Database Schema Outline for NFL Micro-Edge Engine

A well-defined schema is crucial for efficient data storage and retrieval. We'll design a relational schema to handle historical game data, odds, weather, and betting splits.

Here's a proposed schema outline using a SQLite database for simplicity, which is excellent for prototyping and local development. We can always migrate to a more robust database like PostgreSQL later if needed for scaling.

```
-- Table for NFL Game Data
CREATE TABLE IF NOT EXISTS nfl_games (
    game_id TEXT PRIMARY KEY,
    season INTEGER,
    week INTEGER,
    game_date TEXT, -- YYYY-MM-DD
    home_team TEXT,
    away_team TEXT,
    home_score INTEGER,
    away_score INTEGER,
    stadium TEXT,
    stadium_latitude REAL,
    stadium_longitude REAL,
    UNIQUE(game_id)
);

-- Table for Historical Odds Data
CREATE TABLE IF NOT EXISTS nfl_odds (
    odd_id INTEGER PRIMARY KEY AUTOINCREMENT,
    game_id TEXT,
    sportsbook TEXT,
    odd_type TEXT, -- e.g., 'total', 'spread'
    opening_line REAL,
    closing_line REAL,
    line_movement REAL, -- Closing - Opening
    timestamp TEXT, -- When the odd was recorded (for tick-by-tick if
available)
    FOREIGN KEY (game_id) REFERENCES nfl_games(game_id)
);

-- Table for Historical Weather Data
CREATE TABLE IF NOT EXISTS nfl_weather (
    weather_id INTEGER PRIMARY KEY AUTOINCREMENT,
    game_id TEXT,
    timestamp TEXT, -- Hourly timestamp
    temperature_celsius REAL,
    wind_speed_ms REAL,
    precipitation_mm REAL,
    wind_gusts_ms REAL,
    FOREIGN KEY (game_id) REFERENCES nfl_games(game_id)
);

-- Table for Betting Splits (Public Money/Bets)
CREATE TABLE IF NOT EXISTS nfl_betting_splits (
    split_id INTEGER PRIMARY KEY AUTOINCREMENT,
    game_id TEXT,
    timestamp TEXT, -- When the split was recorded
    total_bets_pct_over REAL,
    total_money_pct_over REAL,
    total_bets_pct_under REAL,
    total_money_pct_under REAL,
    FOREIGN KEY (game_id) REFERENCES nfl_games(game_id)
);

-- Table for Engineered Features (Optional, could also be calculated
on the fly)
CREATE TABLE IF NOT EXISTS nfl_features (
    feature_id INTEGER PRIMARY KEY AUTOINCREMENT,
    game_id TEXT,
    feature_date TEXT, -- Date the features were calculated/relevant
    public_bias_index REAL,
    sharp_action INTEGER, -- 0 or 1
    reverse_line_movement INTEGER, -- 0 or 1
    wind_total_adjustment REAL,
    temp_total_adjustment REAL,
    epa_diff REAL,
    recent_epa_trend REAL,
    FOREIGN KEY (game_id) REFERENCES nfl_games(game_id)
);
```

Explanation of Schema Design:

- **nfl_games**: This will be our central table, storing core information about each NFL game. game_id is crucial for linking all other data.
- **nfl_odds**: Stores various odds. odd_type allows flexibility for totals, spreads, money lines, etc. timestamp is vital for capturing line movement chronologically.
- **nfl_weather**: Stores hourly weather data. We'll link this to games based on game_id and the nearest hour to game time.
- **nfl_betting_splits**: Captures public betting percentages. It's important to record the timestamp as these can change.
- **nfl_features**: While features can be engineered on the fly, storing them can speed up model training and backtesting, especially if they are computationally intensive.

Python Scripts for Data Acquisition and Processing

Now, let's create Python scripts that implement the data acquisition strategy you outlined, integrating with the proposed database schema. We'll use sqlite3 for database interaction.

```
import sqlite3
import pandas as pd
import nfl_data_py as nfl
import requests
import time
import json # For The Odds API caching
from datetime import datetime, timedelta

# --- Database Setup ---
DATABASE_NAME = 'nfl_betting_data.db'

def create_database_schema():
    conn = sqlite3.connect(DATABASE_NAME)
    cursor = conn.cursor()

    cursor.execute("""
        CREATE TABLE IF NOT EXISTS nfl_games (
            game_id TEXT PRIMARY KEY,
            season INTEGER,
            week INTEGER,
            game_date TEXT, -- YYYY-MM-DD
            home_team TEXT,
            away_team TEXT,
            home_score INTEGER,
            away_score INTEGER,
            stadium TEXT,
            stadium_latitude REAL,
            stadium_longitude REAL,
            UNIQUE(game_id)
        );
    """)

    cursor.execute("""
        CREATE TABLE IF NOT EXISTS nfl_odds (
            odd_id INTEGER PRIMARY KEY AUTOINCREMENT,
            game_id TEXT,
            sportsbook TEXT,
            odd_type TEXT, -- e.g., 'total', 'spread'
            opening_line REAL,
            closing_line REAL,
            line_movement REAL, -- Closing - Opening
            timestamp TEXT, -- When the odd was recorded (for
tick-by-tick if available)
            FOREIGN KEY (game_id) REFERENCES nfl_games(game_id)
        );
    """)

    cursor.execute("""
        CREATE TABLE IF NOT EXISTS nfl_weather (
            weather_id INTEGER PRIMARY KEY AUTOINCREMENT,
            game_id TEXT,
            timestamp TEXT, -- Hourly timestamp YYYY-MM-DD HH:MM:SS
            temperature_celsius REAL,
            wind_speed_ms REAL,
            precipitation_mm REAL,
            wind_gusts_ms REAL,
            FOREIGN KEY (game_id) REFERENCES nfl_games(game_id)
        );
    """)

    cursor.execute("""
        CREATE TABLE IF NOT EXISTS nfl_betting_splits (
            split_id INTEGER PRIMARY KEY AUTOINCREMENT,
            game_id TEXT,
            timestamp TEXT, -- When the split was recorded YYYY-MM-DD
HH:MM:SS
            total_bets_pct_over REAL,
            total_money_pct_over REAL,
            total_bets_pct_under REAL,
            total_money_pct_under REAL,
            FOREIGN KEY (game_id) REFERENCES nfl_games(game_id)
        );
    """)

    # Note: The nfl_features table is omitted here for simplicity
    # as these features will likely be computed during the backtesting
phase
    # or just before model training, rather than stored permanently in
the DB.

    conn.commit()
    conn.close()
    print(f"Database schema created or verified in {DATABASE_NAME}")

# --- Data Acquisition Functions ---

# Statically defined stadium coordinates for NFL stadiums for weather
data.
# This list is not exhaustive and would need to be expanded.
STADIUM_COORDS = {
    "Lambeau Field": (44.5013, -88.0622),
    "Soldier Field": (41.8623, -87.6167),
    "Gillette Stadium": (42.0627, -71.2643),
    "Arrowhead Stadium": (39.0489, -94.4839),
    "AT&T Stadium": (32.7478, -97.0945),
    "SoFi Stadium": (33.9535, -118.3394),
    "MetLife Stadium": (40.8135, -74.0740),
    "Mercedes-Benz Stadium": (33.7554, -84.4009),
    "Levi's Stadium": (37.4035, -121.9701),
    "Lincoln Financial Field": (39.9009, -75.1675),
    "Allegiant Stadium": (36.0906, -115.1837),
    "State Farm Stadium": (33.5303, -112.2625),
    "Paycor Stadium": (39.0954, -84.5160),
    "Bank of America Stadium": (35.2259, -80.8528),
    "M&T Bank Stadium": (39.2780, -76.6227),
    "Highmark Stadium": (42.7738, -78.7869),
    "NRG Stadium": (29.6847, -95.4093),
    "TIAA Bank Field": (30.3239, -81.6375),
    "Hard Rock Stadium": (25.9579, -80.2393),
    "Nissan Stadium": (36.1664, -86.7713),
    "Acrisure Stadium": (40.4468, -80.0157),
    "FedExField": (38.9077, -76.8647),
    "Lucas Oil Stadium": (39.7600, -86.1517),
    "Ford Field": (42.3401, -83.0456),
    "Huntington Bank Field": (41.4815, -81.6994), # Cleveland Browns
Stadium
    "Empower Field at Mile High": (39.7439, -105.0201),
    "Lumen Field": (47.5952, -122.3316),
    "Raymond James Stadium": (27.9759, -82.5033),
    "Caesars Superdome": (29.9509, -90.0813),
    "US Bank Stadium": (44.9738, -93.2662),
    "EverBank Stadium": (30.3239, -81.6375), # TIAA Bank Field (name
change)
    "Camping World Stadium": (28.5393, -81.3891), # Used for some
early season games/preseason
    "Estadio Azteca": (19.3027, -99.1500), # Occasional international
games
    "Tottenham Hotspur Stadium": (51.6033, -0.0664), # Occasional
international games
    "Deutsche Bank Park": (50.0703, 8.6470), # Occasional
international games
    "Allianz Arena": (48.2188, 11.6247), # Occasional international
games
}


def fetch_nfl_game_data(years):
    """
    Fetches historical NFL game data using nfl_data_py and inserts
into the database.
    """
    print(f"Fetching NFL game data for years: {years}...")
    try:
        # Import schedules for game-level data including stadium info
and game_id
        schedules_df = nfl.import_schedules(years=years)
        # Select relevant columns for our nfl_games table
        games_to_insert = []
        for _, row in schedules_df.iterrows():
            stadium_name = row['stadium']
            lat, lon = STADIUM_COORDS.get(stadium_name, (None, None))
# Get coords or None

            games_to_insert.append((
                row['game_id'],
                row['season'],
                row['week'],
                str(row['game_date']), # Ensure date is string format
YYYY-MM-DD
                row['home_team'],
                row['away_team'],
                row['home_score'],
                row['away_score'],
                stadium_name,
                lat,
                lon
            ))

        conn = sqlite3.connect(DATABASE_NAME)
        cursor = conn.cursor()

        # Use INSERT OR IGNORE to avoid errors on existing game_ids
        cursor.executemany("""
            INSERT OR IGNORE INTO nfl_games (
                game_id, season, week, game_date, home_team,
away_team,
                home_score, away_score, stadium, stadium_latitude,
stadium_longitude
            ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """, games_to_insert)

        conn.commit()
        conn.close()
        print(f"Successfully inserted/updated {len(games_to_insert)}
NFL games.")
        return True
    except Exception as e:
        print(f"Error fetching/inserting NFL game data: {e}")
        return False

def get_historical_weather(latitude, longitude, start_date, end_date):
    """
    Fetches historical hourly weather data from Open-Meteo for a given
```

```
location and date range.
    start_date and end_date should be 'YYYY-MM-DD' strings.
    """
    url = "https://archive-api.open-meteo.com/v1/archive"
    params = {
        "latitude": latitude,
        "longitude": longitude,
        "start_date": start_date,
        "end_date": end_date,
        "hourly":
"temperature_2m,wind_speed_10m,precipitation,wind_gusts_10m",
        "timezone": "America/New_York" # NFL games are typically in
EST/EDT
    }

    try:
        response = requests.get(url, params=params)
        response.raise_for_status() # Raise an HTTPError for bad
responses (4xx or 5xx)
        return response.json()
    except requests.exceptions.RequestException as e:
        print(f"Error fetching weather data for {latitude},{longitude}
on {start_date}: {e}")
        return None

def fetch_and_store_weather_data():
    """
    Fetches weather data for all games in the nfl_games table that
have stadium coordinates
    and inserts it into the nfl_weather table.
    """
    print("Fetching and storing weather data...")
    conn = sqlite3.connect(DATABASE_NAME)
    cursor = conn.cursor()

    # Get games that have stadium coordinates and for which we haven't
fetched weather yet
    cursor.execute("""
        SELECT game_id, game_date, stadium_latitude, stadium_longitude
        FROM nfl_games
        WHERE stadium_latitude IS NOT NULL AND stadium_longitude IS
NOT NULL
        AND game_id NOT IN (SELECT DISTINCT game_id FROM nfl_weather)
    """)
    games_for_weather = cursor.fetchall()

    weather_data_to_insert = []
    fetched_count = 0
    for game_id, game_date, lat, lon in games_for_weather:
        if lat is None or lon is None:
            continue # Skip if coordinates are missing

        # Open-Meteo requires start_date and end_date to be the same
for a single day
        weather_json = get_historical_weather(lat, lon, game_date,
game_date)

        if weather_json and 'hourly' in weather_json:
            hourly_data = weather_json['hourly']
            times = hourly_data.get('time', [])
            temperatures = hourly_data.get('temperature_2m', [])
            wind_speeds = hourly_data.get('wind_speed_10m', [])
            precipitations = hourly_data.get('precipitation', [])
            wind_gusts = hourly_data.get('wind_gusts_10m', [])

            for i in range(len(times)):
                weather_data_to_insert.append((
                    game_id,
                    times[i], # Timestamp for the hour
                    temperatures[i] if i < len(temperatures) else
None,
                    wind_speeds[i] if i < len(wind_speeds) else None,
                    precipitations[i] if i < len(precipitations) else
None,
                    wind_gusts[i] if i < len(wind_gusts) else None
                ))
            fetched_count += 1
            # Be mindful of API rate limits if making many requests
quickly
            time.sleep(0.1) # Small delay

    if weather_data_to_insert:
        cursor.executemany("""
            INSERT OR IGNORE INTO nfl_weather (
                game_id, timestamp, temperature_celsius,
wind_speed_ms,
                precipitation_mm, wind_gusts_ms
            ) VALUES (?, ?, ?, ?, ?, ?)
        """, weather_data_to_insert)
        conn.commit()
        print(f"Successfully inserted weather data for {fetched_count}
games.")
    else:
        print("No new weather data to insert.")
    conn.close()


# --- Odds Data Strategy ---
# Placeholder for The Odds API client. You'll need to replace
'YOUR_FREE_KEY'
# with your actual API key from The Odds API.
THE_ODDS_API_KEY = 'YOUR_FREE_KEY'
THE_ODDS_API_BASE_URL =
'https://api.the-odds-api.com/v4/sports/americanfootball_nfl/odds/'

def fetch_and_store_odds_data(game_date_start, game_date_end):
    """
    Fetches historical NFL odds data from The Odds API and inserts
into the database.
    This fetches 'totals' odds specifically.
    """
    print(f"Fetching odds data from {game_date_start} to
{game_date_end}...")

    # The Odds API historical data is typically accessed by a specific
game ID or date range.
    # We need to iterate through games to fetch their odds.

    conn = sqlite3.connect(DATABASE_NAME)
    cursor = conn.cursor()

    # Get game_ids and dates for which we need odds data
    # Filter for games that haven't been played yet or for which we
don't have odds
    cursor.execute(f"""
        SELECT game_id, game_date
        FROM nfl_games
        WHERE game_date BETWEEN '{game_date_start}' AND
'{game_date_end}'
    """)
    games_to_fetch_odds_for = cursor.fetchall()

    odds_data_to_insert = []
    fetched_count = 0

    for game_id, game_date_str in games_to_fetch_odds_for:
        # Check if we already have odds for this game (simplistic
check)
        cursor.execute("SELECT 1 FROM nfl_odds WHERE game_id = ? LIMIT
1", (game_id,))
        if cursor.fetchone():
            continue # Skip if odds already exist for this game_id
        # The Odds API has a specific format for historical odds.
        # For this example, we'll simulate fetching by date, but
ideally, you'd fetch per game_id
        # if the API supports it efficiently, or in bulk by date.
        # This example uses the 'GET_ODDS' endpoint, which returns
live odds.
        # For historical, you'd use 'GET_HISTORICAL_ODDS', which is a
paid feature.
        # The prompt mentioned "The Odds API client" and
"get_historical_odds"
        # For the free tier, we typically get live odds, not
historical tick-by-tick.
        # For this prototype, let's assume we're fetching a snapshot.

        # For a truly free solution, historical odds are often found
in Kaggle datasets
        # or require manual scraping of archival sites.

        # Simulating fetching live odds or a snapshot for
demonstration
        # This part requires a more robust setup if you want true
historical tick-by-tick.
        # For now, let's assume a simplified API call.
        try:
            # This is a simplified call, typically you query all odds
for a sport
            # and then filter by game.
            response = requests.get(

f"{THE_ODDS_API_BASE_URL}?apiKey={THE_ODDS_API_KEY}&regions=us&markets
=totals"
            )
            response.raise_for_status()
            odds_data = response.json()

            for event in odds_data:
                if event['id'] == game_id: # Assuming The Odds API
`id` matches our `game_id`
                    for bookmaker in event['bookmakers']:
                        sportsbook_name = bookmaker['key']
                        for market in bookmaker['markets']:
                            if market['key'] == 'totals':
                                for outcome in market['outcomes']:
                                    # For totals, we'll store the
point and the price
                                    # For simplicity, we'll just get
the "point" as the line
```

```
                                    # and assume opening/closing will
be derived from multiple snapshots
                                    # This needs refinement for real
tick-by-tick.
                                    line = outcome['point']
                                    price = outcome['price'] #
Probability/price associated with Over/Under

                                    # In a real scenario, you'd
capture opening_line and closing_line
                                    # by taking multiple snapshots
over time.
                                    # For this example, we'll just
store a single snapshot.

                                    odds_data_to_insert.append((
                                        game_id,
                                        sportsbook_name,
                                        'total',
                                        line,        # Simplified as
line for now
                                        line,        # Simplified as
line for now
                                        0.0,         # Line movement
(needs multiple snapshots)
                                        datetime.now().isoformat() #
Timestamp of capture
                                    ))
                    fetched_count += 1
                    break # Found odds for this game_id

        except requests.exceptions.RequestException as e:
            print(f"Error fetching odds for {game_id} from The Odds
API: {e}")
            # Consider adding more robust error handling and retry
logic
        time.sleep(0.1) # Be mindful of API rate limits

    if odds_data_to_insert:
        cursor.executemany("""
            INSERT OR IGNORE INTO nfl_odds (
                game_id, sportsbook, odd_type, opening_line,
closing_line,
                line_movement, timestamp
            ) VALUES (?, ?, ?, ?, ?, ?, ?)
        """, odds_data_to_insert)
        conn.commit()
        print(f"Successfully inserted odds data for {fetched_count}
games.")
    else:
        print("No new odds data to insert.")

    conn.close()

# --- Public Betting Data (Action Network Scraping) ---
# This part requires setting up a Chrome WebDriver.
# Make sure you have Chrome installed and a compatible ChromeDriver in
your PATH.
# Download ChromeDriver from:
https://chromedriver.chromium.org/downloads

from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import TimeoutException,
NoSuchElementException

# This assumes ChromeDriver is in your system's PATH or specified
here.
# For local development, you might specify the path directly:
# CHROME_DRIVER_PATH = '/path/to/chromedriver'
# service = Service(executable_path=CHROME_DRIVER_PATH)
# driver = webdriver.Chrome(service=service)

def scrape_action_network_splits():
    """
    Scrapes public betting split data from Action Network.
    Returns a list of dictionaries with game data or None on failure.
    """
    print("Attempting to scrape Action Network for public betting
splits...")
    driver = None
    try:
        options = webdriver.ChromeOptions()
        options.add_argument('--headless') # Run in headless mode (no
GUI)
        options.add_argument('--no-sandbox')
        options.add_argument('--disable-dev-shm-usage')

        driver = webdriver.Chrome(options=options)
        driver.get('https://www.actionnetwork.com/nfl/public-betting')

        # Wait for dynamic content to load, up to 10 seconds
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CLASS_NAME,
'game-row'))
        )

        games = driver.find_elements(By.CLASS_NAME, 'game-row')
        betting_data = []

        for game in games:
            try:
                # Extract relevant text, handle potential errors if
elements are missing
                game_teams = game.find_element(By.CLASS_NAME,
'teams').text
                total_bets_pct = game.find_element(By.CSS_SELECTOR,
'div[data-total-type="total-bets-pct"]').text
                total_money_pct = game.find_element(By.CSS_SELECTOR,
'div[data-total-type="total-money-pct"]').text

                # Clean and convert percentages
                total_bets_pct_val = float(total_bets_pct.replace('%',
'')) / 100.0 if total_bets_pct else None
                total_money_pct_val =
float(total_money_pct.replace('%', '')) / 100.0 if total_money_pct
else None

                # Action Network usually shows total bets for 'Over'.
We need 'Under' as well.
                # Assuming the displayed percentage is for 'Over',
then 'Under' is 100 - 'Over'
                total_bets_pct_under_val = 1.0 - total_bets_pct_val if
total_bets_pct_val is not None else None
                total_money_pct_under_val = 1.0 - total_money_pct_val
if total_money_pct_val is not None else None


                # We need to link this to a specific game_id. Action
Network doesn't directly
                # provide NFL game_ids from nfl_data_py. This is a
crucial challenge.
                # A robust solution would involve matching team names
and dates.
                # For this prototype, we'll just capture the raw text
and deal with matching later.

                betting_data.append({
                    'game_description': game_teams, # For manual
matching later
                    'total_bets_pct_over': total_bets_pct_val,
                    'total_money_pct_over': total_money_pct_val,
                    'total_bets_pct_under': total_bets_pct_under_val,
                    'total_money_pct_under':
total_money_pct_under_val,
                    'timestamp': datetime.now().isoformat()
                })
            except NoSuchElementException as e:
                print(f"Skipping a game row due to missing element:
{e}")
                continue # Skip to the next game row if elements are
missing

        print(f"Successfully scraped {len(betting_data)} betting split
entries.")
        return betting_data

    except TimeoutException:
        print("Timed out waiting for Action Network content to load.")
    except Exception as e:
        print(f"An error occurred during scraping Action Network:
{e}")
    finally:
        if driver:
            driver.quit()
    return None

def store_betting_splits(betting_data):
    """
    Stores scraped betting split data into the database.
    NOTE: This function currently requires manual matching of
'game_description' to 'game_id'.
    A more advanced solution would involve fuzzy matching team names
and dates.
    """
    if not betting_data:
        print("No betting data to store.")
        return

    conn = sqlite3.connect(DATABASE_NAME)
    cursor = conn.cursor()

    # For prototyping, we'll insert with a placeholder game_id and
require manual matching.
    # In a real system, you'd try to resolve game_id here by querying
nfl_games table
    # based on game_description (e.g., "NYJ vs BUF" and date).

    # Let's try a simple lookup for games played today, or in the near
```

```
future.
    # This is highly simplified and will likely need manual correction
for past games.

    today_str = datetime.now().strftime('%Y-%m-%d')
    cursor.execute(f"""
        SELECT game_id, home_team, away_team, game_date
        FROM nfl_games
        WHERE game_date >= '{today_str}' -- Look for upcoming or
today's games
    """)
    upcoming_games = cursor.fetchall()

    insert_count = 0
    for split_entry in betting_data:
        # Simple string matching for demonstration. This is not robust
for production.
        matched_game_id = None
        for game_id, home_team, away_team, game_date in
upcoming_games:
            game_desc_lower = split_entry['game_description'].lower()
            home_team_lower = home_team.lower()
            away_team_lower = away_team.lower()

            if (home_team_lower in game_desc_lower and away_team_lower
in game_desc_lower) or \
               (away_team_lower in game_desc_lower and home_team_lower
in game_desc_lower):
                matched_game_id = game_id
                break # Found a potential match

        if matched_game_id:
            try:
                cursor.execute("""
                    INSERT INTO nfl_betting_splits (
                        game_id, timestamp, total_bets_pct_over,
total_money_pct_over,
                        total_bets_pct_under, total_money_pct_under
                    ) VALUES (?, ?, ?, ?, ?, ?)
                """, (
                    matched_game_id,
                    split_entry['timestamp'],
                    split_entry['total_bets_pct_over'],
                    split_entry['total_money_pct_over'],
                    split_entry['total_bets_pct_under'],
                    split_entry['total_money_pct_under']
                ))
                insert_count += 1
            except sqlite3.IntegrityError:
                print(f"Betting split for {matched_game_id} at
{split_entry['timestamp']} already exists.")
            except Exception as e:
                print(f"Error inserting betting split for
{matched_game_id}: {e}")
        else:
            print(f"Could not find matching game_id for:
{split_entry['game_description']}")

    conn.commit()
    conn.close()
    print(f"Attempted to store {insert_count} betting split entries.")


# --- Main Execution Flow ---
if __name__ == "__main__":
    create_database_schema()

    # 1. Fetch and store NFL Game Data
    # For a full historical run, you might want to specify a wider
range of years.
    # Be mindful that older nflverse data might have slight
inconsistencies in stadium names.
    fetch_nfl_game_data(years=list(range(2020, 2025))) # Fetch data
from 2020 to 2024 seasons

    # 2. Fetch and store Weather Data
    fetch_and_store_weather_data()

    # 3. Fetch and store Odds Data
    # NOTE: The free tier of The Odds API for historical data is
limited.
    # For robust historical odds, you would need a paid subscription
or alternative sources
    # (e.g., Kaggle datasets for older data, or specific data
providers).
    # This example attempts to fetch for a recent/upcoming period.
    # Adjust the date range as needed.

    # Fetching odds for past 30 days and next 7 days for demonstration
    end_date = datetime.now().strftime('%Y-%m-%d')
    start_date = (datetime.now() -
timedelta(days=30)).strftime('%Y-%m-%d')
    fetch_and_store_odds_data(start_date, end_date)


    # 4. Fetch and store Public Betting Splits
    # This part requires ChromeDriver and can be fragile due to website changes.
    # Run this only if you have ChromeDriver set up and want to capture live splits.
    # It's best to run this regularly via a scheduler for up-to-date information.
    # For historical splits, you would need to scrape archives or purchase data.
    # betting_splits = scrape_action_network_splits()
    # store_betting_splits(betting_splits)
    # print("\n--- Data Acquisition Complete ---")
    # print(f"Check '{DATABASE_NAME}' for your data.")

    # You can then query the database to verify data:
    # conn = sqlite3.connect(DATABASE_NAME)
    # df_games = pd.read_sql_query("SELECT * FROM nfl_games LIMIT 5;", conn)
    # df_weather = pd.read_sql_query("SELECT * FROM nfl_weather LIMIT 5;", conn)
    # df_odds = pd.read_sql_query("SELECT * FROM nfl_odds LIMIT 5;", conn)
    # df_splits = pd.read_sql_query("SELECT * FROM nfl_betting_splits LIMIT 5;", conn)
    # print("\nSample Game Data:\n", df_games)
    # print("\nSample Weather Data:\n", df_weather)
    # print("\nSample Odds Data:\n", df_odds)
    # print("\nSample Betting Splits Data:\n", df_splits)
    # conn.close()
```

How to Implement and Run This Code:

1. **Save the Code**: Save the entire code block above as a Python file (e.g., nfl_data_pipeline.py).
2. **Install Dependencies**:

   ```
   pip install pandas nfl_data_py requests selenium
   ```

3. **ChromeDriver Setup for Selenium (Crucial for Betting Splits)**:
   - Download the appropriate ChromeDriver for your Chrome browser version from the official ChromeDriver downloads page.
   - Place the chromedriver executable in a directory that is in your system's PATH, or specify its path directly in the Service object when initializing the webdriver.Chrome (as commented in the scrape_action_network_splits function).
4. **The Odds API Key**:
   - Register for a free API key at The Odds API.
   - Replace 'YOUR_FREE_KEY' in the THE_ODDS_API_KEY variable with your actual API key.
   - **Important Note on The Odds API**: The free tier of The Odds API generally

provides *live* odds. Historical, tick-by-tick odds are usually a paid feature. For robust backtesting with precise historical line movements, you will likely need to invest in a premium data provider or explore other historical data sources (like Kaggle datasets for older, static odds archives). The current fetch_and_store_odds_data function in the script is a simplification for demonstration purposes.

5. **Run the Script**:
```
python nfl_data_pipeline.py
```

Next Steps and Considerations:

- **Data Cleaning and Preprocessing**: Once data is acquired, the next phase would involve rigorous cleaning, handling missing values, and standardizing formats before feature engineering.
- **Stadium Coordinates**: The STADIUM_COORDS dictionary is not exhaustive. You'll need to expand it to cover all NFL stadiums present in your historical game data. You can programmatically extract unique stadium names from your nfl_games DataFrame and then find their coordinates.
- **Robust Game Matching for Splits**: The current method for matching scraped betting splits (game_description) to game_id is very basic. For a production system, you would need a more sophisticated fuzzy matching algorithm or an external lookup service that maps team names/game times to canonical game IDs.
- **Incremental Data Updates**: As suggested in your initial analysis, setting up a system for incremental updates (only fetching new data since the last run) is vital for long-term maintenance and efficiency.
- **Error Handling and Logging**: Enhance the scripts with more detailed error handling, logging, and retry mechanisms for API calls.
- **Scalability**: For very large datasets or high-frequency updates, consider migrating from SQLite to a more powerful relational database like PostgreSQL or a data warehouse solution.
- **Scheduling**: Automate these scripts to run periodically (e.g., daily or weekly) using tools like cron (Linux/macOS) or Windows Task Scheduler, or a Python scheduler library like APScheduler or schedule.

This first phase sets a strong technical foundation for Project "LUCY." Let me know when you're ready to move on to building out the Signal Stack tiers!