


Trial 1/10
Evaluating hyperparameters: {'lstm_units': 32, 'dropout': 0.29683364680743846, 'lr': 0.00010330948771795018, 'epochs': 23, 'batch_size': 16
, 'loss_type': 'mse'}
Mean PWFCV loss: 4.997079
  New best!

Trial 2/10
Evaluating hyperparameters: {'lstm_units': 32, 'dropout': 0.10716580894028942, 'lr': 0.00013224935709489537, 'epochs': 32, 'batch_size': 16
, 'loss_type': 'mse'}
Mean PWFCV loss: 4.973077
  New best!

Trial 3/10
Evaluating hyperparameters: {'lstm_units': 32, 'dropout': 0.16219705033454196, 'lr': 0.0006848784292355798, 'epochs': 32, 'batch_size': 32,
'loss_type': 'mse'}
Mean PWFCV loss: 5.022202

Trial 4/10
Evaluating hyperparameters: {'lstm_units': 64, 'dropout': 0.353362456389525, 'lr': 0.0003922999695225617, 'epochs': 12, 'batch_size': 64, '
loss_type': 'mse'}
Mean PWFCV loss: 4.982398

Trial 5/10
Evaluating hyperparameters: {'lstm_units': 64, 'dropout': 0.20936422095610396, 'lr': 0.0001595622668486508, 'epochs': 27, 'batch_size': 64,
'loss_type': 'mse'}
Mean PWFCV loss: 5.040113

Trial 6/10
Evaluating hyperparameters: {'lstm_units': 32, 'dropout': 0.235998369888189, 'lr': 0.0002486678408332023, 'epochs': 12, 'batch_size': 32, '
loss_type': 'mse'}
Mean PWFCV loss: 5.018827

Trial 7/10
Evaluating hyperparameters: {'lstm_units': 32, 'dropout': 0.13097470802697372, 'lr': 0.00029337119115384916, 'epochs': 21, 'batch_size': 32
, 'loss_type': 'mse'}
Mean PWFCV loss: 4.982686

Trial 8/10
Evaluating hyperparameters: {'lstm_units': 64, 'dropout': 0.39610235483993594, 'lr': 0.0008962506989759147, 'epochs': 26, 'batch_size': 16,
'loss_type': 'mse'}
Mean PWFCV loss: 4.987687

Trial 9/10
Evaluating hyperparameters: {'lstm_units': 64, 'dropout': 0.2550196541182649, 'lr': 0.0005369990158802189, 'epochs': 16, 'batch_size': 16,
'loss_type': 'mse'}
```

```
Mean PWFCV loss: 4.969933
  New best!

Trial 10/10
Evaluating hyperparameters: {'lstm_units': 64, 'dropout': 0.3283756486290948, 'lr': 0.0004689938951781398, 'epochs': 18, 'batch_size': 64,
'loss_type': 'mse'}
Mean PWFCV loss: 4.982007

================================================================================
Best hyperparameters: {'lstm_units': 64, 'dropout': 0.2550196541182649, 'lr': 0.0005369990158802189, 'epochs': 16, 'batch_size': 16, 'loss_
type': 'mse'}
Best PWFCV loss: 4.969933
================================================
```

In [42]:
```python
##### 6. Split data for final training
train_size = int(len(features_df) * 0.8)
train_features = features_df.iloc[:train_size]
test_features = features_df.iloc[train_size:]
train_prices = close_prices.iloc[:train_size]
test_prices = close_prices.iloc[train_size:]
```

In [43]:
```python
# 7. Train final model
#------------------------------------
# Option 1. Uses dummy validation data
#------------------------------------
'''
final_model, target_scaler, features_scaler = retrain_final_model_returns(
    train_features,
    parameters,
    best_hp,
    use_early_stopping=True
)
'''

#------------------------------------
# Option 2. Uses dedicated functions for cleanest implementation
#------------------------------------
final_model, target_scaler, features_scaler = retrain_final_model_returns_CLEAN(
    train_features,
    parameters,
    best_hp,
    use_early_stopping=True
)
```

```
======================================================================
TRAINING FINAL MODEL (CLEAN VERSION)
======================================================================
Total training samples: 4379
Features: 51
Time step: 180, Horizon: 45

Best Hyperparameters:
  lstm_units: 64
  dropout: 0.255
  lr: 5.37e-04
  epochs: 16
  batch_size: 16
  loss_type: mse
======================================================================

Using early stopping:
  Training samples: 3941
  Validation samples: 438
======================================================================
Epoch 1/16
247/247 ━━━━━━━━━━━━━━━━━━━━ 13s 47ms/step - loss: 0.8880 - val_loss: 1.0162
Epoch 2/16
247/247 ━━━━━━━━━━━━━━━━━━━━ 13s 51ms/step - loss: 0.8782 - val_loss: 1.0160
Epoch 3/16
247/247 ━━━━━━━━━━━━━━━━━━━━ 15s 62ms/step - loss: 0.8756 - val_loss: 1.0157
Epoch 4/16
247/247 ━━━━━━━━━━━━━━━━━━━━ 15s 61ms/step - loss: 0.8730 - val_loss: 1.0163
Epoch 5/16
247/247 ━━━━━━━━━━━━━━━━━━━━ 15s 60ms/step - loss: 0.8706 - val_loss: 1.0182
Epoch 6/16
247/247 ━━━━━━━━━━━━━━━━━━━━ 15s 59ms/step - loss: 0.8676 - val_loss: 1.0178
Epoch 6: early stopping
Restoring model weights from the end of the best epoch: 3.

Training stopped at epoch 6/16
Best validation loss: 1.015689
======================================================================
FINAL MODEL TRAINING COMPLETE
======================================================================
```

In [48]:
```
'''
CHECK 1: Model Summary
----------------------
'''
final_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 180, 64)	29,696
lstm_1 (LSTM)	(None, 64)	33,024
dense (Dense)	(None, 45)	2,925

Total params: 196,937 (769.29 KB)

Trainable params: 65,645 (256.43 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 131,292 (512.86 KB)

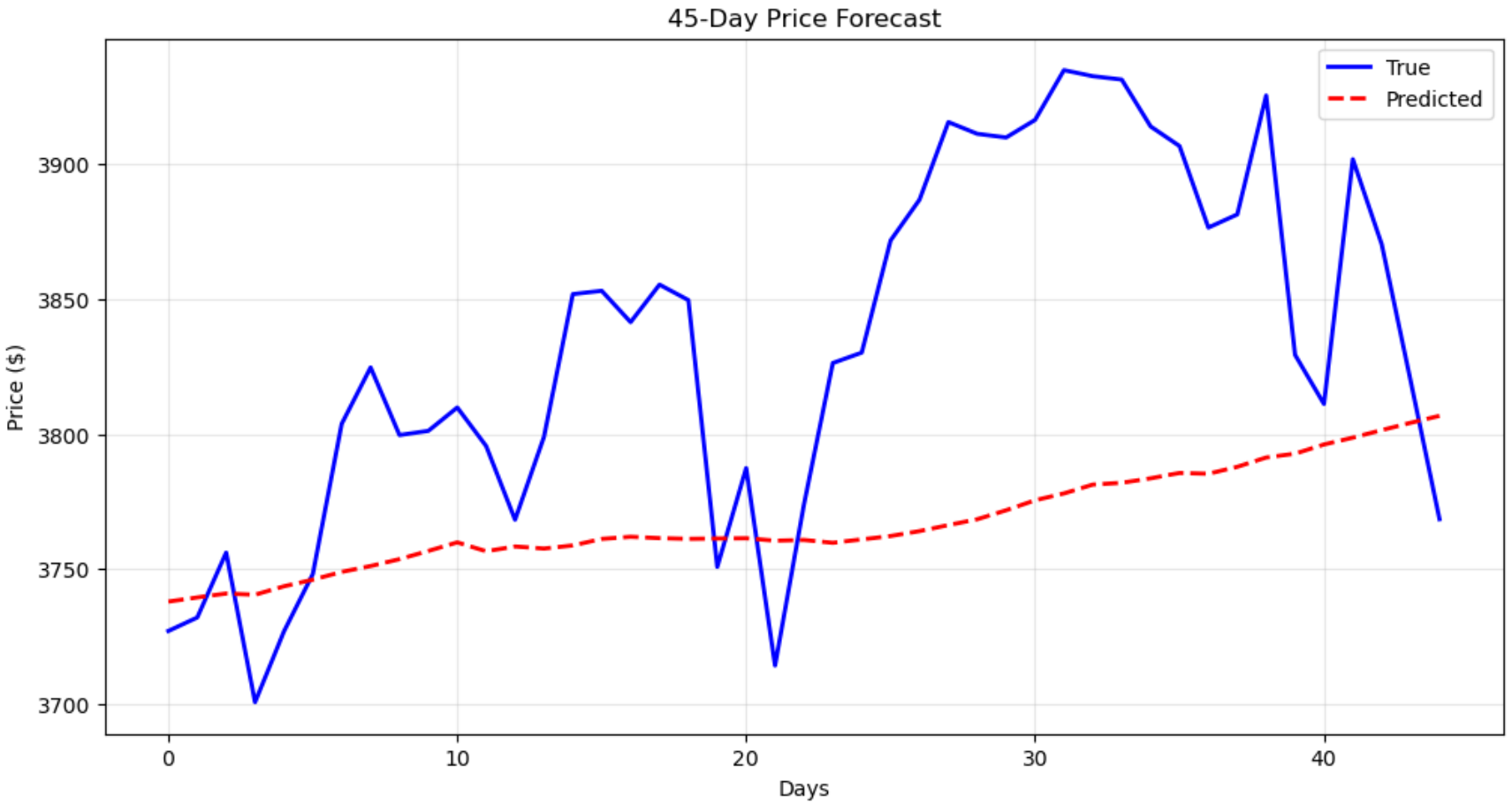
In [45]:
```
# 9. Evaluate on full test set (rolling)
actuals_ret, preds_ret, actuals_prices, preds_prices, _ = evaluate_rolling_test_returns_OPT(
    final_model,
    train_features,
    test_features,
    train_prices,
    test_prices,
    parameters
)
```

```
Running rolling evaluation on test set...
Making 26 predictions (45-day each)
  Completed 5/26 predictions...
  Completed 10/26 predictions...
  Completed 15/26 predictions...
  Completed 20/26 predictions...
  Completed 25/26 predictions...
  Completed 26/26 predictions...
Completed 26 predictions
\n======================================================================
ROLLING EVALUATION RESULTS (26 predictions)
======================================================================
\nTest Set Metrics:
  Direction Accuracy: 80.77% (21/26 correct)
  MAE:  $174.45
  MAPE: 3.78%
  RMSE: $245.80
\nPer-Window MAE Statistics:
  Best:   $35.26
  Worst:  $483.33
  Median: $114.65
  Std:    $127.35
======================================================================\n
```

In [44]:
```
# 8. Evaluate on first prediction
y_true_ret, y_pred_ret, y_true_prices, y_pred_prices, _ = evaluate_on_test_returns(
    final_model,
    train_features,
    test_features,
    train_prices,
    test_prices,
    parameters
)
```

```
======================================================================
RETURNS (Scaled):
TRUE: [-2.05754250e-01  8.22222989e-02  4.93522980e-01 -1.21659315e+00
  5.45467097e-01  4.34699792e-01  1.17200901e+00  4.17110995e-01
 -5.54929491e-01  7.52288663e-03  1.57594331e-01 -3.28892795e-01
 -6.06189957e-01  6.30496890e-01  1.09843767e+00 -4.70673147e-04
 -2.68953120e-01  2.65734758e-01 -1.46160957e-01 -2.09797386e+00
  7.61557233e-01 -1.58426876e+00  1.26918905e+00  1.09541971e+00
  5.53704564e-02  8.49726825e-01  2.88461581e-01  5.71025893e-01
 -1.15877318e-01 -5.38800070e-02  1.08116345e-01  3.54105906e-01
 -7.19616120e-02 -5.18802745e-02 -3.82340553e-01 -1.75698549e-01
 -6.49989382e-01  7.53451725e-02  8.89933937e-01 -2.00122048e+00
 -4.09321536e-01  1.89364814e+00 -6.78070467e-01 -1.08033967e+00
 -1.10866120e+00]
PRED: [ 0.02921564  0.00743581  0.00518296 -0.03560814  0.04111867  0.02693895
  0.03611483  0.02069064  0.02875818  0.03952598  0.04213979 -0.09678978
  0.01221826 -0.04263197 -0.00132596  0.02601986 -0.0091294  -0.0365522
 -0.03216299 -0.02309176 -0.02379241 -0.04567298 -0.02031073 -0.04836059
  0.00088561  0.00086507  0.01295073  0.0217662   0.02041721  0.04522934
  0.05316691  0.02813945  0.04387584 -0.01172931  0.00929383  0.01613774
 -0.03200839  0.02860694  0.04742638  0.00436002  0.04622966  0.02795443
  0.03307891  0.03157637  0.02885197]

======================================================================
RETURNS (Original %):
TRUE: [-0.22273805  0.13415472  0.64388455 -1.47548275  0.70825952  0.57098427
  1.48474038  0.54918627 -0.65547509  0.04157884  0.22756417 -0.37534512
 -0.7190028   0.81363795  1.39356254  0.03167233 -0.30106117  0.36158389
 -0.14888337 -2.56778837  0.9760626  -1.93114752  1.60517676  1.38982235
  0.10087691  1.08533222  0.38974952  0.73993484 -0.1113525  -0.03451849
  0.16624552  0.47110331 -0.05692724 -0.0320402  -0.44158356 -0.18548966
 -0.77328395  0.12563181  1.13516138 -2.44788075 -0.47502142  2.37907635
 -0.80808517 -1.30662219 -1.34172139]%
PRED: [ 0.06846294  0.04147093  0.03867895 -0.01187395  0.08321449  0.0656414
  0.07701318  0.0578978   0.06789601  0.08124066  0.08447999 -0.08769709
  0.04739787 -0.02057867  0.03061236  0.06450236  0.02094146 -0.01304393
 -0.00760434  0.00363775  0.00276942 -0.02434742  0.00708432 -0.02767821
  0.03335318  0.03332774  0.04830563  0.05923076  0.05755894  0.0883089
  0.09814602  0.0671292   0.08663148  0.01771937  0.0437736   0.05225533
 -0.00741273  0.06770856  0.09103172  0.03765907  0.08954861  0.0668999
  0.07325073  0.07138861  0.06801223]%

======================================================================
PRICES (Reconstructed $):
TRUE: [3727.04003906 3732.04003906 3756.07006836 3700.64990234 3726.86010742
 3748.13989258 3803.79003906 3824.67993164 3799.61010742 3801.18994141
 3809.84008789 3795.54003906 3768.25       3798.90991211 3851.85009766
 3853.07006836 3841.4699707  3855.36010742 3849.62011719 3750.77001953
 3787.37988281 3714.23999023 3773.86010742 3826.31005859 3830.16992188
```

```
 3871.73999023 3886.83007813 3915.59008789 3911.22998047 3909.87988281
 3916.37988281 3934.83007813 3932.59008789 3931.33007813 3913.9699707
 3906.70996094 3876.5        3881.37011719 3925.42993164 3829.34008789
 3811.14990234 3901.82006836 3870.29003906 3819.7199707  3768.4699707 ]
PRED: [3737.91740574 3739.46762893 3740.91418269 3740.47001477 3743.58283411
 3746.0399959  3748.92479137 3751.09541703 3753.64247126 3756.69197285
 3759.86576235 3756.56849867 3758.34902249 3757.57549849 3758.72580303
 3761.15033704 3761.93811269 3761.44749967 3761.1614208  3761.29817224
 3761.40219679 3760.48635187 3760.75263311 3759.71164185 3760.96568461
 3762.21924898 3764.03653647 3766.26616712 3768.43381318 3771.76172619
 3775.46352204 3777.99786804 3781.27072457 3781.94055779 3783.59605126
 3785.57295869 3785.29226535 3787.85531992 3791.30333695 3792.73107409
 3796.12746546 3798.66708435 3801.44976406 3804.16378796 3806.7509579 ]
================================================================================
\n================================================
FIRST PREDICTION (45-day forecast)
================================================================================
\nRETURNS (Original %):
TRUE: Mean=0.02%, Std=1.01%
PRED: Mean=0.04%, Std=0.04%
\n------------------------------------------------
PRICES (Reconstructed $):
Start: $3735.36
True End: $3768.47
Pred End: $3806.75
\n------------------------------------------------

Metrics (Price Scale):
MAE:  $71.97
MAPE: 1.86%
RMSE: $86.53
\n================================================

Cumulative Return (45 days):
  True: +1.11%
  Pred: +1.84%
  Direction: ✓ CORRECT
```



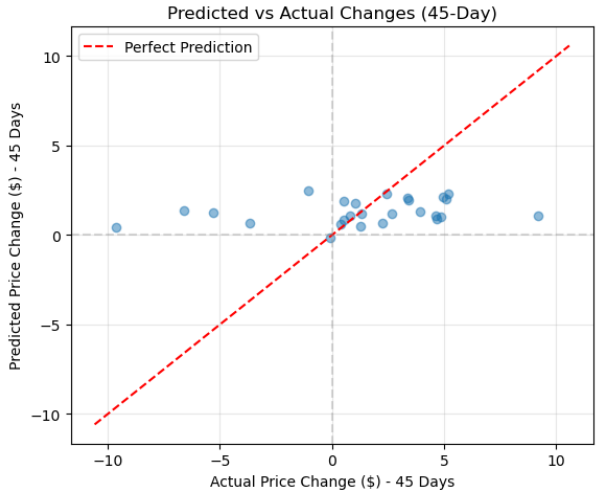
In [46]:
```
# 10. Analyze direction
actual_changes, pred_changes, direction_stats = analyze_direction_accuracy_CORRECTED(
    actuals_prices,  # shape: (n_windows, horizon)
    preds_prices,
    #target_scaler
    #horizon=parameters['forecast_horizon']; horizon from  actuals_prices.shape inside the function
)
```

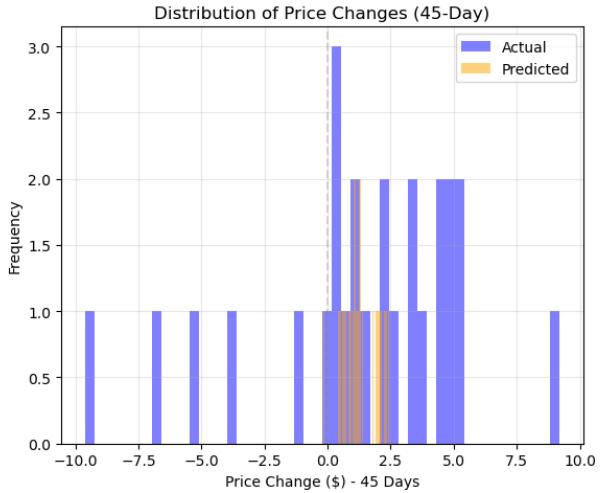
```
======================================================================
DIRECTIONAL ACCURACY ANALYSIS (45-DAY HORIZON)
======================================================================

1. OVERALL DIRECTION (Day 1 → Day 45):
----------------------------------------------------------------------
Accuracy: 80.77%
Correct: 21/26

Sample changes (first 3 windows):
  Window 0: Actual=$41.43, Pred=$68.83, Match=✓
  Window 1: Actual=$390.66, Pred=$81.74, Match=✓
  Window 2: Actual=$180.78, Pred=$54.28, Match=✓

2. WEEKLY DIRECTIONS (5-day intervals):
----------------------------------------------------------------------
Week 1 (Day 5): 42.31%
Week 2 (Day 10): 53.85%
Week 3 (Day 15): 61.54%
Week 4 (Day 20): 61.54%
Week 5 (Day 25): 69.23%
Week 6 (Day 30): 73.08%
Week 7 (Day 35): 65.38%
Week 8 (Day 40): 76.92%
Week 9 (Day 45): 80.77%

Average weekly accuracy: 64.96%

3. DAY-TO-DAY DIRECTIONS (summary by week):
----------------------------------------------------------------------
Week 1 avg (days 1-5): 50.00%
Week 2 avg (days 6-10): 60.58%
Week 3 avg (days 11-15): 38.46%
Week 4 avg (days 16-20): 45.19%
Week 5 avg (days 21-25): 60.58%
Week 6 avg (days 26-30): 50.96%
Week 7 avg (days 31-35): 47.12%
Week 8 avg (days 36-40): 49.04%
Week 9 avg (days 41-45): 50.00%

4. MAGNITUDE ANALYSIS (45-Day Cumulative):
----------------------------------------------------------------------
Avg actual 45-day change (when correct):    $240.27
Avg predicted 45-day change (when correct): $108.06
Ratio (pred/actual): 0.45

5. DETAILED BREAKDOWN (Overall 45-Day Direction):
----------------------------------------------------------------------
True Positives (predicted UP, was UP):     20
```

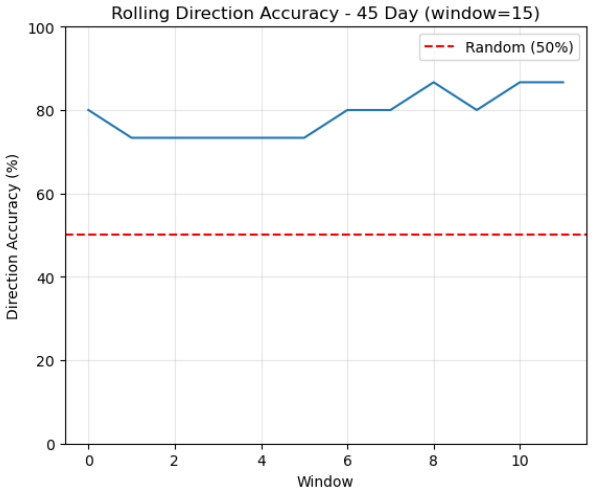
```
True Negatives (predicted DOWN, was DOWN): 1
False Positives (predicted UP, was DOWN):  5
False Negatives (predicted DOWN, was UP):  0

Precision (UP predictions): 80.00%
Recall (catching UP moves): 100.00%

6. DIRECTIONAL ACCURACY OVER TIME:
----------------------------------------------------------------------
Quarter 1 (windows   0-  6): 83.33%
Quarter 2 (windows   6- 12): 66.67%
Quarter 3 (windows  12- 18): 83.33%
Quarter 4 (windows  18- 26): 87.50%

======================================================================
SUMMARY:
======================================================================
Overall 45-Day Direction Accuracy: 80.77%
 EXCELLENT - Model captures 45-day trends very well!
```

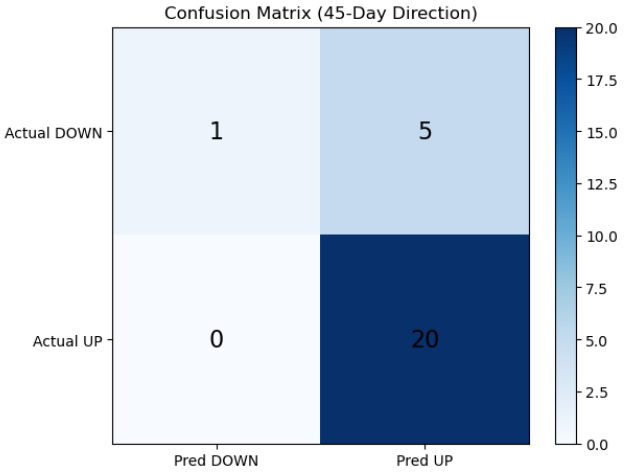
In [ ]:
```
print(f"Actual changes range: [{actual_changes.min():.2f}, {actual_changes.max():.2f}]")
print(f"Pred changes range: [{pred_changes.min():.2f}, {pred_changes.max():.2f}]")
```

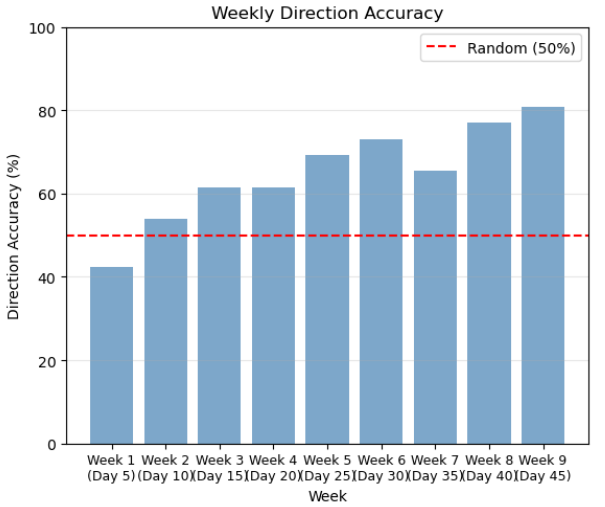
In [47]:
```
# 11. Plot results
#plot_direction_analysis(actuals_prices, preds_prices, target_scaler)
plot_direction_analysis_CORR(actuals_prices, preds_prices)
```

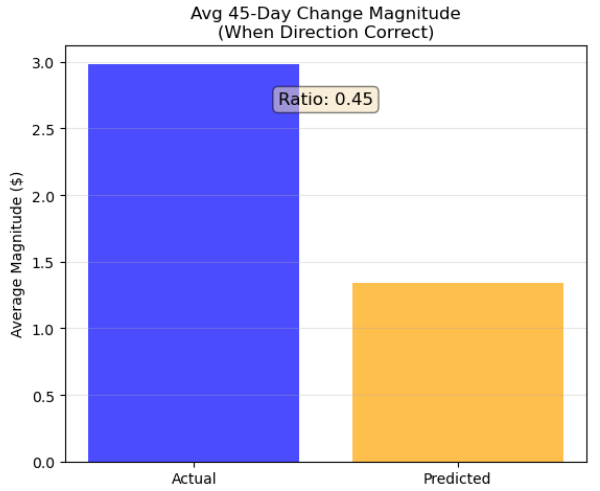












In [3]:
```
print("\n" + "="*70)
print("PIPELINE COMPLETE!")
print("="*70)
```

```
======================================================================
PIPELINE COMPLETE!
======================================================================
```

In [ ]:
```
##################################################################################################################################################
```

In [ ]:
```

```

In [ ]:

In [ ]:

In [ ]:

In [ ]: