Fantasy NBA Analysis Project

1. Problem Statement -

- Objective: Objective: Predict individual NBA player statistics (PTS, REB, AST, STL, BLK, 3PM, FG%, FT%, TO) for the upcoming season to support fantasy basketball decisions in a 9-category Head-to-Head (H2H) league format.
- Why it matters: In fantasy basketball, especially in the 9-cat H2H format, success depends on forecasting player performance across multiple statistical categories, not just points.
- Practical Outcome: A system that predicts performance in each category per player and his total outcome, enabling data-driven team-building recommendations.

2. Data Collection -

- NBA API - Used to get updated data (e.g., average stats).
- Kaggle - "NBA - Player stats - Season 24/25" (per game stats until all-star break), "NBA Players Stats 23/24", "2022-2023 NBA Player Stats"
- Basketball Reference - Manually downloaded per-game and total stats for seasons 2020-2021 ,2021-2022 ,2022-2023, 2023-2024, and 2024-2025. (Consistency in date formatting.)

3. Method -

- Measuring player performance based on the Z-score (a common ranking system in the 9-cat H2H format).
- **Formula** - per-category Z-score is calculated as $Z = (\text{value} - \text{mean})/\text{std}$. For Turnovers (TOV), the sign is flipped: $Z = (\text{mean} - \text{TOV})/\text{std}$ (as higher TOV is detrimental).
- For the first task, I will predict the total Z-score across a season for fantasy players. The model will be trained on data from the 2021-2024 seasons, tested on the 2024-2025 season, and used to provide predictions for the upcoming 2025-2026 season.
- **Regression models** -
  - OLS (ordinary least squares) by statmodels - the foundational technique of linear regression. It works by finding the line (or hyperplane) that minimizes the sum of the squared vertical distances (residuals) between the data points and the line.
  - Linear Regression - the general model that assumes a linear relationship between the input features and the target variable, most often using the OLS method to find the best-fitting coefficients.
  - Ridge Regression - a variation of linear regression that introduces L2 regularization. This technique adds a penalty to the OLS loss function based on the square of the coefficient magnitudes. Its primary purpose is to shrink the coefficients towards zero, which helps prevent overfitting and improves stability in cases where features are highly correlated (multicollinearity).

  - Linear Regression + PCA - a two-stage modeling approach. PCA is first applied to the data to reduce its dimensionality by transforming the original features into a smaller, uncorrelated set of components. Linear Regression is then performed on these new components.
  - Random Forest Regressor - a powerful ensemble method. It constructs multiple independent Decision Trees during training and computes the final prediction by taking the average of the individual tree predictions. This averaging process makes the model robust to outliers and generally highly accurate.
  - XGBoost Regressor (Extreme Gradient Boosting) - a state-of-the-art Gradient Boosting ensemble technique. It builds trees sequentially, where each new tree is designed to correct the errors left by the previous ensemble of trees. It is well-regarded for its high efficiency and top-tier predictive performance across many datasets.
- **Evaluation metrics** -
  - R-squared - a fundamental metric that represents the proportion of the variance in the target variable that is explained by the features in the model. A score of 1.0 indicates a perfect fit, while 0.0 means the model performs no better than simply predicting the mean of the target variable.
  - RMSE (root mean squared error) - measures the square root of the average of the squared errors. Because errors are squared before being averaged, this metric is highly sensitive to large errors (outliers), making it the preferred choice when you need to severely penalize poor predictions.
  - MAE (mean absoulte error) - measures the average of the absolute differences between the model's predictions and the actual values. It is less sensitive to outliers than RMSE and provides an error value that is in the same units as the target variable, making it very interpretable.
  - Mean Absolute Percentage Error (MAPE) - calculates the average of the absolute percentage errors. This is a scale-independent metric that expresses the model's accuracy as a percentage of the actual value, providing a highly intuitive measure for business interpretation.
  - Adjusted R-squared - improves upon R^2 by penalizing the metric when new features are added that do not significantly contribute to the model's predictive power. This makes it a more reliable metric for comparing models that contain different numbers of features.

In [1]:
```
import os
os.environ['OMP_NUM_THREADS'] = '2'
os.environ["LOKY_MAX_CPU_COUNT"] = "4"
```

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import datetime
import time
import statsmodels.api as sm
from xgboost import XGBRegressor
```

```
from sklearn.decomposition import PCA
from nba_api.stats.endpoints import playergamelog, leaguegamelog, playercareerst
from nba_api.stats.static import players
from mpl_toolkits.mplot3d import Axes3D
from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val
from sklearn.inspection import permutation_importance
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression,LogisticRegressionCV, Linear
from sklearn.metrics import roc_auc_score, roc_curve, auc, mean_squared_error,ac
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, QuantileTransformer ,PowerTran
from sklearn.cluster import KMeans
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier,
from requests.exceptions import ReadTimeout
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import GridSearchCV


print("✅ All libraries are working!")
```

```
✅ All libraries are working!
```

In [3]:
```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("eduardopalmieri/nba-player-stats-season-2425"
print("Path to dataset files:", path)

path2 = kagglehub.dataset_download("bryanchungweather/nba-player-stats-dataset-f
print("Path to dataset files:", path2)

path3 = kagglehub.dataset_download("orkunaktas/nba-players-stats-2324")
print("Path to dataset files:", path3)
```

```
Path to dataset files: C:\Users\maors\.cache\kagglehub\datasets\eduardopalmieri\n
ba-player-stats-season-2425\versions\37
Path to dataset files: C:\Users\maors\.cache\kagglehub\datasets\bryanchungweather
\nba-player-stats-dataset-for-the-2023-2024\versions\14
Path to dataset files: C:\Users\maors\.cache\kagglehub\datasets\orkunaktas\nba-pl
ayers-stats-2324\versions\1
```

I used the three cells below to extract the relevant data (avg stats from seasons 21,22,23,24,25) for my project from NBA API and to save it as a CSV on my computer. Total number of players ~500

In [4]:
```
def get_player_stats_from_api(csv_filepath, start_season_str='2020-21', end_seas
    """
    Fetches aggregated season statistics for players listed in the provided CSV,
    starting from the specified minimum season (2020-21) up to the end_season_st

    Args:
        csv_filepath (str): Path to the input CSV file ('nba_5seasons_final_with
        start_season_str (str): The earliest NBA season to fetch data for (e.g.,
        end_season_str (str): The final NBA season to fetch data for (e.g., '202

    Returns:
        pd.DataFrame: A consolidated DataFrame of all requested player season st
    """
```

```
print("--- Step 1: Loading Data and Preparing Player List ---")

# --- 1. Load and process player names and initial seasons from the local CS
df_local = pd.read_csv(csv_filepath)

def season_to_int(season_id):
    # Converts '2022-23' to 2022
    return int(season_id.split('-')[0])

df_local['Start_Year_Original'] = df_local['SEASON_ID'].apply(season_to_int)

# Get the minimum documented start year for each unique player
player_min_years = df_local.groupby('PLAYER_NAME')['Start_Year_Original'].mi

# --- 2. Determine the full list of NBA seasons to query ---
min_query_year = season_to_int(start_season_str) # 2020
end_query_year = season_to_int(end_season_str)   # 2024

# Determine the actual start year for fetching (min_query_year vs. player's
player_min_years['Start_Year_Actual'] = player_min_years['Start_Year_Origina
    lambda x: max(x, min_query_year)
)

# --- 3. Load all NBA players for ID lookup ---
nba_players = players.get_players()
player_id_map = {player['full_name']: player['id'] for player in nba_players

# Combine start year and NBA ID
player_data = player_min_years.copy()
player_data['Player_ID'] = player_data['PLAYER_NAME'].map(player_id_map)

# Identify players who could not be matched
unmatched_players = player_data[player_data['Player_ID'].isna()]
if not unmatched_players.empty:
    print(f"Warning: Could not find NBA ID for {len(unmatched_players)} play
    print(f"Skipping: {list(unmatched_players['PLAYER_NAME'])}")
    player_data = player_data.dropna(subset=['Player_ID']).copy()

player_data['Player_ID'] = player_data['Player_ID'].astype(int)

print(f"Found {len(player_data)} players to process (up to {end_season_str})
print("-" * 50)

# --- Step 2: Fetching Stats from NBA API ---
all_stats_dfs = []

for index, row in player_data.iterrows():
    player_name = row['PLAYER_NAME']
    player_id = row['Player_ID']
    start_year_int = row['Start_Year_Actual'] # Use the adjusted start year

    print(f"Processing ({index + 1}/{len(player_data)}): {player_name} (ID:

    # --- Generate seasons list for this player ---
    seasons_to_fetch = [
        f"{year}-{str(year + 1)[2:]}"
        for year in range(start_year_int, end_query_year + 1)
    ]

    try:
```

```
            # PlayerCareerStats fetches ALL regular season totals for ALL season
            career_stats = playercareerstats.PlayerCareerStats(player_id=player_

            # The 'CareerTotalsRegularSeason' table contains all season-by-seaso
            career_df = career_stats.get_data_frames()[0]

            # Filter the fetched data to include only the requested seasons
            career_df['In_Range'] = career_df['SEASON_ID'].apply(lambda x: x in
            career_df = career_df[career_df['In_Range']].drop(columns=['In_Range

            # Add player name and ID for clarity
            career_df['PLAYER_NAME'] = player_name
            career_df['PLAYER_ID'] = player_id

            all_stats_dfs.append(career_df)

            max_fetched_season = career_df['SEASON_ID'].max() if not career_df.e
            print(f"  -> Fetched {len(career_df)} seasons (up to {max_fetched_se

        except Exception as e:
            print(f"  -> ERROR fetching data for {player_name}: {e}")

        # --- IMPORTANT: Sleep to respect API rate limits (1 second pause) ---
        time.sleep(1)

    # --- Step 3: Consolidate and Save Data ---
    if all_stats_dfs:
        final_df = pd.concat(all_stats_dfs, ignore_index=True)
        # Reorder columns for better readability
        cols = ['PLAYER_NAME', 'PLAYER_ID', 'SEASON_ID'] + [col for col in final
        final_df = final_df[cols]

        # Save the combined results to a new CSV file
        output_filename = f'nba_player_stats_from_{start_season_str}_to_{end_sea
        #final_df.to_csv(output_filename, index=False)
        print("-" * 50)
        print(f"Success! Fetched stats for {len(player_data)} players and saved
        print(f"Data starts at {start_season_str} or the player's true start sea
        return final_df
    else:
        print("Failed to fetch any player data.")
        return pd.DataFrame()
```

In [5]:
```
# --- EXECUTION EXAMPLE ---
# updated_nba_stats = get_player_stats_from_api('nba_5seasons_final_with_nulls.c
# print(updated_nba_stats.head())
```

Z-score calculation for all 9 categories and aggregation

In [6]:
```
df = pd.read_csv("nba_player_stats_from_2020-21_to_2024-25.csv")
#print(df.head())
#print(df.info())
#print(df.nunique())
#df21 = pd.read_excel("datasets\\sportsref_202021.xlsx")
#print(df21.head())
```

In [7]:
```
# --- Step 1: Map columns ---
pts_col = "PTS"
trb_col = "REB"
```

```
ast_col = "AST"
stl_col = "STL"
blk_col = "BLK"
threepm_col = "FG3M"
fg_pct_col = "FG_PCT"
ft_pct_col = "FT_PCT"
tov_col = "TOV"
gp_col = "GP"
fgm_col = "FGM"
fga_col = "FGA"
ftm_col = "FTM"
fta_col = "FTA"
min_col = "MIN"
oreb_col = "OREB"
dreb_col = "DREB"
pf_col = "PF"
fg3a_col = "FG3A"


# --- Step 2: Convert totals → per-game if needed ---
def per_game_if_needed(series, gp_series):
    """If stats look like totals (big numbers), divide by GP."""
    if series.max() > 50:  # heuristic
        return series / gp_series
    return series

for col in [pts_col, trb_col, ast_col, stl_col, blk_col, threepm_col, tov_col, f
    df[col] = per_game_if_needed(df[col], df[gp_col])

# --- Step 3: Clean percentages ---
# Ensure FG% and FT% are fractions [0,1]
if df[fg_pct_col].max() > 1.5:
    df[fg_pct_col] = df[fg_pct_col] / 100
if df[ft_pct_col].max() > 1.5:
    df[ft_pct_col] = df[ft_pct_col] / 100

# --- Step 4: Apply shrinkage to FG% and FT% ---
# This reduces noise for players with very few attempts.
m_fg, m_ft = 100, 50  # shrinkage strength (hyperparameters)
league_fg_mean = df[fg_pct_col].mean()
league_ft_mean = df[ft_pct_col].mean()

df["fg_shrunk"] = (df[fg_pct_col]*df[fga_col] + m_fg*league_fg_mean) / (df[fga_c
df["ft_shrunk"] = (df[ft_pct_col]*df[fta_col] + m_ft*league_ft_mean) / (df[fta_c

# --- Step 5: Build category list ---
cats = {
    "PTS": pts_col,
    "REB": trb_col,
    "AST": ast_col,
    "STL": stl_col,
    "BLK": blk_col,
    "FG3M": threepm_col,
    "FG%": "fg_shrunk",
    "FT%": "ft_shrunk",
    "TOV": tov_col,
}

# Step 6: Compute z-scores within each season and category
for name, col in cats.items():
```

```
    def zscore(x):
        mean, std = x.mean(), x.std(ddof=0)
        if std == 0:
            return (x - mean)  # avoid div/0, all values same
        if name == "TOV":  # turnovers - lower is better
            return (mean - x) / std
        else:
            return (x - mean) / std

    df[f"z_{name}"] = df.groupby("SEASON_ID")[col].transform(zscore)

# Step 7: Aggregate fantasy score (average instead of sum)
z_cols = [f"z_{c}" for c in cats.keys()]
df["fantasy_z_9cat"] = df[z_cols].sum(axis=1)

# --- Step 8: Save ---
#df.to_csv("nba_player_stats_from_2020-21_to_2024-25_with_z_12.csv", index=False

# Show top 10
print(df[["PLAYER_NAME", "SEASON_ID", "fantasy_z_9cat"] + z_cols].sort_values("f
```

	PLAYER_NAME	SEASON_ID	fantasy_z_9cat	z_PTS	z_REB	z_AST	z_STL	z_BLK	z_FG3M	z_FG%	z_FT%	z_TOV
1229	Joel Embiid	2023-24	17.604072	3.656983	3.010286	1.685541	1.281003	3.143920	0.304156	2.326020	5.644804	-3.448640
1887	Nikola Jokić	2024-25	16.588658	2.751711	3.567014	3.898353	2.511163	0.518862	0.772387	3.766506	1.086069	-2.283407
2177	Shai Gilgeous-Alexander	2024-25	15.557988	3.202402	0.370775	1.941280	2.291185	1.498284	0.953734	2.001015	4.579718	-1.280405
2176	Shai Gilgeous-Alexander	2023-24	15.477930	2.974290	0.723238	1.987359	3.392198	1.135660	0.173906	2.348856	4.059659	-1.317236
1228	Joel Embiid	2022-23	15.127221	3.400378	2.720204	0.962296	0.916321	3.367671	-0.141756	2.699813	3.936111	-2.733818
1884	Nikola Jokić	2021-22	14.980551	2.821513	4.128773	2.885343	1.983123	1.099478	0.220316	3.606670	1.370675	-3.135338
98	Anthony Davis	2023-24	14.884312	2.184145	3.684492	0.593435	1.327012	4.777216	-0.803491	2.673008	1.679907	-1.231413
1886	Nikola Jokić	2023-24	14.825487	2.435531	3.563606	3.413305	1.763705	1.053784	-0.064659	3.667408	1.371855	-2.379049
1469	Kevin Durant	2022-23	14.188789	2.898007	1.232113	1.556976	0.425163	2.749699	0.783754	2.896927	4.457499	-2.811350
1885	Nikola Jokić	2022-23	14.154193	2.125983	3.452237	3.881011	1.630178	0.701320	-0.337924	4.232031	1.394694	-2.925337

Originally I added with another data frame the position of each player. Instead of running the code I will just add here the finalize dataframe In addition - in this code cell I merged between the 3 seasons dataframe to the 2021-22 dataframe

In [8]:
```
def add_position_to_existing_data(data_filepath):
    """
    Reads an existing dataset (which must contain a 'PLAYER_NAME' column),
    fetches the primary position for each player using the NBA API, and
    merges it into the dataset under the column name 'POS'.

    Args:
        data_filepath (str): Path to the existing CSV file to be updated.

    Returns:
```

```
    pd.DataFrame: The merged DataFrame.
"""
print(f"--- Starting POS data lookup for file: {data_filepath} ---")
try:
    df_existing = pd.read_csv(data_filepath)
except FileNotFoundError:
    print(f"Error: File not found at {data_filepath}")
    return pd.DataFrame()
except KeyError:
    print(f"Error: The file {data_filepath} must contain a 'PLAYER_NAME' col
    return pd.DataFrame()

# 1. Identify unique players from the existing data
player_names = df_existing['PLAYER_NAME'].unique()
print(f"Found {len(player_names)} unique players to look up.")

# 2. Get NBA IDs for lookup
nba_players = players.get_players()
player_id_map = {player['full_name']: player['id'] for player in nba_players

# Create a temporary DataFrame to hold player lookup data
lookup_df = pd.DataFrame(player_names, columns=['PLAYER_NAME'])
lookup_df['Player_ID'] = lookup_df['PLAYER_NAME'].map(player_id_map)

# Handle unmatched players
unmatched_players = lookup_df[lookup_df['Player_ID'].isna()]
if not unmatched_players.empty:
    print(f"Warning: Could not find NBA ID for {len(unmatched_players)} play
    lookup_df = lookup_df.dropna(subset=['Player_ID']).copy()

lookup_df['Player_ID'] = lookup_df['Player_ID'].astype(int)

# 3. Fetch Player Position (POS)
player_info_list = []
print("--- Fetching Player Position (POS) from NBA API ---")

for index, row in lookup_df.iterrows():
    player_id = row['Player_ID']
    player_name = row['PLAYER_NAME']

    try:
        # Query the CommonPlayerInfo endpoint
        info = commonplayerinfo.CommonPlayerInfo(player_id=player_id)
        info_df = info.get_data_frames()[0]

        info_dict = {
            'Player_ID': player_id,
            'PLAYER_NAME': player_name,
            'POS': info_df.loc[0, 'POSITION'], # Using 'POS' as the final co
        }
        player_info_list.append(info_dict)

    except Exception as e:
        print(f"  -> ERROR fetching info for {player_name}: {e}")

    # Short sleep to prevent rate limiting
    time.sleep(0.5)

position_df = pd.DataFrame(player_info_list).drop(columns=['Player_ID'])
```

```
    # 4. Merge and Save
    print("--- Merging Position Data and Saving New File ---")

    # Merge the position data (POS) onto the existing DataFrame using PLAYER_NAM
    # We drop any pre-existing 'POS' or 'POSITION' columns to ensure a clean upd
    df_merged = pd.merge(
        df_existing.drop(columns=['POS', 'POSITION', 'ROSTERSTATUS'], errors='ig
        position_df,
        on='PLAYER_NAME',
        how='left'
    )

    # Prepare output filename
    output_filename = data_filepath.replace('.csv', '_with_POS.csv')

    # Handle the case where the existing file might already be the output file
    if output_filename == data_filepath:
        output_filename = data_filepath.replace('.csv', '_POS_only.csv')

    df_merged.to_csv(output_filename, index=False)

    print("-" * 50)
    print(f"Success! Added POS data and saved the new file to '{output_filename}
    print(f"Total rows in new file: {len(df_merged)}")
    return df_merged
```

In [9]:
```
# --- EXECUTION EXAMPLE ---
# 'nba_player_stats_from_2020_21_to_2024_25.csv', you would call:
#updated_data_with_pos = add_position_to_existing_data('nba_player_stats_from_20
#print(updated_data_with_pos.head())
```

In [10]:
```
#Loading the final data frame
df_first = pd.read_csv("nba_player_stats_from_2020-21_to_2024-25_with_z.csv")
df_first.rename(columns = {'FG_PCT':'FG%', 'FT_PCT':'FT%'},inplace = True)
df_first.drop(columns = ['Rk', 'TEAM_ABBREVIATION', "3PA", "3P%", "2PA", "2P%",
df.dropna(subset = "PTS", inplace = True)
#print("Sample from the model's dataframe \n")
df_first = df_first.drop_duplicates(subset=['PLAYER_NAME', 'SEASON_ID'], keep='f

#print(df_first.info())

#filtering only on player who played 50 games or more across the last 5 seasons
player_total_gp = df_first.groupby('PLAYER_NAME')['GP'].sum().reset_index(name='
# Merge the total back into the merged DataFrame
df_first = df_first.merge(player_total_gp, on='PLAYER_NAME', how='left')
# Filter for players who meet the total GP minimum
df = df_first[df_first['Total_Career_GP_in_Range'] >= 50].copy()
df = df[df['MIN'] >= 6].copy()
# Drop the temporary column before saving
df = df.drop(columns=['Total_Career_GP_in_Range'])

print(df.sample(6))
#print(df[df["PLAYER_NAME"] == "Victor Wembanyama"])
#print(df.info())
#print(df[df[['ft_shrunk']].isnull().any(axis=1)])
```

```
            PLAYER_NAME  PLAYER_ID SEASON_ID  LEAGUE_ID       TEAM_ID  \
7          Aaron Gordon   203932.0   2024-25        0.0  1.610613e+09
974   Jonas Valančiūnas   202685.0   2022-23        0.0  1.610613e+09
2133         Yves Missi        NaN   2024-25        NaN           NaN
537      Doug McDermott   203926.0   2023-24        0.0  0.000000e+00
581        Enes Freedom   202683.0   2020-21        0.0  1.610613e+09
1103    Kelly Oubre Jr.  1626162.0   2020-21        0.0  1.610613e+09

      PLAYER_AGE    GP    GS        MIN       FGM  ...     z_REB     z_AST  \
7           29.0  51.0  42.0  28.372549  5.176471  ...  0.396319  0.512934
974         31.0  79.0  79.0  24.911392  5.645570  ...  2.594681 -0.245041
2133        20.0  73.0  67.0  26.800000  3.700000  ...  1.779801 -0.440451
537         32.0  64.0   0.0  14.125000  1.921875  ... -1.219772 -0.601258
581         29.0  72.0  35.0  24.402778  4.708333  ...  2.950952 -0.562971
1103        25.0  55.0  50.0  30.690909  5.781818  ...  0.865610 -0.478639

         z_STL     z_BLK    z_FG3M     z_FG%     z_FT%     z_TOV  \
7    -0.584545 -0.387484  0.408866  1.217882  0.658963 -0.316742
974  -1.092977  0.549017 -0.683361  1.596114  0.685882 -1.016591
2133 -0.458553  2.115325 -1.214270  1.044244 -1.649859  0.082221
537  -1.101111 -0.981184  0.406838 -0.314347 -0.428975  0.879850
581  -0.565646  0.569613 -1.198941  2.060631  0.027456  0.102287
1103  0.920025  0.806277  0.591838 -0.578452 -0.911102 -0.125738

      fantasy_z_9cat  TEAM_ABBREVIATION
7           2.635650                NaN
974         3.033817                NaN
2133        1.148367                NOP
537        -4.015344                NaN
581         3.589168                NaN
1103        1.935358                NaN

[6 rows x 41 columns]
```

Used the cell below originally to merge between the datasets, after megind I saved it on the same file path

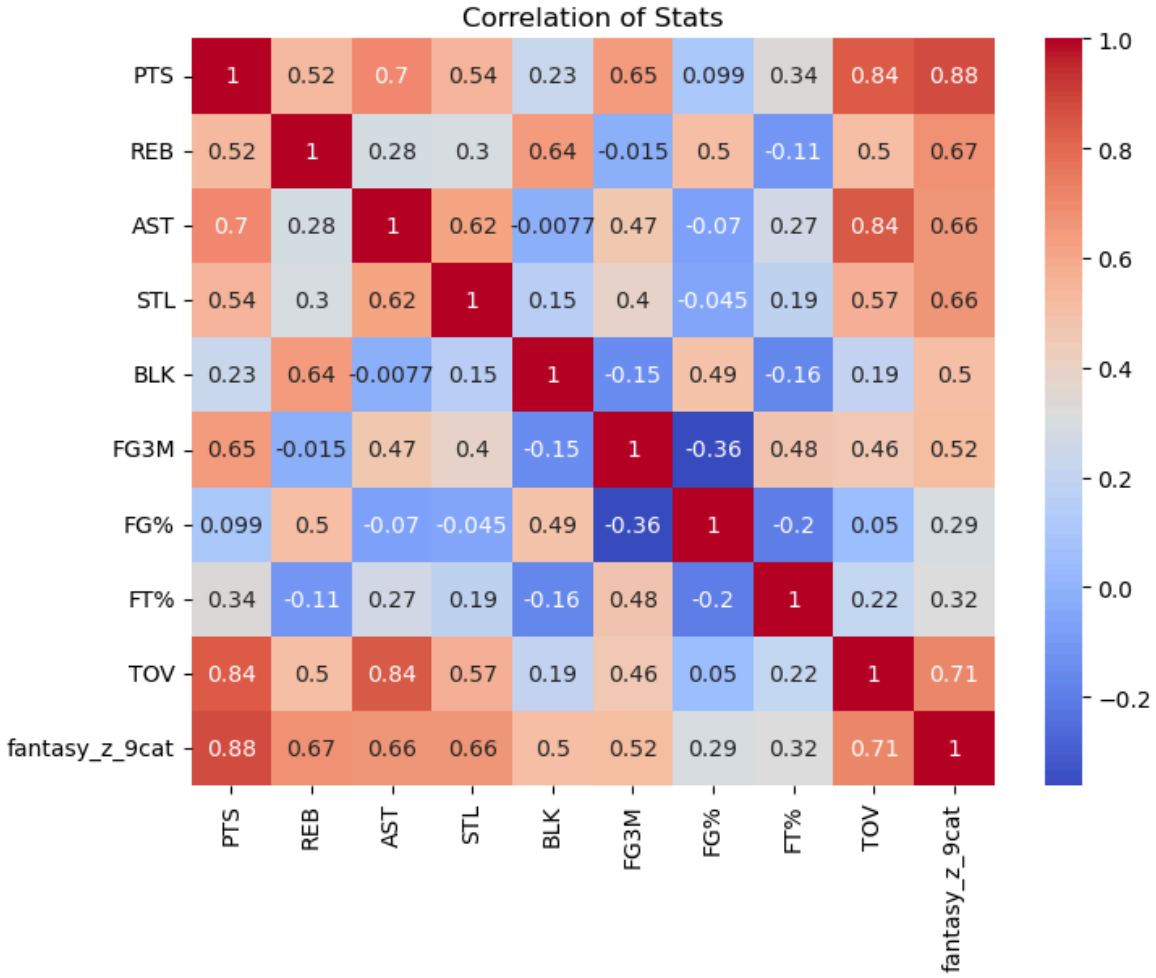
df25 = pd.read_excel("datasets\\Basketball_ref_2425.xlsx") #print(df25.info()) df24 = pd.read_excel("datasets\\Basketball_ref_2324.xlsx") #print(df24.info()) concat_df = pd.concat([df ,df24, df25], ignore_index = True) final_df = concat_df.drop_duplicates(subset=['PLAYER_NAME', 'SEASON_ID'], keep='first') print(final_df.sample(7)) print(final_df[final_df["PLAYER_NAME"] == "Victor Wembanyama"]) print(final_df.info())

Adding missing players as rookies and sophmors that got omitted in the process

EDA

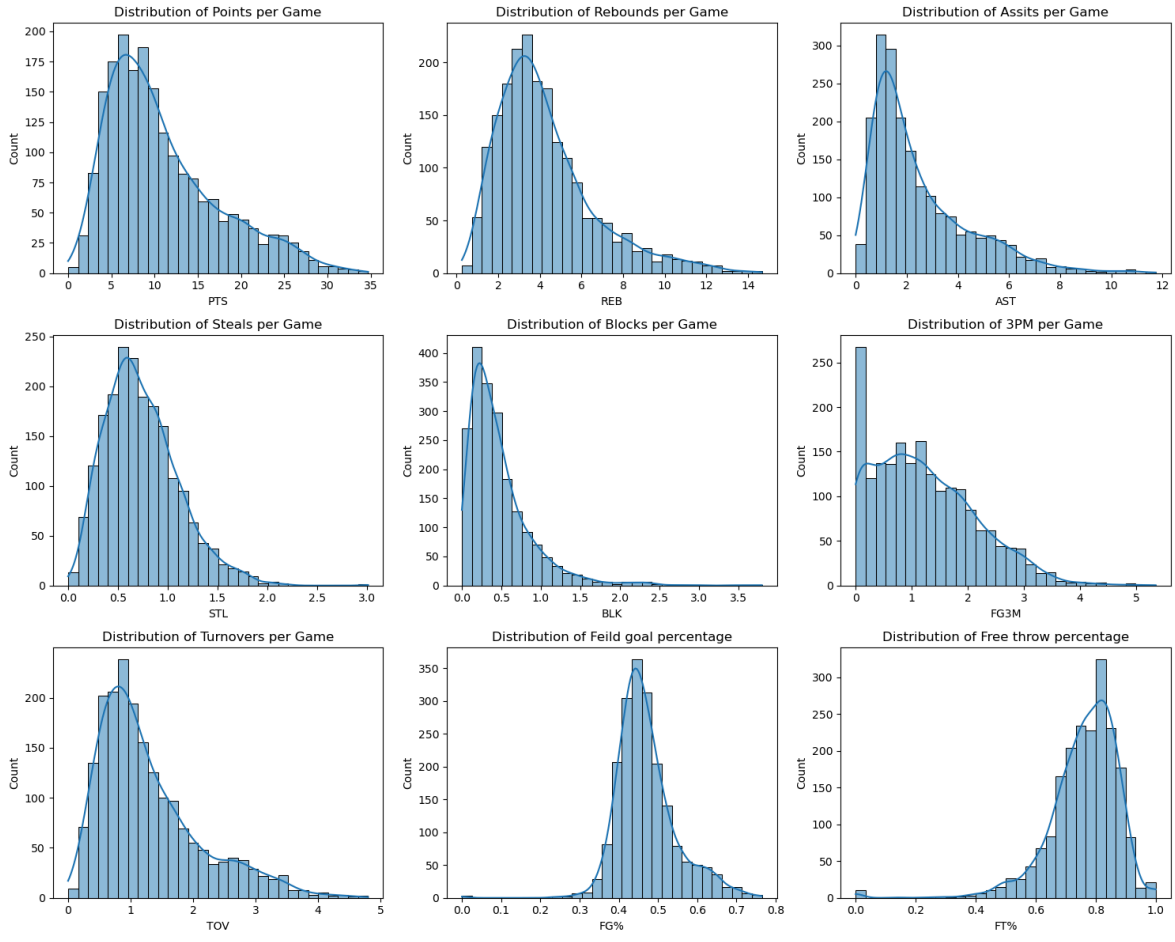
Exploring the different distributions of the relevant features for 9-CAT H2H format

In [11]:
```
stats_cols = ["PLAYER_NAME", "SEASON_ID", "GP", "PTS", "REB", "AST", "STL", "BLK
df_stats = df[stats_cols].copy()

plt.figure(figsize=(8,6))
sns.heatmap(df_stats.drop(columns=["PLAYER_NAME", "SEASON_ID", "GP"], inplace=Fa
plt.title("Correlation of Stats")
plt.show()
```



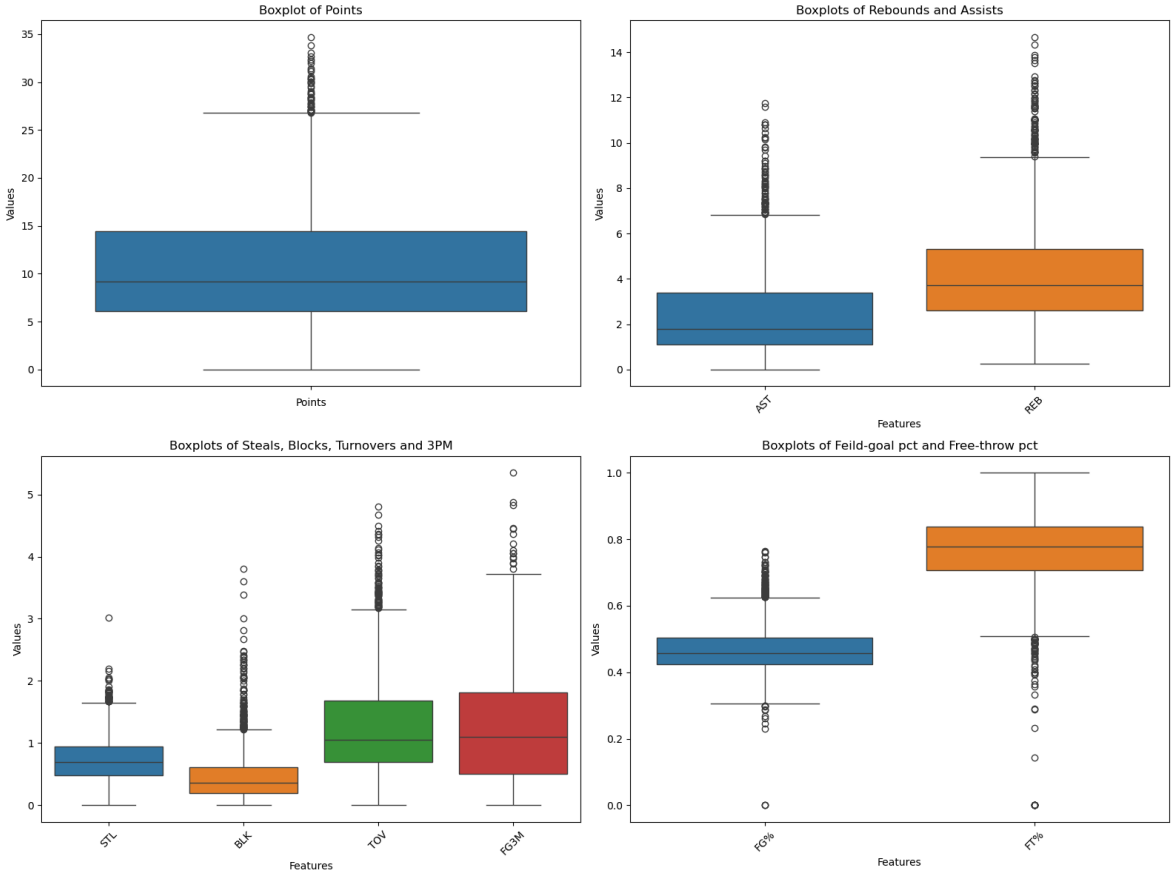
In [12]:
```
fig, axs = plt.subplots(3,3, figsize=(15, 12))

#Histogram of Points
#plt.figure(figsize=(6,4))
sns.histplot(df_stats["PTS"], bins=30, kde=True, ax=axs[0,0])
axs[0,0].set_title("Distribution of Points per Game")

#Histogram of Rebounds
#plt.figure(figsize=(6,4))
sns.histplot(df_stats["REB"], bins=30, kde=True, ax=axs[0,1])
axs[0,1].set_title("Distribution of Rebounds per Game")

#Histogram of Assits
#plt.figure(figsize=(6,4))
sns.histplot(df_stats["AST"], bins=30, kde=True, ax=axs[0,2])
axs[0,2].set_title("Distribution of Assits per Game")

#Histogram of Steals
#plt.figure(figsize=(6,4))
sns.histplot(df_stats["STL"], bins=30, kde=True, ax=axs[1,0])
axs[1,0].set_title("Distribution of Steals per Game")

#Histogram of Blocks
#plt.figure(figsize=(6,4))
sns.histplot(df_stats["BLK"], bins=30, kde=True, ax=axs[1,1])
axs[1,1].set_title("Distribution of Blocks per Game")

#Histogram of 3PM
#plt.figure(figsize=(6,4))
```

```
sns.histplot(df_stats["FG3M"], bins=30, kde=True, ax=axs[1,2])
axs[1,2].set_title("Distribution of 3PM per Game")

#Histogram of Turnovers
#plt.figure(figsize=(6,4))
sns.histplot(df_stats["TOV"], bins=30, kde=True, ax=axs[2,0])
axs[2,0].set_title("Distribution of Turnovers per Game")

#Histogram of Feild Goal percentage
#plt.figure(figsize=(6,4))
sns.histplot(df_stats["FG%"], bins=30, kde=True, ax=axs[2,1])
axs[2,1].set_title("Distribution of Feild goal percentage")

#Histogram of Free throw percentage
#plt.figure(figsize=(6,4))
sns.histplot(df_stats["FT%"], bins=30, kde=True, ax=axs[2,2])
axs[2,2].set_title("Distribution of Free throw percentage")

plt.tight_layout()
plt.show()
```



In [13]:
```
#need to split it to categories with high values (points, assists etc) to low va
high_value_features, medium_value_features, low_value_feaures = ["AST", "REB"],
fig, axs = plt.subplots(2, 2, figsize=(16, 12))
#Points boxplots
sns.boxplot(data=df["PTS"], ax=axs[0, 0])
axs[0,0].set_title('Boxplot of Points')
axs[0,0].set_xlabel('Points')
axs[0,0].set_ylabel('Values')
axs[0,0].tick_params(axis='x', rotation=45) # Rotate x-axis labels for clarity
```

```
#Rebounds and Assists boxplots
sns.boxplot(data=df[high_value_features], ax = axs[0,1])
axs[0,1].set_title('Boxplots of Rebounds and Assists')
axs[0,1].set_xlabel('Features')
axs[0,1].set_ylabel('Values')
axs[0,1].tick_params(axis='x', rotation=45)

#Steals, Blocks, Turnovers, 3-point made boxplots
sns.boxplot(data=df[medium_value_features], ax = axs[1,0])
axs[1,0].set_title('Boxplots of Steals, Blocks, Turnovers and 3PM')
axs[1,0].set_xlabel('Features')
axs[1,0].set_ylabel('Values')
axs[1,0].tick_params(axis='x', rotation=45)

#Feild goal percentage and free throw percentage boxplots
sns.boxplot(data=df[low_value_feaures], ax = axs[1,1])
axs[1,1].set_title('Boxplots of Feild-goal pct and Free-throw pct')
axs[1,1].set_xlabel('Features')
axs[1,1].set_ylabel('Values')
axs[1,1].tick_params(axis='x', rotation=45)

plt.tight_layout()
plt.show()

print("League average points:", df_stats["PTS"].mean(), "Median points", df_stat
print("League average rebounds:", df_stats["REB"].mean())
print("League average asstits:", df_stats["AST"].mean())
print("League average steals:", df_stats["STL"].mean())
print("League average blocks:", df_stats["BLK"].mean())
print("League average 3-points made:", df_stats["FG3M"].mean())
print("League average turnovers:", df_stats["TOV"].mean())
print("League median field-goal percentage:", df_stats["FG%"].mean())
print("League median field-goal percentage:", df_stats["FT%"].mean())
```



```
League average points: 10.953886768163883 Median points 9.2
League average rebounds: 4.246218887704603
League average asstits: 2.4824804127840165
League average steals: 0.7406439041942338
League average blocks: 0.4730731680991401
League average 3-points made: 1.2250292073940314
League average turnovers: 1.2919964135619626
League median field-goal percentage: 0.4701896813353566
League median field-goal percentage: 0.7609468892261002
```

#Top 10 Per quantative categories top_pts = df_stats.sort_values(by="PTS", ascending=False).head(10) print("Top 10 Scorers (Avg Points per Game):") print(top_pts[["PLAYER_NAME", "SEASON_ID", "PTS"]],"\n") top_reb = df_stats.sort_values(by="REB", ascending=False).head(10) print("Top 10 Rebounders (Avg Rebounds per Game):") print(top_reb[["PLAYER_NAME", "SEASON_ID", "REB"]],"\n") top_ast = df_stats.sort_values(by="AST", ascending=False).head(10) print("Top 10 Passers (Avg Assists per Game):") print(top_ast[["PLAYER_NAME", "SEASON_ID", "AST"]],"\n") top_3PM = df_stats.sort_values(by="FG3M", ascending=False).head(10) print("Top 10 3 pointers (Avg 3 points made per Game):") print(top_3PM[["PLAYER_NAME", "SEASON_ID", "FG3M"]])

Initial EDA conclusions

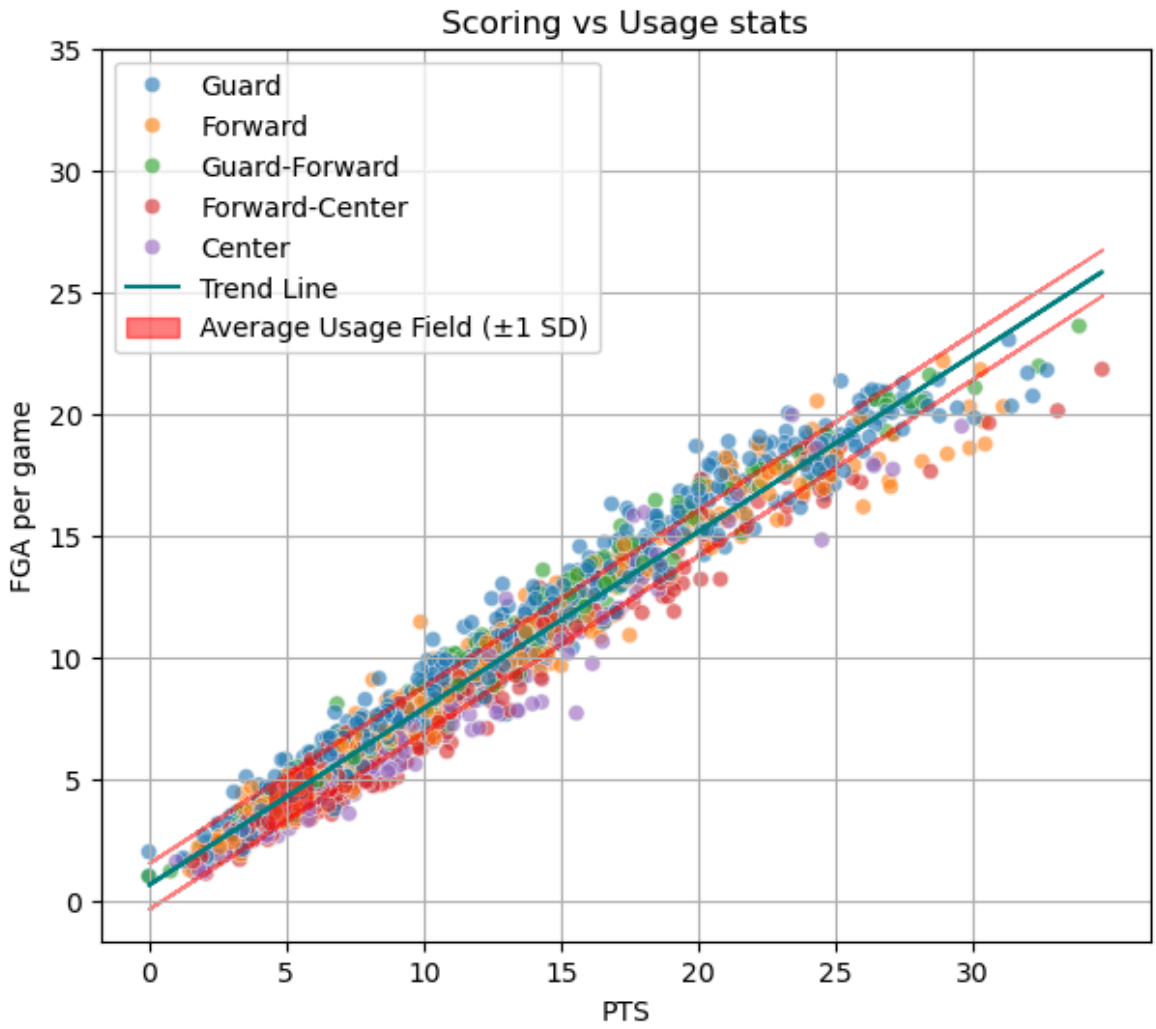
The analysis of the data reveals several key correlations and distribution patterns among the features.

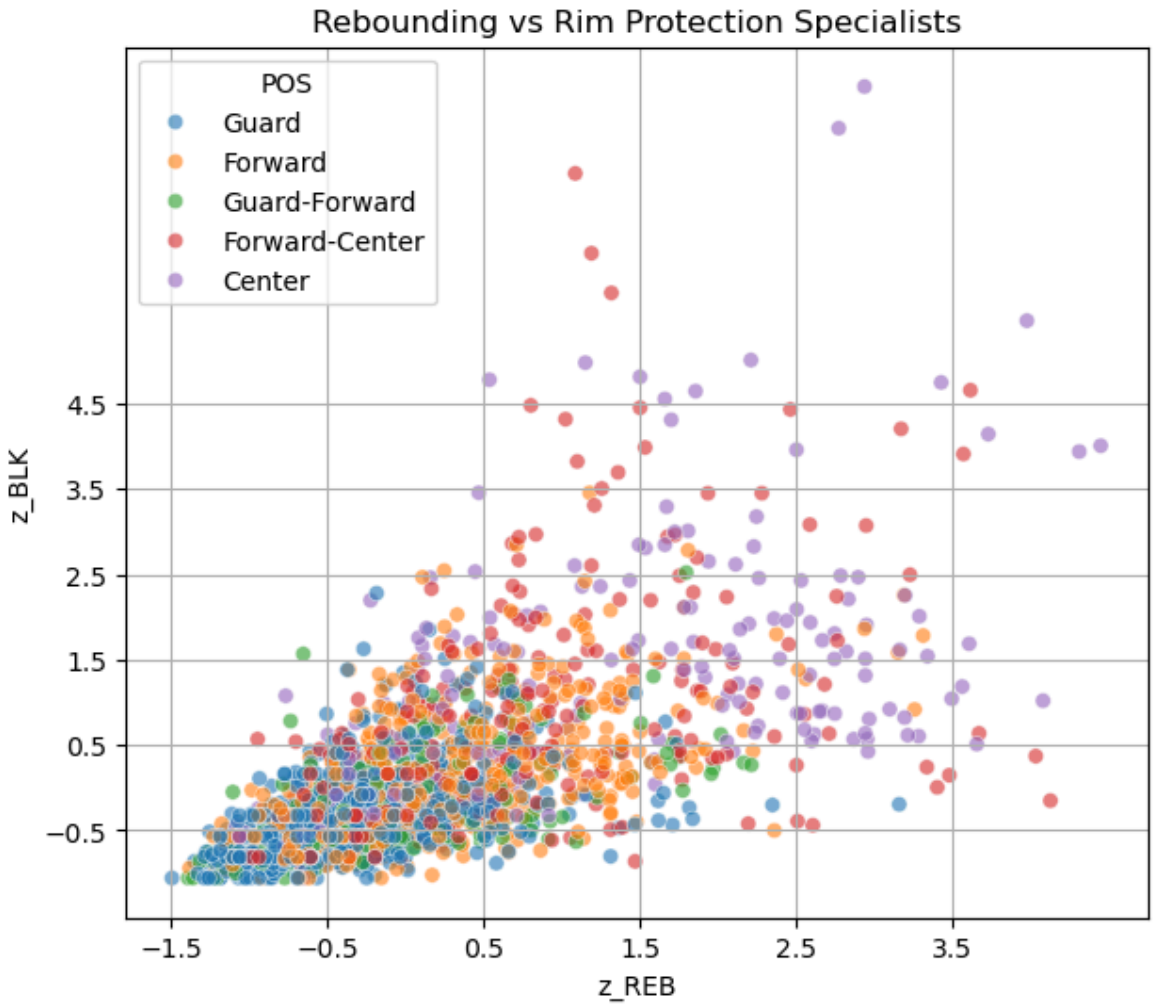
- A **strong positive correlation** is observed among points, assists, and turnovers. This suggests that players who are primary scorers and ball-handlers tend to have more opportunities for assists but also assume greater risk, which can lead to a higher frequency of turnovers.
- Rebounds and blocks also exhibit a **high positive correlation**. This is a logical relationship, as taller players, who typically play frontcourt positions, are more likely to achieve higher numbers in both categories. Furthermore, these two categories show a **positive correlation with field goal percentage**. This can be attributed to frontcourt players generally taking shots from closer to the basket, which results in a higher shooting percentage compared to players who shoot from a distance.
- Field goal percentage appears to follow a **t-distribution centered around the 47% mark** (I determined it is a t-distribution and not normal because of the wider tails). This indicates that most players' field goal percentages cluster near this average, with fewer players having extremely high or low percentages. On the other hand, Free throw percentage appears to follow a **t-distribution centered around the 75% mark** but the tails are much wider than those of the eild goal percentage stat. It means that large amount of players is considered as outliers in the FT% category.
- The distributions of points, rebounds, and assists are **right-skewed**, indicating a **heavy-tailed distribution**. This distribution pattern confirms the presence of outliers and extreme values that are significantly greater than the majority of the data. Consistent with this skew, the **mean value for each of these features is higher than the median**, a direct result of the influence of these high-value outliers pulling the mean upwards.
- Lastly, according to the correlation heatmap, the feature **Points** has a the strongest correlation with the total Z-score, while the feature **FG percentage** has the lowest correlation (though still solid) among all features.

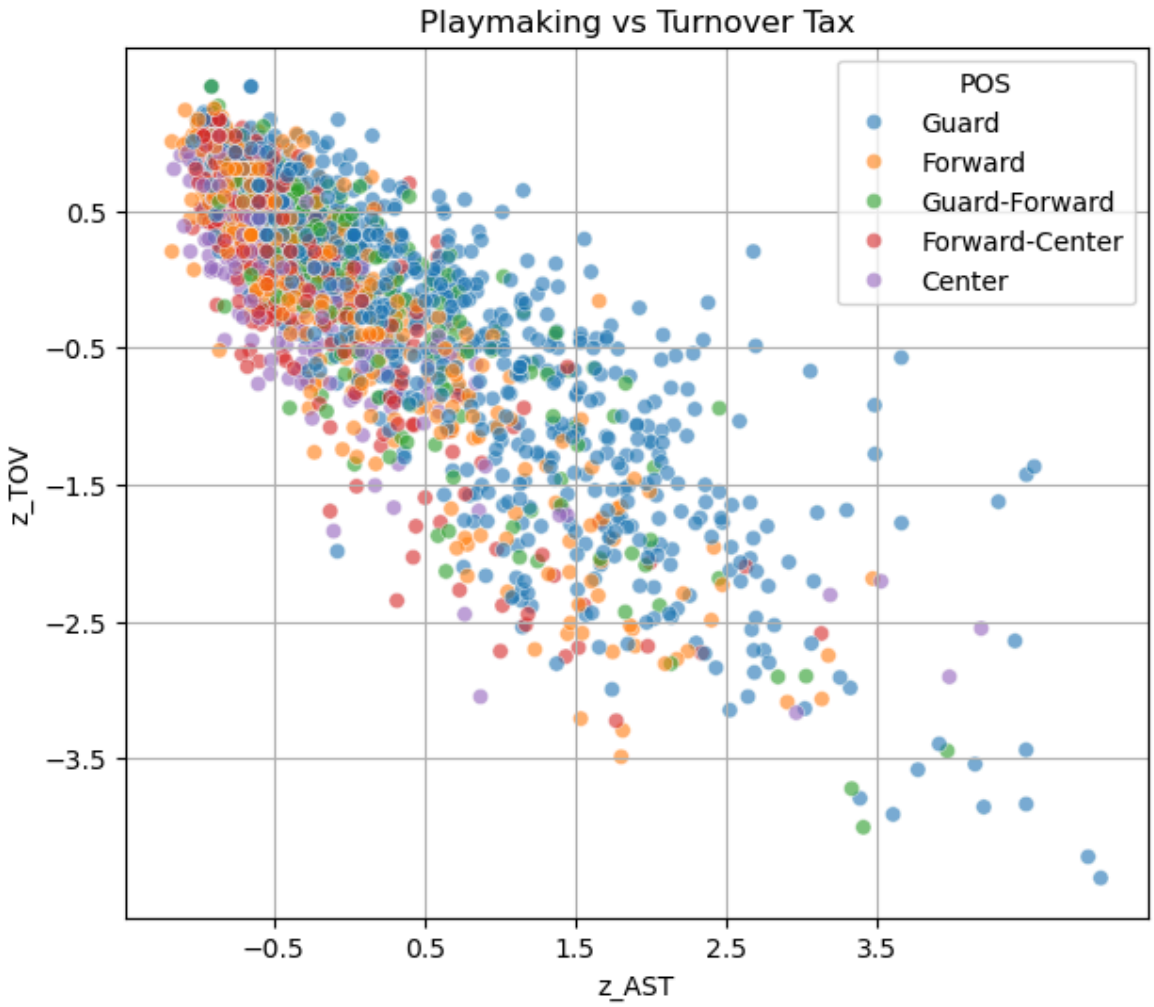
Exploring the different connections between the 9cat features

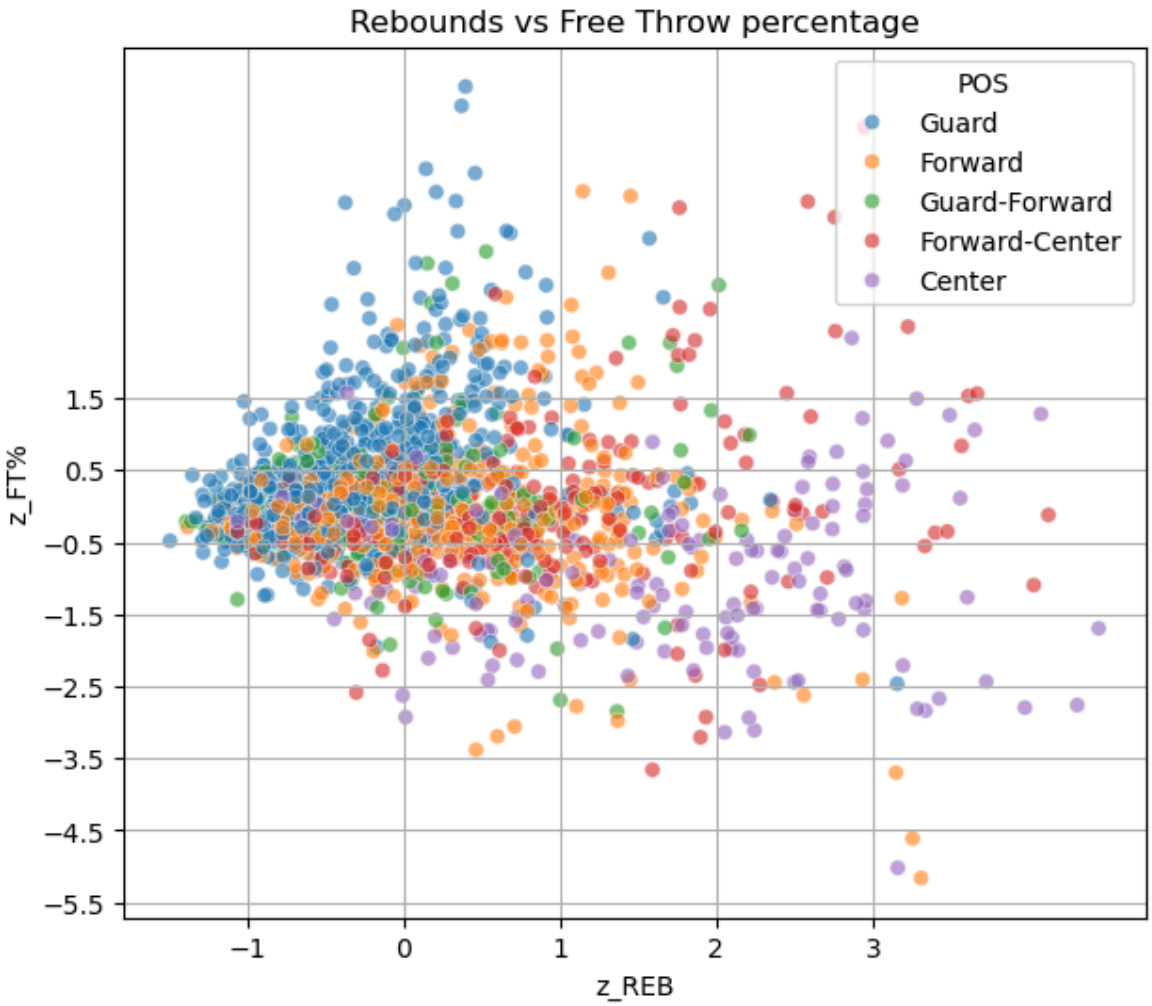
In [14]:
```
df["Total_PTS"] = df["PTS"]*df["GP"]
# Example 1: Scoring vs Assists
plt.figure(figsize=(7,6))
sns.scatterplot(x="PTS", y="FGA", hue = "POS", data=df, alpha=0.6)
plt.title("Scoring vs Usage stats")
plt.xlabel("PTS")
plt.ylabel("FGA per game")
model = LinearRegression()
X = df[['PTS']]
y = df['FGA']
model.fit(X, y)

# Plot the trend line
plt.plot(X, model.predict(X), color='teal', linestyle='-', linewidth=1.5, label=

# Add a filled area for the average usage field
# Calculate residuals and standard deviation
residuals = y - model.predict(X)
std_dev = np.std(residuals)
plt.fill_between(
    X['PTS'],
    model.predict(X) - std_dev,
    model.predict(X) + std_dev,
    color='red',
    alpha=0.5,
    label='Average Usage Field (±1 SD)'
)
plt.legend()
tick_locations = np.arange(0,35,5)
tick_y_locations = np.arange(0,40,5)
plt.xticks(tick_locations)
plt.yticks(tick_y_locations)
plt.grid(True)
plt.show()

# Example 2: Rebounding vs Blocking
plt.figure(figsize=(7,6))
sns.scatterplot(x="z_REB", y="z_BLK",hue = "POS", data=df, alpha=0.6)
plt.title("Rebounding vs Rim Protection Specialists")
plt.xlabel("z_REB")
plt.ylabel("z_BLK")
tick_locations = np.arange(-1.5,4,1)
tick_y_locations = np.arange(-0.5,5.5,1)
plt.xticks(tick_locations)
plt.yticks(tick_y_locations)
plt.grid(True)
plt.show()

# Example 3: Assists vs Turnovers (risk/reward guards)
plt.figure(figsize=(7,6))
sns.scatterplot(x="z_AST", y="z_TOV", hue = "POS", data=df, alpha=0.6)
plt.title("Playmaking vs Turnover Tax")
plt.xlabel("z_AST")
plt.ylabel("z_TOV")
tick_locations = np.arange(-0.5, 4, 1)
tick_y_locations = np.arange(-3.5,1.5,1)
plt.xticks(tick_locations)
```

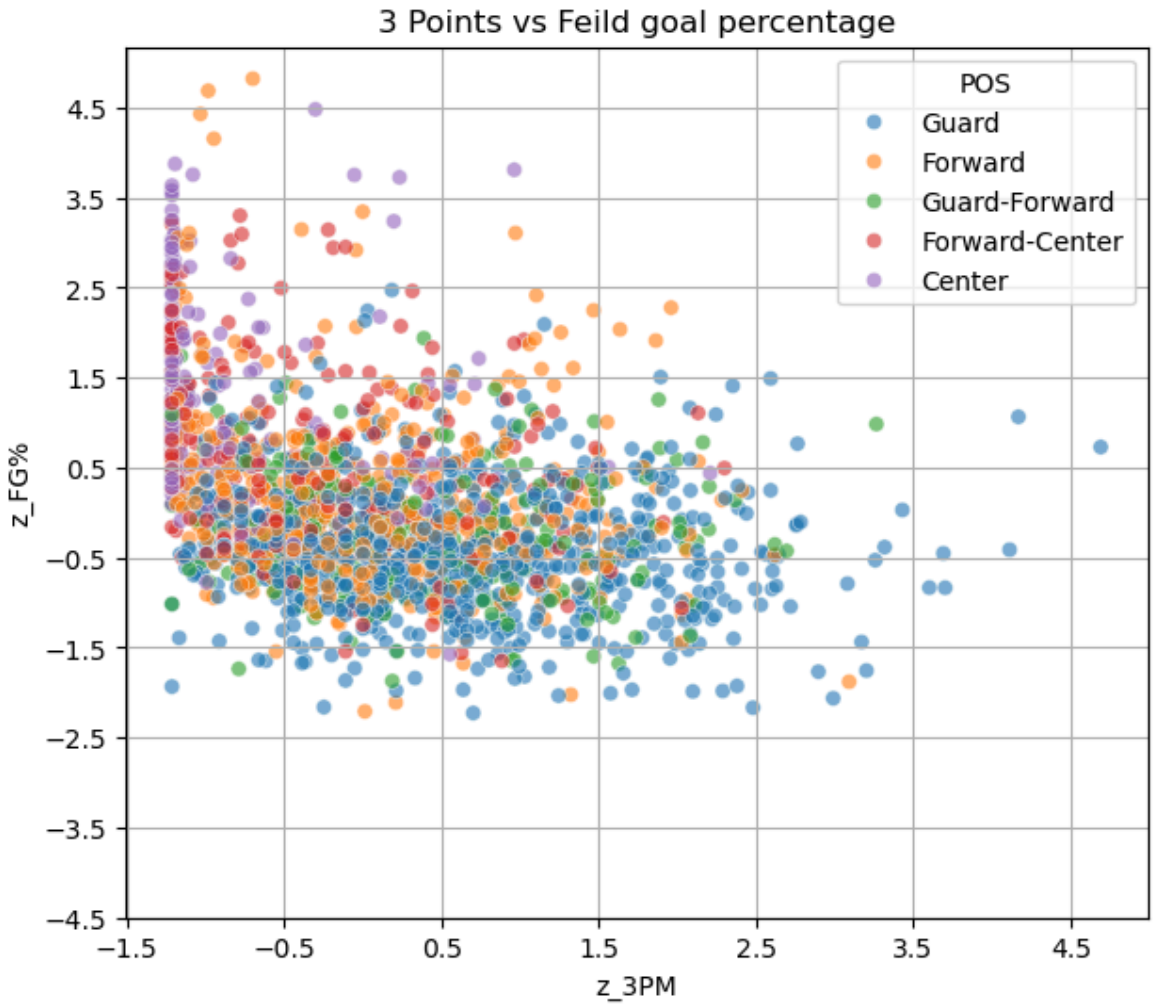
```
plt.yticks(tick_y_locations)
plt.grid(True)
plt.show()

# Example 4: Rebounds vs Free Throw percentage
plt.figure(figsize=(7,6))
sns.scatterplot(x="z_REB", y="z_FT%", hue = "POS", data=df, alpha=0.6)
plt.title("Rebounds vs Free Throw percentage")
plt.xlabel("z_REB")
plt.ylabel("z_FT%")
tick_locations = np.arange(-1, 4, 1)
tick_y_locations = np.arange(-5.5,2.5,1)
plt.xticks(tick_locations)
plt.yticks(tick_y_locations)
plt.grid(True)
plt.show()

# Example 5: 3 Points vs Feild goal percentage
plt.figure(figsize=(7,6))
sns.scatterplot(x="z_FG3M", y="z_FG%", hue = "POS", data=df, alpha=0.6)
plt.title("3 Points vs Feild goal percentage")
plt.xlabel("z_3PM")
plt.ylabel("z_FG%")
tick_locations = np.arange(-1.5, 5, 1)
tick_y_locations = np.arange(-4.5,5.5,1)
plt.xticks(tick_locations)
plt.yticks(tick_y_locations)
plt.grid(True)
plt.show()
```


Scoring vs Usage stats
Guard
Forward
Guard-Forward
Forward-Center
Center
Trend Line
Average Usage Field (±1 SD)
FGA per game
PTS
0
5
10
15
20
25
30
35


Rebounding vs Rim Protection Specialists
POS
Guard
Forward
Guard-Forward
Forward-Center
Center
z_BLK
4.5
3.5
2.5
1.5
0.5
−0.5
−1.5
−0.5
0.5
1.5
2.5
3.5
z_REB

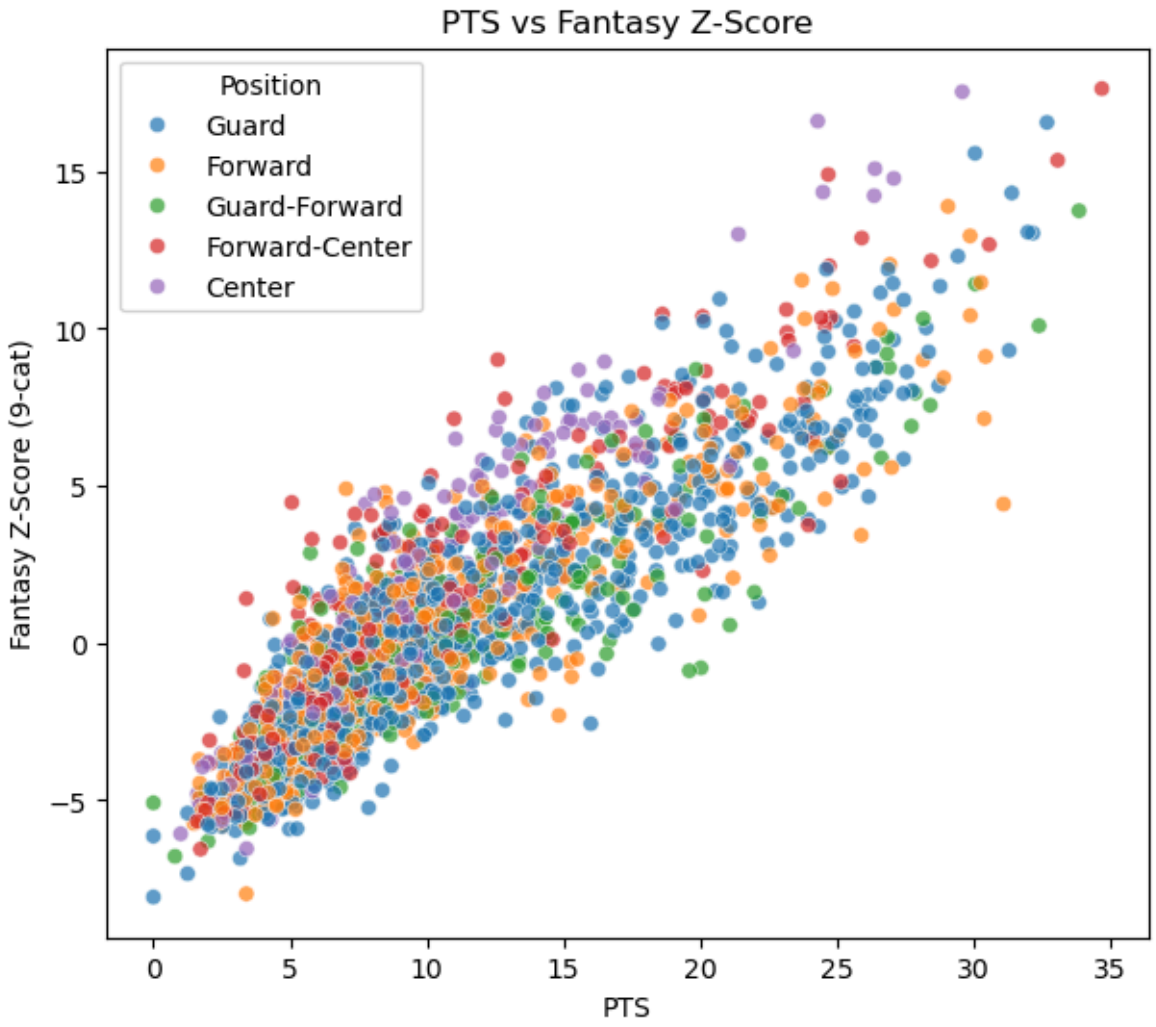

Playmaking vs Turnover Tax
POS
Guard
Forward
Guard-Forward
Forward-Center
Center
z_TOV
0.5
−0.5
−1.5
−2.5
−3.5
−0.5
0.5
1.5
2.5
3.5
z_AST

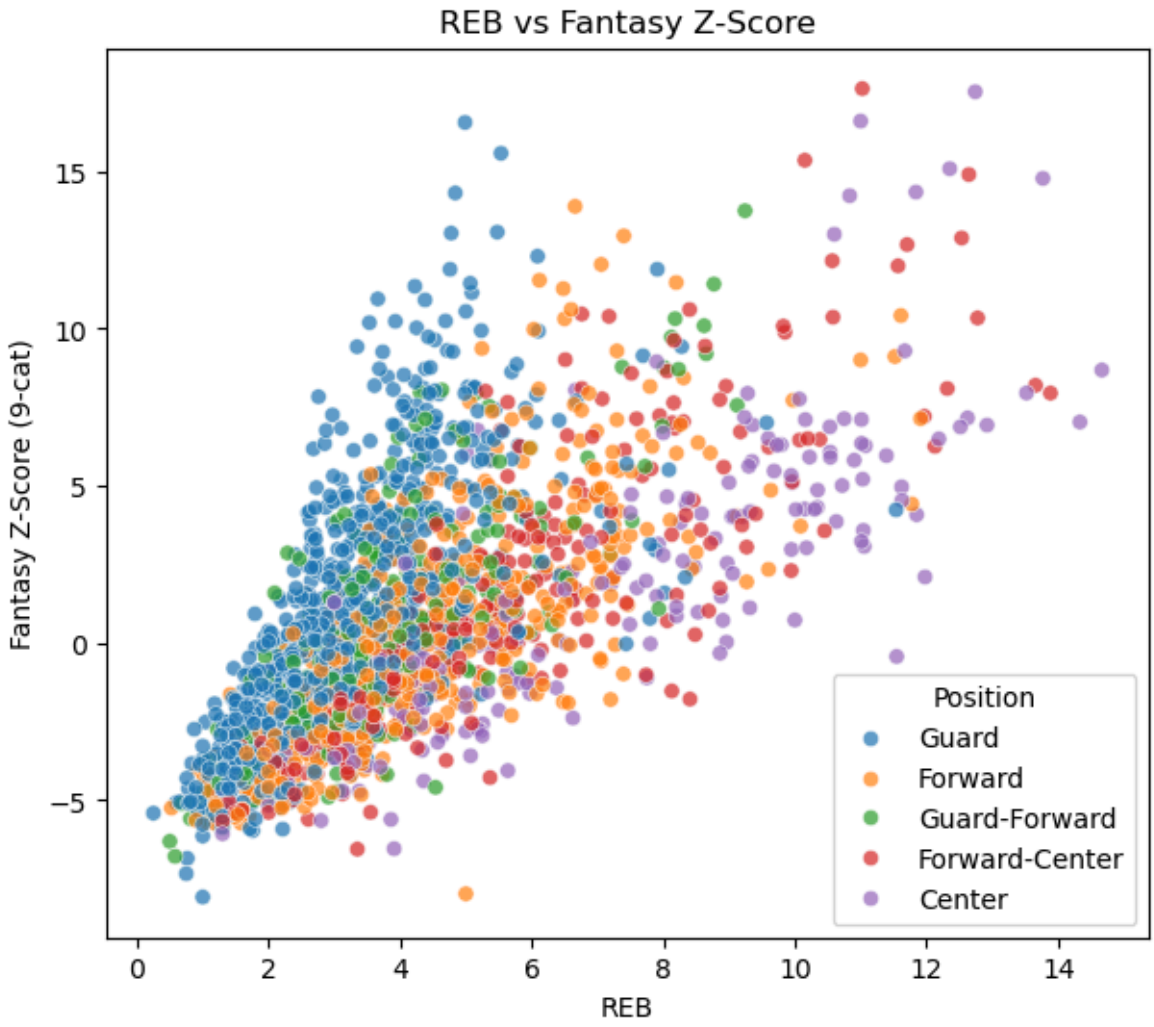

Rebounds vs Free Throw percentage
POS
Guard
Forward
Guard-Forward
Forward-Center
Center
z_FT%
1.5
0.5
−0.5
−1.5
−2.5
−3.5
−4.5
−5.5
−1
0
1
2
3
z_REB



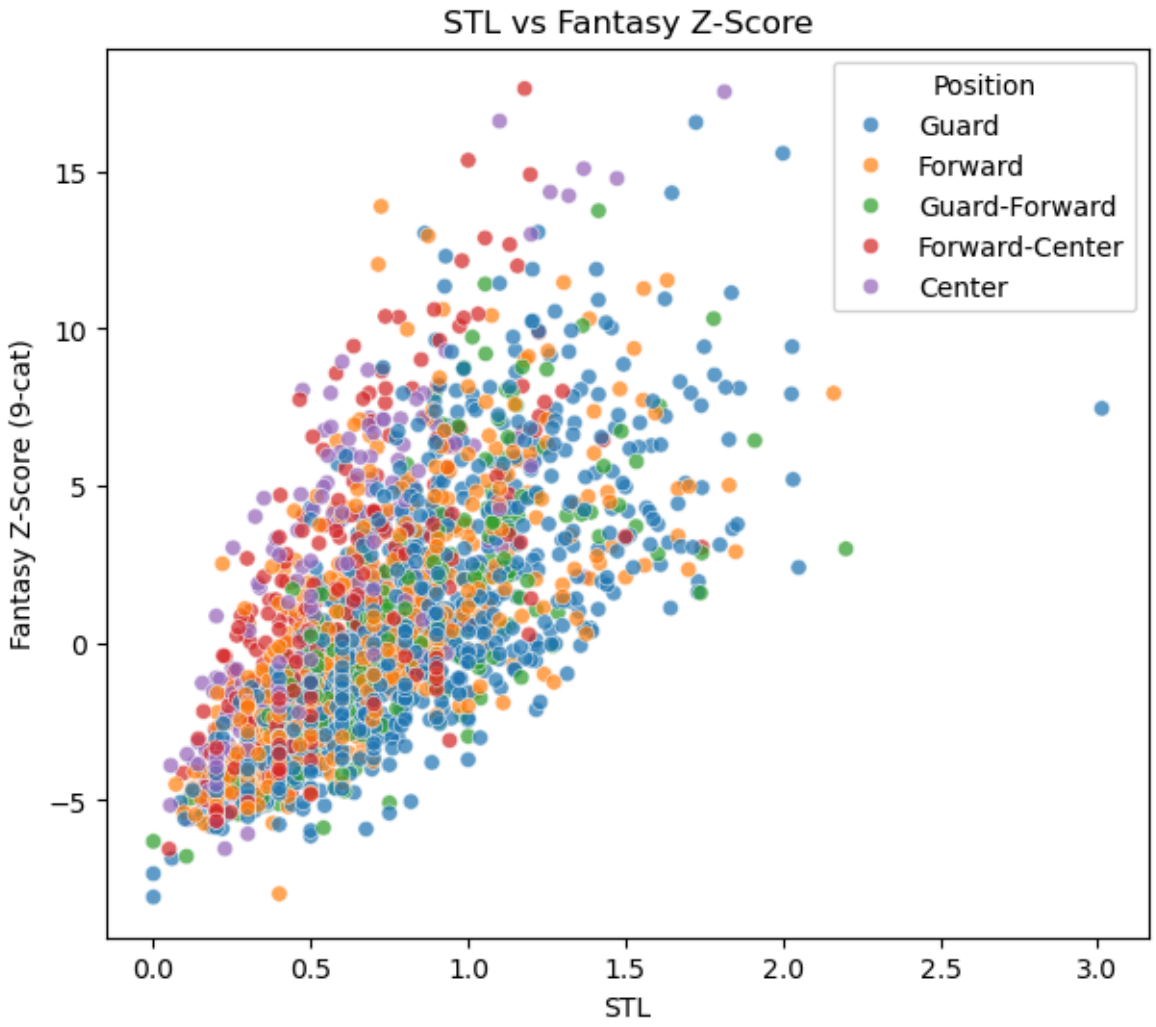
Correlation between key stats to total Z-score (fantasy_z_9cat) - Testing how well the stats predict the fantasy rank of a player.

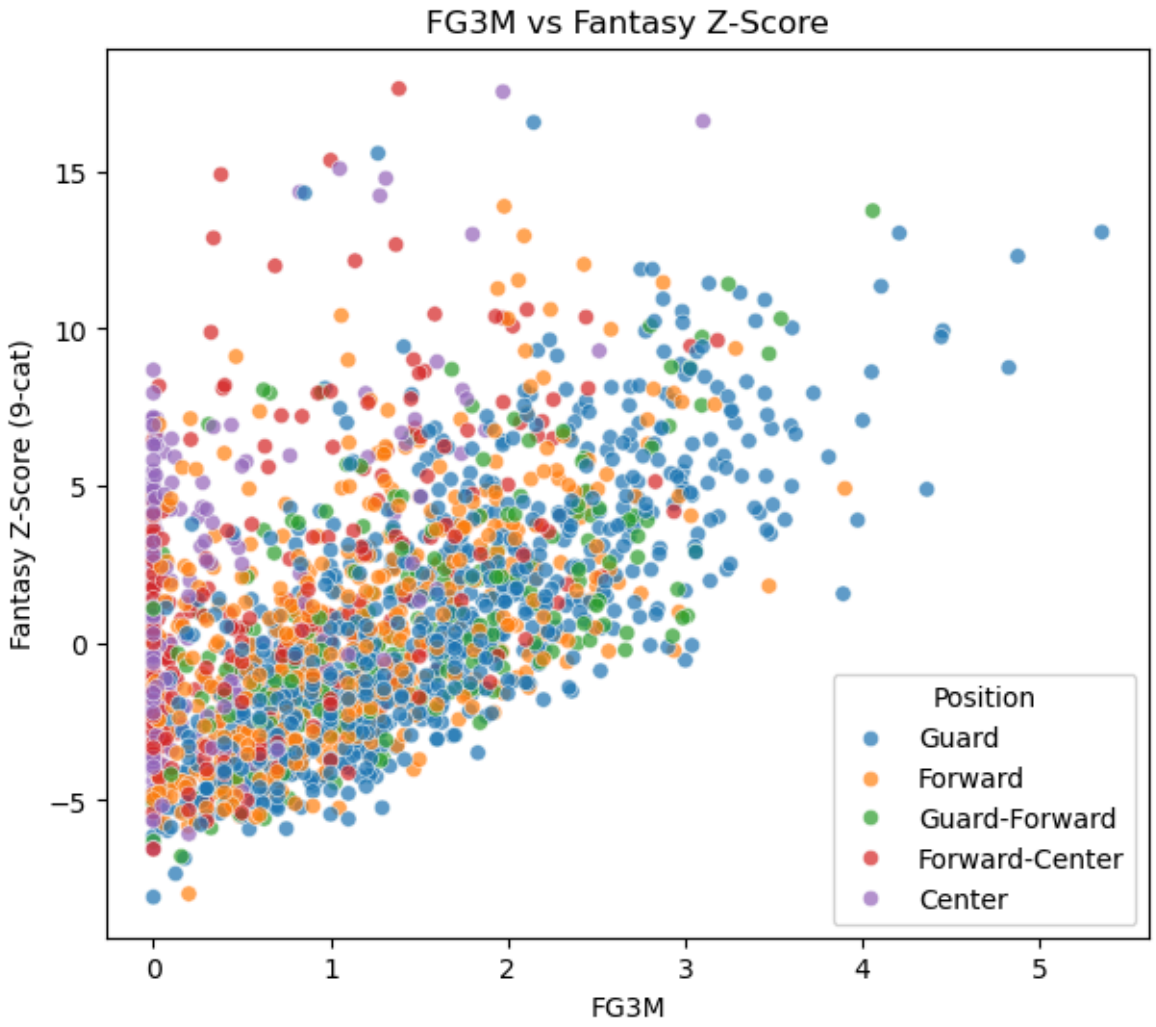
In [15]:
```
key_stats = ['PTS', 'REB', 'AST', 'STL', 'FG3M']

for stat in key_stats:
    if stat in df.columns:
        plt.figure(figsize=(7,6))
        sns.scatterplot(x=df[stat], y=df['fantasy_z_9cat'], hue=df['POS'], alpha
        plt.title(f"{stat} vs Fantasy Z-Score")
        plt.xlabel(stat)
        plt.ylabel("Fantasy Z-Score (9-cat)")
        plt.legend(title="Position")
        plt.show()
```


PTS vs Fantasy Z-Score
Position
Guard
Forward
Guard-Forward
Forward-Center
Center
Fantasy Z-Score (9-cat)
PTS
15
10
5
0
−5
0
5
10
15
20
25
30
35


REB vs Fantasy Z-Score
Fantasy Z-Score (9-cat)
REB
Position
Guard
Forward
Guard-Forward
Forward-Center
Center
15
10
5
0
−5
0
2
4
6
8
10
12
14


AST vs Fantasy Z-Score
Fantasy Z-Score (9-cat)
AST
Position
Guard
Forward
Guard-Forward
Forward-Center
Center
15
10
5
0
−5
0
2
4
6
8
10
12


STL vs Fantasy Z-Score
Position
Guard
Forward
Guard-Forward
Forward-Center
Center
Fantasy Z-Score (9-cat)
15
10
5
0
−5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
STL



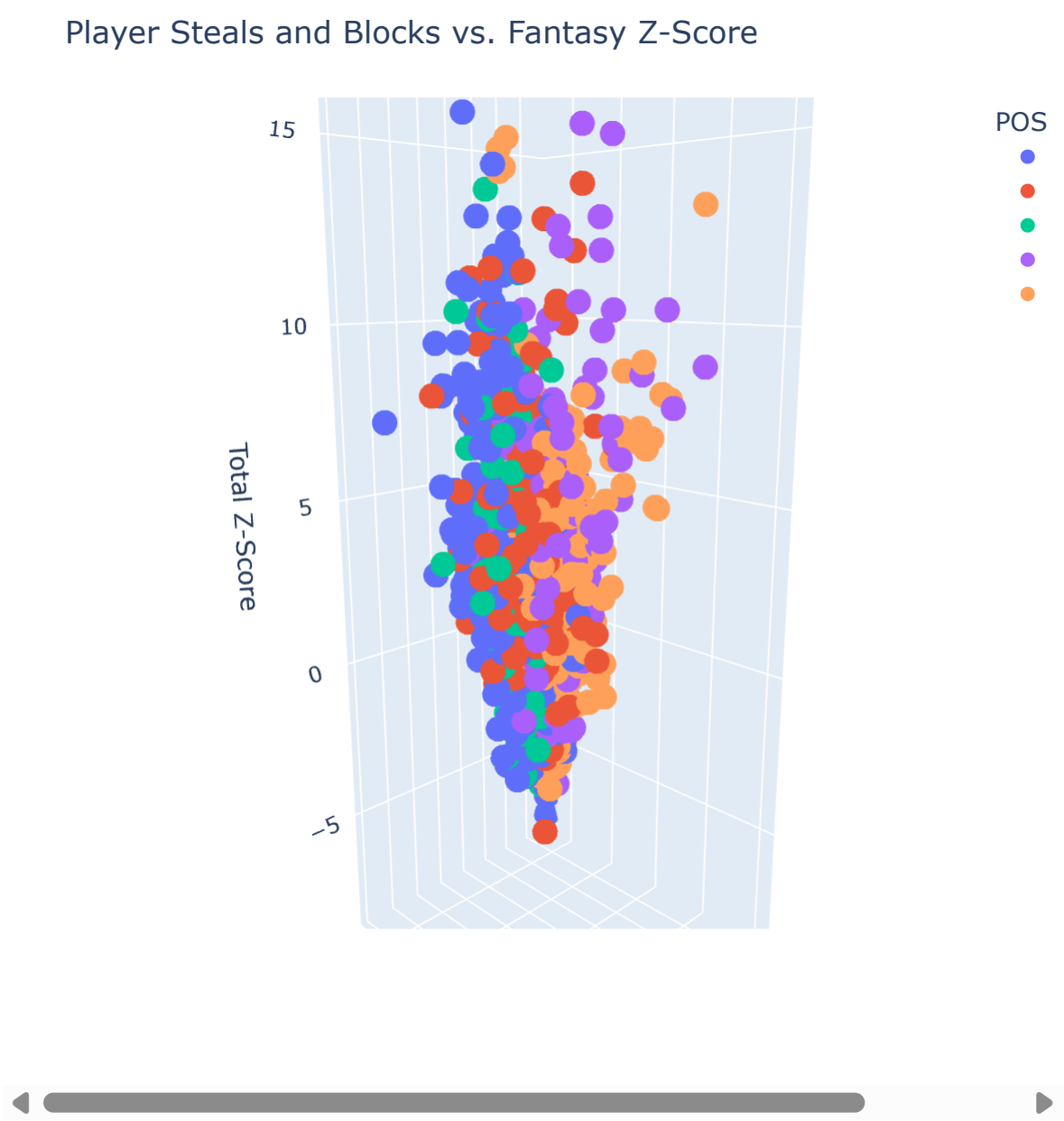
Exploring the cirrelations between the seasons stats, to check player's stability

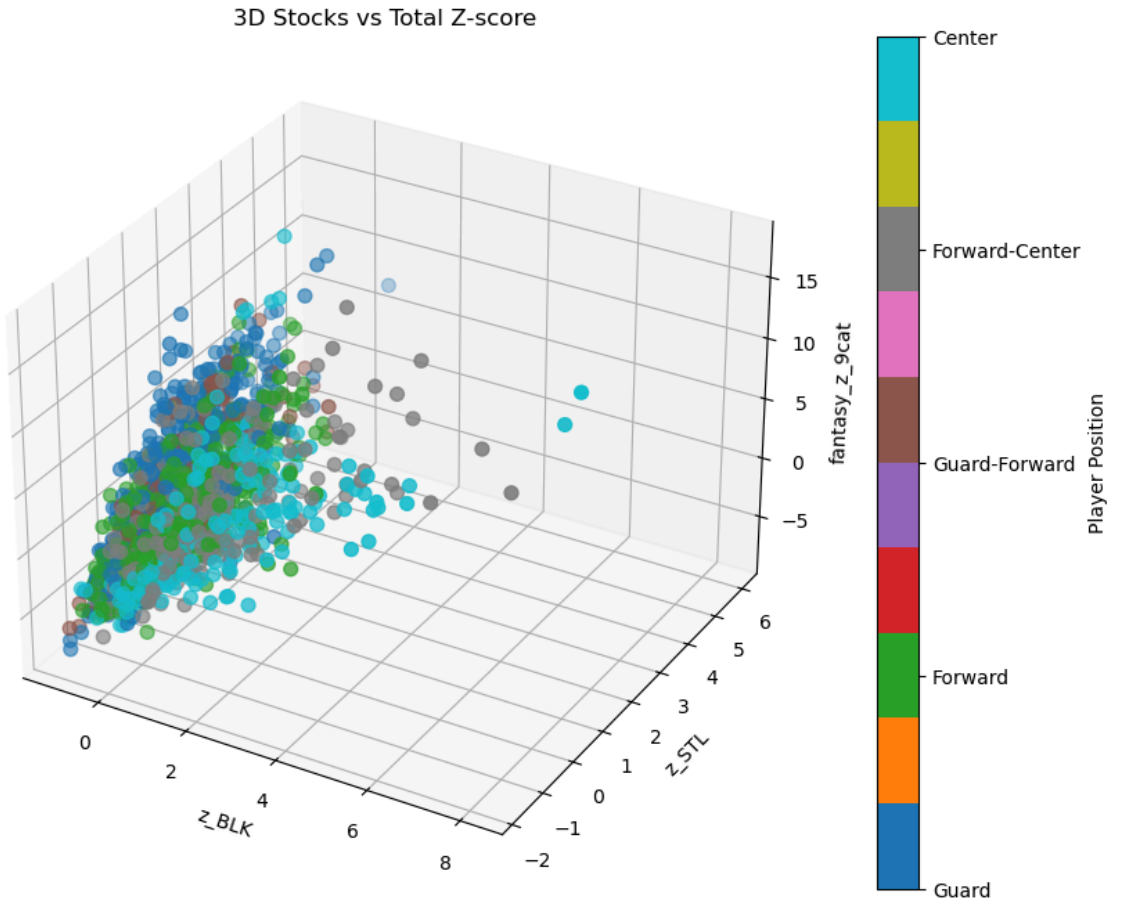
Using MAE which Measures how much the fantasy score changed on average from one season to the next.

The last output is the MAE between 2023-24 to 2024-25.

In [16]:
```
#Example 2: 3D- scatter plot to show the connection between Rebounds, Blocks and
positions = df['POS'].unique()
pos_to_index = {pos: i for i, pos in enumerate(positions)}
color_indices = df['POS'].map(pos_to_index)
fig = px.scatter_3d(
    df,
    x='z_STL',
    y='z_BLK',
    z='fantasy_z_9cat',
    color='POS',
    hover_data={'PLAYER_NAME': True, 'z_STL': ':.2f', 'z_BLK': ':.2f', 'fantasy_
)
# Customize the plot layout
fig.update_layout(
    title='Player Steals and Blocks vs. Fantasy Z-Score',
    scene=dict(xaxis_title='Steals', yaxis_title='Blocks', zaxis_title='Total Z-
            eye=dict(x=1.5, y=1.5, z=0.5))), width=700, height=600)
fig.show()

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
sc = ax.scatter(df["z_BLK"], df['z_STL'], df['fantasy_z_9cat'], c=color_indices,
```

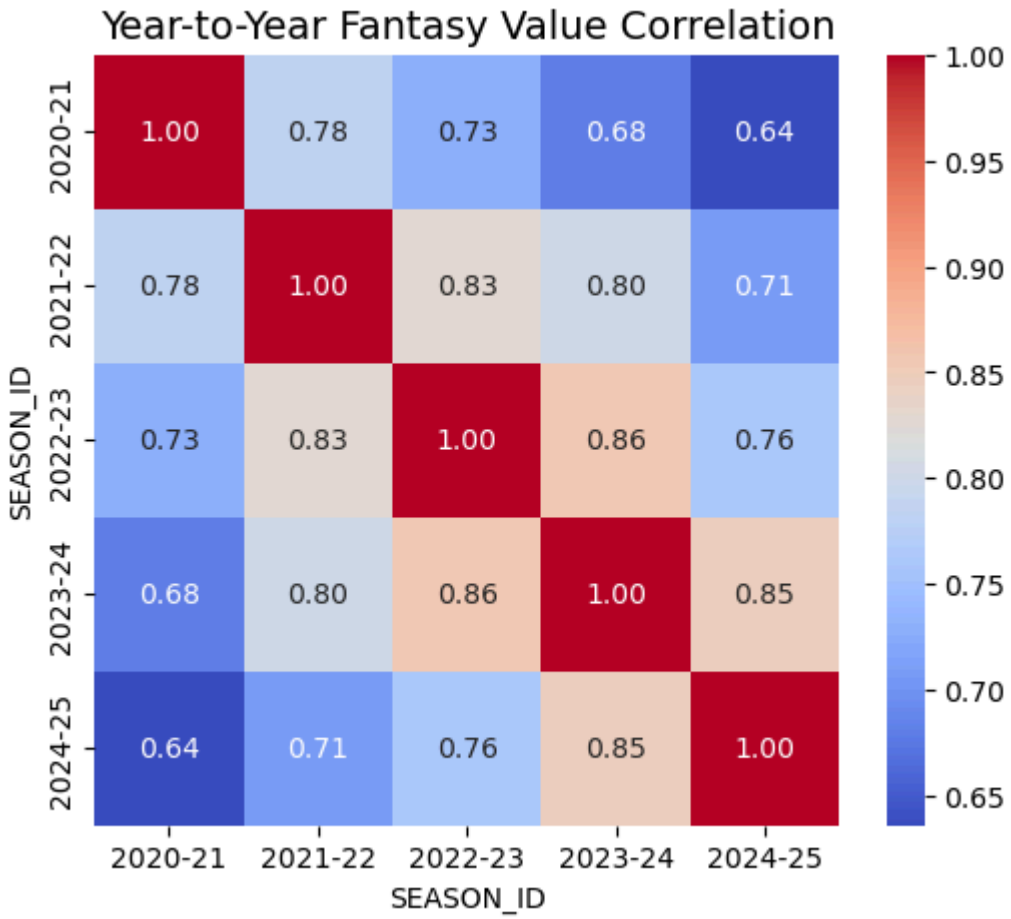
```
cbar = plt.colorbar(sc, ticks=range(len(positions)))
cbar.ax.set_yticklabels(positions)
cbar.set_label('Player Position')
ax.set_xlabel('z_BLK')
ax.set_ylabel('z_STL')
ax.set_zlabel('fantasy_z_9cat')
ax.set_title('3D Stocks vs Total Z-score')
plt.show()
```





In [17]:
```
if 'PLAYER_NAME' in df.columns and 'SEASON_ID' in df.columns:
    pivot = df.pivot_table(index='PLAYER_NAME', columns='SEASON_ID', values='fan
    # Compute correlation between seasons
    corr_matrix = pivot.corr()
    print("Year-to-year fantasy z-score correlation:")
    print(corr_matrix)

    plt.figure(figsize=(6,5))
    sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f")
    plt.title("Year-to-Year Fantasy Value Correlation", fontsize=14)
    plt.show()

    #print(pivot.head())
```

```
Year-to-year fantasy z-score correlation:
SEASON_ID   2020-21   2021-22   2022-23   2023-24   2024-25
SEASON_ID
2020-21    1.000000  0.782966  0.725782  0.679224  0.636100
2021-22    0.782966  1.000000  0.828416  0.800936  0.706001
2022-23    0.725782  0.828416  1.000000  0.857705  0.761919
2023-24    0.679224  0.800936  0.857705  1.000000  0.846091
2024-25    0.636100  0.706001  0.761919  0.846091  1.000000
```



Additional EDA Conclusions

Based on the scatter plots and specialist analysis, here are the key conclusions:

Player Performance and Efficiency

- **Scoring Efficiency:** Frontcourt players, especially centers, are the most efficient scorers.
- **Passing Efficiency:** Players in the upper-right quadrant of the Assists vs. Turnovers plot are efficient passers. A select group of elite players with a Z-score above 1.5 in assists and a turnover Z-score of -0.5 or better are particularly valuable. These players significantly boost a team's assist numbers without negatively impacting turnovers, making them ideal for teams built around big men. *(Note: Be mindful of outliers like Jeff Dowtin Jr. and Keon Ellis, whose stats may be skewed due to a small sample size of games.)*
- **Frontcourt Versatility:** Some frontcourt players contribute positively to both rebounds and free-throw percentage. These players are an excellent fit for "combo guard" teams that prioritize 3-pointers and assists, as they won't lower the team's free-throw percentage average.
- **Steals & Fantasy Value:** Dyson Daniels from the 2024-2025 season is an extreme outlier in steals, with numbers unmatched by any player in the last three seasons. His exceptional performance in this category significantly elevated his fantasy rank. Excluding him, there is a clear positive correlation between a player's steals and their total fantasy score.

Positional Analysis and Correlations

- **3-Pointers and Field Goal Percentage:** There is a small group of players who excel in both 3-pointers and field goal percentage. These players, primarily in backcourt positions, are a good fit for a balanced team that values both 3-pointers made and steals.
- **Overall Player Rankings:** Nikola Jokić has been the most efficient player in the 9-cat H2H format over the last three seasons.
- **Points' Impact:** Points have a major effect on overall fantasy rankings. A strong positive correlation is observed between points and `fantasy_z_9cat` across all positions.
- **3-Pointers' Impact:** While not immediately obvious from the graph, excluding players at the center position, there is a high correlation between 3-pointers made and the total Z-score. This suggests that 3-point shooting significantly boosts the fantasy ranks of power forwards and shooting guards.
- **Category-Specific Value:** A player's rank is heavily influenced by their added value in specific categories. For example, a guard who secures a high number of rebounds (relative to other guards) will be ranked higher than other players with similar or even higher rebound totals if their overall statistical profile is strong. The same principle applies to centers and power forwards who excel in assists.
- **"Stocks" Impact:** The graph visually demonstrates that "stocks" (steals and blocks) are a strong positive indicator of a player's total fantasy value in a 9-category league. The more steals and blocks a player accumulates, the more likely they are to be a top-ranked fantasy player.

Season-to-Season Consistency

- **Correlations:** There is a strong positive correlation between a player's fantasy value in consecutive seasons. The correlation remains strong between non-consecutive seasons, though it is slightly weaker. This suggests that a player's performance is a reliable indicator of future fantasy value.
- **Mean Absolute Error (MAE):** The Mean Absolute Error between a player's fantasy value in consecutive seasons is 1.53. While a lower MAE is generally desirable, this value indicates a notable level of volatility in player performance. A player with a Z-score of 1.5 in one season could have a Z-score ranging from 0 to 3 the next season. Our predictive models will aim to outperform this benchmark.

In [18]:
```
# Define threshold for "specialist"
specialist_threshold = 1.5 # z > 1.5 is elite, only ~6% of the players are in th

profiles = []

for _, row in df.iterrows():
    player = row["PLAYER_NAME"]
    season = row["SEASON_ID"]
    z_scores = {c.replace("z_",""): row[c] for c in z_cols}
```

```
    strong_cats = [cat for cat, val in z_scores.items() if val > specialist_thre
    weak_cats   = [cat for cat, val in z_scores.items() if val < -1.25]

    if len(strong_cats) >= 3 and len(weak_cats) == 0:
        profiles.append((player, season, "All-Rounder", strong_cats))
    elif len(strong_cats) >= 1 and len(weak_cats) >= 1:
        profiles.append((player, season, "Specialist/Punt", (strong_cats, weak_c
    elif len(strong_cats) >= 1:
        profiles.append((player, season, "Specialist", strong_cats))
    else:
        profiles.append((player, season, "Balanced/Neutral", []))

profiles_df = pd.DataFrame(profiles, columns=["PLAYER_NAME","SEASON_ID","Profile

# Show examples
print("=== Example Profiles ===")
print(profiles_df.sample(10))
```

```
=== Example Profiles ===
             PLAYER_NAME SEASON_ID           Profile  \
1388     P.J. Washington   2022-23        Specialist
1110        Kyrie Irving   2023-24       All-Rounder
1928        Ben Sheppard   2024-25  Balanced/Neutral
1606       Terrence Ross   2020-21  Balanced/Neutral
1244         Maxi Kleber   2023-24  Balanced/Neutral
360      Davion Mitchell   2023-24  Balanced/Neutral
569         Franz Wagner   2024-25   Specialist/Punt
549          Eric Gordon   2024-25  Balanced/Neutral
670   Isaiah Hartenstein   2020-21  Balanced/Neutral
668          Isaac Okoro   2023-24  Balanced/Neutral

                        Details
1388                      [BLK]
1110  [PTS, AST, STL, FG3M, FT%]
1928                          []
1606                          []
1244                          []
360                           []
569          ([PTS, FT%], [TOV])
549                           []
670                           []
668                           []
```

In [19]:
```
def generate_profiles_per_season(df_general,season):
    df = df_general[df_general["SEASON_ID"] == season].copy()
    results = []
    # 1. Swiss Army Knife (all-rounders)
    all_rounders = df[(df[[f"z_{c}" for c in ["PTS","REB","AST","STL","BLK","FG3
    swiss_top10 = all_rounders.sort_values("fantasy_z_9cat", ascending=False).he
    results.append(swiss_top10)

    # 2. Best assists-turnover ratio (z-score style)
    df["ast_tov_ratio"] = df["AST"] / df["TOV"].replace(0, np.nan)
    ast_tov_top10 = df.sort_values("ast_tov_ratio", ascending=False).head(10)
    results.append(ast_tov_top10)

    # 3. Punt FT specialists (reb + fg% + blk, but weak FT%)
    punt_ft = df[(df["z_FT%"] < -0.5)].copy()
    punt_ft["bigman_combo"] = df["z_REB"] + df["z_BLK"] + df["z_FG%"]
    punt_ft_top10 = punt_ft.sort_values("bigman_combo", ascending=False).head(10
```

```
    results.append(punt_ft_top10)

    # 4. Defensive kings (steals + blocks)
    df["defense_combo"] = df["z_STL"] + df["z_BLK"]
    defense_top10 = df.sort_values("defense_combo", ascending=False).head(10)
    results.append(defense_top10)

    # 5. Best usage players (points + assists + rebounds)
    df["usage_combo"] = df["z_PTS"] + df["z_AST"] + df["z_REB"]
    usage_top10 = df.sort_values("usage_combo", ascending=False).head(10)
    results.append(usage_top10)

    # 6. Combo guard team (points + assists + 3pt + steals)
    df["combo_guard"] = df["z_PTS"] + df["z_AST"] + df["z_FG3M"] + df["z_STL"]
    guards_top10 = df.sort_values("combo_guard", ascending=False).head(10)
    results.append(guards_top10)

    return results
```

In [20]:

```
results_2122 = generate_profiles_per_season(df,"2021-22")
results_2223 = generate_profiles_per_season(df,"2022-23")
results_2324 = generate_profiles_per_season(df,"2023-24")
results_2425 = generate_profiles_per_season(df,"2024-25")

swiss_top10_2122, ast_tov_top10_2122, punt_ft_top10_2122, defense_top10_2122, us
swiss_top10_2223, ast_tov_top10_2223, punt_ft_top10_2223, defense_top10_2223, us
swiss_top10_2324, ast_tov_top10_2324, punt_ft_top10_2324, defense_top10_2324, us
swiss_top10_2425, ast_tov_top10_2425, punt_ft_top10_2425, defense_top10_2425, us

# Show 2022-2023 results
print("2022-2023 Leaders:\n")
print("Top 10 Swiss Army Knife (all-rounders)", swiss_top10_2223[["PLAYER_NAME",
print("Top 10 AST/TOV (great playmakers)", ast_tov_top10_2223[["PLAYER_NAME","GP
print("Top 10 Punt FT (rim protectors)", punt_ft_top10_2223[["PLAYER_NAME","GP",
print("Top 10 Defensive Kings (stocks monsters)", defense_top10_2223[["PLAYER_NA
print("Top 10 Usage players (PTS, REB, AST)", usage_top10_2223[["PLAYER_NAME","G
print("Top 10 Combo Guards", guards_top10_2223[["PLAYER_NAME","GP","combo_guard"
```

2022-2023 Leaders:

Top 10 Swiss Army Knife (all-rounders)

	PLAYER_NAME	GP	fantasy_z_9cat
953	Joel Embiid	66.0	15.358694
1453	Nikola Jokić	69.0	14.346907
1672	Shai Gilgeous-Alexander	68.0	14.318434
1138	Kevin Durant	47.0	13.892392
348	Damian Lillard	58.0	13.040528
76	Anthony Davis	56.0	12.887413
1688	Stephen Curry	56.0	12.305920
1206	Kyrie Irving	60.0	11.444130
909	Jayson Tatum	74.0	11.422048
1825	Tyrese Haliburton	56.0	10.947757

Top 10 AST/TOV (great playmakers)

	PLAYER_NAME	GP	ast_tov_ratio
912	Jeff Dowtin Jr.	25.0	6.200000
1835	Tyus Jones	80.0	5.635135
1838	Cody Martin	7.0	5.500000
24	Al Horford	63.0	5.108108
2024	Kemba Walker	9.0	4.750000
1156	Kevon Looney	82.0	4.600000
274	Chris Paul	59.0	4.596491
1811	Ty Jerome	45.0	4.500000
991	Jordan McLaughlin	43.0	4.484848
447	Delon Wright	50.0	4.409091

Top 10 Punt FT (rim protectors)

	PLAYER_NAME	GP	bigman_combo
1443	Nic Claxton	76.0	10.750085
1847	Walker Kessler	74.0	9.253264
1620	Rudy Gobert	70.0	8.327960
291	Clint Capela	65.0	7.798128
1386	Mitchell Robinson	59.0	7.614041
666	Giannis Antetokounmpo	63.0	7.319665
1600	Robert Williams III	35.0	6.781004
754	Ivica Zubac	76.0	6.776000
597	Evan Mobley	79.0	6.718709
804	Jakob Poeltl	72.0	6.662757

Top 10 Defensive Kings (stocks monsters)

	PLAYER_NAME	GP	defense_combo
864	Jaren Jackson Jr.	63.0	7.172498
1443	Nic Claxton	76.0	5.466484
76	Anthony Davis	56.0	4.875161
185	Brook Lopez	78.0	4.457007
1417	Myles Turner	62.0	4.202299
1386	Mitchell Robinson	59.0	4.147321
1471	OG Anunoby	67.0	3.930512
953	Joel Embiid	66.0	3.910730
1847	Walker Kessler	74.0	3.807656
1672	Shai Gilgeous-Alexander	68.0	3.765124

Top 10 Usage players (PTS, REB, AST)

	PLAYER_NAME	GP	usage_combo
1453	Nikola Jokić	69.0	9.475544
1263	Luka Dončić	66.0	8.389468
666	Giannis Antetokounmpo	63.0	8.278056
512	Domantas Sabonis	79.0	7.509456
1239	LeBron James	55.0	7.106751
953	Joel Embiid	66.0	7.094261

```
849            James Harden  58.0     7.014443
348          Damian Lillard  58.0     6.410225
759               Ja Morant  61.0     6.343863
1213            LaMelo Ball  36.0     6.331443

Top 10 Combo Guards             PLAYER_NAME    GP  combo_guard
1825  Tyrese Haliburton  56.0    10.343125
1263        Luka Dončić  66.0    10.076789
1213        LaMelo Ball  36.0    10.027794
348      Damian Lillard  58.0     9.942701
1688      Stephen Curry  56.0     9.890150
849        James Harden  58.0     9.350893
616       Fred VanVleet  69.0     8.961757
1777         Trae Young  73.0     8.813322
520    Donovan Mitchell  68.0     8.689660
1206       Kyrie Irving  60.0     7.645884
```

Archtypes conclusions

- Swiss Army Knives (all-rounders) - These are the rare unicorns of fantasy basketball. They don't just contribute in one or two stats — they fill up every column of the box score. Players in this archetype give managers flexibility: you don't need to punt categories or worry about glaring weaknesses when you draft one. They're the safest building blocks because they help you win multiple categories every week.

- AST/TOV Specialists (great playmakers) - Not all playmakers are created equal. Some pile up assists but also turn the ball over recklessly. The elite in this group thread the needle: they run the offense, rack up dimes, and still keep their mistakes to a minimum. They're the hidden gems for managers who want steady guard production without tanking turnovers.

- Punt FT Bigs (rim protectors) - The classic fantasy archetype: towering big men who dominate rebounds, blocks, and field goal percentage, but drag down your free throw shooting. On the surface they look like flawed players — but if you commit to punting FT%, they suddenly become game-changing anchors. They embody the strategy of embracing a weakness to supercharge your strengths.

- Defensive Kings (stocks monsters) - Steals and blocks can swing matchups, and the players in this archetype specialize in them. Even if they don't score much, their defensive presence makes them fantasy gold. Managers who lock down one or two of these guys can often secure the defense categories week after week, turning what looks like "glue guys" in real life into fantasy MVPs.

- Usage Monsters (PTS, REB, AST) - Some players are simply the engine of their teams. Every possession runs through them, and that shows up in fantasy: big points, high assist numbers, and strong rebounding for their position. They may not be perfect in efficiency, but their sheer volume gives them unmatched influence in head-to-head matchups.

- Combo Guards (PTS, AST, 3PM, STL) - This modern archetype thrives in pace-and-space basketball. Combo guards score in bunches, stretch defenses with threes,

create for teammates, and disrupt passing lanes on defense. They might not offer elite rebounding or blocks, but their mix of offensive punch and opportunistic steals makes them fantasy darlings in guard builds.

Modeling

- In this section we will use several regression models to predict a player's fantasy value based on his average season statistics. Since fantasy value is a continuous target (z-score across categories), regression is a natural choice. We experiment with both linear and non-linear approaches to balance interpretability and predictive power.

- **Linear Regression** – A statistical and machine learning method for finding the linear relationship between a continuous dependent variable and one or more independent variables. It serves as our baseline model because of its simplicity and interpretability.
- **OLS by statsmodels** - As linear regression this model works to based on Ordinary Least Squares but in differ from the scikit-learn Linear Regression the models focuses Explanatory analysis - understanding the underlying relationships between variables and their statistical significance.
- **Ridge Regression** – A variant of linear regression that adds an L2 penalty to shrink coefficients of correlated features. This helps reduce overfitting and improve stability when predictors are highly collinear (common in basketball stats such as PTS, USG%, and FGA).
- **PCA + Linear Regression** – Principal Component Analysis (PCA) is first applied to reduce dimensionality and remove multicollinearity among features while retaining most of the variance (95%). A linear regression model is then trained on the compressed feature space, allowing us to capture the main signal with fewer dimensions.
- **Random Forest Regressor** – An ensemble of decision trees where each tree is trained on a random subset of data and features. Random Forests can capture non-linear relationships and feature interactions, making them useful when the relationship between raw stats and fantasy value is not strictly linear.
- **XGBoost Regressor** – A gradient boosting method that builds decision trees sequentially, where each tree corrects errors from the previous ones. XGBoost often outperforms Random Forest by focusing on difficult-to-predict cases and is widely used in applied machine learning competitions due to its high accuracy.

Because of linear models limitations to handle with null values, we created a copy of the data frame where all NaN (located only on _roll and _delta columns) will get the value -999.

Our modeling strategy combines different types of features:

- **Quantitative features:**
  Box score statistics: traditional per-game averages such as points, rebounds, assists,

steals, blocks, turnovers, and shooting percentages.
Durability metrics: games played and minutes per game, since fantasy value is strongly affected by availability.
Rolling averages and deltas: aggregated features from the previous two seasons to capture both consistency and trends in player development.

- **Qualitative feature:**
  Demographics: player age and position, as younger players often improve while older players may decline, and positions reflect different statistical profiles.

We then compare the model's predictive accuracy using MAE, RMSE, and R^2.

Feature engineering for model's performance:

- **Age:** Age often has a nonlinear relationship: Players improve until ~27–29, then decline. A quadratic term captures that curve. Instead of the model assuming "+1 year = always better/worse," it can learn the "arc" of improvement then decline.
- **Per 36 and totals:** Creating them to normalize the model's performance. We use them only for stats where per-minute scaling makes sense (PTS, REB, AST, STL, BLK) and not for stats as percetages.

Summary of OLS and the difference between the two models:

- The core difference between statsmodels.OLS and scikit-learn.LinearRegression lies in their primary objective and the information they provide.
- **statsmodels.OLS** is built for statistical inference and explanation. Its purpose is to help you understand the relationship between variables and the statistical significance of their coefficients. It provides a comprehensive summary table with statistical details like p-values, standard errors, and confidence intervals.It does not automatically add an intercept, forcing the user to explicitly add one with sm.add_constant().It may produce a better predictive model in cases where its slightly different implementation details or numerical procedures are a better fit for the specific characteristics of your data.
- **scikit-learn.LinearRegression** is built for predictive modeling and machine learning workflows. It prioritizes ease of use, speed, and integration into larger pipelines for tasks like cross-validation and hyperparameter tuning.Its output is more concise, focusing on coefficients and basic performance scores like (R^{2}). It does not provide detailed statistical summaries with p-values.It includes an intercept by default, which can sometimes lead to different initial results compared to statsmodels if not handled correctly.

The function below helps us to evaluate the baseline that our models has to beat.

As baseline I used the value of a player's last season and I calculate the difference between following seasons fantasy value (total Z-score)

In [21]:
```
def evaluate_baseline(df, target_col="next_fantasy_z_9cat"):
    """
    Evaluate naive baseline: predict next season's fantasy value = last season's

    Parameters
    ----------
    df : pd.DataFrame
        Data with player_id, season, fantasy_z_9cat, and target_col.
    target_col : str, default="next_fantasy_z_9cat"
        Column representing the actual next-season fantasy value.

    Returns
    -------
    dict : baseline performance metrics
    """
    # Make sure data is sorted
    df = df.sort_values(["PLAYER_NAME", "SEASON_ID"])

    # Baseline = current season's fantasy value
    df["baseline_pred"] = df["fantasy_z_9cat"]

    # Keep rows where we actually have a next-season target
    valid = df.dropna(subset=[target_col, "baseline_pred"])

    y_true = valid[target_col]
    y_pred = valid["baseline_pred"]

    n = len(y_true)
    p = len(df.columns.tolist())

    # Metrics
    mae = mean_absolute_error(y_true, y_pred)
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    r2 = r2_score(y_true, y_pred)
    adjusted_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)

    return {
        "MAE": mae,
        "RMSE": rmse,
        "R-squared": r2,
        "Adjusted R-squared": adjusted_r2,
    }
```

The functions below helps to preproces the features for the linear and non-linear models

In [22]:
```
## Helper function for filling the 2021/22 season and rookies from other seasoms
def preprocess_linear_features(df, season_col="SEASON_ID", player_col="PLAYER_NA
    """
    Preprocess linear feature columns:
      1. Fill rolling/delta NaNs with -999 (benefit: When a linear model sees a
          "no prior history" cases, effectively treating it as a new, separate c
      2. Add has_history flag: 0 if first season for player, else 1

    Args:
        df (pd.DataFrame): input dataframe with rolling & delta features
        season_col (str): season column name
        player_col (str): player column name
```

```
    Returns:
        pd.DataFrame: cleaned dataframe ready for modeling
    """
    df_clean = df.copy()

    # 1. Identify rolling & delta columns
    rolling_cols = [c for c in df_clean.columns if "_roll2" in c]
    delta_cols = [c for c in df_clean.columns if "_delta" in c]

    # 2. Fill rolling & delta features with -999
    df_clean[rolling_cols + delta_cols] = df_clean[rolling_cols + delta_cols].fi

    # 3. Add has_history flag (0 if all deltas are 0, else 1)
    df_clean["has_history"] = (df_clean[delta_cols].abs().sum(axis=1) > 0).astyp

    return df_clean
```

In [23]:
```
def preprocess_features(df, season_col="SEASON_ID", player_col="PLAYER_NAME"):
    """
    Preprocess linear feature columns:
      1. Adding per36 features.
      2. Creating sample weights for games played

    Args:
        df (pd.DataFrame): input dataframe with rolling & delta features
        season_col (str): season column name
        player_col (str): player column name

    Returns:
        pd.DataFrame: cleaned dataframe ready for modeling
    """
    df_clean = df.copy()

    # 1. Per36 stats (selected features only)
    per36_cols = ["PTS", "REB", "AST", "STL", "BLK", "TOV"]
    for col in per36_cols:
        df_clean[f"{col}_per36"] = df_clean[col] / df_clean["MIN"] * 36

    #2. Adding sample-weight - Give more weight to examples where the season is
    df_clean["sample_weight"] = np.sqrt(df_clean["GP"].fillna(0))
    # Min-max normalize weights to [0.2,1] if you want to avoid extremely large
    w = df_clean["sample_weight"].values
    w = (w - w.min()) / (w.max() - w.min() + 1e-9)
    df_clean["sample_weight_norm"] = 0.2 + 0.8 * w
    df_clean.drop(columns = ['sample_weight'],inplace=True)

    return df_clean
```

In [24]:
```
df = df.sort_values(by=["PLAYER_NAME", "SEASON_ID"]).reset_index(drop=True)
# List of stats to use for rolling + delta
stats = ['PTS', 'REB', 'AST', 'STL', 'BLK', 'FG3M', 'FG%', 'FT%', 'TOV', 'GP', '
# Calculating rolling averages and delta
for col in stats:
    df[f"{col}_roll2"] = df.groupby("PLAYER_NAME")[col].transform(lambda x: x.ro
    df[f"{col}_delta"] = df.groupby("PLAYER_NAME")[col].diff()
#print(df.loc[:,"PTS_roll2":].head(8))
# Target variable: fantasy z-score (you can change if named differently), shift(
df["next_fantasy_z_9cat"] = df.groupby("PLAYER_NAME")["fantasy_z_9cat"].shift(-1
```

```
for col in cats:
    df[f"next_z_{col}"] = df.groupby("PLAYER_NAME")[f"z_{col}"].shift(-1)
# delta target = next season - current season
df["delta_fantasy"] = df["next_fantasy_z_9cat"] - df["fantasy_z_9cat"]
#print(df.loc[: , :'delta_fantasy'].head())
```

In [25]:
```
df_linear = preprocess_linear_features(df) #filling NaN with -999, creating an h
#print(df_linear.info())
df_linear = preprocess_features(df_linear) #adding per36 stats and age squared
df = preprocess_features(df)

#print(df_linear.loc[: , :'delta_fantasy'].head())

# baseline (last season) is simply the current fantasy_z_9cat value
baseline_results = evaluate_baseline(df)

print("Baseline stats (last season → current season):")
print(f"MAE: {baseline_results['MAE']:.3f}")
print(f"RMSE: {baseline_results['RMSE']:.3f}")
print(f"R-squared: {baseline_results['R-squared']:.3f}")
print(f"Adjusted R-squared: {baseline_results['Adjusted R-squared']:.3f}")
```

```
Baseline stats (last season → current season):
MAE: 1.857
RMSE: 2.399
R-squared: 0.669
Adjusted R-squared: 0.649
```

In [26]:
```
# Target = delta in fantasy z-scor
df["target_delta"] = df["next_fantasy_z_9cat"] - df["fantasy_z_9cat"]
df_linear["target_delta"] = df_linear["next_fantasy_z_9cat"] - df_linear["fantas

# --- Features we'll use ---
base_features = ["PTS", "REB", "AST", "STL", "BLK", "TOV", "FG%", "FT%", "FG3M"]
trend_features = [c for c in df.columns if "_roll2" in c or "_delta" in c or "_p
durability = ["GP", "MIN"]
demographics = ["AGE", "POS"]

feature_cols = base_features + trend_features + demographics
feature_cols.remove("target_delta")
#print(feature_cols)

# Add age^2
df["AGE_squared"] = df["PLAYER_AGE"] ** 2
df_linear["AGE_squared"] = df_linear["PLAYER_AGE"] ** 2

feature_cols.append("AGE_squared")
feature_cols.remove("AGE")

# Train and test data = up to 2023-24, Prediction data = 2024-25
train_val = df[df["SEASON_ID"] < "2024-25"]   # 2020-21 → 2023-24
linear_train_val = df_linear[df_linear["SEASON_ID"] < "2024-25"]
predict_only = df_linear[df_linear["SEASON_ID"] == "2024-25"]  # 2024-25
#train_val.to_csv("train_set.csv", index = False)

X = train_val[feature_cols]
y = train_val["next_fantasy_z_9cat"]

# Example: hold out 2023-24 as test
train = train_val[train_val["SEASON_ID"] < "2023-24"].dropna(subset=["target_del
```

```
linear_train = linear_train_val[linear_train_val["SEASON_ID"] < "2023-24"].dropn
test = train_val[train_val["SEASON_ID"] == "2023-24"].dropna(subset=["target_del
linear_test = linear_train_val[linear_train_val["SEASON_ID"] == "2023-24"].dropn

linear_X_train, X_train, y_train = linear_train[feature_cols], train[feature_col
linear_X_test, X_test, y_test = linear_test[feature_cols] ,test[feature_cols], t

prediction_df_features = predict_only[feature_cols]

# Sample weights (durability)
sample_weights = train["sample_weight_norm"].values

print("Train shape:", X_train.shape)
print("Test shape:", X_test.shape)
print("Prediction shape:", prediction_df_features.shape)

#print(linear_X_train.info())#loc[:,:].tail(8))
#print(train_val.loc[:,"z_TOV":].tail(8))
#print(y_train.info())
#print(linear_X_test.info())
#print(df_train.loc[:,:].head(8))
#print(df_train.info())
```

```
Train shape: (1022, 39)
Test shape: (388, 39)
Prediction shape: (434, 39)
```

#creating train and test splits using the train-test-split function model_df = df_train.drop(columns=[c for c in df_train.columns if c not in model_stats], inplace = False) model_target = df_train["delta_fantasy"] model_df_train, model_df_test, model_target_train, model_target_test = train_test_split(model_df, model_target, test_size=0.2, random_state=42) linear_model_df = df_linear_train.drop(columns=[c for c in df_linear_train.columns if c not in model_stats], inplace = False) linear_model_target = df_linear_train["next_fantasy_z_9cat"] linear_model_df_train, linear_model_df_test, linear_model_target_train, linear_model_target_test = train_test_split(linear_model_df, linear_model_target, test_size=0.2, random_state=42) #print(linear_model_df_train.info()) #print(model_df_train.head()) #print(model_df_test.info())

In [27]:
```
#creating a preprocessor - one hot encoding and standard scaler for numeric feat
position = ["POS"] if "POS" in X_train.columns else []
numeric = [c for c in X_train.columns if c not in position]
preprocessor = ColumnTransformer(transformers=[("pos", OneHotEncoder(drop="first
results = {}
results["Baseline"] = baseline_results
fitted_models = {}
```

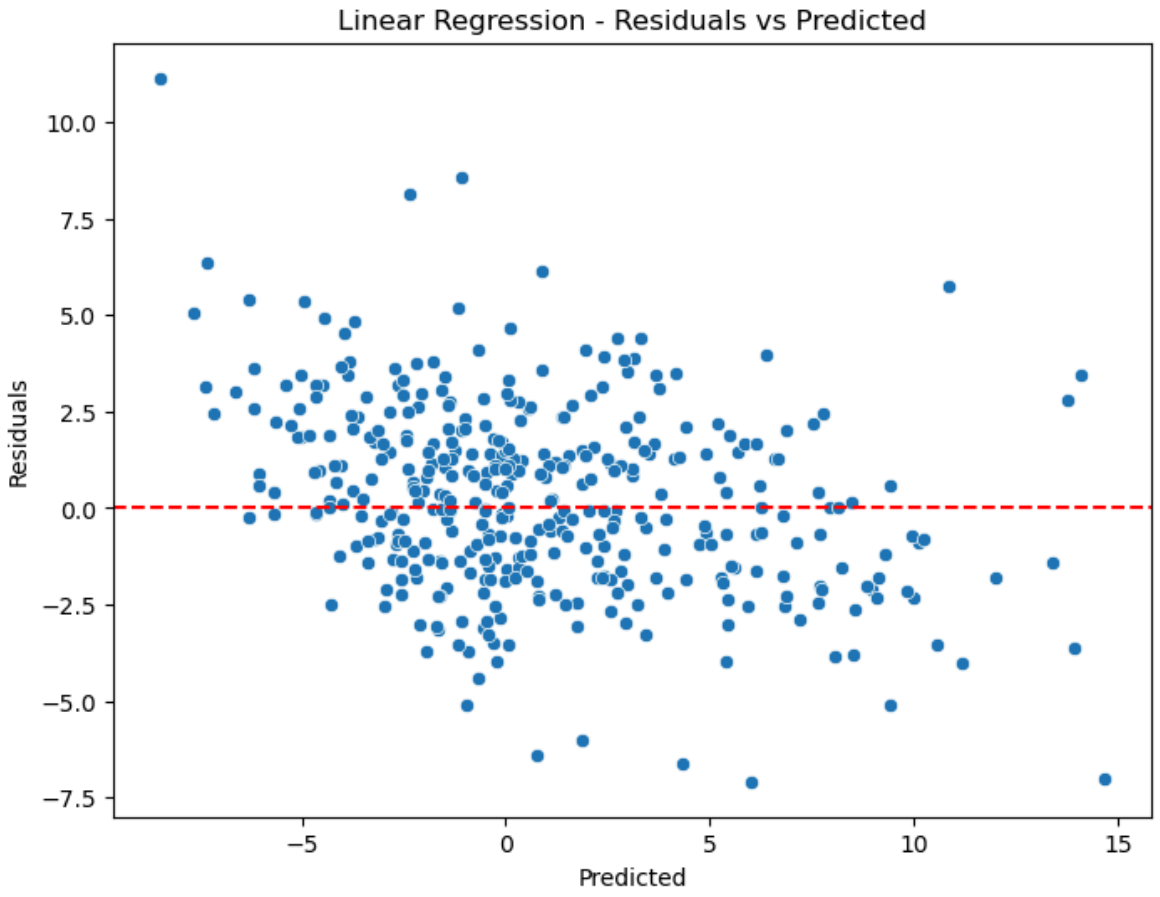
In [28]:
```
def evaluate(y_true, y_pred, p):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    non_zero_indices = y_true != 0
    n = len(y_true)

    mae = mean_absolute_error(y_true, y_pred)
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    r2 = r2_score(y_true, y_pred)
    adjusted_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)
    return {"MAE" :mae,"RMSE": rmse,"R-squared": r2, "Adjusted R-squared": adjus
```

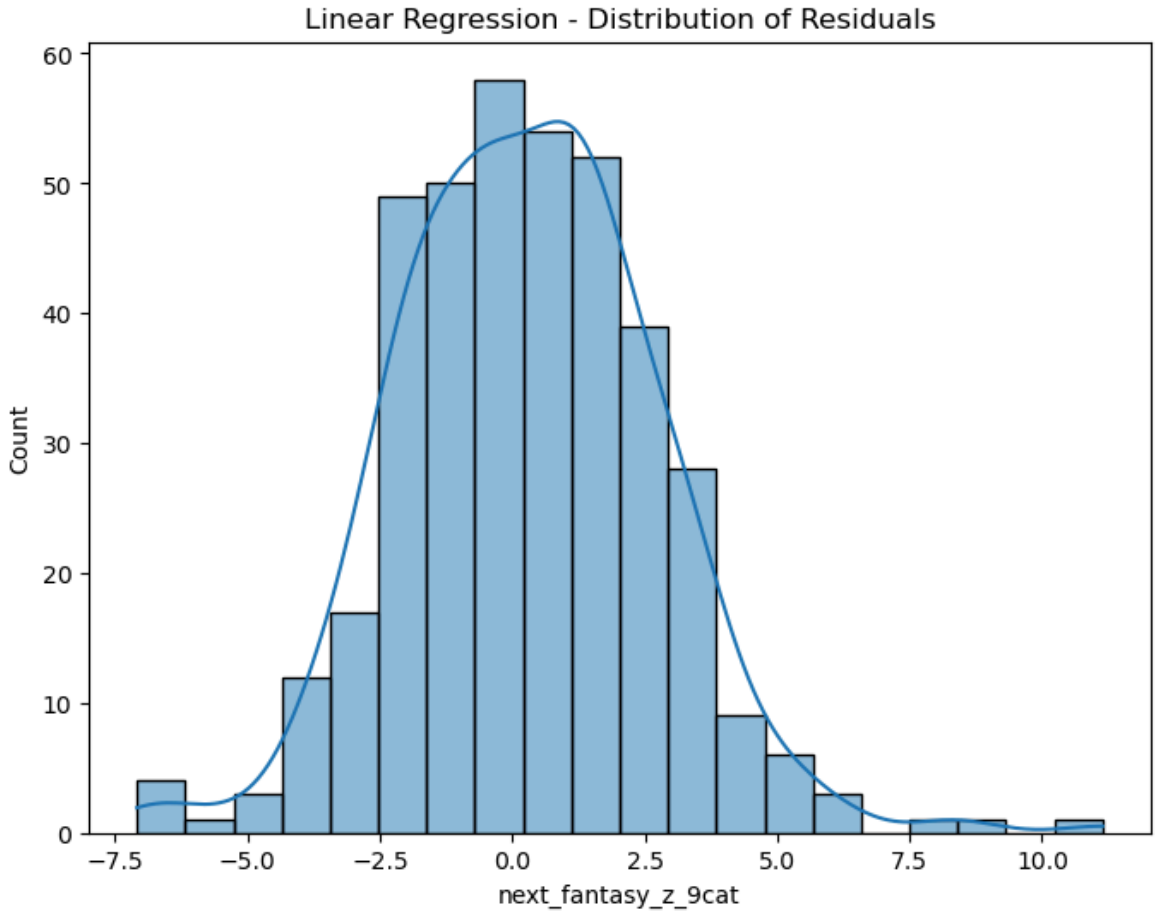
In [29]:
```
def plot_residaul_analysis(model, name):
    best_model = model  # or PCA+LR
    y_pred = best_model.predict(linear_X_test)

    residuals = y_test - y_pred

    plt.figure(figsize=(8, 6))
```

```
    sns.scatterplot(x=y_pred, y=residuals)
    plt.axhline(0, color="red", linestyle="--")
    plt.xlabel("Predicted")
    plt.ylabel("Residuals")
    plt.title(f"{name} - Residuals vs Predicted")
    plt.show()

    plt.figure(figsize=(8, 6))
    sns.histplot(residuals, bins=20, kde=True)
    plt.title(f"{name} - Distribution of Residuals")
    plt.show()
```

In [30]:
```
#Linear regression
pipe_LR = Pipeline([('preprocessor', preprocessor),("LinearRegression", LinearRe
pipe_LR.fit(linear_X_train, y_train, LinearRegression__sample_weight=sample_weig
y_preds_LR = pipe_LR.predict(linear_X_test)
p_LR = len(pipe_LR.named_steps['preprocessor'].get_feature_names_out())
#print(evaluate(y_test, y_pred_delta_LR))

results["LinearRegression"] = evaluate(y_test ,y_preds_LR, p_LR)
fitted_models["LinearRegression"] = pipe_LR
```

In [31]:
```
plot_residaul_analysis(pipe_LR, "Linear Regression")
```



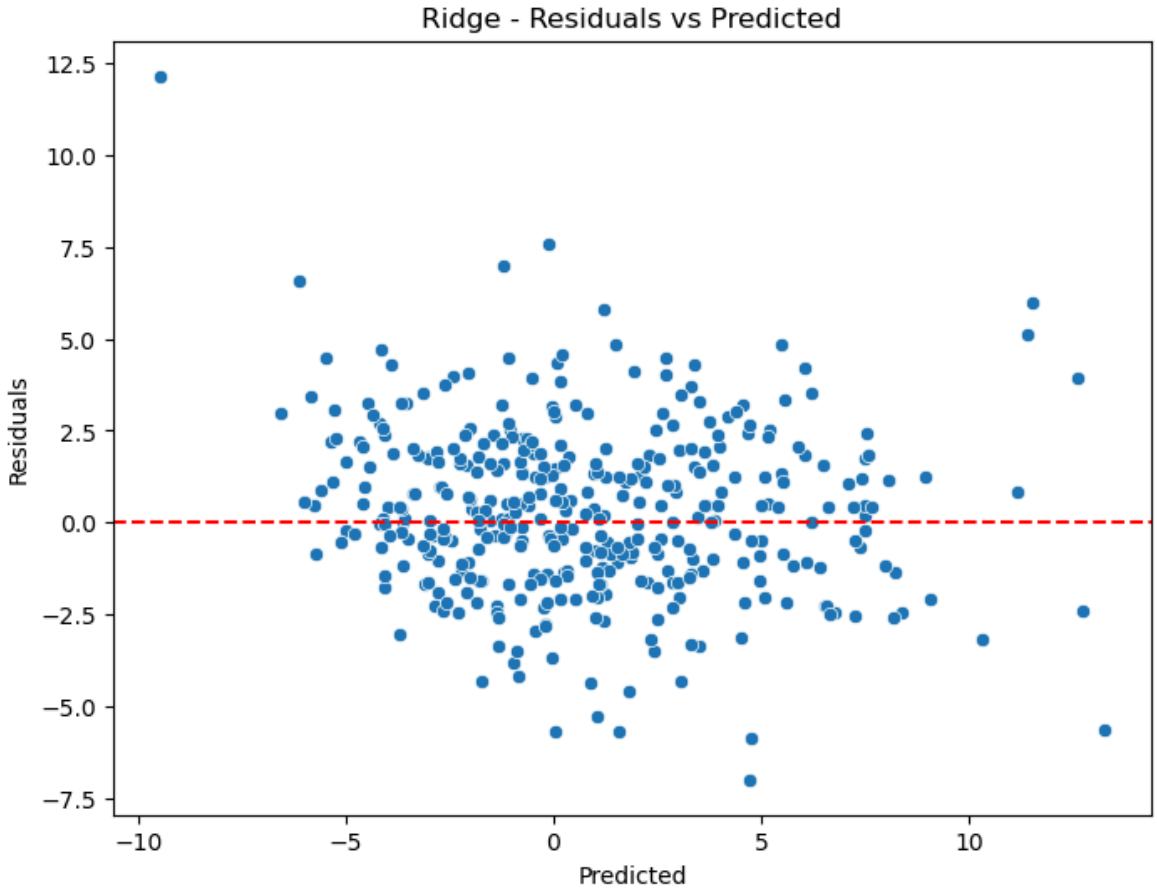


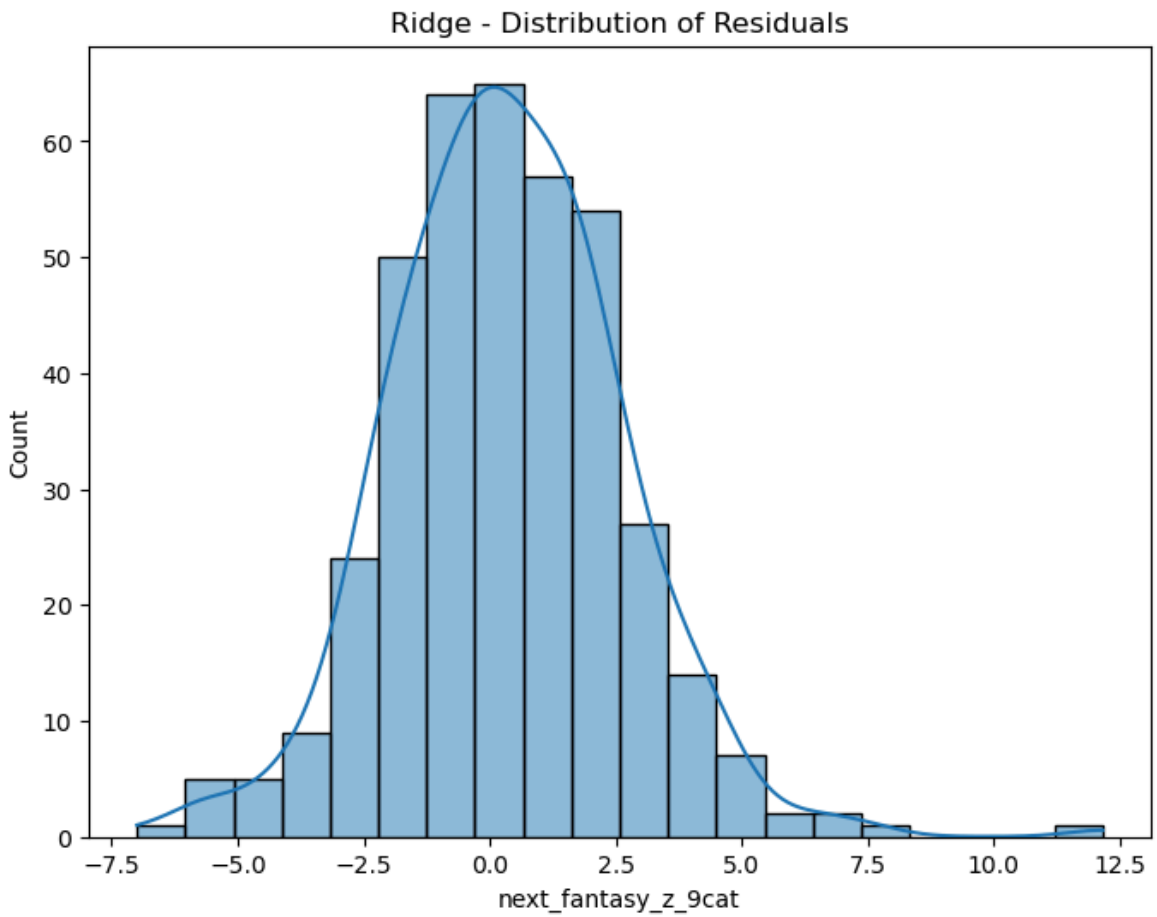
Trying to find the optimal alpha by cross-validation, unfortunately it led to poor results on test-set

pipe_R = Pipeline([('preprocessor', preprocessor), ('Ridge', Ridge())]) # Define alpha values to test alpha_values = [0.01, 0.1, 1.0, 10.0, 100.0, 150.0] # Set up GridSearchCV grid = GridSearchCV(estimator=pipe_R, param_grid={'Ridge__alpha': alpha_values}, cv=5, scoring='r2', return_train_score=False) # 💡 FIX: Pass sample_weights correctly using fit_params # The key is '<step_name>__sample_weight' fit_params = {'Ridge__sample_weight': sample_weights} # Fit with sample weights # You pass the fit_params dictionary as keyword arguments (using **) grid.fit(linear_X_train, y_train, **fit_params) # Note: In newer scikit-learn versions, this syntax: # grid.fit(linear_X_train, y_train, Ridge__sample_weight=sample_weights) # *should* work if sample_weights is a one-dimensional array, # but using the dict and ** is the canonical and often safer way. y_pred = grid.predict(linear_X_test) # Best alpha and score print("Best alpha:", grid.best_params_['Ridge__alpha']) print("Best CV score:", grid.best_score_) test_r2_score = r2_score(y_test, y_pred) print("-" * 30) print(f"Optimal Alpha found via CV: {grid.best_params_['Ridge__alpha']}") print(f"Cross-Validation (Training) R2 Score: {grid.best_score_:.4f}") print(f"Final **Unseen Test Set** R2 Score: {test_r2_score:.4f}") print("-" * 30)

In [32]:
```
#Ridge
pipe_R = Pipeline([('preprocessor', preprocessor),("Ridge", Ridge(alpha = 100))])
pipe_R.fit(linear_X_train, y_train, Ridge__sample_weight=sample_weights)
y_preds_R = pipe_R.predict(linear_X_test)
p_R = len(pipe_LR.named_steps['preprocessor'].get_feature_names_out())
#print(evaluate(y_test, y_pred_delta_R))

results["Ridge"] = evaluate(y_test ,y_preds_R, p_R)
fitted_models["Ridge"] = pipe_R
```

In [33]:
```
plot_residaul_analysis(pipe_R, "Ridge")
```

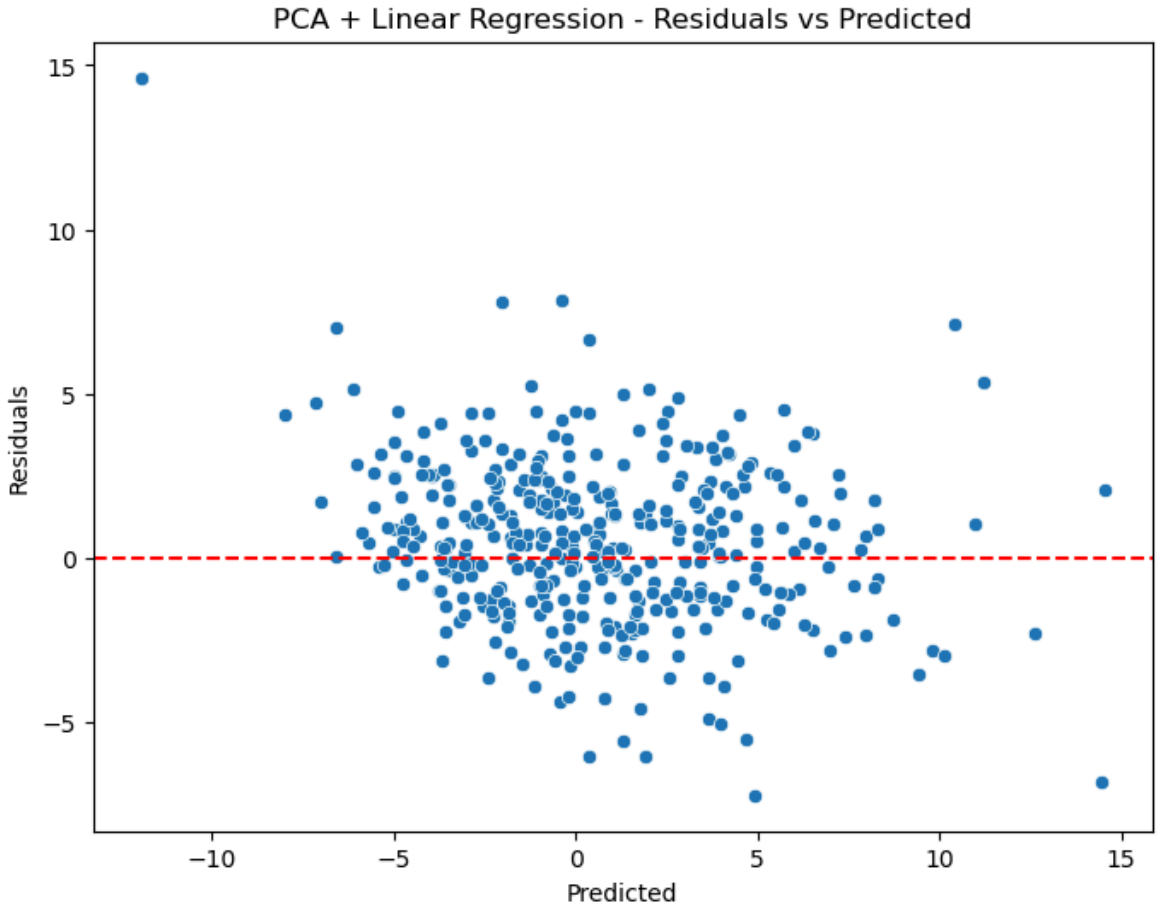


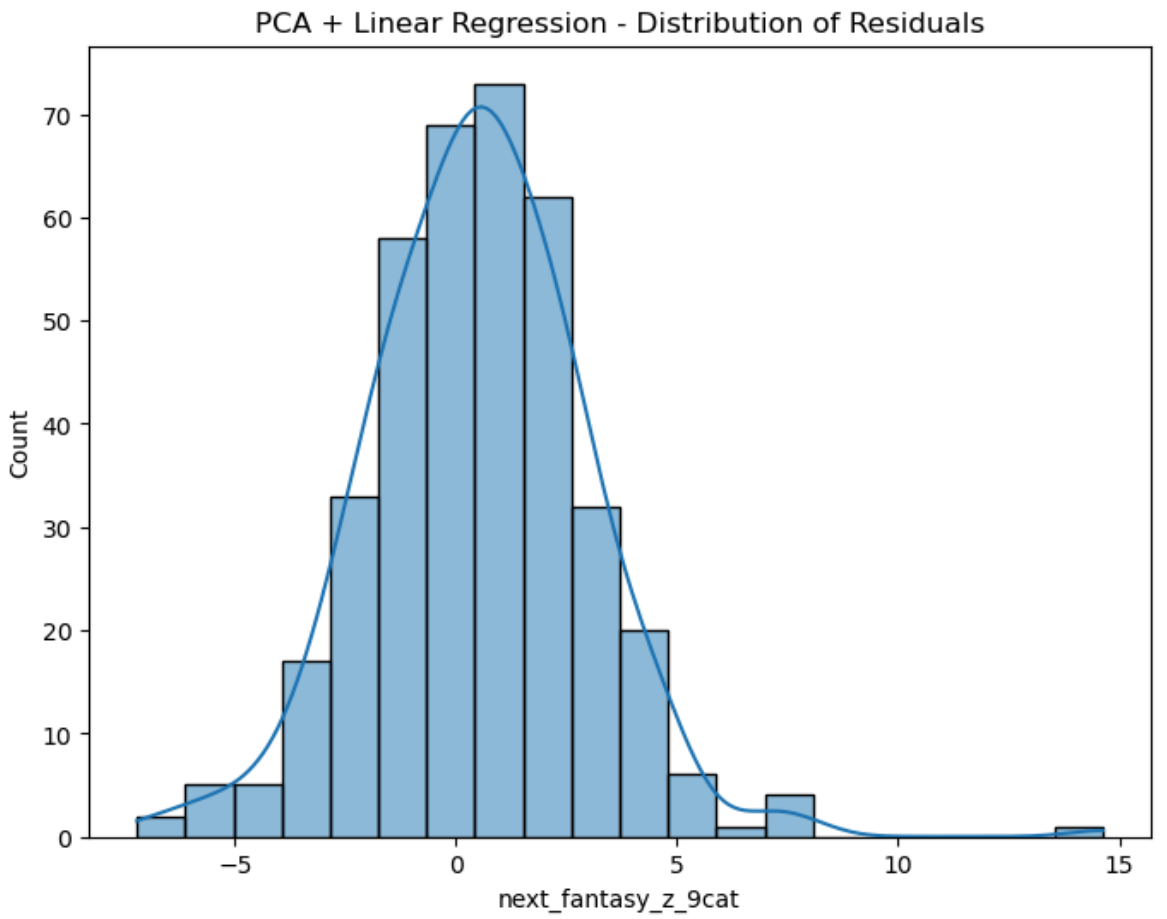


In [34]:
```
#PCA + Linear Regression - keeping more than 95% of the variance - play with num
pca_pipe = Pipeline([('preprocessor', preprocessor),("pca", PCA(n_components=0.9
pca_pipe.fit(linear_X_train, y_train, LinearRegression__sample_weight=sample_wei
y_preds_pca = pca_pipe.predict(linear_X_test)
```

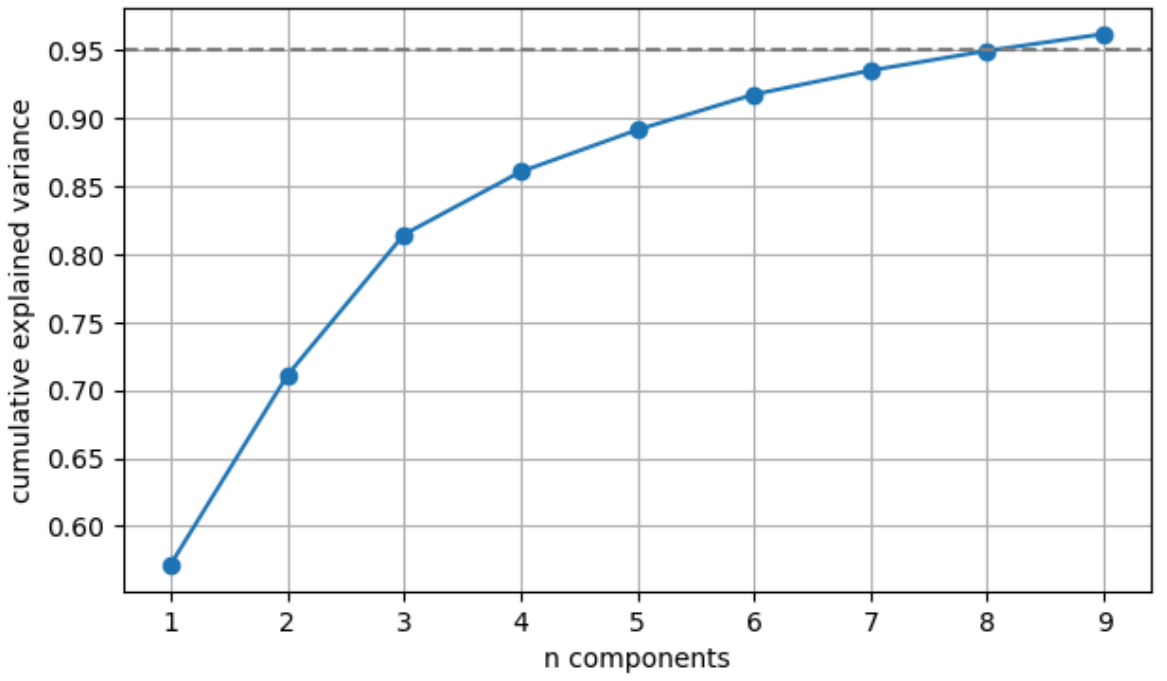
```
p_pca = len(pipe_LR.named_steps['preprocessor'].get_feature_names_out())
#print(evaluate(y_test, y_pred_delta_pca))

results["PCA + Linear Regression"] = evaluate(y_test ,y_preds_pca, p_pca)
fitted_models["PCA + Linear Regression"] = pca_pipe
```

In [35]:
```
plot_residaul_analysis(pca_pipe, "PCA + Linear Regression")
```





In [36]:
```
cum_var = np.cumsum(pca_pipe[1].explained_variance_ratio_)
plt.figure(figsize=(7,4))
plt.plot(np.arange(1,len(cum_var)+1), cum_var, marker='o')
plt.axhline(0.95, color='grey', linestyle='--')
plt.xlabel("n components"); plt.ylabel("cumulative explained variance"); plt.gri
```

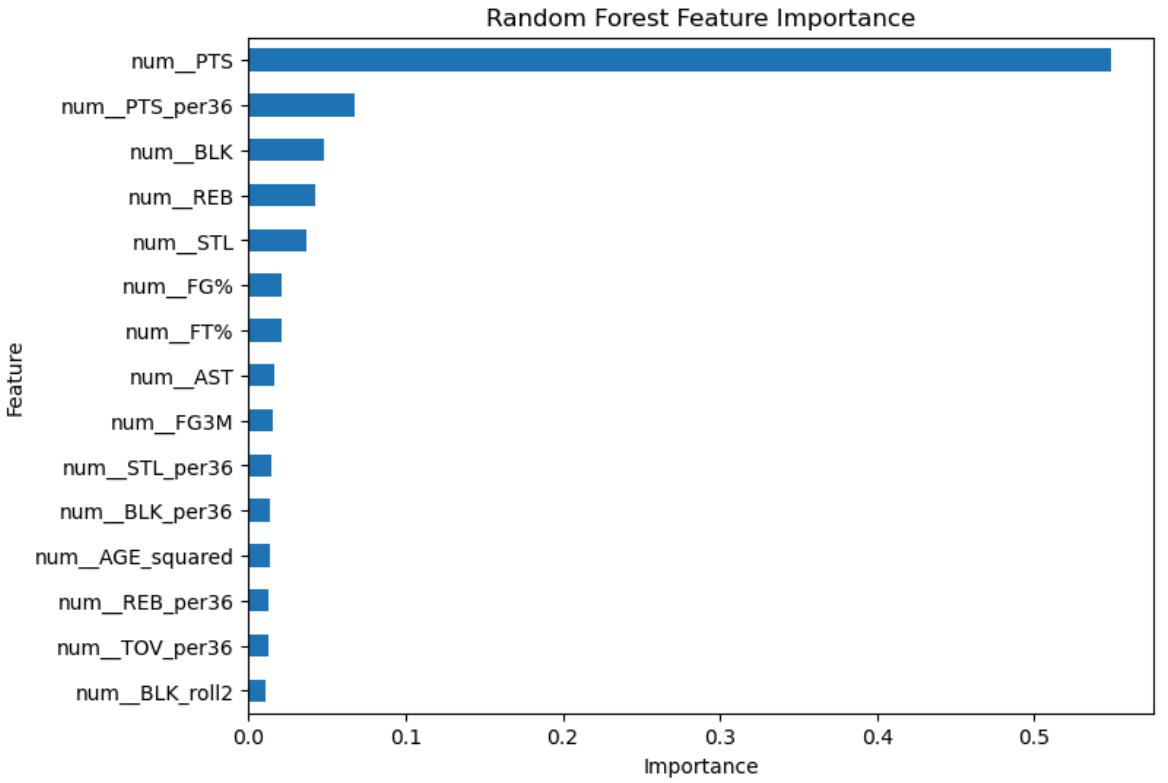


In [37]:
```
def plot_feature_importance(model, feature_names, title):
    features = model[0].get_feature_names_out()
    feat_imp = pd.Series(model[1].feature_importances_, index=features).sort_val
    plt.figure(figsize=(8,6))
```

```
    feat_imp.head(15).plot(kind='barh')
    plt.gca().invert_yaxis()
    plt.title(title)
    plt.xlabel("Importance")
    plt.ylabel("Feature")
    plt.show()
```

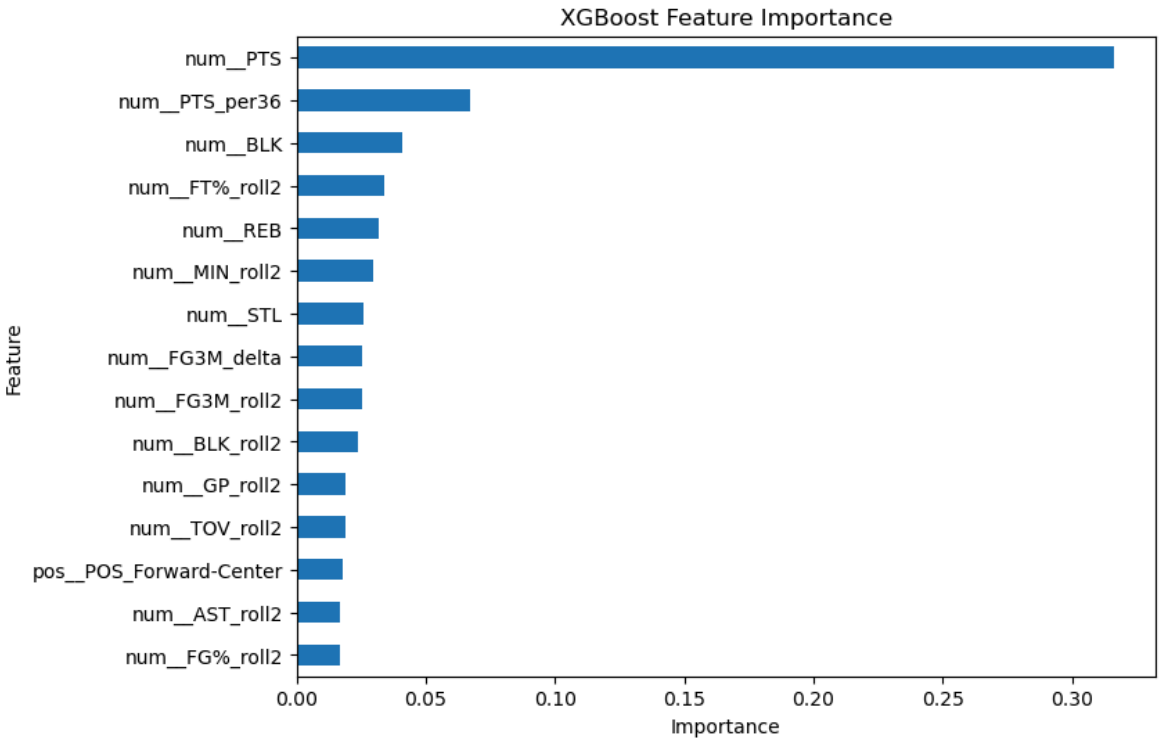
In [38]:
```
#Random Forrest regressor
pipe_RF = Pipeline([('preprocessor', preprocessor),("RandomForest", RandomForest
pipe_RF.fit(linear_X_train, y_train)
y_preds_RF = pipe_RF.predict(linear_X_test)
p_RF = len(pipe_LR.named_steps['preprocessor'].get_feature_names_out())
#print(evaluate(y_test, y_pred_delta_RF))

results["RandomForest"] = evaluate(y_test ,y_preds_RF, p_RF)
fitted_models["RandomForest"] = pipe_RF
```

In [39]:
```
plot_feature_importance(pipe_RF , feature_cols, "Random Forest Feature Importanc
```



In [40]:
```
#XGBoost
try:
    pipe_XG = Pipeline([('preprocessor', preprocessor),("XGBoost", XGBRegressor(
    pipe_XG.fit(X_train, y_train, XGBoost__sample_weight=sample_weights)
    y_preds_XG = pipe_XG.predict(X_test)
    p_XG = len(pipe_LR.named_steps['preprocessor'].get_feature_names_out())
    #print(evaluate(y_test, y_pred_delta_XG))

    results["XGBoost"] = evaluate(y_test ,y_preds_XG, p_XG)
    fitted_models["XGBoost"] = pipe_XG
except ImportError:
    print("⚠️ XGBoost not installed, skipping that model.")
```

In [41]:
```
plot_feature_importance(pipe_XG, feature_cols, "XGBoost Feature Importance")
```



In [42]:
```python
def backward_elimination_ols_named(X_initial, y, feature_names=None, p_threshold
    """
    Backward elimination OLS. Returns (ols_model, final_feature_list).
    X_initial: DataFrame or ndarray (rows must align with y)
    y: Series or ndarray
    """
    # Prepare X DataFrame
    if isinstance(X_initial, np.ndarray):
        if feature_names is None:
            raise ValueError("feature_names required when X_initial is ndarray")
        X = pd.DataFrame(X_initial.copy(), columns=feature_names)
    elif isinstance(X_initial, pd.DataFrame):
        X = X_initial.copy()
    else:
        raise TypeError("X_initial must be numpy array or DataFrame")

    # Ensure y is Series and aligned to X's index
    if not isinstance(y, pd.Series):
        y = pd.Series(y)
    y = y.reindex(X.index)

    # Iteratively remove worst p-value feature
    while True:
        Xc = sm.add_constant(X, has_constant="add")
        model = sm.OLS(y, Xc).fit()
        pvals = model.pvalues.drop('const', errors='ignore')
        if pvals.empty or pvals.max() <= p_threshold:
            break
        worst = pvals.idxmax()
        # If worst not in X (shouldn't happen) break
        if worst not in X.columns:
            break
        X = X.drop(columns=[worst])
        if X.shape[1] == 0:
            break
```

```
    final_features = list(X.columns)
    return model, final_features
```

In [43]:
```
# OLS linear regression, every run omitting the feature with the highest P-value
X_train_arr = preprocessor.transform(linear_X_train)
X_train_df = pd.DataFrame(X_train_arr, columns=preprocessor.get_feature_names_ou

y = y_train

ols_model, final_features = backward_elimination_ols_named(X_train_df, y , X_tra

p_OLS = len(final_features)
preds_OLS = ols_model.fittedvalues
results["OLS"] = evaluate(y ,preds_OLS, p_OLS)
fitted_models["OLS"] = ols_model
#print(ols_model.summary())
```

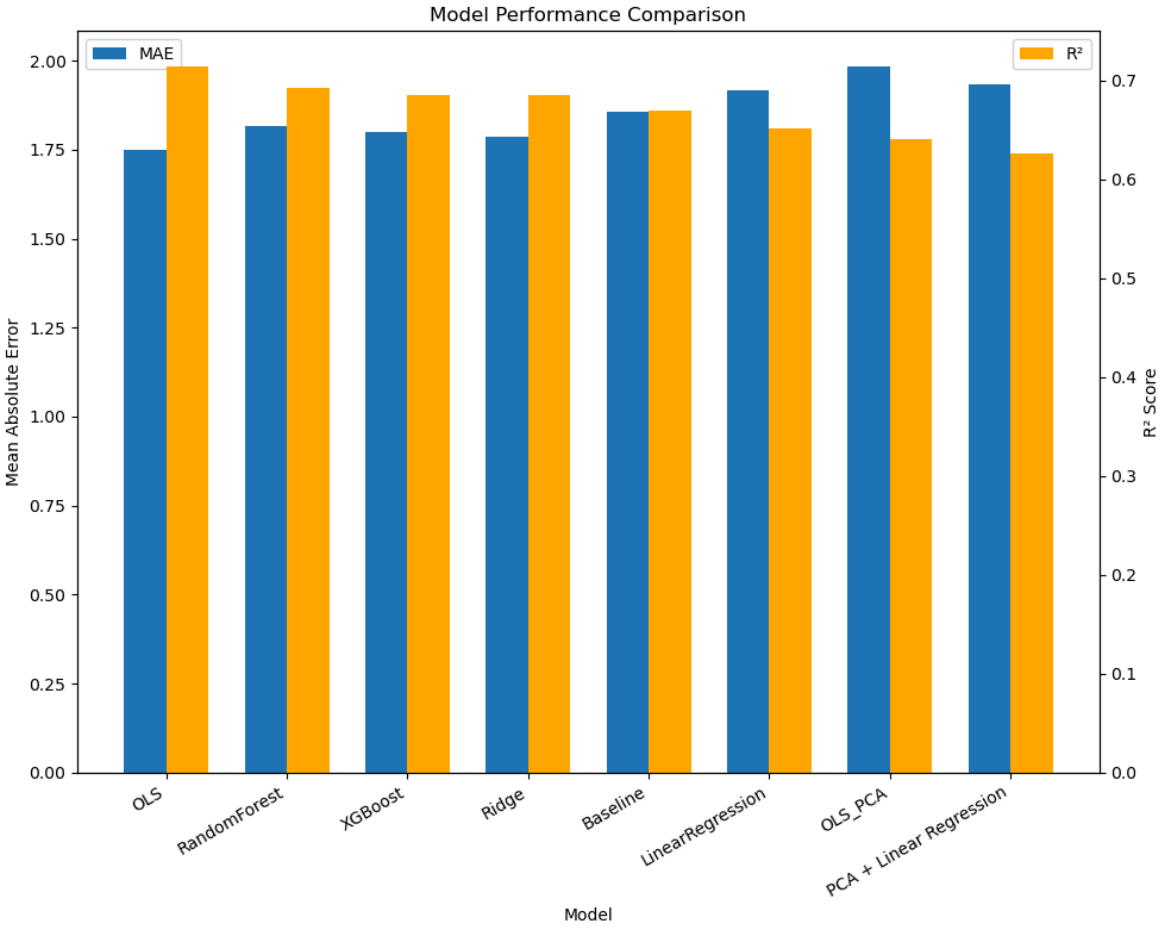
In [44]:
```
# OLS linear regression with PCA, doesn't omit any features
pca = PCA(n_components = 8, random_state=42) # 95% of the variance
X_pca = pca.fit_transform(X_train_df)
X_pca_df = pd.DataFrame(X_pca, columns=[f"PCA_{i}" for i in range(X_pca.shape[1]

ols_PCA_model, final_features_pca = backward_elimination_ols_named(X_pca_df, y ,

p_ols_pca = len(final_features_pca)
preds_OLS_PCA = ols_PCA_model.fittedvalues
results["OLS_PCA"] = evaluate(y ,preds_OLS_PCA, p_ols_pca)
fitted_models["OLS_PCA"] = ols_PCA_model

#print(ols_PCA_model.summary())
```

In [45]:
```
# Show results
results_df = pd.DataFrame(results).T.sort_values("R-squared", ascending = False)
styled_results = (
    results_df.style
    .format({"MAE": "{:.3f}", "R-squared": "{:.3f}", "RMSE":"{:.3f}", "Adjusted
    .highlight_min("MAE", color="#ABEBC6")
    .highlight_max("R-squared", color="#ABEBC6")
    .highlight_min("RMSE", color="#ABEBC6")
    .highlight_max("RMSE", color="#F5B7B1")
    .highlight_max("MAE", color = "#F5B7B1")
    .highlight_min("R-squared", color = "#F5B7B1")
    .highlight_max("Adjusted R-squared", color = "#ABEBC6")
    .highlight_min("Adjusted R-squared", color = "#F5B7B1")
)
display(styled_results)

print("Baseline stats (last season → current season):")
print(f"MAE: {baseline_results['MAE']:.3f}")
print(f"RMSE: {baseline_results['RMSE']:.3f}")
print(f"R-squared: {baseline_results['R-squared']:.3f}")
print(f"Adjusted R-squared: {baseline_results['Adjusted R-squared']:.3f}")
```

	MAE	RMSE	R-squared	Adjusted R-squared
OLS	1.750	2.237	0.714	0.709
RandomForest	1.816	2.294	0.692	0.655
XGBoost	1.801	2.318	0.686	0.647
Ridge	1.785	2.320	0.685	0.647
Baseline	1.857	2.399	0.669	0.649
LinearRegression	1.915	2.441	0.651	0.609
OLS_PCA	1.984	2.511	0.640	0.637
PCA + Linear Regression	1.934	2.528	0.626	0.580

```
Baseline stats (last season → current season):
MAE: 1.857
RMSE: 2.399
R-squared: 0.669
Adjusted R-squared: 0.649
```

In [46]:
```
# Bar plot for MAE and R²
fig, ax1 = plt.subplots(figsize=(10,8))

results_df_plot = results_df.copy()
models_list = results_df_plot.index

x = np.arange(len(models_list))
width = 0.35

ax1.bar(x - width/2, results_df_plot["MAE"], width, label="MAE")
ax1.set_ylabel("Mean Absolute Error")
ax1.set_xlabel("Model")
ax1.set_xticks(x)
ax1.set_xticklabels(models_list, rotation=30, ha="right")

# Add secondary axis for R²
ax2 = ax1.twinx()
ax2.bar(x + width/2, results_df_plot["R-squared"], width, color="orange", label=
ax2.set_ylabel("R² Score")

# Legends
ax1.legend(loc="upper left")
ax2.legend(loc="upper right")

plt.title("Model Performance Comparison")
plt.tight_layout()
plt.show()
```



Modeling Conclusions

OLS is the clear best performer, while **PCA + Linear Regression** and **OLS_PCA** performed poorly. Only **OLS** and the tree-based models (**XGBoost** and **Random Forest**) managed to outperform the **Baseline** in terms of R-squared, MAE and RMSE.

---

1. The Dominance of OLS and the Regression Paradox

The key takeaway is that **OLS (Ordinary Least Squares) is the best model** ($\text{MAE} = 1.750, R^2 = 0.714$).

- **Strong Linear Relationship Confirmed:** The high R^2 value ($\mathbf{0.714}$) for OLS still confirms a **strong linear relationship** between the selected features and the target variable, `next_fantasy_z_9cat`.
- **The Paradox of Poor Linear Model Performance:** It is highly unusual for **Linear Regression** ($\text{MAE} = 1.915, R^2 = 0.651$), **Ridge** ($\text{MAE} = 1.813, R^2 = 0.674$), and the PCA-based models to perform worse than the **Baseline** ($\text{MAE} = 1.857, R^2 = 0.669$).
  - **The Likely Culprit: Multicollinearity.** Your correlation matrix (image **image_cfeb8f.png**) shows **very strong correlations** between features, such as `PTS` vs. `TOV` (-0.84), `PTS` vs. `AST` (0.70), and `REB` vs. `BLK` (0.64). When using the full feature set, this **multicollinearity** can make the standard Linear Regression coefficients highly unstable and inflate their variance, leading to

poor generalization (i.e., poor out-of-sample performance, resulting in the low R^2 and high MAE).

- **Why OLS Excels:** The OLS model used here appears to have successfully applied a **feature selection** step (likely by dropping features with high p-values) which is the most effective strategy for this dataset. By dropping the redundant/unstable features, it successfully stabilized the model, leading to the best performance.
- **Why Ridge Fails:** Ridge Regression is designed to handle multicollinearity by shrinking coefficients, but in this specific instance, the regularization penalty may have been too strong or the combination of highly correlated features was too complex for a standard Ridge approach to stabilize effectively without severely harming predictive power (resulting in $R^2 = 0.674$, only marginally better than the Baseline).

---

2. Failure of Dimensionality Reduction (PCA)

The **PCA-based models are the worst performers** ($\mathbf{R^2 \approx 0.63}$).

- **PCA + Linear Regression** and **OLS_PCA** performed poorly, suggesting that simply transforming the features via **PCA destroyed some of the most critical predictive information**.
- PCA is a rotation that maximizes variance but does not account for the target variable. In this case, **feature selection (as done by OLS)** was far more effective than feature transformation (PCA) at dealing with the feature redundancy.

---

3. Tree Models Outperform Most Linear Models

The tree-based models, **XGBoost** ($\mathrm{MAE} = 1.801, R^2 = 0.686$) and **Random Forest** ($\mathrm{MAE} = 1.816, R^2 = 0.692$), both significantly **outperformed the standard Linear Regression and Ridge** models.

- This suggests that while the overall relationship is mostly linear, the tree-based models' ability to handle **feature interactions** and **multicollinearity implicitly** allowed them to stabilize better than the unstable linear models. They are robust alternatives to the highly tuned OLS model.

---

4. Robust Feature Importance

The feature importance plots for **Random Forest** and **XGBoost** (images **image_d03d68.png** and **image_d03d82.png**) consistently highlight **scoring ability** as the top predictor.

- `num_PTS` and `num_PTS_per36` are the **most important features by a substantial margin**. This is a highly stable finding across all model iterations and corroborates the strong correlation of `PTS` with the target variable (0.88).

---

5. Examination of Residuals

The residual plots (images **image_d03d4a.png**, **image_cfebaa.png**, and **image_cfebad.png**) offer important insights:

- **Residual Distribution:** The histogram (image **image_d03d4a.png**) shows the residuals for PCA + Linear Regression **approximate a normal distribution** centered around zero, which is a key assumption for linear models.
- **Homoscedasticity:** The **Linear Regression** and **Ridge** Residuals vs. Predicted plots (images **image_cfebaa.png** and **image_cfebad.png**) show a relatively **random scatter of points** around the $y = 0$ line. This suggests the models' variance of errors is roughly constant (**homoscedasticity**), which is another positive assumption check for linear regression. The points do not fan out or show a clear pattern.

Additional Metrics to Consider

To get a more complete picture of model performance, especially when dealing with poor-performing models, you should consider adding:

1. **Mean Absolute Percentage Error (MAPE):**
   - **Formula:** $\text{MAPE} = \frac{100\%}{n} \sum_{t=1}^{n} \left| \frac{A_t - F_t}{A_t} \right|$
   - **Purpose:** Measures accuracy as a percentage of the actual value, which is more intuitive for interpretation (e.g., "The prediction is off by X% on average").
2. **Adjusted R-squared:**
   - **Purpose:** This metric penalizes models that add features that do not significantly improve the fit. Since your OLS model is the best performer and likely used feature selection, comparing its R^2 to its **Adjusted** R^2 would confirm that the discarded features were indeed non-contributory.
3. **Cross-Validation Scores (if not already done):**
   - **Purpose:** Ensures the reported scores are stable and not due to a single fortunate train/test split. Since the linear models are struggling to generalize, comparing the **standard deviation of MAE** across 5-10 folds would show the volatility of each model.

Forecasting player's fantasy performance at 2025-26

Based on our modelling section we will now proceed with predicting the leadgue ranking toward the upcoming season

In [47]:
```
df_forecast = predict_only.copy()
forecasts = {}
```

In [48]:
```
# Random Forest predictions
X_forecast = predict_only[feature_cols]
```

```
forecast_RF = pipe_RF.predict(X_forecast)
forecasts["pred_RandomForest"] = forecast_RF
df_forecast["pred_RandomForest"] = forecast_RF
```

In [49]:
```
def forecast_with_ols(ols_model, X_forecast_df, final_features):
    """
    ols_model: statsmodels fitted OLS
    X_forecast_df: DataFrame of preprocessor.get_feature_names_out() columns (an
    final_features: list of columns used during training (no 'const')
    Returns: pandas Series of predictions aligned to X_forecast_df.index
    """
    Xf = X_forecast_df.copy()
    # add missing training cols as zeros
    for c in final_features:
        if c not in Xf.columns:
            Xf[c] = 0.0
    # reduce to exactly final_features and cast to float
    Xf = Xf[final_features].astype(float)
    Xf_const = sm.add_constant(Xf, has_constant='add')
    preds = ols_model.predict(Xf_const)
    preds.index = Xf.index
    return preds
```

In [50]:
```
# OLS predictions
X_fore_arr = preprocessor.transform(predict_only)
X_fore_df = pd.DataFrame(X_fore_arr, columns=preprocessor.get_feature_names_out(

preds_OLS = forecast_with_ols(ols_model, X_fore_df, final_features)

forecasts["OLS"] = preds_OLS
df_forecast = df_forecast.copy()
df_forecast.loc[:, "pred_OLS"] = preds_OLS

print("NaNs in predictions:", preds_OLS.isna().sum())
```

```
NaNs in predictions: 0
```

In [51]:
```
forecast_results = df_forecast[["PLAYER_NAME", "SEASON_ID", "fantasy_z_9cat", "p

# Rank top players by prediction
forecast_top = forecast_results.sort_values("pred_OLS", ascending=False).head(20
print(forecast_top)

#df_forecast.to_csv("forecast_to_2025-26.csv", index = False)
```

	PLAYER_NAME	SEASON_ID	fantasy_z_9cat	pred_OLS	pred_RandomForest
1921	Victor Wembanyama	2024-25	16.605037	16.719948	10.269204
1506	Nikola Jokić	2024-25	17.539191	15.624058	11.293070
1740	Shai Gilgeous-Alexander	2024-25	16.562903	14.886174	12.230451
87	Anthony Davis	2024-25	11.999300	12.661495	12.058671
692	Giannis Antetokounmpo	2024-25	7.142473	11.306681	9.825232
1312	Luka Dončić	2024-25	10.321897	11.207628	10.555666
1903	Tyrese Haliburton	2024-25	10.194490	10.673968	6.313034
1909	Tyrese Maxey	2024-25	9.427006	9.956577	8.808089
941	Jayson Tatum	2024-25	9.204196	9.810708	8.661132
988	Joel Embiid	2024-25	7.645750	9.179041	11.603762
898	Jaren Jackson Jr.	2024-25	7.665074	9.140564	9.743778
92	Anthony Edwards	2024-25	8.628321	8.915679	8.904511
1233	Kristaps Porziņģis	2024-25	8.101312	8.914970	6.288431
232	Cade Cunningham	2024-25	7.898586	8.702767	9.906508
1179	Kevin Durant	2024-25	9.979623	8.511844	10.838263
519	Devin Booker	2024-25	7.689415	8.411127	6.544807
1122	Karl-Anthony Towns	2024-25	10.342544	8.407029	8.635041
1254	Kyrie Irving	2024-25	9.270574	8.228396	8.778872
1851	Trae Young	2024-25	6.835737	8.201882	6.701368
279	Chet Holmgren	2024-25	6.687919	8.192794	5.549691

Applying best models on forecastig all categories - will help later to create a team-building reccomandation which will be based on punt strategies

In [52]:
```
def train_models_per_target(train_df, test_df, raw_feature_columns, target_cols,
                            rf_params=None):
    """
    train_df/test_df: dataframes with rows and target cols present
    raw_feature_columns: columns used as input to preprocessor (raw features)
    target_cols: list of target columns to train e.g. ['z_PTS', 'z_REB', ..., 'f
    preprocessor: fitted ColumnTransformer/processor (must be fitted on training
    returns: dicts ols_models, rf_models, and performance summary
    """
    rf_params = rf_params or {"n_estimators":300, "max_depth":10, "random_state"
    ols_models = {}
    rf_models  = {}
```

```
    pca_lr_models = {}
    perf = []

    # Transform training and test raw features to named DataFrames
    X_train_arr = preprocessor.transform(train_df[raw_feature_columns])
    feat_names = list(preprocessor.get_feature_names_out())
    X_train_df = pd.DataFrame(X_train_arr, columns=feat_names, index=train_df.in

    X_test_arr = preprocessor.transform(test_df[raw_feature_columns])
    X_test_df = pd.DataFrame(X_test_arr, columns=feat_names, index=test_df.index

    # For PCA we will standardize inside PCA pipeline; here we use raw arrays
    for target in target_cols:
        # prepare y aligned
        y_train = train_df[target].reindex(X_train_df.index).copy()
        y_test  = test_df[target].reindex(X_test_df.index).copy()

        # 1) OLS + backward elimination
        try:
            ols_model, final_features = backward_elimination_ols_named(X_train_d
            # prepare X_test subset for prediction
            X_test_for_ols = X_test_df.reindex(columns=final_features).astype(fl
            X_test_for_ols_const = sm.add_constant(X_test_for_ols, has_constant=
            y_pred_ols = ols_model.predict(X_test_for_ols_const)
            r2_ols = r2_score(y_test, y_pred_ols)
        except Exception as e:
            print(f"OLS failed for {target}: {e}")
            ols_model, final_features, r2_ols, y_pred_ols = None, [], np.nan, np

        ols_models[target] = {"model": ols_model, "features": final_features}

        # 2) Random Forest
        try:
            rf = RandomForestRegressor(**rf_params)
            rf.fit(X_train_df, y_train)
            y_pred_rf = rf.predict(X_test_df)
            r2_rf = r2_score(y_test, y_pred_rf)
        except Exception as e:
            print(f"RF failed for {target}: {e}")
            rf, r2_rf, y_pred_rf = None, np.nan, np.full(len(X_test_df), np.nan)
        rf_models[target] = {"model": rf}

        perf.append({
            "target": target,
            "r2_ols": float(r2_ols) if np.isfinite(r2_ols) else np.nan,
            "r2_rf" : float(r2_rf)  if np.isfinite(r2_rf) else np.nan,
        })

    perf_df = pd.DataFrame(perf).set_index("target")
    return ols_models, rf_models, perf_df, feat_names
```

In [53]:

```
def forecast_targets_on_predict_only(predict_df, preprocessor, ols_models, rf_mo
    """
    predict_df: predict_only dataframe (2024-25)
    preprocessor: fitted transformer
    ols_models, rf_models, pca_lr_models: outputs from train_models_per_target
    feat_names: names returned by preprocessor.get_feature_names_out()
    raw_feature_columns: raw columns before preprocessor
    target_cols: list of target names
    Returns predict_df with appended prediction columns
```

```
    """
    # Transform predict set
    X_fore_arr = preprocessor.transform(predict_df[raw_feature_columns])
    X_fore_df = pd.DataFrame(X_fore_arr, columns=feat_names, index=predict_df.in

    # prepare result df
    res = predict_df.copy()

    for target in target_cols:
        # OLS
        ols_meta = ols_models.get(target, {})
        ols_model = ols_meta.get("model", None)
        final_features = ols_meta.get("features", [])
        if ols_model is not None and len(final_features)>0:
            try:
                preds_ols = forecast_with_ols(ols_model, X_fore_df, final_featur
                res[f"pred_OLS_{target}"] = preds_ols
            except Exception as e:
                print(f"Failed OLS forecast for {target}: {e}")
                res[f"pred_OLS_{target}"] = np.nan
        else:
            res[f"pred_OLS_{target}"] = np.nan
        # RF
        rf_meta = rf_models.get(target, {})
        rf_model = rf_meta.get("model", None)
        if rf_model is not None:
            try:
                res[f"pred_RF_{target}"] = rf_model.predict(X_fore_df)
            except Exception as e:
                print(f"Failed RF forecast for {target}: {e}")
                res[f"pred_RF_{target}"] = np.nan
        else:
            res[f"pred_RF_{target}"] = np.nan

    return res
```

In [54]:

```
# 1. define raw_feature_columns exactly as used when fitting preprocessor:
# Example: raw_feature_columns = feature_cols  # if feature_cols are the raw col
raw_feature_columns = feature_cols  # adjust if different in your notebook

# 2. split train/test as in your notebook - ensure indexes remain intact
train_df = df[df["SEASON_ID"] < "2024-25"].copy()       # 2020-21 -> 2023-24
test_df  = train_df[train_df["SEASON_ID"] == "2023-24"].copy()  # validation sea
train_df = train_df[train_df["SEASON_ID"] < "2023-24"].copy()  # earlier seasons

stats = ['PTS', 'REB', 'AST', 'STL', 'BLK', 'FG3M', 'FG%', 'FT%', 'TOV']
# target columns: 9 cat z-scores + total
target_cols = [f"next_z_{c}" for c in stats] + ["next_fantasy_z_9cat"]

linear_X_train, y_train = linear_train[feature_cols], train[target_cols]
linear_X_test, y_test = linear_test[feature_cols], test[target_cols]
train_df = pd.concat([linear_X_train, y_train], axis=1)
test_df = pd.concat([linear_X_test, y_test], axis=1)

# 4. train models
ols_models, rf_models, perf_df, feat_names = train_models_per_target(
    train_df=train_df,
    test_df=test_df,
    raw_feature_columns=raw_feature_columns,
    target_cols=target_cols,
```

```
    preprocessor=preprocessor,
    rf_params={"n_estimators":300,"max_depth":10,"random_state":42},
)

print("Per-target performance (R2):")
display(perf_df)

# 5. forecast for predict_only (2024-25)
df_forecast = forecast_targets_on_predict_only(
    predict_df=predict_only,
    preprocessor=preprocessor,
    ols_models=ols_models,
    rf_models=rf_models,
    feat_names=feat_names,
    raw_feature_columns=raw_feature_columns,
    target_cols=target_cols
)

# 6. Aggregate predicted total z-score if desired (average over categories)
zcols_ols = [f"pred_OLS_{col}" for col in [f"next_z_{c}" for c in stats]]
zcols_RF = [f"pred_RF_{col}" for col in [f"next_z_{c}" for c in stats]]

df_forecast["pred_OLS_fantasy_z_9cat"] = df_forecast[zcols_ols].mean(axis=1)
df_forecast["pred_RF_fantasy_z_9cat"] = df_forecast[zcols_RF].mean(axis=1)

# 7. inspect top players
df_forecast.sort_values("pred_OLS_fantasy_z_9cat", ascending=False).head(20)
```

```
Per-target performance (R2):
```

target	r2_ols	r2_rf
next_z_PTS	0.802367	0.814476
next_z_REB	0.793509	0.784694
next_z_AST	0.828924	0.800337
next_z_STL	0.478949	0.477391
next_z_BLK	0.795663	0.770997
next_z_FG3M	0.745881	0.739467
next_z_FG%	0.559642	0.588275
next_z_FT%	0.398407	0.587413
next_z_TOV	0.767357	0.750265
next_fantasy_z_9cat	0.659242	0.692121

Out[54]:

	PLAYER_NAME	PLAYER_ID	SEASON_ID	LEAGUE_ID	TEAM_ID	PLAYER_AGE
1921	Victor Wembanyama	NaN	2024-25	NaN	NaN	21.0
1506	Nikola Jokić	203999.0	2024-25	0.0	1.610613e+09	30.0
1740	Shai Gilgeous-Alexander	1628983.0	2024-25	0.0	1.610613e+09	26.0
87	Anthony Davis	203076.0	2024-25	0.0	0.000000e+00	32.0
1312	Luka Dončić	1629029.0	2024-25	0.0	0.000000e+00	26.0
1903	Tyrese Haliburton	1630169.0	2024-25	0.0	1.610613e+09	25.0
692	Giannis Antetokounmpo	203507.0	2024-25	0.0	1.610613e+09	30.0
1909	Tyrese Maxey	1630178.0	2024-25	0.0	1.610613e+09	24.0
941	Jayson Tatum	1628369.0	2024-25	0.0	1.610613e+09	27.0
898	Jaren Jackson Jr.	1628991.0	2024-25	0.0	1.610613e+09	25.0
92	Anthony Edwards	1630162.0	2024-25	0.0	1.610613e+09	23.0
988	Joel Embiid	203954.0	2024-25	0.0	1.610613e+09	31.0
1233	Kristaps Porziņģis	204001.0	2024-25	0.0	1.610613e+09	29.0
1179	Kevin Durant	201142.0	2024-25	0.0	1.610613e+09	36.0
551	Donovan Mitchell	1628378.0	2024-25	0.0	1.610613e+09	28.0
1254	Kyrie Irving	202681.0	2024-25	0.0	1.610613e+09	33.0
841	Jalen Brunson	1628973.0	2024-25	0.0	1.610613e+09	28.0
519	Devin Booker	1626164.0	2024-25	0.0	1.610613e+09	28.0
232	Cade Cunningham	1630595.0	2024-25	0.0	1.610613e+09	23.0
1122	Karl-Anthony Towns	1626157.0	2024-25	0.0	1.610613e+09	29.0

20 rows × 107 columns

In [55]:
```
#df_forecast.to_csv("forecast_to_2025-26_including_cats.csv", index = False)
```