NHL Stanley Cup Prediction Pipeline

16 Seasons | December 2025

```
import pandas as pd
import numpy as np
from pathlib import Path
import glob
import warnings
warnings.filterwarnings('ignore')

from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.calibration import calibration_curve
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from catboost import CatBoostClassifier
import matplotlib.pyplot as plt


RAW_DATA_DIR = Path("./NHL_Data")
FEATURES_DIR = Path("./NHL_Features")
MODELS_DIR = Path("./NHL_Models_Goal1")
MODELS_GOAL2_DIR = Path("./NHL_Models_Goal2")

for d in [FEATURES_DIR, MODELS_DIR, MODELS_GOAL2_DIR]:
    d.mkdir(exist_ok=True)

RANDOM_STATE = 42
np.random.seed(RANDOM_STATE)

SEASONS = [
    '2007-2008', '2008-2009', '2009-2010', '2010-2011', '2011-2012',
    '2013-2014', '2014-2015', '2015-2016', '2016-2017', '2017-2018',
    '2018-2019', '2020-2021', '2021-2022', '2022-2023', '2023-2024', '2024-2025'
]

SEASON_PAIRS = [
    ('2007-2008', '2008-2009'),
    ('2008-2009', '2009-2010'),
    ('2009-2010', '2010-2011'),
    ('2010-2011', '2011-2012'),
    ('2011-2012', '2013-2014'),
    ('2013-2014', '2014-2015'),
    ('2014-2015', '2015-2016'),
    ('2015-2016', '2016-2017'),
    ('2016-2017', '2017-2018'),
    ('2017-2018', '2018-2019'),
    ('2018-2019', '2020-2021'),
    ('2020-2021', '2021-2022'),
    ('2021-2022', '2022-2023'),
    ('2022-2023', '2023-2024'),
]

FINAL_PREDICTION_PAIR = ('2023-2024', '2024-2025')

CUP_WINNERS_ABBR = {
    '2007-2008': 'DET', '2008-2009': 'PIT', '2009-2010': 'CHI', '2010-2011': 'BOS',
    '2011-2012': 'LAK', '2013-2014': 'LAK', '2014-2015': 'CHI', '2015-2016': 'PIT',
    '2016-2017': 'PIT', '2017-2018': 'WSH', '2018-2019': 'STL', '2020-2021': 'TBL',
    '2021-2022': 'COL', '2022-2023': 'VGK', '2023-2024': 'FLA', '2024-2025': 'FLA'
}

MIN_GP_SKATERS_REG = 10
MIN_GP_SKATERS_PLAYOFFS = 3
MIN_GP_GOALIES_REG = 5
MIN_GP_GOALIES_PLAYOFFS = 2

CLEAR_STARTER_THRESHOLD = 0.65
TANDEM_LOWER = 0.45

ABBR_TO_FULL = {
    "ANA": "Anaheim Ducks", "ARI": "Arizona Coyotes", "PHX": "Phoenix Coyotes",
    "ATL": "Atlanta Thrashers", "BOS": "Boston Bruins", "BUF": "Buffalo Sabres",
    "CGY": "Calgary Flames", "CAR": "Carolina Hurricanes", "CHI": "Chicago Blackhawks",
    "COL": "Colorado Avalanche", "CBJ": "Columbus Blue Jackets", "DAL": "Dallas Stars",
    "DET": "Detroit Red Wings", "EDM": "Edmonton Oilers", "FLA": "Florida Panthers",
```

```
    "L.A": "Los Angeles Kings", "LAK": "Los Angeles Kings", "MIN": "Minnesota Wild",
    "MTL": "Montreal Canadiens", "NSH": "Nashville Predators", "N.J": "New Jersey Devils",
    "NJD": "New Jersey Devils", "NYI": "New York Islanders", "NYR": "New York Rangers",
    "OTT": "Ottawa Senators", "PHI": "Philadelphia Flyers", "PIT": "Pittsburgh Penguins",

    "S.J": "San Jose Sharks", "SJS": "San Jose Sharks", "SEA": "Seattle Kraken",
    "STL": "St Louis Blues", "T.B": "Tampa Bay Lightning", "TBL": "Tampa Bay Lightning",
    "TOR": "Toronto Maple Leafs", "UTA": "Utah Hockey Club", "VAN": "Vancouver Canucks",
    "VGK": "Vegas Golden Knights", "WSH": "Washington Capitals", "WPG": "Winnipeg Jets"
}

FULL_TO_ABBR = {
    "Anaheim Ducks": "ANA", "Arizona Coyotes": "ARI", "Phoenix Coyotes": "PHX",
    "Atlanta Thrashers": "ATL", "Boston Bruins": "BOS", "Buffalo Sabres": "BUF",
    "Calgary Flames": "CGY", "Carolina Hurricanes": "CAR", "Chicago Blackhawks": "CHI",
    "Colorado Avalanche": "COL", "Columbus Blue Jackets": "CBJ", "Dallas Stars": "DAL",
    "Detroit Red Wings": "DET", "Edmonton Oilers": "EDM", "Florida Panthers": "FLA",
    "Los Angeles Kings": "LAK", "Minnesota Wild": "MIN", "Montreal Canadiens": "MTL",
    "Nashville Predators": "NSH", "New Jersey Devils": "NJD", "New York Islanders": "NYI",
    "New York Rangers": "NYR", "Ottawa Senators": "OTT", "Philadelphia Flyers": "PHI",
    "Pittsburgh Penguins": "PIT", "San Jose Sharks": "SJS", "Seattle Kraken": "SEA",
    "St Louis Blues": "STL", "Tampa Bay Lightning": "TBL", "Toronto Maple Leafs": "TOR",
    "Utah Hockey Club": "UTA", "Vancouver Canucks": "VAN", "Vegas Golden Knights": "VGK",
    "Washington Capitals": "WSH", "Winnipeg Jets": "WPG"
}


def standardize_season(season_str):

    if pd.isna(season_str):
        return None
    s = str(season_str).strip()
    if ' - ' in s:
        s = s.replace(' - ', '-')
    return s


def standardize_team_abbr(team_str):

    if pd.isna(team_str):
        return None
    t = str(team_str).strip()
    if ',' in t:
        teams = [x.strip() for x in t.split(',')]
        t = teams[-1]
    replacements = {'L.A': 'LAK', 'S.J': 'SJS', 'N.J': 'NJD', 'T.B': 'TBL'}
    return replacements.get(t, t)


def get_full_team_name(abbr):

    return ABBR_TO_FULL.get(abbr, abbr)


def load_all_data():

    print("=" * 70)
    print("LOADING DATA (16 SEASONS)")
    print("=" * 70)

    all_data = {
        'team_reg': [], 'team_playoffs': [],
        'skaters_reg': [], 'skaters_playoffs': [],
        'goalies_reg': [], 'goalies_playoffs': []
    }

    for season in SEASONS:
        print(f"  Loading {season}...")

        team_reg = pd.read_csv(RAW_DATA_DIR / 'regular_season' / f'{season}_team_reg.csv')
        skaters_reg = pd.read_csv(RAW_DATA_DIR / 'regular_season' / f'{season}_skaters_reg.csv')
        goalies_reg = pd.read_csv(RAW_DATA_DIR / 'regular_season' / f'{season}_goalies_reg.csv')

        team_playoffs = pd.read_csv(RAW_DATA_DIR / 'playoffs' / f'{season}_team_playoffs.csv')
        skaters_playoffs = pd.read_csv(RAW_DATA_DIR / 'playoffs' / f'{season}_skaters_playoffs.csv')
        goalies_playoffs = pd.read_csv(RAW_DATA_DIR / 'playoffs' / f'{season}_goalies_playoffs.csv')

        for df in [team_reg, skaters_reg, goalies_reg]:
            df['Season'] = season
        for df in [team_playoffs, skaters_playoffs, goalies_playoffs]:
            df['Season'] = season

        all_data['team_reg'].append(team_reg)
        all_data['team_playoffs'].append(team_playoffs)
```

```
        all_data['skaters_reg'].append(skaters_reg)

        all_data['skaters_playoffs'].append(skaters_playoffs)
        all_data['goalies_reg'].append(goalies_reg)
        all_data['goalies_playoffs'].append(goalies_playoffs)

    data = {}
    for key in all_data:
        data[key] = pd.concat(all_data[key], ignore_index=True)
        print(f"  {key}: {len(data[key])} rows")

    playoff_results = pd.read_csv(RAW_DATA_DIR / 'playoff_results.csv')
    playoff_results['Season'] = playoff_results['Season'].apply(standardize_season)
    playoff_results['Team'] = playoff_results['Team'].apply(standardize_team_abbr)
    data['playoff_results'] = playoff_results

    return data


def clean_column_names(df):

    new_cols = {}
    for col in df.columns:
        new_col = col.replace('%', 'Pct').replace('/', '_').replace(' ', '_')
        new_col = new_col.replace('[', '').replace(']', '').replace('{', '').replace('}', '')
        new_cols[col] = new_col
    return df.rename(columns=new_cols)


def clean_and_standardize(data):

    print("\n" + "=" * 70)
    print("CLEANING &amp;amp; STANDARDIZING DATA")
    print("=" * 70)

    for key in data:
        if isinstance(data[key], pd.DataFrame):
            data[key] = clean_column_names(data[key])

    for key in ['team_reg', 'team_playoffs']:
        if 'Team' in data[key].columns:
            sample = data[key]['Team'].iloc[0]
            if len(sample) &gt; 5:
                data[key]['Team_Full'] = data[key]['Team']
                data[key]['Team_Abbr'] = data[key]['Team'].apply(lambda x: FULL_TO_ABBR.get(x, x))
            else:
                data[key]['Team_Abbr'] = data[key]['Team'].apply(standardize_team_abbr)
                data[key]['Team_Full'] = data[key]['Team_Abbr'].apply(get_full_team_name)

    for key in ['skaters_reg', 'skaters_playoffs', 'goalies_reg', 'goalies_playoffs']:
        data[key]['Team'] = data[key]['Team'].apply(standardize_team_abbr)

    column_renames = {
        'Total Points': 'Points',
        'Total Assists': 'Assists',
        'Faceoffs %': 'FOW%'
    }

    for key in ['skaters_reg', 'skaters_playoffs']:
        data[key] = data[key].rename(columns=column_renames)

    print("\nApplying GP filters...")
    before = len(data['skaters_reg'])
    data['skaters_reg'] = data['skaters_reg'][data['skaters_reg']['GP'] &gt;= MIN_GP_SKATERS_REG]
    print(f"  skaters_reg: {before} → {len(data['skaters_reg'])}")

    before = len(data['skaters_playoffs'])
    data['skaters_playoffs'] = data['skaters_playoffs'][data['skaters_playoffs']['GP'] &gt;= MIN_GP_SKATERS_PLAYOFFS]
    print(f"  skaters_playoffs: {before} → {len(data['skaters_playoffs'])}")

    before = len(data['goalies_reg'])
    data['goalies_reg'] = data['goalies_reg'][data['goalies_reg']['GP'] &gt;= MIN_GP_GOALIES_REG]
    print(f"  goalies_reg: {before} → {len(data['goalies_reg'])}")

    before = len(data['goalies_playoffs'])
    data['goalies_playoffs'] = data['goalies_playoffs'][data['goalies_playoffs']['GP'] &gt;= MIN_GP_GOALIES_PLAYOFFS]
    print(f"  goalies_playoffs: {before} → {len(data['goalies_playoffs'])}")

    print("\nMerging playoff results...")
    data['team_reg'] = data['team_reg'].merge(
        data['playoff_results'][['Season', 'Team', 'Round_Reached']],
        left_on=['Season', 'Team_Abbr'],

        right_on=['Season', 'Team'],
```

```
        how='left',
        suffixes=('', '_result')
    )
    data['team_reg']['Round_Reached'] = data['team_reg']['Round_Reached'].fillna(0).astype(int)

    print("\nCup winners verification:")
    for season in SEASONS:
        winners = data['team_reg'][(data['team_reg']['Season'] == season) &amp;
                                   (data['team_reg']['Round_Reached'] == 5)]
        if len(winners) &gt; 0:
            print(f"  {season}: {winners['Team_Abbr'].values[0]}")
        else:
            print(f"  {season}: NOT FOUND!")

    return data


def aggregate_skater_features(skaters_df, team_abbr, season):

    team_skaters = skaters_df[(skaters_df['Team'] == team_abbr) &amp;
                              (skaters_df['Season'] == season)]

    if len(team_skaters) == 0:
        return None

    forwards = team_skaters[team_skaters['Position'] != 'D']
    defensemen = team_skaters[team_skaters['Position'] == 'D']

    points_col = 'Total_Points' if 'Total_Points' in team_skaters.columns else 'Points'

    forwards_sorted = forwards.sort_values(points_col, ascending=False)
    defensemen_sorted = defensemen.sort_values(points_col, ascending=False)

    top6_fwd = forwards_sorted.head(6)
    bottom6_fwd = forwards_sorted.iloc[6:12] if len(forwards_sorted) &gt; 6 else pd.DataFrame()
    top4_def = defensemen_sorted.head(4)

    def safe_sum(df, col):
        return df[col].sum() if col in df.columns and len(df) &gt; 0 else 0

    features = {
        'top6_fwd_points': safe_sum(top6_fwd, points_col),
        'top6_fwd_goals': safe_sum(top6_fwd, 'Goals'),
        'bottom6_fwd_points': safe_sum(bottom6_fwd, points_col),
        'top4_def_points': safe_sum(top4_def, points_col),
        'top4_def_goals': safe_sum(top4_def, 'Goals'),

        'scoring_depth_ratio': (safe_sum(bottom6_fwd, points_col) / safe_sum(top6_fwd, points_col))
                               if safe_sum(top6_fwd, points_col) &gt; 0 else 0,
        'scoring_gini': team_skaters[points_col].std() / team_skaters[points_col].mean()
                        if team_skaters[points_col].mean() &gt; 0 else 0,

        'team_ixG': safe_sum(team_skaters, 'ixG'),

        'team_hits': safe_sum(team_skaters, 'Hits'),
        'team_hits_taken': safe_sum(team_skaters, 'Hits_Taken'),
        'net_hits': (safe_sum(team_skaters, 'Hits') - safe_sum(team_skaters, 'Hits_Taken')),

        'team_pim': safe_sum(team_skaters, 'Total_Penalties'),
        'team_penalties_drawn': safe_sum(team_skaters, 'Penalties_Drawn'),

        'team_iHDCF': safe_sum(team_skaters, 'iHDCF'),
        'team_iSCF': safe_sum(team_skaters, 'iSCF'),

        'total_skater_TOI': safe_sum(team_skaters, 'TOI'),
        'avg_games_played': team_skaters['GP'].mean(),
        'roster_size': len(team_skaters)
    }

    return features


def aggregate_goalie_features(goalies_df, team_abbr, season):

    team_goalies = goalies_df[(goalies_df['Team'] == team_abbr) &amp;
                              (goalies_df['Season'] == season)]

    if len(team_goalies) == 0:
        return None

    team_goalies_sorted = team_goalies.sort_values('TOI', ascending=False)
    starter = team_goalies_sorted.iloc[0]
    backup = team_goalies_sorted.iloc[1] if len(team_goalies_sorted) &gt; 1 else None
    total_toi = team_goalies['TOI'].sum()
```

```
    def safe_get(row, col):
        return row[col] if col in row and pd.notna(row[col]) else 0

    def safe_sum(df, col):
        return df[col].sum() if col in df.columns else 0

    features = {
        'starter_GP': starter['GP'],
        'starter_TOI': starter['TOI'],
        'starter_SVPct': safe_get(starter, 'SVPct'),
        'starter_GAA': safe_get(starter, 'GAA'),
        'starter_GSAA': safe_get(starter, 'GSAA'),
        'starter_HDSVPct': safe_get(starter, 'HDSVPct'),
        'starter_HDGSAA': safe_get(starter, 'HDGSAA'),
        'starter_workload_share': starter['TOI'] / total_toi if total_toi &gt; 0 else 1,

        'backup_GP': backup['GP'] if backup is not None else 0,
        'backup_GSAA': safe_get(backup, 'GSAA') if backup is not None else 0,

        'team_total_GSAA': safe_sum(team_goalies, 'GSAA'),
        'team_total_HDGSAA': safe_sum(team_goalies, 'HDGSAA'),
        'num_goalies_used': len(team_goalies),
        'total_goalie_TOI': total_toi
    }

    return features


def create_team_derived_features(team_reg):

    df = team_reg.copy()

    df['goal_differential'] = df['GF'] - df['GA']

    if 'xGF' in df.columns:
        df['xG_differential'] = df['xGF'] - df['xGA']
        df['goals_above_expected'] = df['GF'] - df['xGF']

    pp_col = 'PPPct' if 'PPPct' in df.columns else ('PP%' if 'PP%' in df.columns else None)
    pk_col = 'PKPct' if 'PKPct' in df.columns else ('PK%' if 'PK%' in df.columns else None)
    if pp_col and pk_col:
        df['special_teams_index'] = df[pp_col] + df[pk_col]

    if 'HDCF' in df.columns and 'HDCA' in df.columns:
        df['HD_chance_differential'] = df['HDCF'] - df['HDCA']
    if 'HDGF' in df.columns and 'HDGA' in df.columns:
        df['HD_goal_differential'] = df['HDGF'] - df['HDGA']

    df['win_pct'] = df['W'] / df['GP']
    if 'RW' in df.columns:
        df['regulation_win_pct'] = df['RW'] / df['GP']

    return df


def build_goal1_features(data):

    print("\n" + "=" * 70)
    print("BUILDING GOAL 1 FEATURES")
    print("=" * 70)

    team_reg = data['team_reg']
    skaters_reg = data['skaters_reg']
    goalies_reg = data['goalies_reg']

    team_reg = create_team_derived_features(team_reg)

    print("\nAggregating skater features...")
    skater_aggs = []
    for _, row in team_reg.iterrows():
        features = aggregate_skater_features(skaters_reg, row['Team_Abbr'], row['Season'])
        if features:
            features['Team_Abbr'] = row['Team_Abbr']

            features['Season'] = row['Season']
            skater_aggs.append(features)
    skater_agg_df = pd.DataFrame(skater_aggs)
    print(f"  Created {len(skater_agg_df)} skater aggregations")

    print("\nAggregating goalie features...")
    goalie_aggs = []
    for _, row in team_reg.iterrows():
        features = aggregate_goalie_features(goalies_reg, row['Team_Abbr'], row['Season'])
        if features:
```

```
            features['Team_Abbr'] = row['Team_Abbr']
            features['Season'] = row['Season']
            goalie_aggs.append(features)
    goalie_agg_df = pd.DataFrame(goalie_aggs)
    print(f"  Created {len(goalie_agg_df)} goalie aggregations")

    print("\nMerging features...")
    modeling_df = team_reg.merge(skater_agg_df, on=['Team_Abbr', 'Season'], how='left')
    modeling_df = modeling_df.merge(goalie_agg_df, on=['Team_Abbr', 'Season'], how='left')

    numeric_cols = modeling_df.select_dtypes(include=[np.number]).columns
    for col in numeric_cols:
        if modeling_df[col].isna().any():
            modeling_df[col] = modeling_df[col].fillna(modeling_df[col].median())

    print(f"\nGoal 1 feature dataset: {modeling_df.shape}")

    modeling_df.to_csv(FEATURES_DIR / 'goal1_features.csv', index=False)

    return modeling_df


def calculate_toi_weighted_continuity(team_abbr, current_season, prior_season,
                                      skaters_reg, top6_fwd_prior, top4_def_prior):

    prior_roster = skaters_reg[(skaters_reg['Team'] == team_abbr) &amp;
                               (skaters_reg['Season'] == prior_season)]

    current_roster = skaters_reg[(skaters_reg['Team'] == team_abbr) &amp;
                                 (skaters_reg['Season'] == current_season)]

    if len(prior_roster) == 0:
        return {
            'toi_pct_returning': np.nan,
            'top6_fwd_toi_pct_returning': np.nan,
            'top4_def_toi_pct_returning': np.nan
        }

    points_col = 'Total_Points' if 'Total_Points' in prior_roster.columns else 'Points'

    prior_players = set(prior_roster['Player'].unique())
    current_players = set(current_roster['Player'].unique())
    returning_players = prior_players.intersection(current_players)

    total_prior_toi = prior_roster['TOI'].sum()
    returning_toi = prior_roster[prior_roster['Player'].isin(returning_players)]['TOI'].sum()
    toi_pct_returning = returning_toi / total_prior_toi if total_prior_toi &gt; 0 else 0

    prior_forwards = prior_roster[prior_roster['Position'] != 'D'].sort_values(points_col, ascending=False)
    top6_fwd = prior_forwards.head(6)
    top6_fwd_toi = top6_fwd['TOI'].sum()
    top6_fwd_returning = top6_fwd[top6_fwd['Player'].isin(returning_players)]
    top6_fwd_toi_returning = top6_fwd_returning['TOI'].sum()
    top6_fwd_toi_pct = top6_fwd_toi_returning / top6_fwd_toi if top6_fwd_toi &gt; 0 else 0

    prior_defense = prior_roster[prior_roster['Position'] == 'D'].sort_values(points_col, ascending=False)
    top4_def = prior_defense.head(4)
    top4_def_toi = top4_def['TOI'].sum()
    top4_def_returning = top4_def[top4_def['Player'].isin(returning_players)]
    top4_def_toi_returning = top4_def_returning['TOI'].sum()
    top4_def_toi_pct = top4_def_toi_returning / top4_def_toi if top4_def_toi &gt; 0 else 0

    return {
        'toi_pct_returning': toi_pct_returning,
        'top6_fwd_toi_pct_returning': top6_fwd_toi_pct,
        'top4_def_toi_pct_returning': top4_def_toi_pct
    }


def calculate_goalie_stability(team_abbr, current_season, prior_season, goalies_reg):

    prior_goalies = goalies_reg[(goalies_reg['Team'] == team_abbr) &amp;
                                (goalies_reg['Season'] == prior_season)]
    current_goalies = goalies_reg[(goalies_reg['Team'] == team_abbr) &amp;
                                  (goalies_reg['Season'] == current_season)]

    if len(prior_goalies) == 0 or len(current_goalies) == 0:
        return {
            'starter_returning': np.nan,
            'goalie_stability_cat': np.nan,
            'goalie_toi_pct_returning': np.nan
        }

    prior_starter = prior_goalies.sort_values('TOI', ascending=False).iloc[0]
```

```
    prior_starter_name = prior_starter['Player']
    prior_total_toi = prior_goalies['TOI'].sum()
    prior_starter_share = prior_starter['TOI'] / prior_total_toi if prior_total_toi &gt; 0 else 0

    current_starter = current_goalies.sort_values('TOI', ascending=False).iloc[0]
    current_starter_name = current_starter['Player']
    current_total_toi = current_goalies['TOI'].sum()
    current_starter_share = current_starter['TOI'] / current_total_toi if current_total_toi &gt; 0 else 0

    starter_returning = 1 if prior_starter_name == current_starter_name else 0

    prior_goalie_names = set(prior_goalies['Player'].unique())
    current_goalie_names = set(current_goalies['Player'].unique())
    returning_goalies = prior_goalie_names.intersection(current_goalie_names)

    returning_toi = prior_goalies[prior_goalies['Player'].isin(returning_goalies)]['TOI'].sum()
    goalie_toi_pct_returning = returning_toi / prior_total_toi if prior_total_toi &gt; 0 else 0


    if starter_returning and prior_starter_share &gt;= CLEAR_STARTER_THRESHOLD:
        goalie_stability_cat = 2
    elif goalie_toi_pct_returning &gt;= TANDEM_LOWER:
        goalie_stability_cat = 1
    else:
        goalie_stability_cat = 0

    return {
        'starter_returning': starter_returning,
        'goalie_stability_cat': goalie_stability_cat,
        'goalie_toi_pct_returning': goalie_toi_pct_returning
    }


def build_playoff_experience_safe(team_abbr, feature_season, skaters_reg, skaters_playoffs,
                                  goalies_playoffs, all_seasons, cup_finals_teams, cup_winners):

    roster = skaters_reg[(skaters_reg['Team'] == team_abbr) &amp;amp;
                         (skaters_reg['Season'] == feature_season)]

    if len(roster) == 0:
        return {
            'roster_total_playoff_gp': 0,
            'roster_avg_playoff_gp': 0,
            'roster_playoff_veterans': 0,
            'roster_cup_finals_exp': 0,
            'roster_cup_winners': 0
        }

    roster_players = set(roster['Player'].unique())

    season_idx = all_seasons.index(feature_season)
    prior_seasons = all_seasons[:season_idx]

    prior_skater_playoffs = skaters_playoffs[skaters_playoffs['Season'].isin(prior_seasons)]
    prior_goalie_playoffs = goalies_playoffs[goalies_playoffs['Season'].isin(prior_seasons)]

    total_playoff_gp = 0
    playoff_veterans = 0
    cup_finals_exp = 0
    cup_winners_count = 0

    for player in roster_players:
        player_playoffs = prior_skater_playoffs[prior_skater_playoffs['Player'] == player]

        if len(player_playoffs) &gt; 0:
            player_gp = player_playoffs['GP'].sum()

            total_playoff_gp += player_gp
            playoff_veterans += 1

            for _, row in player_playoffs.iterrows():
                playoff_season = row['Season']
                playoff_team = row['Team']

                if playoff_team in cup_finals_teams.get(playoff_season, set()):
                    cup_finals_exp += 1
                    break

            for _, row in player_playoffs.iterrows():
                playoff_season = row['Season']
                playoff_team = row['Team']

                if playoff_team == cup_winners.get(playoff_season):
                    cup_winners_count += 1
```

```
                    break

        for player in roster_players:
            player_goalie_playoffs = prior_goalie_playoffs[prior_goalie_playoffs['Player'] == player]
            if len(player_goalie_playoffs) &gt; 0:
                pass

        roster_size = len(roster_players)

        return {
            'roster_total_playoff_gp': total_playoff_gp,
            'roster_avg_playoff_gp': total_playoff_gp / roster_size if roster_size &gt; 0 else 0,
            'roster_playoff_veterans': playoff_veterans,
            'roster_cup_finals_exp': cup_finals_exp,
            'roster_cup_winners': cup_winners_count
        }


    def calculate_yoy_deltas(team_abbr, current_season, prior_season, team_reg):

        current = team_reg[(team_reg['Team_Abbr'] == team_abbr) &amp;amp; (team_reg['Season'] == current_season)]
        prior = team_reg[(team_reg['Team_Abbr'] == team_abbr) &amp;amp; (team_reg['Season'] == prior_season)]

        if len(current) == 0 or len(prior) == 0:
            return {
                'delta_point_pct': np.nan,
                'delta_gf_pct': np.nan,
                'delta_goal_diff': np.nan
            }

        curr, prev = current.iloc[0], prior.iloc[0]

        point_col = 'Point_Pct' if 'Point_Pct' in curr else 'Point %'
        gf_col = 'GFPct' if 'GFPct' in curr else 'GF%'

        return {
            'delta_point_pct': curr.get(point_col, 0) - prev.get(point_col, 0),
            'delta_gf_pct': curr.get(gf_col, 0) - prev.get(gf_col, 0) if gf_col in curr else np.nan,
            'delta_goal_diff': (curr['GF'] - curr['GA']) - (prev['GF'] - prev['GA'])
        }


    def build_goal2_features(data, goal1_features):

        print("\n" + "=" * 70)
        print("BUILDING GOAL 2 FEATURES (PROBABILITY-BASED)")
        print("=" * 70)

        team_reg = data['team_reg']
        skaters_reg = data['skaters_reg']
        skaters_playoffs = data['skaters_playoffs']
        goalies_reg = data['goalies_reg']
        goalies_playoffs = data['goalies_playoffs']
        team_playoffs = data['team_playoffs']

        cup_finals_teams = {}
        cup_winners = {}

        for season in SEASONS:
            season_teams = team_reg[team_reg['Season'] == season]
            finals_teams = season_teams[season_teams['Round_Reached'] &gt;= 4]['Team_Abbr'].tolist()
            cup_finals_teams[season] = set(finals_teams)
            winner = season_teams[season_teams['Round_Reached'] == 5]['Team_Abbr'].values

            cup_winners[season] = winner[0] if len(winner) &gt; 0 else None

        print(f"\nCup winners: {cup_winners}")

        all_pairs = SEASON_PAIRS + [FINAL_PREDICTION_PAIR]

        rows = []
        for feature_season, target_season in all_pairs:
            print(f"\n  Processing: {feature_season} → {target_season}")

            feature_idx = SEASONS.index(feature_season)
            prior_season = SEASONS[feature_idx - 1] if feature_idx &gt; 0 else None

            feature_teams = set(team_reg[team_reg['Season'] == feature_season]['Team_Abbr'].unique())
            target_teams = set(team_reg[team_reg['Season'] == target_season]['Team_Abbr'].unique())
            common_teams = feature_teams.intersection(target_teams)

            for team_abbr in common_teams:
                g1_row = goal1_features[(goal1_features['Team_Abbr'] == team_abbr) &amp;amp;
                                        (goal1_features['Season'] == feature_season)]
```

```
            if len(g1_row) == 0:
                continue
            g1_row = g1_row.iloc[0].to_dict()

            target_row = team_reg[(team_reg['Team_Abbr'] == team_abbr) &amp;amp;
                                  (team_reg['Season'] == target_season)]
            if len(target_row) == 0:
                continue
            target_round = target_row.iloc[0]['Round_Reached']

            is_cup_winner = 1 if target_round == 5 else 0


            continuity = calculate_toi_weighted_continuity(
                team_abbr, target_season, feature_season, skaters_reg, None, None
            )

            goalie_stability = calculate_goalie_stability(
                team_abbr, target_season, feature_season, goalies_reg
            )

            playoff_exp = build_playoff_experience_safe(
                team_abbr, feature_season, skaters_reg, skaters_playoffs,
                goalies_playoffs, SEASONS, cup_finals_teams, cup_winners
            )

            prior_round = g1_row.get('Round_Reached', 0)

            if prior_season:
                yoy = calculate_yoy_deltas(team_abbr, feature_season, prior_season, team_reg)
            else:
                yoy = {'delta_point_pct': np.nan, 'delta_gf_pct': np.nan, 'delta_goal_diff': np.nan}

            row = {
                'Team_Abbr': team_abbr,
                'Team_Full': get_full_team_name(team_abbr),
                'Feature_Season': feature_season,
                'Target_Season': target_season,
                'Target_Round_Reached': target_round,
                'Is_Cup_Winner': is_cup_winner,
                'Prior_Round_Reached': prior_round
            }

            exclude_cols = ['Team', 'Team_Full', 'Team_Abbr', 'Season', 'Round_Reached',
                            'Team_result', 'Season_End_Year']
            for col, val in g1_row.items():
                if col not in exclude_cols and not isinstance(val, str):
                    row[f'reg_{col}'] = val

            row.update(continuity)
            row.update(goalie_stability)
            row.update(playoff_exp)
            row.update(yoy)

            rows.append(row)

    goal2_df = pd.DataFrame(rows)
    print(f"\nGoal 2 feature dataset: {goal2_df.shape}")
    print(f"  Season pairs: {len(all_pairs)}")
    print(f"  Teams per pair: ~{len(goal2_df) / len(all_pairs):.0f}")

    goal2_df.to_csv(FEATURES_DIR / 'goal2_features.csv', index=False)

    return goal2_df


def get_models():

    return {
        'XGBoost': XGBClassifier(
            n_estimators=100, max_depth=4, learning_rate=0.1,
            subsample=0.8, colsample_bytree=0.8,
            random_state=RANDOM_STATE, verbosity=0
        ),
        'LightGBM': LGBMClassifier(
            n_estimators=100, max_depth=4, learning_rate=0.1,
            subsample=0.8, colsample_bytree=0.8,
            random_state=RANDOM_STATE, verbosity=-1
        ),
        'CatBoost': CatBoostClassifier(
            iterations=100, depth=4, learning_rate=0.1,
            random_state=RANDOM_STATE, verbose=False
        ),
        'RandomForest': RandomForestClassifier(
```

```
            n_estimators=100, max_depth=6, min_samples_leaf=3,
            random_state=RANDOM_STATE, class_weight='balanced'
        )
    }


def softmax_normalize(probs):

    exp_probs = np.exp(probs - np.max(probs))
    return exp_probs / exp_probs.sum()


def calculate_log_loss_single(true_prob, pred_prob, eps=1e-15):

    pred_prob = np.clip(pred_prob, eps, 1 - eps)
    return -np.log(pred_prob)


def run_goal1_model(goal1_features):

    print("\n" + "=" * 70)
    print("GOAL 1: SAME-SEASON PREDICTION (16 SEASONS)")
    print("=" * 70)

    df = goal1_features.copy()

    exclude_cols = ['Team', 'Team_Full', 'Team_Abbr', 'Season', 'Round_Reached',
                    'Team_result', 'Season_End_Year']
    feature_cols = [c for c in df.columns if c not in exclude_cols
                    and df[c].dtype in ['int64', 'float64']]

    def clean_feature_name(name):
        return name.replace('%', 'Pct').replace('/', '_').replace(' ', '_').replace('[', '').replace(']', '').replace('{', '').replace('}', '')

    clean_feature_cols = [clean_feature_name(c) for c in feature_cols]

    X = df[feature_cols].fillna(df[feature_cols].median())
    X.columns = clean_feature_cols
    y = df['Round_Reached']

    print(f"\nFeatures: {len(feature_cols)}")
    print(f"Samples: {len(X)}")

    models = get_models()
    scaler = StandardScaler()

    print("\nLeave-One-Season-Out Cross-Validation:")
    results = []

    for test_season in SEASONS:
        train_mask = df['Season'] != test_season
        test_mask = df['Season'] == test_season

        X_train, y_train = X[train_mask], y[train_mask]
        X_test, y_test = X[test_mask], y[test_mask]
        test_teams = df[test_mask]['Team_Abbr'].values

        actual_winner = CUP_WINNERS_ABBR.get(test_season)

        test_df = df[test_mask].copy()
        point_col = None
        for col in ['Point_Pct', 'Point Pct', 'PointPct']:
            if col in test_df.columns:
                point_col = col
                break
        if point_col is None:
            for col in test_df.columns:
                if 'Point' in col and ('Pct' in col or '%' in col):
                    point_col = col
                    break
        if point_col is None:
            point_col = 'Points'
        baseline_sorted = test_df.sort_values(point_col, ascending=False)
        baseline_rank = list(baseline_sorted['Team_Abbr']).index(actual_winner) + 1 \
                        if actual_winner in baseline_sorted['Team_Abbr'].values else 999

        results.append({
            'season': test_season, 'model': 'Baseline',
            'winner_rank': baseline_rank, 'top5_hit': baseline_rank &lt;= 5
        })

        for model_name, model in models.items():
            try:
                model.fit(X_train, y_train)
```

```
                probs = model.predict_proba(X_test)
                classes = model.classes_

                if 5 in classes:
                    cup_idx = list(classes).index(5)
                    cup_probs = probs[:, cup_idx]
                else:
                    cup_probs = probs[:, -1]

                pred_df = pd.DataFrame({'Team_Abbr': test_teams, 'cup_prob': cup_probs})
                pred_df = pred_df.sort_values('cup_prob', ascending=False).reset_index(drop=True)

                winner_idx = pred_df[pred_df['Team_Abbr'] == actual_winner].index
                winner_rank = winner_idx[0] + 1 if len(winner_idx) &gt; 0 else 999

                results.append({
                    'season': test_season, 'model': model_name,
                    'winner_rank': winner_rank, 'top5_hit': winner_rank &lt;= 5
                })

            except Exception as e:
                print(f"    {model_name} error: {e}")

    results_df = pd.DataFrame(results)

    print("\n" + "-" * 50)
    print("Model Performance Summary:")
    summary = results_df.groupby('model').agg({
        'top5_hit': ['sum', 'mean'],
        'winner_rank': 'mean'
    }).round(3)
    summary.columns = ['Top5_Hits', 'Top5_Rate', 'Avg_Rank']
    print(summary.sort_values('Top5_Rate', ascending=False))

    results_df.to_csv(MODELS_DIR / 'goal1_cv_results.csv', index=False)

    return results_df


def run_goal2_model(goal2_features):

    print("\n" + "=" * 70)
    print("GOAL 2: NEXT-SEASON PREDICTION (PROBABILITY-BASED)")
    print("=" * 70)

    df = goal2_features.copy()

    train_df = df[df['Target_Season'] != '2024-2025'].copy()
    final_pred_df = df[df['Target_Season'] == '2024-2025'].copy()

    print(f"\nTraining pairs: {train_df['Target_Season'].nunique()}")

    print(f"Teams in training: {len(train_df)}")
    print(f"Teams for final prediction: {len(final_pred_df)}")

    exclude_cols = ['Team_Abbr', 'Team_Full', 'Feature_Season', 'Target_Season',
                    'Target_Round_Reached', 'Is_Cup_Winner']
    feature_cols = [c for c in df.columns if c not in exclude_cols
                    and df[c].dtype in ['int64', 'float64']]

    def clean_feature_name(name):
        return name.replace('%', 'Pct').replace('/', '_').replace(' ', '_').replace('[', '').replace(']', '').replace('{', '').replace('}', '')

    clean_feature_cols = [clean_feature_name(c) for c in feature_cols]
    feature_col_map = dict(zip(feature_cols, clean_feature_cols))

    X_train_full = train_df[feature_cols].fillna(train_df[feature_cols].median())
    X_train_full.columns = clean_feature_cols
    y_train_full = train_df['Is_Cup_Winner']

    print(f"Features: {len(feature_cols)}")

    models = get_models()

    print("\n" + "-" * 50)
    print("Leave-One-Season-Out Cross-Validation")
    print("-" * 50)

    cv_results = []
    all_predictions = []

    target_seasons = train_df['Target_Season'].unique()

    for test_target in target_seasons:
```

```
    train_mask = train_df['Target_Season'] != test_target
    test_mask = train_df['Target_Season'] == test_target

    X_train = X_train_full[train_mask].copy()
    y_train = y_train_full[train_mask]
    X_test = X_train_full[test_mask].copy()

    test_subset = train_df[test_mask].copy()
    test_teams = test_subset['Team_Abbr'].values
    actual_winner = CUP_WINNERS_ABBR.get(test_target)

    print(f"\n  Holdout: {test_target} (Winner: {actual_winner})")

    ensemble_probs = []

    for model_name, model in models.items():
        try:
            model.fit(X_train, y_train)

            probs = model.predict_proba(X_test)

            if len(model.classes_) == 2 and 1 in model.classes_:
                cup_idx = list(model.classes_).index(1)
                raw_probs = probs[:, cup_idx]
            else:
                raw_probs = probs[:, -1]

            normalized_probs = softmax_normalize(raw_probs)
            ensemble_probs.append(normalized_probs)

        except Exception as e:
            print(f"    {model_name} error: {e}")

    if len(ensemble_probs) == 0:
        continue

    avg_probs = np.mean(ensemble_probs, axis=0)
    final_probs = softmax_normalize(avg_probs)

    pred_df = pd.DataFrame({
        'Team_Abbr': test_teams,
        'cup_prob': final_probs,
        'is_actual_winner': [1 if t == actual_winner else 0 for t in test_teams]
    })
    pred_df = pred_df.sort_values('cup_prob', ascending=False).reset_index(drop=True)

    winner_row = pred_df[pred_df['Team_Abbr'] == actual_winner]
    if len(winner_row) &gt; 0:

        winner_prob = winner_row['cup_prob'].values[0]
        winner_rank = winner_row.index[0] + 1
    else:
        winner_prob = 0
        winner_rank = 999

    log_loss_winner = calculate_log_loss_single(1, winner_prob)

    cv_results.append({
        'target_season': test_target,
        'actual_winner': actual_winner,
        'predicted_top1': pred_df.iloc[0]['Team_Abbr'],
        'winner_prob': winner_prob,
        'winner_rank': winner_rank,
        'log_loss': log_loss_winner,
        'top3_hit': winner_rank &lt;= 3,
        'top5_hit': winner_rank &lt;= 5
    })

    pred_df['target_season'] = test_target
    all_predictions.append(pred_df)

    print(f"    Winner rank: {winner_rank}, Prob: {winner_prob:.4f}, Log loss: {log_loss_winner:.4f}")

cv_results_df = pd.DataFrame(cv_results)
all_preds_df = pd.concat(all_predictions, ignore_index=True)

print("\n" + "=" * 70)
print("CROSS-VALIDATION SUMMARY")
print("=" * 70)

print(f)

print("\n" + "-" * 50)
print("CALIBRATION ANALYSIS")
```

```
print("-" * 50)

prob_bins = [0, 0.02, 0.05, 0.10, 0.15, 1.0]
bin_labels = ['0-2%', '2-5%', '5-10%', '10-15%', '15%+']

all_preds_df['prob_bin'] = pd.cut(all_preds_df['cup_prob'], bins=prob_bins, labels=bin_labels)

calibration_table = all_preds_df.groupby('prob_bin').agg({
    'is_actual_winner': ['sum', 'count', 'mean'],
    'cup_prob': 'mean'
}).round(4)
calibration_table.columns = ['Actual_Winners', 'Total_Teams', 'Observed_Rate', 'Predicted_Prob']

print("\nCalibration Table:")
print(calibration_table)

try:
    fig, ax = plt.subplots(figsize=(8, 6))

    y_true = all_preds_df['is_actual_winner'].values
    y_prob = all_preds_df['cup_prob'].values

    prob_true, prob_pred = calibration_curve(y_true, y_prob, n_bins=5, strategy='quantile')

    ax.plot([0, 1], [0, 1], 'k--', label='Perfectly calibrated')
    ax.plot(prob_pred, prob_true, 'o-', label='Model')
    ax.set_xlabel('Predicted probability')
    ax.set_ylabel('Observed frequency')
    ax.set_title('Goal 2 Model Calibration (Reliability Diagram)')
    ax.legend()
    ax.grid(True, alpha=0.3)

    plt.savefig(MODELS_GOAL2_DIR / 'calibration_plot.png', dpi=150, bbox_inches='tight')
    plt.close()
    print("\nCalibration plot saved to: calibration_plot.png")
except Exception as e:
    print(f"\nCould not create calibration plot: {e}")

print("\n" + "=" * 70)
print("FINAL PREDICTION: 2024-25 CUP WINNER")
print("=" * 70)
print("\nNote: This is a PRE-OFFSEASON forecast based on 2023-24 data.")
print("No trades, free agency, or injury information is included.")

X_train_all = X_train_full

y_train_all = y_train_full

X_final = final_pred_df[feature_cols].fillna(X_train_full.median())
X_final.columns = clean_feature_cols
final_teams = final_pred_df['Team_Abbr'].values

ensemble_probs = []

for model_name, model in models.items():
    try:
        model.fit(X_train_all, y_train_all)
        probs = model.predict_proba(X_final)

        if len(model.classes_) == 2 and 1 in model.classes_:
            cup_idx = list(model.classes_).index(1)
            raw_probs = probs[:, cup_idx]
        else:
            raw_probs = probs[:, -1]

        normalized_probs = softmax_normalize(raw_probs)
        ensemble_probs.append(normalized_probs)

    except Exception as e:
        print(f"  {model_name} error: {e}")

avg_probs = np.mean(ensemble_probs, axis=0)
final_probs = softmax_normalize(avg_probs)

final_pred_result = pd.DataFrame({
    'Team_Abbr': final_teams,
    'Team_Full': [get_full_team_name(t) for t in final_teams],
    'Cup_Probability': final_probs,
    'Prior_Round_Reached': final_pred_df['Prior_Round_Reached'].values
})
final_pred_result = final_pred_result.sort_values('Cup_Probability', ascending=False).reset_index(drop=True)
final_pred_result['Rank'] = range(1, len(final_pred_result) + 1)

actual_winner = 'FLA'
```

```
    winner_row = final_pred_result[final_pred_result['Team_Abbr'] == actual_winner]

    print(f"\n{'Rank':&lt;6}{'Team':&lt;30}{'Probability':&lt;15}{'2023-24 Result':&lt;15}")
    print("-" * 66)

    round_names = {0: 'Missed', 1: 'R1', 2: 'R2', 3: 'Conf Finals', 4: 'Finals', 5: 'Cup Winner'}

    for _, row in final_pred_result.head(16).iterrows():
        prior = round_names.get(int(row['Prior_Round_Reached']), '?')
        marker = "★" if row['Team_Abbr'] == actual_winner else ""
        print(f"{row['Rank']:&lt;6}{row['Team_Full']:&lt;30}{row['Cup_Probability']*100:&gt;6.2f}%{'':&lt;8}{prior:&lt;15}{marker}")

    print(f"\n  → Actual Winner: Florida Panthers")
    if len(winner_row) &gt; 0:
        print(f"  → Model Rank: #{winner_row['Rank'].values[0]}")
        print(f"  → Model Probability: {winner_row['Cup_Probability'].values[0]*100:.2f}%")

    print("\n" + "-" * 50)
    print("TOP 20 MOST IMPORTANT FEATURES")
    print("-" * 50)

    importance_df = pd.DataFrame({'feature': clean_feature_cols})

    for model_name in ['XGBoost', 'CatBoost', 'RandomForest']:
        try:
            model = models[model_name]
            model.fit(X_train_all, y_train_all)
            importance_df[model_name] = model.feature_importances_
        except Exception as e:
            print(f"  {model_name} feature importance error: {e}")

    importance_cols = [c for c in importance_df.columns if c != 'feature']
    if importance_cols:
        importance_df['avg_importance'] = importance_df[importance_cols].mean(axis=1)
        importance_df = importance_df.sort_values('avg_importance', ascending=False)

        for i, (_, row) in enumerate(importance_df.head(20).iterrows()):
            print(f"  {i+1:2d}. {row['feature']:&lt;45} {row['avg_importance']:.4f}")

    print("\n" + "=" * 70)
    print("SAVING OUTPUTS")
    print("=" * 70)

    cv_results_df.to_csv(MODELS_GOAL2_DIR / 'goal2_cv_results.csv', index=False)
    all_preds_df.to_csv(MODELS_GOAL2_DIR / 'goal2_all_predictions.csv', index=False)
    final_pred_result.to_csv(MODELS_GOAL2_DIR / 'goal2_2024_25_predictions.csv', index=False)
    importance_df.to_csv(MODELS_GOAL2_DIR / 'goal2_feature_importance.csv', index=False)
    calibration_table.to_csv(MODELS_GOAL2_DIR / 'goal2_calibration_table.csv')

    print(f"  Saved to: {MODELS_GOAL2_DIR}")

    summary = {
        'cv_log_loss_mean': cv_results_df['log_loss'].mean(),
        'cv_log_loss_median': cv_results_df['log_loss'].median(),
        'cv_winner_rank_mean': cv_results_df['winner_rank'].mean(),
        'cv_winner_rank_median': cv_results_df['winner_rank'].median(),
        'cv_top5_rate': cv_results_df['top5_hit'].mean(),
        'cv_winner_prob_mean': cv_results_df['winner_prob'].mean(),
        'final_winner_rank': winner_row['Rank'].values[0] if len(winner_row) &gt; 0 else None,
        'final_winner_prob': winner_row['Cup_Probability'].values[0] if len(winner_row) &gt; 0 else None
    }

    return cv_results_df, final_pred_result, importance_df, summary


def run_full_pipeline():

    print("=" * 70)
    print("STANLEY CUP PREDICTION - EXPANDED PIPELINE (16 SEASONS)")
    print("=" * 70)

    data = load_all_data()
    data = clean_and_standardize(data)

    goal1_features = build_goal1_features(data)

    goal1_results = run_goal1_model(goal1_features)

    goal2_features = build_goal2_features(data, goal1_features)

    goal2_cv, goal2_pred, goal2_importance, goal2_summary = run_goal2_model(goal2_features)

    print("\n" + "=" * 70)
    print("PIPELINE COMPLETE")
```

```
    print("=" * 70)

    return {
        'data': data,
        'goal1_features': goal1_features,
        'goal1_results': goal1_results,
        'goal2_features': goal2_features,
        'goal2_cv_results': goal2_cv,
        'goal2_predictions': goal2_pred,
        'goal2_importance': goal2_importance,
        'goal2_summary': goal2_summary
    }


if __name__ == "__main__":
    results = run_full_pipeline()
```