Advanced Models for Football Match Outcome Prediction

Bayesian Hierarchical Models

Bayesian hierarchical models use **multi-level structures** to capture team and player effects with uncertainty. Instead of fixed coefficients for each team, they treat parameters (like a team's attacking/defensive strength or home advantage) as random variables drawn from a league-wide distribution. This **partial pooling** improves estimates for teams with less data and naturally provides probability distributions for outcomes. Bayesian models can also incorporate **expert knowledge or historical data as priors**, such as using previous season performance or FIFA rankings to inform current team strengthfrontiersin.org. These models are applicable to predicting win/draw/loss or scores by estimating the scoring rates of each team.

- **Application to football:** Hierarchical Poisson models are common, where each team has attack and defense parameters that vary around a league average. For example, Baio and Blangiardo (2010) used a Bayesian hierarchical model to predict Italian Serie A match scores, assigning each team its own offense/defense ability while accounting for home advantagediscovery.ucl.ac.uk. Such models were shown to improve predictive fit but needed tweaks to avoid **over-shrinkage** of team strengths, leading to mixture models for better accuracydiscovery.ucl.ac.uk. Recent research on Brazil's Série A compared **hierarchical vs non-hierarchical Poisson regressions**, confirming that a two-level (team-level) random-effects model can enhance predictionsfrontiersin.org.
- **Notable use cases:** Bayesian approaches have been used in tournament forecasts – e.g. **Suzuki et al. (2006)** incorporated FIFA rankings and expert opinions as priors in a Bayesian model for World Cup matches, achieving good prediction ratesfrontiersin.org. Other studies applied dynamic Bayesian updating of team abilities over time (using state-space models or Kalman filters) to capture form changesfrontiersin.org. Overall, hierarchical Bayesian models offer a flexible framework to include various factors (team strength, venue, player injuries) with uncertainty, often yielding well-calibrated probabilities for match outcomes.

Survival Analysis (Time-to-Event Models)

Survival analysis focuses on **time until an event occurs** – in football, this typically means modeling the timing of goals (or other events like red cards). Instead of directly predicting final scores, a survival model treats goals as events in a timeline, using hazard functions or survival curves to estimate the probability of a goal as time progresses. By modeling the **distribution of goal times**, one can derive the likelihood of different scorelines or which team scores first. This approach captures the dynamic nature of a match – e.g. the longer a match stays 0–0, the more likely a late goal might occur from pressing, or how a goal hazard changes after a red card.

- **Application to football:** One can use a **Cox proportional hazards model** or parametric survival model to examine scoring. For example, Glasson and Bedford (2009) applied survival analysis to **time of first goal**, finding how the hazard of a goal relates to teams' relative strength (rank) and other factorsjournals.sagepub.com. Similarly, researchers have studied the effect of scoring the first goal on the timing of the second using Cox models (treating the first goal as a time-dependent variable).
- **Advanced survival models:** A recent approach by Boshnakov et al. used a **Weibull inter-arrival time model** for goalsfrontiersin.org. In their framework, each team's goal occurrences follow a Weibull distribution (a flexible survival distribution) for the time gaps between goals. They then coupled the two teams' processes with a **copula** to form a joint distribution of home and away goalsfrontiersin.org. This effectively produces a full match score prediction model: the Weibull hazard captures how goal likelihood changes over time, and the copula introduces a correlation between teams (e.g. accounting for the fact that a very defensive match yields low goals for both sides). By treating goal-scoring as a time-to-event problem, these models can simulate or estimate probabilities for win/draw/loss or exact scores given how long it typically takes teams to score. Survival analysis thus adds a temporal dimension to match outcome modeling, beyond what static regression models provide.

Poisson, Negative Binomial, and Zero-Inflated Count Models

Predicting exact scores or total counts (goals, corners, cards, etc.) often involves **count regression models**. The classic choice is the Poisson model, which assumes events occur at a constant rate and the probability of scoring k goals follows a Poisson distribution. In football analytics, it's common to use a **"double Poisson"** setup – one Poisson model for home goals and one for away goals – possibly with different means to reflect team strengths and home advantagemdpi.com. However, real match data can violate Poisson assumptions (equal mean and variance, independent team scores). Advanced count models address these issues:

- **Negative Binomial Regression:** If the variance of goals or other counts is higher than the mean (a frequent case in sports due to occasional blowout scores or high variability in cards/fouls), a negative binomial model can be used. It adds an extra parameter to model **over-dispersion**. In fact, early studies found that while Poisson is a good first model for goals, a Negative Binomial may fit better for datasets where some teams or matches produce unpredictably high scoresmdpi.com. Bäcklund and Nils (2021), for instance, compared negative binomial vs Poisson for football scores and found the former could better accommodate the variance in goal countsfrontiersin.org.

- **Zero-Inflated Models:** Football data often have **excess zeros**, particularly in goals (e.g. 0–0 draws, or a team failing to score). A standard Poisson might underestimate the frequency of zero-goal outcomesmdpi.com. *Zero-inflated Poisson (ZIP)* or *zero-inflated negative binomial* models address this by blending two processes: one that generates structural zeros (e.g. with some probability a team simply doesn't score at all), and another that generates counts according to a Poisson/NB for cases when a team *does* scorefrontiersin.org. This effectively boosts the probability of zero goals beyond the Poisson expectation. In a recent Premier League analysis, a simple Poisson model severely under-predicted 0–0 matches (assigning only 0.9% chance to a Man City vs Liverpool match ending 0–0, when goalless draws actually occurred in 23 of 380 matches)mdpi.com. The authors suggest adding a zero-inflation term to correct this biasmdpi.com. By estimating a zero-inflation parameter, the model explicitly accounts for the likelihood of a goal drought scenariomdpi.com.
- **Use in football:** Poisson regression has been a cornerstone of football score prediction since Maher's seminal 1982 paper. It forms the basis of many betting models and was later extended by Dixon & Coles (1997) with a slight covariance adjustment for low-scoring games. Negative binomial models have been tested as alternatives when data show greater variabilitymdpi.com. Zero-inflated models are newer in this domain, but studies have begun to apply them. For example, a team-specific zero-inflated Poisson was proposed in 2022 to improve goal predictions between certain teamsarxiv.org. Overall, these count models are well-suited for predicting **exact scores** (via two coupled distributions for home/away goals) or **totals** (goals, corners, cards, etc.), with more advanced versions capturing the idiosyncrasies of football data (lots of 0–0, occasional high counts, etc.)frontiersin.org.

Generalized Additive Models (GAMs)

Generalized Additive Models extend traditional linear regression by allowing **non-linear effects** of predictors while retaining interpretability. A GAM predicts an outcome as a sum of smooth functions of the inputs: $g(E[Y]) = \beta_0 + f_1(X_1) + f_2(X_2) + \dots$, where f_i are spline or loess functions fitted from data. This flexibility is useful for football data, where relationships may not be purely linear (e.g., the effect of possession percentage on win probability might increase steeply up to a point then level off). GAMs can handle both classification (via logistic link for win/draw/loss probability) and regression (via identity/log link for counts) by choosing an appropriate link function in the GLM family.

- **Application to football:** GAMs have been successfully used to model **expected goals (xG)** and other performance metrics in an interpretable way. For instance, Van Haaren et al. (2021) introduced an **explainable GAM for predicting the value of shots**, i.e. the chance a shot will result in a goalarxiv.org. By using smooth functions of shot features (like distance, angle, assist type), the GAM could match the accuracy of complex black-box models while being **easier to interpret for coaches**arxiv.org. The model broke down the contribution of each feature (e.g. shot distance effect curve) so practitioners could trust the numbersarxiv.org.

- **Why GAMs are useful:** They capture **non-linear patterns** (which are common in sports data) without manual feature engineering. For example, a GAM can learn that very low or very high possession values both lead to lower win probability (an inverted U-shape), or that the effect of a player's age on performance is non-linear. Indeed, studies on player performance trends have used GAMs to reveal subtle non-linear age curves and match intensity effectsnature.com. In football outcome prediction, one could use a GAM to allow, say, team Elo rating to have a diminishing returns effect, or different smooth terms for a team's form over the last N games. Decroos and Davis (2019) applied a GAM to predict the probability of a goal being scored in a sequence of play, as a more interpretable alternative to tree-based models.
- **Notable example:** Decroos & Davis replaced a complicated ensemble (used in a **VAEP** player value model) with a **simpler GAM using 10 features** to predict goal events, making the model's reasoning transparentsemanticscholar.org. This shows how GAMs can serve in **sports analytics** to balance accuracy and interpretability. Overall, GAMs are well-suited for football data, which often has nonlinear relationships and domain-specific thresholds (like fatigue effects after 70 minutes, or exponential drop in win probability as goal deficit increases). By visualizing each feature's effect, analysts can **explain the model's predictions**, an important factor for adoption by coaches and clubs.

Bivariate Outcome Models (Joint Score Distributions)

While one can model goals for each team separately, treating them jointly can improve exact score forecasts. **Bivariate distribution models** capture the correlation between two teams' scores. Two main approaches have emerged: (1) direct multivariate distributions (like bivariate Poisson or bivariate Negative Binomial), and (2) modeling the **goal difference** and total (or one team's goals and the goal difference). These methods acknowledge that the scores are not independent – for instance, very high scoring games typically involve both teams scoring more (or one scoring a lot and the other less), and low scoring games often mean low for both (a defensive battle).

- **Bivariate Poisson models:** A bivariate Poisson allows the two goal counts to share a component. In the Karlis & Ntzoufras model, each match has three Poisson components: one for home goals, one for away goals, and one shared component that adds to both (creating correlation). Their 2003 paper showed that a **bivariate Poisson fit** could better match observed score distributions in soccermdpi.com. Notably, they found it useful to include a **common home advantage parameter** and that one could often treat attack and defense as constant across home/away for simplicitymdpi.commdpi.com. Bivariate Poisson models have been widely cited in the literature as an improvement over independent Poissons for exact score predictions.

- **Skellam (goal-difference) models:** The Skellam distribution (difference of two independent Poissons) gives the distribution of goal difference. Modeling goal difference directly is another route: fit a Skellam for the score margin and maybe a separate model for total goals or one team's goals. Karlis & Ntzoufras also proposed a **Bayesian Skellam model** that included **zero-inflation for draws** and a correlation between teams' attack/defense strengthsfrontiersin.org. In effect, they inflated the probability of zero goal difference (draws) beyond the Skellam baseline, acknowledging that draws can be more frequent than naive models expectfrontiersin.org. This approach focuses on getting win/draw/loss probabilities right, since those relate directly to goal difference.

- **Copula and other multivariate methods:** Instead of a specific parametric form, one can model each team's goals with a marginal distribution (Poisson or other) and then join them with a **copula** to induce a correlation. Boshnakov et al. (2017) did this by using a **Weibull count process** for each team's goals (as discussed earlier) and coupling them with a copula to produce a full joint distribution for (home_goals, away_goals)frontiersin.org. The copula allows flexible correlation structures (for example, ensuring that if one team scores a lot, the other's likelihood of also scoring might increase in open-play scenarios or decrease if one dominates completely, depending on the chosen copula). Copula models are less common but offer flexibility to fit empirical joint distributions of scores.

- **Use cases:** Bivariate and goal-difference models are particularly useful for **exact score forecasting**. For example, a betting analyst might use a Dixon-Coles model (a slight tweak on independent Poisson with a covariance adjustment) to predict the probabilities of each possible scoreline for setting odds. Academic studies have found that incorporating a correlation term (like Dixon-Coles' adjustment for low scores or a full bivariate model) improves predictive log-likelihood of match scoresfrontiersin.org. Additionally, these models can be extended: **dynamic bivariate models** allow team strengths to evolve over time (Koopman & Lit (2015) used a state-space bivariate Poisson to update team attack/defense each weekfrontiersin.org), and **multivariate extensions** can include extra outcomes (like modeling goals and red cards together in a multivariate framework to see how a red card influences goals). In summary, jointly modeling scores yields more accurate and realistic predictions for exact scores, since it captures the dependency between teams' performances in a match.

Elo Ratings and Bradley-Terry Models

Not all effective football prediction models are based on traditional regression or ML – **rating systems** like Elo and Bradley-Terry are powerful for win/draw/loss classification. These approaches convert past match results into team strength ratings, which in turn predict future outcomes.

- **Elo Ratings:** Originally developed for chess, Elo ratings have each team start with a rating (e.g., 1500) and then **update after every match** based on the result and the expected result. Beating a strong opponent yields a big rating gain, while losing to a weaker team causes a big loss; draws cause small adjustments depending on expectations. Elo ratings are **dynamic** and self-correcting – they reflect current form and strength. To predict a match, the difference in Elo ratings between two teams is mapped to an expected win probability (usually via a logistic curve). Elo models are **simple but effective**: they've been shown to rival more complex models in accuracy. For example, during World Cups, Elo-based predictions often outperform FIFA's official rankings in predicting match outcomesdubstat.com. Elo ratings provide a natural probability estimate for each outcome; by comparing two teams' ratings, one can calculate the chance of win/draw/loss (draws are handled by assuming a certain draw probability or by using an Elo variant that includes draws). Sports analysts and bettors favor Elo because of its **predictive accuracy and real-time adaptability**dubstat.com. It's common to see websites like ClubElo or FiveThirtyEight use Elo systems for league and international match forecasts.

- **Bradley-Terry and extensions:** The Bradley-Terry model is a classic statistical model for pairwise comparisons (matches). It assigns each team a strength parameter p_i; the probability team A beats team B is $P(A\text{ wins}) = \frac{p_A}{p_A + p_B}$. This can be reparameterized in logistic form $\logit(P_A) = \beta_A - \beta_B$. Fitting a Bradley-Terry model to past results yields a rating for each team (similar to Elo but derived from a maximum likelihood estimation rather than sequential updating). **Extensions** like the Davidson model incorporate draws by adding a draw parameter. In football analytics, Bradley-Terry models have been used to rate teams or even players (e.g., who wins more duels). They are essentially equivalent to **logistic regression on match outcomes** with team indicators. One advantage is that they naturally account for strength of schedule – beating strong teams raises p_i more significantly. These models are less frequently mentioned by name in soccer literature (often overshadowed by Elo), but the concept underlies many ranking systems.

- **Use in football:** Rating-based predictions are well-suited for structured data and **time-series forecasting of matchups** because they carry information from past matches into future predictions. A system like Elo implicitly captures momentum and form. For instance, if a mid-table team suddenly improves and wins several games, Elo will quickly raise its rating, and thus the model will predict higher chances for that team in upcoming fixtures – effectively a form of time-series adaptation. Ratings can also integrate player stats (some advanced versions adjust team ratings if key players are injured, etc.). Empirically, Elo ratings have demonstrated strong predictive power: one article notes that Elo's mathematically-updated rankings yield more accurate forecasts of match results than subjective or static systemsdubstat.comdubstat.com. Many bookmakers use Elo-like systems as part of their models. Bradley-Terry models, being equivalent to a logistic regression, can incorporate additional features (home advantage can be added as a constant term β_{home}; one can extend it to account for team form by making p_i time-dependent, etc.). These rating approaches are not in the user's PDF but are **commonly used in sports analytics** for classification tasks (will team A win, draw, or lose). They are particularly appealing when one has **structured historical data** (season-by-season results) and wants a quick, interpretable prediction model.

Transformer-Based Time-Series Models

Modern deep learning has introduced **transformer architectures** that excel at sequence modeling. Unlike RNNs (e.g. LSTM in the PDF) which process time steps sequentially, transformers use **self-attention** to learn long-range dependencies in parallel. In time-series forecasting, transformers can capture complex patterns, seasonality, and interactions across multiple series. For football, this could mean modeling sequences of past match features or player statistics over time to predict future outcomes or metrics. For example, one could feed a transformer the timeline of team performances (with features like goals, shots, possession in recent matches, lineup changes, etc.) to forecast the probability of winning the next game or to predict next match's goal counts.

- **Why transformers:** Transformers have shown **state-of-the-art performance** on many forecasting benchmarks, often outperforming traditional models and RNN-based modelsnature.com. They handle long sequences well and can incorporate covariates (e.g., known schedule, opponent features, weather) through appropriate architecture design. In a sports context, a transformer could learn patterns like "Team A tends to struggle in away games after a Champions League midweek match" or capture interactions like "if key player X is absent, and the team faces a high-pressing opponent, their goal rate drops significantly" – patterns that might be hard to encode manually.

- **Application potential:** While not yet prevalent in published football literature, these models are beginning to appear. Researchers are exploring transformers for sports time-series, such as injury prediction or trajectory forecasting. In match outcome prediction, one could use a **Temporal Fusion Transformer (TFT)** or similar model that takes as input: team form (as a sequence of past match stats), opponent strength series, and static info (team budget, coach, etc.), and outputs win/draw/loss probabilities or expected goals for the next match. The **attention mechanism** would allow the model to weigh the importance of specific past games (e.g., giving more weight to recent matches or to games against similar-caliber opponents). Because transformers are data-hungry, this approach works best with rich data (multiple seasons, player-level stats, etc.). Some contemporary studies in other domains (economics, meteorology) show transformers achieving **top accuracy in multivariate forecasting**nature.com, which suggests that as football clubs accumulate more data (tracking data, detailed event data over years), transformer models could become very powerful for predictions like total shots, possession, or even *in-game win probability* that evolves over time.
- **Example and performance:** A 2025 study proposed *MultiPatchFormer*, a transformer variant for multivariate time series, and it outperformed many baseline models on economic and energy datasetsnature.com. This indicates the model's ability to capture multi-scale temporal patterns and cross-series relationships – analogous to capturing multi-season cycles or team interactions in sports. While specific football-focused transformer models are still emerging, the technique is promising. **In summary**, transformer-based models represent the cutting edge for structured time-series prediction, offering the ability to model complex, non-linear temporal dynamics in football data. As data availability grows, we expect to see them perform strongly in problems like **match outcome forecasting, player performance over a season, or injury risk prediction**, complementing the more classical statistical models above.

Sources: The above approaches have been synthesized from sports analytics literature and recent machine learning research. Key references include traditional studies on Poisson and Bayesian modelsmdpi.comfrontiersin.orgfrontiersin.org, modern developments like zero-inflated and dynamic modelsmdpi.comfrontiersin.org, explainable models for expected goalsarxiv.org, and general insights from time-series forecasting advancesnature.com. These models, though not covered in the user's PDF of 17 basic ML models, are highly relevant for football (soccer) outcome prediction and have demonstrated their value in both research and practice. Each can be selected or combined depending on the task – classification (win/draw/loss), exact score prediction, or regression for match statistics – to improve predictive performance while accounting for the unique distributional characteristics of football data.

Sources

You said:

other statistical models or predictions models that are highly relevant for football (soccer) outcome prediction and have demonstrated their value in both research and practice

ChatGPT said:

To help you further, could you confirm if you're looking for:

1. **Only statistical models** (like those from econometrics or applied statistics), or are **machine learning models** also okay if they are used in practice for football?
2. Should the focus be only on **match outcome prediction** (win/draw/loss), or also include **in-play events** (e.g., real-time predictions, expected goals, player performance, etc.)?
3. Are you interested in **real-world implementations** (e.g., models used by betting companies or clubs) or just **peer-reviewed research models**?

This will help me prioritize the most applicable and evidence-backed models.

You said:

1. Only statistical models and predictions models only 2. both but specify with model is good for in play eents and which one is good for outcome match predictions 3. real world implementation in a pyton app

ChatGPT said:

Thanks for clarifying. I'll compile a list of statistical and predictive models—excluding pure ML models—that are highly relevant for football match outcome prediction or in-play event forecasting, and that have been implemented in real-world Python applications.

I’ll indicate for each model whether it's suited for match outcome prediction (e.g., win/draw/loss, scoreline) or for in-play events (e.g., expected goals, cards, shots), and include examples of real-world usage where available. I’ll let you know when the research is ready!

Research completed in 17m · 13 sources · 138 searches

Match Outcome Prediction Models

Poisson Goal Models (and Variants)

Use case: Best for predicting full-time scores and match outcomes (win/draw/loss probabilities).
How it works: Each team’s goals are modeled as a Poisson-distributed random variable. In the basic model, home and away goals are assumed independent, with each team having an **expected goals** value based on its attack strength, opponent’s defense, and home-field advantage. For example, if Team A averages 1.5 goals and Team B 1.0 goals, a Poisson model can estimate the probability of any scoreline (e.g. 2–1, 0–0, etc.). The **Skellam distribution** (difference of two Poissons) gives a direct handle on win/draw/loss probabilities. This approach is effective because real soccer scores roughly follow a Poisson pattern, making it straightforward to calculate outcome oddspinnacle.bet. However, the basic independent Poisson tends to **under-predict draws** (especially 0–0 or 1–1 outcomes) since it doesn’t account for correlation in scoring.
Extensions and why effective: *Dixon-Coles model* is a famous improvement that adds a correlation adjustment (parameter ρ) to boost the probability of low-scoring drawspypi.org. In practice this means the model explicitly increases the likelihood of outcomes like 0–0, 1–0 or 0–1, aligning better with empirical datapypi.org. Dixon-Coles also proposed down-weighting older matches so that recent form counts more. Other variants include *bivariate Poisson* models, which introduce a shared factor so that one team’s high scoring can slightly reduce the other’s scoring rate (capturing defensive interplay), and using a Skellam distribution directly for goal **difference** modeling (useful for handicap betting). These statistical tweaks improve predictive accuracy by fitting soccer’s tendency for extra draws and the defensive-offensive balance between teams.
Real-world use and Python implementation: Bookmakers and analysts commonly use Poisson-based models for odds-making – for instance, Pinnacle’s betting guides demonstrate Poisson to forecast soccer scorespinnacle.bet. Data scientists often fit Poisson regression (a GLM) to estimate each team’s attack/defense parameters. In Python, one can implement this with statsmodels or optimize a likelihood function manually (as shown in many blog tutorials). There are also ready libraries – e.g. the **penaltyblog** package – which offers high-performance Poisson, Dixon-Coles, and bivariate Poisson models for match outcome predictiongithub.com. These models have been validated in academic research and are used in practice for setting odds and powering prediction apps.

Rating Systems (Elo and Elo-based Models)

Use case: Primarily for predicting match results (win/draw/loss) and updating team strength over time.
How it works: Elo ratings assign each team a numerical strength rating. After each game, ratings are updated based on the result (teams gain points for wins, lose points for losses, with the exchange amount depending on the expected result). The **rating difference** between two teams can be converted into an expected win probability via a logistic formula. For example, a 100-point Elo advantage corresponds to roughly a 64% win chance for the higher-rated sideen.wikipedia.org. Draws are handled by smaller rating exchanges or an explicit draw probability formula. Elo's strength is its iterative nature – a team's rating evolves as results come in, reflecting form and quality changes. It has a solid statistical basis and is **widely used in soccer** – FIFA's women's world ranking and many public models use Elo or variantsen.wikipedia.org.
Why it's effective: Elo is a **proven predictor** because it encapsulates a lot of historical performance information in one number per team. It's especially good for head-to-head outcome probabilities and inherently accounts for regression to the mean (a string of upset wins boosts a team's rating, raising expectations going forward, and vice versa). Unlike Poisson models, Elo doesn't directly yield scorelines, but it focuses on win/lose likelihood. Extensions like **Glicko** (which adds rating uncertainty) or **TrueSkill** (a Bayesian Elo for games with draws) similarly see use in sports analytics to maintain up-to-date team strength estimates.
Real-world use and Python implementation: Many betting and prediction systems incorporate Elo-type ratings. FiveThirtyEight's well-known club soccer forecasts, for example, include an Elo component in their Soccer Power Indexfivethirtyeight.comfivethirtyeight.com. Clubs and analysts might use Elo to track team performance or as features in larger models. Implementing Elo in Python is straightforward – it's essentially a few lines of code to update ratings after each match. Libraries like elo-python or the penaltyblog toolkit provide convenience functions as well (the latter even includes Elo and other rating systems built-ingithub.com). Overall, Elo provides a simple yet effective baseline model for match outcome prediction and is often combined with other approaches for more nuanced forecasting.

Bayesian Hierarchical Models

Use case: Best suited for match outcome and score predictions, especially when we want to estimate team abilities with limited data or incorporate uncertainty.
How it works: These models extend the Poisson goal framework by treating team **attack and defense strengths as random effects** (latent variables) rather than fixed values. In a typical hierarchical model, each team *i* has an attack strength α_i and defense strength β_i drawn from a population distribution (often centered around league-average). Goals scored by Team A against Team B are then modeled as Poisson(exp(α_A – β_B + H)) for home team (H is home

advantage) and Poisson(exp(α_B – β_A)) for awaypypi.orgpypi.org. The Bayesian approach puts priors on these parameters and uses the data (matches played) to **update the distributions (posterior)** of team strengths. This yields a **partial pooling** effect: extreme values get "shrunk" toward the average unless strongly supported by data.
Why it's effective: Hierarchical Bayesian models can **simultaneously estimate team qualities and predict match outcomes**statmodeling.stat.columbia.edu. They borrow strength across teams and seasons – for example, a newly promoted team's attack rating will be informed by the overall prior (league average) until enough matches are observed. This prevents overfitting on small sample sizes and naturally quantifies uncertainty in predictions with credible intervals. Such models were shown to improve predictive accuracy in research. Baio and Blangiardo (2010) famously built a Bayesian model for Italian Serie A that could estimate the factors leading to wins/losses and predict match scoresstatmodeling.stat.columbia.edu. Similarly, R. Bååth (2015) demonstrated a Bayesian Poisson model on Spanish league data, producing not only match predictions but also **probabilistic team rankings** (with uncertainty bounds for each team's strength)lucs.lu.se.
Real-world use and Python implementation: While computationally heavier, these models are used in practice by advanced analysts and academics – for instance, to power club decision-making or betting syndicate models that continually update team ratings. In Python, one can implement hierarchical soccer models using libraries like **PyMC3/PyMC** or **Stan (via CmdStanPy/Pystan)**. There are examples in the community (e.g. blog posts and notebooks porting the Baio & Blangiardo model to PyMC3) that show how to use Markov Chain Monte Carlo to fit such modelsmedium.com. The open-source penaltyblog package also supports Bayesian formulations and even MCMC fitting for some modelspypi.org. These Bayesian models, validated in peer-reviewed studies, strike a good balance between **interpretability** (attack/defense ratings with uncertainty) and predictive performance, making them highly relevant for serious football outcome forecasting.

In-Play Event Forecasting Models

Expected Goals (xG) Models

Use case: Used for **in-play event prediction**, especially for evaluating chances of scoring from shots and forecasting total goals. (While xG is often a post-game analytic, it's fundamentally a predictive model for each shot's outcome, and summing xG gives an expectation of goals.)
How it works: An xG model typically uses a logistic regression (or another binary outcome model) to estimate the probability that a given shot results in a goal. Features like shot distance, angle to goal, type of assist, body part, etc., are inputs. The model outputs a probability between 0 and 1 (this is the "expected goal" value of the shot). For example, a shot from the center of the six-yard box might have xG ≈ 0.3 (30% chance of goal), whereas a long-range

attempt might be xG ≈ 0.02. These probabilities are learned from historical data of thousands of shots. **Why effective:** xG models quantify chance quality far better than raw shot counts – a team that had 10 shots worth 0.1 xG each (total 1.0 expected goals) likely posed more threat than a team with 10 speculative 0.01 xG shots (0.1 expected goals). Over many games, cumulative xG correlates strongly with goals scored, so it's predictive of future performance and scoring. In live forecasting, summing the xG of chances a team creates can update our expectation of how many goals they "should" score.

Real-world use: Expected goals have become **standard in clubs and broadcasting** to assess performance. Many betting models incorporate xG to adjust live odds (if a team is creating high-xG chances, their win probability in-play should increase even if they haven't scored yet). FiveThirtyEight's soccer SPI model, for instance, uses shot-based expected goals as a key componentfivethirtyeight.com. On the implementation side, building an xG model in Python can be done with libraries like scikit-learn or statsmodels (logistic regression). There are public datasets and notebooks demonstrating this. Additionally, some Python libraries (e.g., sklearn pipelines in **Metrica Sports** or open-source codes on GitHub) allow you to plug in shot data and train an xG model. In practice, companies like Opta/StatsPerform provide pre-trained xG models as part of their analytics offerings, and open-source projects exist for learning xG from event data. Overall, xG is a **practical, validated tool** for in-play goal forecasting and is often the foundation for more complex live win probability models.

Time-to-Event (Survival) Models for In-Play Forecasting

Use case: These approaches are used for **live match prediction** – e.g. forecasting the timing of the next goal, the evolving win/draw/lose probabilities as a match progresses, or the likelihood of events like a red card in the remainder of a game.

How it works: Survival analysis and point process models treat goals (or other events) as events in continuous time. A simple version is to assume goal arrivals follow a Poisson process with some rate (goals per minute). Under a constant rate assumption, the time until the next goal is exponentially distributed. More sophisticated models make the rate **state-dependent** – i.e. the scoring intensity λ_home and λ_away can change depending on the current score, time remaining, or other factors. For example, if a team is trailing late, their scoring rate might increase while the leading team's rate drops (parking the bus). Dixon & Robinson (1998) introduced a two-team **"birth process"** model: essentially each team's goals arrive as a Poisson process, and after each goal the intensities are adjusted for the new scoreline. In survival terms, one can also use a **Cox proportional hazards model** to include covariates (like which team scored first, or if a red card has occurred) when modeling time to next goal. These models naturally account for *censoring* (e.g. "no goal happened before full-time" is a censored outcome) and allow updating probabilities as time ticks away.

Why it's effective: Time-to-event models directly tackle the question "Given the match state now, what is the probability X will happen in the next interval (or by game end)?" This is crucial for in-play betting and coaching decisions. For instance, a survival model can tell us the probability that the trailing team will equalize before the match ends, based on how much time is left and historical goal rates in similar situations. Unlike static pre-match models, survival models are **dynamic** – as the clock runs and no goal occurs, the chances of a comeback naturally decline (the hazard of goal is integrated over less time). These methods have been validated in research (e.g. using survival analysis to show how the first goal's timing affects the second goal) and align with how odds swing during matches.

Real-world use: Modern win probability calculators often use a form of this. A practical implementation is to **estimate the instantaneous scoring probabilities** for each team and then simulate the rest of the match many times. For example, *American Soccer Analysis* described a model that at any game minute takes inputs like time remaining, score, team strengths, etc., and produces the probability each team scores in each of the next minutes – then runs Monte Carlo simulations to get win/draw/lose oddsamericansocceranalysis.com. In that approach, if Team A has (say) a 0.7% chance of scoring in the next minute and Team B a 3.8% chance (perhaps Team B is pushing hard), by simulating minute-by-minute you can compute that Team B might have, for example, a ~49% chance to overturn a 1–2 deficit in the remaining 40 minutesamericansocceranalysis.comamericansocceranalysis.com. Betting markets also implicitly use such models (the live odds of a draw or comeback are essentially derived from estimated scoring intensities given the state).

Python implementation: One can implement a basic hazard model by fitting an exponential or Weibull distribution to goal times, or use lifelines library for Cox models to include covariates. For the simulation approach, you can use a loop or vectorized simulation (or even a custom Markov chain) to roll out many random scenarios of goals given estimated per-minute scoring probabilities. While specialized libraries for sports survival models are rare, the logic can be coded with standard Python scientific stacks (numpy random draws for Poisson events, etc.). Some open-source notebooks (e.g., by soccer analytics enthusiasts) demonstrate live win-probability simulation using expected goals and time decay as inputs. These time-to-event models, whether analytical or simulation-based, are grounded in statistics (Poisson processes and survival analysis) and have been **validated by both peer-reviewed studies and real betting outcomes**, making them a cornerstone of in-play forecasting.

Other Statistical Approaches

Beyond the major categories above, a few other noteworthy modeling approaches are used in football analytics:

- **Ordered Outcome Models:** Instead of predicting exact scores, one can model the match result (win/draw/loss) as an **ordinal outcome**. Techniques like ordered logistic regression or probit models treat a draw as the “middle” outcome between a home win and away win. These can incorporate team strengths similarly to Elo or Poisson models. They have a solid statistical basis, though in practice they often perform similarly to ratings or Poisson-based methods when features are similar.

- **Rating Systems Extensions:** The basic Elo can be extended with factors for goals difference (to reward big wins) or adjusted for tournament importance. **TrueSkill**, a Bayesian rating system, has been used in some research to rate teams or even players, accounting for draw probabilities explicitly. These are statistically grounded (assuming underlying normal skill distributions) and have seen practical use in ranking teams and simulating tournament outcomes.
- **Markov Chain Models for Game Flow:** Some research models the progression of a match as a Markov chain with states (possession, attacking move, etc.) culminating in shots or goals. For example, one can define states for “team A in possession” vs “team B in possession” and absorbing states for goals. Transition probabilities (estimated from data) then yield insights into scoring likelihood. While more common in evaluating **tactical sequences** than directly predicting final scores, this approach is statistical (based on transition matrices) and has been explored in football analytics literature. It’s particularly useful for computing metrics like **expected possession value (EPV)** or **expected threat (xT)**, which forecast the chance of a goal given the ball’s current state. Python implementations often use custom code with numpy for matrix exponentiation or simulation to realize these Markov models.
- **Hawkes Processes (self-exciting events):** An emerging statistical approach is to model event sequences (like shots, passes, or fouls) with Hawkes point processes, where one event can temporarily raise the probability of follow-up events. In soccer, a flurry of shots might be modeled this way (a rebound shot is more likely right after an initial shot on goal, for instance). This is a research frontier rather than a widespread practice, but it is grounded in statistical point process theory. A few academic papers have applied Hawkes processes to soccer for things like modeling the momentum of attacks. While not yet common in production models, open-source libraries (like hawkesspatial in R, or custom Python code using tick library) have been used in experiments to forecast sequences of events in matches.

Each of these approaches – from ordered models to Markov and point processes – adds a twist to how we statistically view a football match. They are **unified by statistical principles** (probability distributions, Markov property, etc.) and several have been validated in studies or pilot implementations. For instance, a **hierarchical model for yellow cards** has been explored with Bayesian methods (treating referees as a factor)dm13450.github.io, and zero-inflated

Poisson models have been used to handle the excess of matches with 0 cards or 0 goalsresearchgate.net. While general machine learning (like SVMs or neural networks) can also tackle these problems, the approaches listed here have the advantage of interpretability and often require less data, making them highly relevant for both academic research and practical deployment in football outcome prediction and in-play forecasting.

Sources: The above guide is informed by a wide range of soccer analytics research and applications, including Dixon & Coles's seminal statistical model for scorespypi.org, Bayesian team-strength models from academic literaturestatmodeling.stat.columbia.edu, bookmaker methodologies for Poisson predictions, Elo rating adaptations to socceren.wikipedia.orgen.wikipedia.org, and modern implementations like the penaltyblog Python library which provides efficient tools for Poisson, Dixon-Coles, bivariate models and Elo ratingsgithub.comgithub.com. In-play forecasting insights draw from survival analysis studies and industry examples of live win probability simulationsamericansocceranalysis.com, as well as widely used metrics like expected goalsfivethirtyeight.com that underpin many real-time predictive systems. These approaches have all been validated either by peer-reviewed research or by their successful use in betting markets, professional clubs, and popular analytics platforms.

Alright, let's turn that architecture into a *real* Python app you could actually start coding.

I'll give you:

- A **project structure**
- **requirements**
- Concrete **Python files** with code (DB, ingestion from APIfootball, features, models, training, prediction API)

You'll be able to copy–paste into files and have a working skeleton you can extend with more models.

---

1. Project structure

You can use this as your base:

football_predictor/

  requirements.txt

```
.env                    # API keys, DB URL
app/
 __init__.py
 config.py

 db/
  __init__.py
  base.py
  session.py
  models.py

 ingestion/
  __init__.py
  apifootball_client.py
  sync_jobs.py

 features/
  __init__.py
  prematch_builder.py
  inplay_builder.py

 models_core/
  __init__.py
  base.py
  registry.py
  statistical/
```

```
    __init__.py
    elo.py
    poisson_dc.py
   ml/
    __init__.py
    logistic_outcome.py
    random_forest_outcome.py

  services/
   __init__.py
   training_service.py
   prediction_service.py

  api/
   __init__.py
   routes_matches.py
   routes_predictions.py

  schemas/
   __init__.py
   prediction_schemas.py

  main.py

 scripts/
  backfill_matches.py
```

train_prematch_outcome.py

We'll fill key files now.

---

2. requirements.txt

```
fastapi
uvicorn[standard]
python-dotenv

SQLAlchemy>=2.0
psycopg2-binary

pandas
numpy
scipy

scikit-learn
xgboost

requests
joblib
```

Install with:

```
pip install -r requirements.txt
```

---

3. Config & DB setup

3.1. .env (example)

```
APIFOOTBALL_API_KEY=YOUR_API_KEY_HERE
```

```
DATABASE_URL=postgresql+psycopg2://user:password@localhost:5432/football
DEFAULT_LEAGUE_ID=152
```

3.2. app/config.py

```python
import os
from dotenv import load_dotenv

load_dotenv()

APIFOOTBALL_API_KEY = os.getenv("APIFOOTBALL_API_KEY", "")
DATABASE_URL = os.getenv("DATABASE_URL")
DEFAULT_LEAGUE_ID = int(os.getenv("DEFAULT_LEAGUE_ID", "152"))  # example EPL
MODEL_STORE_DIR = os.getenv("MODEL_STORE_DIR", "model_store")
```

3.3. app/db/base.py

```python
from sqlalchemy.orm import declarative_base

Base = declarative_base()
```

3.4. app/db/session.py

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from app.config import DATABASE_URL

engine = create_engine(
    DATABASE_URL,
    pool_pre_ping=True,
)
```

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

3.5. app/db/models.py

```
from sqlalchemy import (
    Column, Integer, String, Date, Time, Float, JSON, ForeignKey
)
from sqlalchemy.orm import relationship
from app.db.base import Base


class Team(Base):
    __tablename__ = "teams"


    id = Column(Integer, primary_key=True)  # API team_id
    name = Column(String, nullable=False)
    country = Column(String)


    home_matches = relationship("Match", foreign_keys="Match.home_team_id",
back_populates="home_team")
    away_matches = relationship("Match", foreign_keys="Match.away_team_id",
back_populates="away_team")



class Match(Base):
    __tablename__ = "matches"


    id = Column(Integer, primary_key=True)  # API match_id
    league_id = Column(Integer, index=True)
    home_team_id = Column(Integer, ForeignKey("teams.id"))
```

```python
    away_team_id = Column(Integer, ForeignKey("teams.id"))
    date = Column(Date, index=True)
    time = Column(Time)
    status = Column(String)  # "Finished", "Not Started", "13'"
    home_ft_goals = Column(Integer)
    away_ft_goals = Column(Integer)

    raw_json = Column(JSON)

    home_team = relationship("Team", foreign_keys=[home_team_id],
back_populates="home_matches")
    away_team = relationship("Team", foreign_keys=[away_team_id],
back_populates="away_matches")
    stats = relationship("MatchStats", back_populates="match", uselist=False)


class MatchStats(Base):
    __tablename__ = "match_stats"

    id = Column(Integer, primary_key=True)
    match_id = Column(Integer, ForeignKey("matches.id"), index=True)
    period = Column(String)  # e.g. "full"

    home_shots_total = Column(Integer)
    away_shots_total = Column(Integer)
    home_corners = Column(Integer)
    away_corners = Column(Integer)
```

```python
    home_cards_yellow = Column(Integer)
    away_cards_yellow = Column(Integer)
    home_cards_red = Column(Integer)
    away_cards_red = Column(Integer)
    home_fouls = Column(Integer)
    away_fouls = Column(Integer)

    raw_stats = Column(JSON)

    match = relationship("Match", back_populates="stats")


class RatingHistory(Base):
    __tablename__ = "rating_history"

    id = Column(Integer, primary_key=True)
    team_id = Column(Integer, ForeignKey("teams.id"))
    date = Column(Date, index=True)
    rating_type = Column(String)  # "elo", etc.
    rating = Column(Float)
```

Run migrations quickly by:

```python
# small helper script
from app.db.base import Base
from app.db.session import engine
from app.db import models  # noqa
```

```
Base.metadata.create_all(bind=engine)
```

---

4. Ingestion from APIfootball

4.1. app/ingestion/apifootball_client.py

```
import requests
from typing import Any, Dict, List, Optional
from app.config import APIFOOTBALL_API_KEY


BASE_URL = "https://apiv3.apifootball.com/"


class APIFootballClient:
    def __init__(self, api_key: Optional[str] = None):
        self.api_key = api_key or APIFOOTBALL_API_KEY


    def _get(self, params: Dict[str, Any]) -> List[Dict[str, Any]]:
        params["APIkey"] = self.api_key
        resp = requests.get(BASE_URL, params=params, timeout=30)
        resp.raise_for_status()
        data = resp.json()
        if isinstance(data, dict) and data.get("error"):
            raise RuntimeError(f"API error: {data}")
        return data


    def get_events(
        self,
        date_from: str,
```

```
        date_to: str,
        league_id: Optional[int] = None,
        match_live: Optional[int] = None,
    ) -> List[Dict[str, Any]]:
        params = {
            "action": "get_events",
            "from": date_from,
            "to": date_to,
        }
        if league_id is not None:
            params["league_id"] = league_id
        if match_live is not None:
            params["match_live"] = match_live
        return self._get(params)
```

4.2. app/ingestion/sync_jobs.py

```
from datetime import date
from sqlalchemy.orm import Session
from app.ingestion.apifootball_client import APIFootballClient
from app.db.models import Team, Match, MatchStats


STAT_MAP = {
    "Shots Total": ("shots_total", int),
    "Corners": ("corners", int),
    "Yellow Cards": ("cards_yellow", int),
    "Red Cards": ("cards_red", int),
    "Fouls": ("fouls", int),
```

```
}

def get_or_create_team(session: Session, team_id: int, team_name: str, country: str | None):
    team = session.get(Team, team_id)
    if not team:
        team = Team(id=team_id, name=team_name, country=country)
        session.add(team)
        session.flush()
    return team

def sync_day(session: Session, target_date: date, league_id: int | None = None):
    client = APIFootballClient()
    events = client.get_events(
        date_from=target_date.isoformat(),
        date_to=target_date.isoformat(),
        league_id=league_id,
    )

    for ev in events:
        match_id = int(ev["match_id"])
        league_id = int(ev["league_id"])
        home_team_id = int(ev["match_hometeam_id"])
        away_team_id = int(ev["match_awayteam_id"])

        home_team = get_or_create_team(
            session, home_team_id, ev["match_hometeam_name"], ev.get("country_name")
```

```
)
away_team = get_or_create_team(
    session, away_team_id, ev["match_awayteam_name"], ev.get("country_name")
)

match = session.get(Match, match_id) or Match(id=match_id)
match.league_id = league_id
match.home_team_id = home_team.id
match.away_team_id = away_team.id
match.date = date.fromisoformat(ev["match_date"])
match.time = None  # parse ev["match_time"] if you want
match.status = ev["match_status"]
match.home_ft_goals = int(ev.get("match_hometeam_ft_score") or 0)
match.away_ft_goals = int(ev.get("match_awayteam_ft_score") or 0)
match.raw_json = ev

session.add(match)

# stats
stats_full = ev.get("statistics", [])
home_stats = {}
away_stats = {}
for s in stats_full:
    t = s["type"]
    if t not in STAT_MAP:
        continue
```

```
            fname, cast = STAT_MAP[t]
            home_stats[fname] = cast(str(s["home"]).replace("%", "") or 0)
            away_stats[fname] = cast(str(s["away"]).replace("%", "") or 0)

        ms = session.query(MatchStats).filter_by(match_id=match_id,
period="full").one_or_none()
        if not ms:
            ms = MatchStats(match_id=match_id, period="full")
            session.add(ms)

        ms.home_shots_total = home_stats.get("shots_total")
        ms.away_shots_total = away_stats.get("shots_total")
        ms.home_corners = home_stats.get("corners")
        ms.away_corners = away_stats.get("corners")
        ms.home_cards_yellow = home_stats.get("cards_yellow")
        ms.away_cards_yellow = away_stats.get("cards_yellow")
        ms.home_fouls = home_stats.get("fouls")
        ms.away_fouls = away_stats.get("fouls")
        ms.raw_stats = stats_full

    session.commit()
```

4.3. script scripts/backfill_matches.py

```
# scripts/backfill_matches.py
from datetime import date, timedelta
from app.db.session import SessionLocal
from app.ingestion.sync_jobs import sync_day
```

```python
from app.config import DEFAULT_LEAGUE_ID


if __name__ == "__main__":
    session = SessionLocal()
    # example: last 30 days
    today = date.today()
    for i in range(30):
        d = today - timedelta(days=i)
        print(f"Syncing {d}")
        sync_day(session, d, league_id=DEFAULT_LEAGUE_ID)
    session.close()
```

---

5. Feature builders

5.1. app/features/prematch_builder.py

```python
from dataclasses import dataclass
from typing import List, Optional
import numpy as np
from sqlalchemy.orm import Session
from app.db.models import Match, MatchStats, RatingHistory
from app.config import DEFAULT_LEAGUE_ID


@dataclass
class PreMatchExample:
    match_id: int
    X: np.ndarray
    y_outcome: Optional[int]
```

```python
    y_home_goals: Optional[int]
    y_away_goals: Optional[int]

class PreMatchFeatureBuilder:
    def __init__(self, session: Session, rating_type: str = "elo"):
        self.session = session
        self.rating_type = rating_type

    def _get_team_rating(self, team_id: int, match_date) -> float:
        rh = (
            self.session.query(RatingHistory)
            .filter_by(team_id=team_id, rating_type=self.rating_type)
            .filter(RatingHistory.date <= match_date)
            .order_by(RatingHistory.date.desc())
            .first()
        )
        return rh.rating if rh else 1500.0

    def build_examples(self, league_id: int = DEFAULT_LEAGUE_ID) -> List[PreMatchExample]:
        matches = (
            self.session.query(Match)
            .filter_by(league_id=league_id)
            .order_by(Match.date)
            .all()
        )
        examples: List[PreMatchExample] = []
```

```
for m in matches:
    if not m.stats:
        continue

    ms: MatchStats = m.stats

    home_rating = self._get_team_rating(m.home_team_id, m.date)
    away_rating = self._get_team_rating(m.away_team_id, m.date)
    rating_diff = home_rating - away_rating

    features = [
        home_rating,
        away_rating,
        rating_diff,
        ms.home_shots_total or 0,
        ms.away_shots_total or 0,
        ms.home_corners or 0,
        ms.away_corners or 0,
    ]
    X = np.array(features, dtype=float)

    if m.status == "Finished":
        home_goals = m.home_ft_goals or 0
        away_goals = m.away_ft_goals or 0
```

```
            if home_goals > away_goals:
                outcome = 0
            elif home_goals == away_goals:
                outcome = 1
            else:
                outcome = 2
        else:
            home_goals = away_goals = outcome = None

        examples.append(
            PreMatchExample(
                match_id=m.id,
                X=X,
                y_outcome=outcome,
                y_home_goals=home_goals,
                y_away_goals=away_goals,
            )
        )
    return examples
```

5.2. app/features/inplay_builder.py

```
from dataclasses import dataclass
from typing import Optional
import numpy as np


@dataclass
class InplaySnapshot:
```

```
    match_id: int
    minute: int
    X: np.ndarray


def parse_minute(status: str) -> int:
    if "'" in status:
        try:
            return int(status.replace("'", "").strip())
        except ValueError:
            return 0
    if status == "Half Time":
        return 45
    if status == "Finished":
        return 90
    return 0


class InplayFeatureBuilder:
    def build_from_event_json(self, ev: dict) -> Optional[InplaySnapshot]:
        status = ev["match_status"]
        if status in ("Not Started", "Postponed", "Cancelled"):
            return None


        minute = parse_minute(status)
        stats_full = ev.get("statistics", [])
        stat_dict = {s["type"]: (s["home"], s["away"]) for s in stats_full}
```

```
    def get_int(type_name, side="home"):
        if type_name not in stat_dict:
            return 0
        val = stat_dict[type_name][0 if side == "home" else 1]
        return int(str(val).replace("%", "") or 0)

    home_goals = int(ev.get("match_hometeam_score") or 0)
    away_goals = int(ev.get("match_awayteam_score") or 0)

    features = [
        minute,
        home_goals,
        away_goals,
        get_int("Shots Total", "home"),
        get_int("Shots Total", "away"),
        get_int("Corners", "home"),
        get_int("Corners", "away"),
        get_int("Yellow Cards", "home"),
        get_int("Yellow Cards", "away"),
    ]
    X = np.array(features, dtype=float)
    return InplaySnapshot(match_id=int(ev["match_id"]), minute=minute, X=X)
```

---

6. Model core: base + registry

6.1. app/models_core/base.py

```
from abc import ABC, abstractmethod
```

```python
from typing import Any
import numpy as np
import joblib


class BaseModel(ABC):
    name: str
    task: str    # "outcome", "goals", etc.
    mode: str    # "prematch", "inplay", "both"

    @abstractmethod
    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        ...

    @abstractmethod
    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        ...

    def save(self, path: str) -> None:
        joblib.dump(self, path)

    @classmethod
    def load(cls, path: str) -> "BaseModel":
        return joblib.load(path)
```

6.2. app/models_core/registry.py

```python
from typing import Dict, Type
from .base import BaseModel
```

```
MODEL_REGISTRY: Dict[str, Type[BaseModel]] = {}


def register_model(cls: Type[BaseModel]):
    MODEL_REGISTRY[cls.__name__] = cls
    return cls


def get_model_class(name: str) -> Type[BaseModel]:
    return MODEL_REGISTRY[name]
```

---

7. Example model implementations

7.1. Logistic outcome (pre-match)

app/models_core/ml/logistic_outcome.py

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from app.models_core.base import BaseModel
from app.models_core.registry import register_model


@register_model
class LogisticOutcome(BaseModel):
    def __init__(self) -> None:
        self.name = "LogisticOutcome"
        self.task = "outcome"
        self.mode = "prematch"
        self.model = LogisticRegression(
            max_iter=500,
```

```
        multi_class="multinomial"
    )

  def fit(self, X: np.ndarray, y: np.ndarray) -> None:
    self.model.fit(X, y)

  def predict_proba(self, X: np.ndarray) -> np.ndarray:
    return self.model.predict_proba(X)
```

7.2. RandomForest outcome

app/models_core/ml/random_forest_outcome.py

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from app.models_core.base import BaseModel
from app.models_core.registry import register_model

@register_model
class RandomForestOutcome(BaseModel):
  def __init__(self) -> None:
    self.name = "RandomForestOutcome"
    self.task = "outcome"
    self.mode = "prematch"
    self.model = RandomForestClassifier(
      n_estimators=300,
      min_samples_leaf=2,
      random_state=42,
      n_jobs=-1,
```

```python
    )

  def fit(self, X: np.ndarray, y: np.ndarray) -> None:
    self.model.fit(X, y)

  def predict_proba(self, X: np.ndarray) -> np.ndarray:
    return self.model.predict_proba(X)
```

7.3. Simple Elo rating update

app/models_core/statistical/elo.py

```python
from sqlalchemy.orm import Session
from datetime import date
from app.db.models import Match, RatingHistory

DEFAULT_K = 20.0

def expected_score(rating_a: float, rating_b: float) -> float:
  return 1.0 / (1.0 + 10.0 ** ((rating_b - rating_a) / 400.0))

def get_latest_rating(session: Session, team_id: int, rating_type: str, up_to: date) -> float:
  rh = (
    session.query(RatingHistory)
    .filter_by(team_id=team_id, rating_type=rating_type)
    .filter(RatingHistory.date <= up_to)
    .order_by(RatingHistory.date.desc())
    .first()
  )
```

```python
    return rh.rating if rh else 1500.0


def update_elo_for_league(session: Session, league_id: int, rating_type: str = "elo", k: float =
DEFAULT_K):
    matches = (
        session.query(Match)
        .filter_by(league_id=league_id)
        .filter(Match.status == "Finished")
        .order_by(Match.date)
        .all()
    )

    for m in matches:
        home_rating = get_latest_rating(session, m.home_team_id, rating_type, m.date)
        away_rating = get_latest_rating(session, m.away_team_id, rating_type, m.date)

        exp_home = expected_score(home_rating, away_rating)
        exp_away = 1.0 - exp_home

        if (m.home_ft_goals or 0) > (m.away_ft_goals or 0):
            act_home, act_away = 1.0, 0.0
        elif (m.home_ft_goals or 0) == (m.away_ft_goals or 0):
            act_home, act_away = 0.5, 0.5
        else:
            act_home, act_away = 0.0, 1.0
```

```
        new_home = home_rating + k * (act_home - exp_home)
        new_away = away_rating + k * (act_away - exp_away)


        session.add(RatingHistory(team_id=m.home_team_id, date=m.date,
rating_type=rating_type, rating=new_home))
        session.add(RatingHistory(team_id=m.away_team_id, date=m.date,
rating_type=rating_type, rating=new_away))


    session.commit()
```

(Dixon–Coles Poisson is more code; you can plug in the DC likelihood from earlier once you're ready. The architecture already supports it.)

---

8. Training service & script

8.1. app/services/training_service.py

```
import os
import numpy as np
from typing import List
from sqlalchemy.orm import Session
from app.features.prematch_builder import PreMatchFeatureBuilder
from app.models_core.registry import MODEL_REGISTRY
from app.config import MODEL_STORE_DIR


class TrainingService:
    def __init__(self, session: Session):
        self.session = session


    def train_prematch_outcome_models(self, league_id: int, model_names: List[str]) -> None:
```

```
        builder = PreMatchFeatureBuilder(self.session)
        examples = builder.build_examples(league_id)

        train_examples = [e for e in examples if e.y_outcome is not None]
        if not train_examples:
            raise RuntimeError("No finished matches to train on.")

        X = np.vstack([e.X for e in train_examples])
        y = np.array([e.y_outcome for e in train_examples], dtype=int)

        os.makedirs(MODEL_STORE_DIR, exist_ok=True)

        for name in model_names:
            cls = MODEL_REGISTRY[name]
            model = cls()
            model.fit(X, y)
            path = os.path.join(MODEL_STORE_DIR, f"{name}_league{league_id}.joblib")
            model.save(path)
            print(f"Saved {name} model to {path}")
```

8.2. scripts/train_prematch_outcome.py

```
# scripts/train_prematch_outcome.py
import argparse
from app.db.session import SessionLocal
from app.services.training_service import TrainingService
from app.models_core import ml  # noqa: F401  # ensure registration
from app.config import DEFAULT_LEAGUE_ID
```

```python
# this import path ensures the @register_model decorators run
import app.models_core.ml.logistic_outcome  # noqa: F401
import app.models_core.ml.random_forest_outcome  # noqa: F401


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--league", type=int, default=DEFAULT_LEAGUE_ID)
    args = parser.parse_args()

    session = SessionLocal()
    svc = TrainingService(session)
    svc.train_prematch_outcome_models(
        league_id=args.league,
        model_names=["LogisticOutcome", "RandomForestOutcome"],
    )
    session.close()


if __name__ == "__main__":
    main()
```

---

9. Prediction service & API

9.1. app/schemas/prediction_schemas.py

```python
from typing import Dict, List, Optional
from pydantic import BaseModel
```

```
class PrematchRequest(BaseModel):
    match_id: int
    league_id: Optional[int] = None
    models: List[str]


class InplayRequest(BaseModel):
    match_id: int
    league_id: Optional[int] = None
    models: List[str]


class PredictionResponse(BaseModel):
    match_id: int
    models: Dict[str, Dict[str, float]]
```

9.2. app/services/prediction_service.py

```
import os
import numpy as np
from typing import Dict, Any, List
from sqlalchemy.orm import Session
from app.db.session import SessionLocal
from app.db.models import Match
from app.features.prematch_builder import PreMatchFeatureBuilder
from app.features.inplay_builder import InplayFeatureBuilder
from app.models_core.registry import get_model_class
from app.ingestion.apifootball_client import APIFootballClient
from app.config import MODEL_STORE_DIR, DEFAULT_LEAGUE_ID
```

```
class PredictionService:
    def __init__(self, league_id: int | None = None):
        self.league_id = league_id or DEFAULT_LEAGUE_ID

    def _load_model(self, model_name: str):
        cls = get_model_class(model_name)
        path = os.path.join(MODEL_STORE_DIR, f"{model_name}_league{self.league_id}.joblib")
        return cls.load(path)

    def prematch_outcome(self, match_id: int, model_names: List[str]) -> Dict[str, Any]:
        session: Session = SessionLocal()
        try:
            builder = PreMatchFeatureBuilder(session)
            examples = builder.build_examples(self.league_id)
            ex = next(e for e in examples if e.match_id == match_id)
            X = ex.X.reshape(1, -1)

            result: Dict[str, Any] = {}
            for name in model_names:
                model = self._load_model(name)
                probs = model.predict_proba(X)[0]
                result[name] = {
                    "home_win": float(probs[0]),
                    "draw": float(probs[1]),
                    "away_win": float(probs[2]),
                }
```

```python
            return result
        finally:
            session.close()

    def inplay_outcome(self, match_id: int, model_names: List[str]) -> Dict[str, Any]:
        client = APIFootballClient()
        # NOTE: you'd normally use 'livescore' or match_live endpoint; this is simplified
        events = client.get_events(date_from="", date_to="", league_id=self.league_id,
match_live=1)

        ev = next(e for e in events if int(e["match_id"]) == match_id)
        builder = InplayFeatureBuilder()
        snap = builder.build_from_event_json(ev)
        if not snap:
            raise RuntimeError("Match not in-play")

        X = snap.X.reshape(1, -1)
        result: Dict[str, Any] = {"minute": snap.minute}
        for name in model_names:
            model = self._load_model(name)
            probs = model.predict_proba(X)[0]
            result[name] = {
                "home_win": float(probs[0]),
                "draw": float(probs[1]),
                "away_win": float(probs[2]),
            }
```

```python
        return result
```

(For real in-play, adapt the API call to use action=livescore or the exact APIfootball live endpoint; I'm keeping it framed around get_events to show the structure.)

9.3. app/api/routes_predictions.py

```python
from fastapi import APIRouter, HTTPException

from app.schemas.prediction_schemas import PrematchRequest, InplayRequest, PredictionResponse

from app.services.prediction_service import PredictionService


router = APIRouter(prefix="/predictions", tags=["predictions"])


@router.post("/prematch", response_model=PredictionResponse)

def prematch(req: PrematchRequest):

    svc = PredictionService(req.league_id)

    try:

        models = svc.prematch_outcome(req.match_id, req.models)

    except StopIteration:

        raise HTTPException(status_code=404, detail="Match not found")

    return PredictionResponse(match_id=req.match_id, models=models)


@router.post("/inplay", response_model=PredictionResponse)

def inplay(req: InplayRequest):

    svc = PredictionService(req.league_id)

    models = svc.inplay_outcome(req.match_id, req.models)

    return PredictionResponse(match_id=req.match_id, models=models)
```

9.4. Minimal matches route (optional)

app/api/routes_matches.py

```
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from app.db.session import SessionLocal
from app.db.models import Match
from app.config import DEFAULT_LEAGUE_ID

router = APIRouter(prefix="/matches", tags=["matches"])


def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()


@router.get("/")
def list_matches(league_id: int = DEFAULT_LEAGUE_ID, db: Session = Depends(get_db)):
    matches =
db.query(Match).filter_by(league_id=league_id).order_by(Match.date.desc()).limit(50).all()
    return [
        {
            "match_id": m.id,
            "league_id": m.league_id,
            "date": m.date.isoformat() if m.date else None,
            "home_team_id": m.home_team_id,
            "away_team_id": m.away_team_id,
```

```
        "home_ft_goals": m.home_ft_goals,
        "away_ft_goals": m.away_ft_goals,
        "status": m.status,
    }
    for m in matches
]
```

9.5. FastAPI main

app/main.py

```
from fastapi import FastAPI
from app.api import routes_matches, routes_predictions


app = FastAPI(title="Football Prediction Platform")


app.include_router(routes_matches.router)
app.include_router(routes_predictions.router)
```

Run server:

```
uvicorn app.main:app --reload
```

---

10. How to actually use this

1. **Create DB & tables**
    - Set DATABASE_URL in .env.
    - Run a small script to call Base.metadata.create_all(engine).
2. **Backfill data**
3. python -m scripts.backfill_matches
4. **Compute Elo ratings**

Write a tiny script that calls update_elo_for_league from elo.py.

5. **Train models**
6. python -m scripts.train_prematch_outcome --league 152
7. **Run API**
8. uvicorn app.main:app --reload
9. **Call endpoints**
    - GET /matches → list recent matches.
    - POST /predictions/prematch with JSON:
    - {
    - "match_id": 123456,
    - "league_id": 152,
    - "models": ["LogisticOutcome", "RandomForestOutcome"]
    - }

→ You get model-wise probabilities for home/draw/away.

This is a **concrete implementation skeleton** of the architecture you designed:

- APIfootball v3 VIP as data source.
- PostgreSQL DB with Teams / Matches / MatchStats / RatingHistory.
- Feature builders for pre-match & in-play.
- Plugin-style model layer (statistical + ML).
- Training + prediction services.
- FastAPI endpoints for pre-match & in-play predictions.

From here you can:

- Add more models (XGBoost, SVM, Poisson/DC, xG models, etc.) by just creating new classes and registering them.
- Extend features (more stats, player info).
- Build a frontend that calls these endpoints and compares models visually.

You said:

refactor the entire implementation for a python app for all of these # Model Category Mainly used for Typical football use 1 Poisson goal model (independent Poisson) Statistical Pre-match outcomes & exact score Model home/away goals as Poisson to get probabilities for each scoreline and win/draw/loss. 2 Dixon–Coles Poisson Statistical Pre-match outcomes & exact score Poisson with correlation tweak for low scores; better calibration for 0–0, 1–0, 0–1 etc. 3 Bivariate Poisson Statistical Exact score Joint model for home & away goals with shared component (correlation between teams' goals). 4 Skellam (goal-difference) model Statistical Win/draw/loss (handicap bets) Model goal difference as Skellam (difference of two Poissons) to price match result & handicaps. 5 Negative Binomial count model Statistical Totals (goals/cards/corners) Like Poisson but handles over-dispersion (variance > mean) in goals or cards. 6 Zero-inflated Poisson / NB Statistical Exact score & totals Adds extra probability mass at zero to handle many 0-goal or 0-card outcomes. 7 Bayesian hierarchical Poisson (team attack/defense) Statistical Pre-match outcomes & exact score Team strengths as random effects; great for updating attack/defense ratings over a season. 8 Ordered logistic / probit (Win–Draw–Loss) Statistical Match outcome only Treat W/D/L as ordered categories; predicts outcome directly without modeling goals. 9 Generalized Additive Model (GAM) Statistical Outcomes, totals & xG-type metrics Flexible non-linear regression for things like win probability or expected goals with interpretable smooth effects. 10 Survival / time-to-event model Statistical In-play (next goal, time of goal) Model hazard of next goal given current minute/score, used for live win/draw/loss probabilities. 11 Copula-based joint score model Statistical Exact score Build marginals for each team's goals (e.g. Weibull/Poisson) and couple them with a copula to get a realistic joint score distribution. 12 Markov chain (possession / EPV / xT) Statistical / Prediction In-play events & long-run scoring States = ball locations/possessions; transition matrix gives expected chance of eventually scoring from current state. 13 Hawkes process (self-exciting events) Statistical In-play event sequences Point-process model where one event (e.g. shot) temporarily increases chance of follow-up events (rebounds, more shots). 14 Expected Goals (xG) logistic model Statistical / Prediction In-play chance quality & total goals Logistic regression per shot; sum xG for team totals or to feed live win-probability models. 15 Elo rating system Prediction system Pre-match outcome Dynamic team rating updated after every game; rating difference → win/draw/loss probabilities. 16 Glicko / TrueSkill ratings Prediction system Pre-match outcome Elo-style but Bayesian with rating uncertainty; useful when match frequency or opposition strength varies. 17 Bradley–Terry / Davidson model Prediction system Pre-match outcome Pairwise comparison model; each team has a strength parameter, extended to handle draws (Davidson). 18 Monte-Carlo simulation on top of a score model Prediction system Both (pre-match & in-play) Simulate many future match paths using Poisson / survival intensities to estimate full distributions of results. 19 Linear Regression ML Regression Predict numeric targets (e.g. total goals, xG per team, number of cards). 20 Logistic Regression ML Classification Predict binary/multiclass outcomes (win/draw/loss, over/under line, team to score next). 21

Decision Tree ML Classification / Regression Simple tree rules for outcome or totals; interpretable but easy to overfit. 22 Random Forest ML Classification / Regression Ensemble of trees; strong baseline for outcome and totals using many features. 23 K-Nearest Neighbors (KNN) ML Classification / Regression Outcome or totals based on similar historic matches (feature-based similarity). 24 Naive Bayes ML Classification Simple probabilistic classifier (e.g. W/D/L) assuming feature independence. 25 Support Vector Machine (SVM) ML Classification / Regression (SVR) Margin-based model for match outcome or regression targets with engineered features. 26 AdaBoost ML Classification Boosted ensemble of weak learners (often trees) for outcome prediction. 27 XGBoost (Gradient Boosting Trees) ML Classification / Regression Very strong general-purpose tabular model; popular for match outcome / goals / totals. 28 K-Means Clustering ML Unsupervised Cluster teams, playing styles, or match types (not direct prediction, but useful preprocessing). 29 Hierarchical (Agglomerative) Clustering ML Unsupervised Group teams/players by similarity to explore segments (e.g. similar attack styles). 30 DBSCAN ML Unsupervised Detect dense clusters / outlier matches (e.g. weird scorelines or extreme stats). 31 PCA (Principal Component Analysis) ML Dimensionality reduction Compress many match/player stats into a few components before modeling. 32 Multi-Layer Perceptron (MLP) ML Classification / Regression Generic feed-forward NN for outcome, score, or totals with many nonlinear interactions. 33 Convolutional Neural Network (CNN) ML Mostly classification If you encode data as "images" (e.g. spatial maps or sequence heatmaps), CNNs can learn spatial patterns. 34 Long Short-Term Memory (LSTM) ML Sequence / time-series prediction Use sequences of past matches or in-play event streams to predict future outcome or next events. 35 Gaussian Mixture Model (GMM) ML Unsupervised (density / clustering) Model mixture of match types or team styles; can be used to weight predictions by cluster. 36 Transformer-based time-series model ML Sequence / time-series prediction Modern attention-based model for long match histories or rich temporal features (forms, player stats over time).

ChatGPT said:

Thought for 1m 52s

I'll refactor the whole thing so your Python app can **host all 36 models** cleanly, without turning into spaghetti.

Instead of dumping 36 giant classes, I'll give you a **modular framework** where:

- Every model plugs into a small set of **base interfaces**.
- You can train / save / load them with a **single orchestrator**.
- API endpoints don't care which algorithm is behind them.

I'll show concrete code for at least **one representative model in each family**, and stubs for the rest (so you know exactly where to implement them).

---

1. New project structure (refactored for all 36 models)

```
football_predictor/
 requirements.txt
 .env

 app/
  __init__.py
  config.py

  db/
   __init__.py
   base.py
   session.py
   models.py     # Teams, Matches, Stats, Ratings, etc.

  ingestion/
   __init__.py
   apifootball_client.py
   sync_jobs.py   # pulls data from APIfootball v3

  features/
   __init__.py
   prematch.py    # team/match aggregated features
```

```
 shot_level.py    # per-shot features for xG
 inplay.py        # live minute/score/stats features
 sequences.py     # sequences of matches/events for LSTM/Transformer

core/
 __init__.py
 base.py          # base interfaces for all model types
 registry.py      # central registry & metadata
 tasks.py         # enums describing tasks & capabilities

models/
 __init__.py

 statistical/
  __init__.py
  poisson_goal.py         # #1
  dixon_coles.py          # #2
  bivariate_poisson.py    # #3
  skellam_diff.py         # #4
  negbin_totals.py        # #5
  zero_inflated.py        # #6
  bayes_hier_poisson.py   # #7
  ordered_outcome.py      # #8
  gam_models.py           # #9
  survival_goal.py        # #10
  copula_score.py         # #11
```

```
 markov_epv_xt.py      # #12
 hawkes_events.py      # #13
 xg_logistic.py        # #14

ratings/
 __init__.py
 elo.py                # #15
 glicko_trueskill.py   # #16
 bradley_terry.py      # #17

simulation/
 __init__.py
 score_monte_carlo.py  # #18

ml/
 __init__.py
 classical/
  linear_reg.py        # #19
  logistic_cls.py      # #20
  decision_tree.py     # #21
  random_forest.py     # #22
  knn.py               # #23
  naive_bayes.py       # #24
  svm.py               # #25
  adaboost.py          # #26
  xgboost_models.py    # #27
```

```
    gam_sklearn_wrapper.py # (alt GAM impl if needed)

   unsupervised/
    kmeans_clusters.py     # #28
    hierarchical_clusters.py# #29
    dbscan_clusters.py     # #30
    gmm_clusters.py        # #35

   dim_reduction/
    pca_reducer.py         # #31

   deep/
    mlp.py                 # #32
    cnn.py                 # #33
    lstm_seq.py            # #34
    transformer_seq.py     # #36

 services/
  __init__.py
  training.py        # orchestrates training for any model
  prediction.py      # orchestrates predictions (prematch & inplay)
  ratings_update.py  # updates Elo/Glicko/BT over time

 api/
  __init__.py
  routes_matches.py
```

routes_predictions.py

routes_analysis.py # clustering, PCA, EPV/xT, etc.

schemas/

__init__.py

prediction.py   # request/response models

main.py

scripts/

backfill_matches.py

update_ratings.py

train_all_models.py

Data ingestion + DB models from previous answer stay almost the same (APIfootball v3 → PostgreSQL). What changes is the **core model layer** and how training & prediction are wired.

---

2. Core model abstractions for all 36 models

We'll categorise models by what they *do*, not just by algorithm:

- **OutcomeModel** – predicts W/D/L or similar (classification).
- **ScorelineModel** – predicts joint distribution P(HomeGoals=g, AwayGoals=h).
- **TotalsModel** – distribution of totals (goals, cards, corners).
- **SurvivalModel** – hazard for next goal/time-to-event.
- **RatingModel** – maintains team ratings over time (Elo, Glicko, BT).
- **SimulationModel** – Monte-Carlo on top of other models.
- **PreprocessorModel** – PCA, clustering, GMM, etc. (feature transformers / analyzers).
- **SequenceModel** – LSTM / Transformer / CNN time-series models.

2.1. app/core/tasks.py

```
from enum import Enum, auto


class TaskType(Enum):
    OUTCOME = auto()        # W/D/L, team to score next, etc.
    SCORELINE = auto()      # full score distribution
    TOTALS = auto()         # goals/cards/corners totals
    SURVIVAL = auto()       # hazard/time-to-goal
    RATING = auto()         # team strength ratings
    SIMULATION = auto()     # Monte Carlo on top of another model
    PREPROCESSOR = auto()   # PCA, clustering, density
    SEQUENCE = auto()       # LSTM/Transformer forecasting



class Mode(Enum):
    PREMATCH = auto()
    INPLAY = auto()
    BOTH = auto()
```

2.2. app/core/base.py

```
from abc import ABC, abstractmethod
from typing import Any, Dict, Optional
import numpy as np
import joblib
from .tasks import TaskType, Mode


class BaseComponent(ABC):
```

```python
    """Root type for everything: models, preprocessors, rating systems."""
    name: str
    task_type: TaskType
    mode: Mode

    def save(self, path: str) -> None:
        joblib.dump(self, path)

    @classmethod
    def load(cls, path: str) -> "BaseComponent":
        return joblib.load(path)


class SupervisedModel(BaseComponent):
    """
    Generic supervised model:
    - For OUTCOME: returns probabilities [home, draw, away]
    - For TOTALS: probabilities over bins (e.g. 0,1,2,...)
    - For REGRESSION-like outputs, we use predict() instead.
    """
    @abstractmethod
    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        ...

    @abstractmethod
    def predict_proba(self, X: np.ndarray) -> np.ndarray:
```

```python
        ...


class ScorelineModel(BaseComponent):
    """
    Model that outputs a full scoreline PMF matrix for each sample.
    P[home_goals, away_goals].
    """
    max_goals: int = 8

    @abstractmethod
    def fit(self, matches: Any) -> None:
        """Matches = iterable of objects with (home_team, away_team, goals, date,...)."""
        ...

    @abstractmethod
    def predict_score_matrix(self, match_state: Any) -> np.ndarray:
        """
        match_state: object containing teams, league, maybe current score/state.
        Returns: (max_goals+1, max_goals+1) matrix that sums to 1.
        """
        ...

    def outcome_probs_from_matrix(self, matrix: np.ndarray) -> Dict[str, float]:
        home_win = matrix[np.tril_indices_from(matrix, -1)].sum()
        draw = matrix.diagonal().sum()
```

```
        away_win = matrix[np.triu_indices_from(matrix, 1)].sum()
        return {"home_win": float(home_win),
                "draw": float(draw),
                "away_win": float(away_win)}


class TotalsModel(BaseComponent):
    """
    Models distribution of totals (goals, cards, corners) for a team or match.
    """
    @abstractmethod
    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        ...

    @abstractmethod
    def pmf(self, X: np.ndarray, max_total: int = 10) -> np.ndarray:
        """Return P(T = 0..max_total) for each row in X."""
        ...


class SurvivalModel(BaseComponent):
    """
    Hazard model for time-to-next-goal or similar.
    """
    @abstractmethod
    def fit(self, X: np.ndarray, durations: np.ndarray, events: np.ndarray) -> None:
```

```python
        ...

    @abstractmethod
    def hazard(self, X: np.ndarray, t: float) -> np.ndarray:
        """Hazard at time t (minutes remaining or elapsed)."""
        ...


class RatingModel(BaseComponent):
    """
    Systems like Elo, Glicko, TrueSkill, Bradley-Terry.
    """
    @abstractmethod
    def update_from_matches(self, matches: Any) -> None:
        ...

    @abstractmethod
    def get_rating(self, team_id: int, when: Optional[Any] = None) -> float:
        ...


class SimulationModel(BaseComponent):
    """
    Monte Carlo simulator that uses some base model(s).
    """
    @abstractmethod
```

```python
    def simulate(self, match_state: Any, n_sims: int = 10000) -> Dict[str, Any]:
        ...


class PreprocessorModel(BaseComponent):
    """
    PCA, KMeans, GMM, etc.
    """
    @abstractmethod
    def fit_transform(self, X: np.ndarray) -> np.ndarray:
        ...

    @abstractmethod
    def transform(self, X: np.ndarray) -> np.ndarray:
        ...


class SequenceModel(BaseComponent):
    """
    LSTM/Transformer-style models on sequences of feature vectors.
    """
    @abstractmethod
    def fit_sequences(self, sequences: Any, targets: np.ndarray) -> None:
        ...

    @abstractmethod
```

```python
    def predict_sequence(self, sequence: Any) -> Any:
        ...
```

2.3. app/core/registry.py

```python
from typing import Dict, Type
from .base import BaseComponent

COMPONENT_REGISTRY: Dict[str, Type[BaseComponent]] = {}

def register_component(cls: Type[BaseComponent]):
    COMPONENT_REGISTRY[cls.__name__] = cls
    return cls

def get_component_class(name: str) -> Type[BaseComponent]:
    return COMPONENT_REGISTRY[name]

def list_components() -> Dict[str, Type[BaseComponent]]:
    return COMPONENT_REGISTRY.copy()
```

Now, each of your 36 models becomes a **concrete class** inheriting from one of these, registered with @register_component.

---

3. Mapping the 36 models to base types

Here's how each row in your table maps:

#	Model	Base type
1	Poisson goal	ScorelineModel
2	Dixon–Coles Poisson	ScorelineModel

#	Model	Base type
3	Bivariate Poisson	ScorelineModel
4	Skellam	SupervisedModel (OUTCOME on goal difference)
5	Negative Binomial	TotalsModel
6	Zero-inflated Poisson/NB	ScorelineModel + TotalsModel
7	Bayesian hier. Poisson	ScorelineModel
8	Ordered logit/probit	SupervisedModel (OUTCOME)
9	GAM	SupervisedModel (OUTCOME or REGRESSION)
10	Survival/time-to-event	SurvivalModel
11	Copula joint score	ScorelineModel
12	Markov chain (EPV/xT)	SimulationModel or bespoke xT component; used as feature provider
13	Hawkes process	SurvivalModel / event-intensity component
14	xG logistic	SupervisedModel on shot-level data
15	Elo	RatingModel
16	Glicko/TrueSkill	RatingModel
17	Bradley–Terry/Davidson	RatingModel
18	Monte-Carlo on score model	SimulationModel
19	Linear Regression	SupervisedModel (REGRESSION)
20	Logistic Regression	SupervisedModel (OUTCOME)
21	Decision Tree	SupervisedModel
22	Random Forest	SupervisedModel

#	Model	Base type
23	KNN	SupervisedModel
24	Naive Bayes	SupervisedModel
25	SVM	SupervisedModel
26	AdaBoost	SupervisedModel
27	XGBoost	SupervisedModel
28	K-Means	PreprocessorModel (clustering)
29	Hierarchical clustering	PreprocessorModel
30	DBSCAN	PreprocessorModel
31	PCA	PreprocessorModel (dim reduction)
32	MLP	SupervisedModel
33	CNN	SupervisedModel / SequenceModel (if using spatial/temporal encoding)
34	LSTM	SequenceModel
35	GMM	PreprocessorModel (density/clustering)
36	Transformer TS model	SequenceModel

---

4. Representative implementations for each family

I'll show code for a few key ones; the others follow exactly the same pattern.

4.1. #1 – Poisson goal model (independent Poisson)

app/models/statistical/poisson_goal.py

import numpy as np

from dataclasses import dataclass

from typing import Any

```python
from app.core.base import ScorelineModel

from app.core.registry import register_component

from app.core.tasks import TaskType, Mode


@dataclass

class MatchState:

    home_attack: float

    away_attack: float

    home_defense: float

    away_defense: float

    home_advantage: float = 0.0

    # You can add league, date, etc.


@register_component

class PoissonGoalModel(ScorelineModel):

    def __init__(self, max_goals: int = 8) -> None:

        self.name = "PoissonGoalModel"

        self.task_type = TaskType.SCORELINE

        self.mode = Mode.PREMATCH

        self.max_goals = max_goals

        # In a full implementation you'd learn attack/defence params per team.

        # Here we assume those are precomputed and passed via MatchState.


    def fit(self, matches: Any) -> None:

        """

        Implement maximum-likelihood fitting for team attack/defence strengths
```

```python
        using match history (Poisson regression or log-linear model).
        For brevity: left as TODO.
        """
        raise NotImplementedError("Fit Poisson attack/defence per team here.")

    def _poisson_pmf(self, k: int, lam: float) -> float:
        from math import exp, factorial
        return exp(-lam) * lam**k / factorial(k)

    def predict_score_matrix(self, match_state: MatchState) -> np.ndarray:
        lam_home = np.exp(match_state.home_attack - match_state.away_defense +
                          match_state.home_advantage)
        lam_away = np.exp(match_state.away_attack - match_state.home_defense)

        g = self.max_goals
        pm = np.zeros((g+1, g+1))
        for h in range(g+1):
            for a in range(g+1):
                pm[h, a] = self._poisson_pmf(h, lam_home) * self._poisson_pmf(a, lam_away)
        pm /= pm.sum()
        return pm
```

You'll have similar classes for:

- DixonColesPoissonModel – same base, but apply DC correlation factor.
- BivariatePoissonModel – same base, with bivariate Poisson PMF.
- ZeroInflatedScoreModel – uses zero-inflation factor.

4.2. #4 – Skellam (goal difference) model as outcome

app/models/statistical/skellam_diff.py

```python
import numpy as np
from scipy.stats import skellam
from app.core.base import SupervisedModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@register_component
class SkellamOutcomeModel(SupervisedModel):
    """
    Uses lambda_home and lambda_away (from Poisson) and models goal difference.
    """
    def __init__(self) -> None:
        self.name = "SkellamOutcomeModel"
        self.task_type = TaskType.OUTCOME
        self.mode = Mode.PREMATCH

    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        """
        Typically you don't 'fit' Skellam directly; you fit lambdas via Poisson model.
        Here we treat X as [lambda_home, lambda_away] so there is no training.
        """
        # No training required if lambdas come from another model.
        pass

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
```

```
        """
        X[:,0] = lambda_home, X[:,1] = lambda_away
        Outputs [P(home_win), P(draw), P(away_win)].
        """
        probs = []
        for lam_home, lam_away in X:
            # Skellam distribution for D = home_goals - away_goals
            # We'll approximate by summing D from -10..10
            d_vals = np.arange(-10, 11)
            pmf = skellam.pmf(d_vals, lam_home, lam_away)
            p_home = pmf[d_vals > 0].sum()
            p_draw = pmf[d_vals == 0].sum()
            p_away = pmf[d_vals < 0].sum()
            probs.append([p_home, p_draw, p_away])
        return np.array(probs)
```

4.3. #5 – Negative Binomial totals model

app/models/statistical/negbin_totals.py

```
import numpy as np
from statsmodels.discrete.discrete_model import NegativeBinomial
from app.core.base import TotalsModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@register_component
class NegBinTotalsModel(TotalsModel):
    """
```

```
Models total goals / cards / corners with Negative Binomial regression.
"""
def __init__(self) -> None:
   self.name = "NegBinTotalsModel"
   self.task_type = TaskType.TOTALS
   self.mode = Mode.PREMATCH
   self.model = None


def fit(self, X: np.ndarray, y: np.ndarray) -> None:
   # Add intercept
   import statsmodels.api as sm
   X_ = sm.add_constant(X)
   self.model = NegativeBinomial(y, X_).fit(disp=0)


def pmf(self, X: np.ndarray, max_total: int = 10) -> np.ndarray:
   import statsmodels.api as sm
   if self.model is None:
      raise RuntimeError("Model not fit")
   X_ = sm.add_constant(X)
   mu = self.model.predict(X_)
   # Convert NB parameters to pmf for each integer 0..max_total
   # Here we use a simple approximation: treat as Poisson with mean mu
   # In full implementation, you'd use NB pmf with alpha from model.
   probs = []
   from math import exp, factorial
   for m in mu:
```

```
        p = np.array([exp(-m) * m**k / factorial(k) for k in range(max_total+1)])
        p /= p.sum()
        probs.append(p)
    return np.vstack(probs)
```

(Real NB pmf would use model's dispersion; kept short here.)

4.4. #10 – Survival / time-to-event model

We'll sketch with lifelines (you can add to requirements):

app/models/statistical/survival_goal.py

```
import numpy as np
from lifelines import CoxPHFitter
from app.core.base import SurvivalModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@register_component
class CoxGoalSurvivalModel(SurvivalModel):
    """
    Cox proportional hazards model for time-to-next-goal.
    X contains in-play state at the start of an interval; durations are minutes
    until next goal (or censoring).
    """
    def __init__(self) -> None:
        self.name = "CoxGoalSurvivalModel"
        self.task_type = TaskType.SURVIVAL
        self.mode = Mode.INPLAY
        self.cph = CoxPHFitter()
```

```
    def fit(self, X: np.ndarray, durations: np.ndarray, events: np.ndarray) -> None:
        import pandas as pd
        df = pd.DataFrame(X)
        df["T"] = durations
        df["E"] = events
        self.cph.fit(df, duration_col="T", event_col="E")

    def hazard(self, X: np.ndarray, t: float) -> np.ndarray:
        import pandas as pd
        df = pd.DataFrame(X)
        # Partial estimation: we use baseline hazard from fitted CPH
        base_cum_haz = self.cph.baseline_cumulative_hazard_.iloc[:, 0]
        # pick closest time to t
        idx = (base_cum_haz.index - t).abs().argmin()
        base_h = base_cum_haz.iloc[idx]
        # hazard ~ base_h * exp(beta^T x)
        lin_pred = self.cph.predict_partial_hazard(df).values.flatten()
        return base_h * lin_pred
```

4.5. #14 – Expected Goals (xG) logistic

app/models/statistical/xg_logistic.py

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from app.core.base import SupervisedModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode
```

```python
@register_component
class XGShotLogisticModel(SupervisedModel):
    """
    Shot-level xG model: predicts probability that a shot is a goal.
    """
    def __init__(self) -> None:
        self.name = "XGShotLogisticModel"
        self.task_type = TaskType.OUTCOME  # binary (goal vs no-goal)
        self.mode = Mode.BOTH
        self.model = LogisticRegression(max_iter=500)

    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        self.model.fit(X, y)

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """
        Returns [P(no-goal), P(goal)] per shot.
        """
        return self.model.predict_proba(X)
```

Your **shot-level feature builder** (distance, angle, body part, etc.) plugs into this.

4.6. #15–17 – rating systems

Example: Elo.

app/models/ratings/elo.py (refactoring previous version):

```python
from dataclasses import dataclass
from typing import Any, Optional
```

```python
from sqlalchemy.orm import Session
from datetime import date
from app.core.base import RatingModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode
from app.db.models import Match, RatingHistory


def expected_score(r_a: float, r_b: float) -> float:
    return 1.0 / (1.0 + 10 ** ((r_b - r_a) / 400))


@dataclass
class EloConfig:
    k: float = 20.0
    initial_rating: float = 1500.0


@register_component
class EloRatingSystem(RatingModel):
    def __init__(self, config: EloConfig | None = None) -> None:
        self.name = "EloRatingSystem"
        self.task_type = TaskType.RATING
        self.mode = Mode.BOTH
        self.config = config or EloConfig()
        self._cache: dict[int, float] = {}

    def update_from_matches(self, matches: list[Match], session: Session) -> None:
        for m in sorted(matches, key=lambda x: x.date):
```

```
        h = self.get_rating(m.home_team_id, m.date)
        a = self.get_rating(m.away_team_id, m.date)
        exp_h = expected_score(h, a)
        exp_a = 1 - exp_h

        hg = m.home_ft_goals or 0
        ag = m.away_ft_goals or 0
        if hg > ag:
            act_h, act_a = 1.0, 0.0
        elif hg == ag:
            act_h, act_a = 0.5, 0.5
        else:
            act_h, act_a = 0.0, 1.0

        new_h = h + self.config.k * (act_h - exp_h)
        new_a = a + self.config.k * (act_a - exp_a)

        session.add(RatingHistory(team_id=m.home_team_id, date=m.date,
                                  rating_type="elo", rating=new_h))
        session.add(RatingHistory(team_id=m.away_team_id, date=m.date,
                                  rating_type="elo", rating=new_a))
        self._cache[m.home_team_id] = new_h
        self._cache[m.away_team_id] = new_a
    session.commit()

def get_rating(self, team_id: int, when: Optional[date] = None) -> float:
```

```
    # In memory cache fallback; in production, read from DB for date <= when
    return self._cache.get(team_id, self.config.initial_rating)
```

Similar pattern for:

- **Glicko/TrueSkill** → GlickoRatingSystem, TrueSkillRatingSystem.
- **Bradley–Terry** → BradleyTerrySystem (fit strengths via MLE).

4.7. #18 – Monte Carlo on top of score model

app/models/simulation/score_monte_carlo.py

```
import numpy as np
from dataclasses import dataclass
from typing import Any, Dict
from app.core.base import SimulationModel, ScorelineModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@dataclass
class MCConfig:
  n_sims: int = 20000
  max_goals: int = 8


@register_component
class ScoreMonteCarloSimulator(SimulationModel):
  """
  Wraps any ScorelineModel (Poisson, DC, Bivariate, Copula etc.)
  and simulates full match or remaining time (if you adjust lambdas).
  """
  def __init__(self, base_model: ScorelineModel, config: MCConfig | None = None):
```

```python
        self.name = "ScoreMonteCarloSimulator"
        self.task_type = TaskType.SIMULATION
        self.mode = Mode.BOTH
        self.base_model = base_model
        self.config = config or MCConfig()

    def simulate(self, match_state: Any, n_sims: int | None = None) -> Dict[str, Any]:
        n = n_sims or self.config.n_sims
        matrix = self.base_model.predict_score_matrix(match_state)
        # Flatten PMF for sampling
        pmf_flat = matrix.flatten()
        outcomes = np.random.choice(
            matrix.size,
            size=n,
            p=pmf_flat
        )
        hg, ag = np.divmod(outcomes, matrix.shape[1])
        home_win = (hg > ag).mean()
        draw = (hg == ag).mean()
        away_win = (hg < ag).mean()
        return {
            "home_win": float(home_win),
            "draw": float(draw),
            "away_win": float(away_win),
            "sampled_scores": list(zip(hg.tolist(), ag.tolist()))
        }
```

4.8. #19–27 – classical ML models

Refactor earlier LogisticOutcome etc. to inherit from SupervisedModel.

Example: app/models/ml/classical/logistic_cls.py for #20.

```
import numpy as np

from sklearn.linear_model import LogisticRegression

from app.core.base import SupervisedModel

from app.core.registry import register_component

from app.core.tasks import TaskType, Mode


@register_component

class LogisticOutcomeModel(SupervisedModel):

    def __init__(self) -> None:

        self.name = "LogisticOutcomeModel"

        self.task_type = TaskType.OUTCOME

        self.mode = Mode.PREMATCH

        self.model = LogisticRegression(max_iter=500, multi_class="multinomial")


    def fit(self, X: np.ndarray, y: np.ndarray) -> None:

        self.model.fit(X, y)


    def predict_proba(self, X: np.ndarray) -> np.ndarray:

        return self.model.predict_proba(X)
```

RandomForest, SVM, XGBoost, etc. are analogous—same base, different underlying estimator.

4.9. #28–31, 35 – clustering & PCA

app/models/ml/unsupervised/kmeans_clusters.py:

```
import numpy as np
```

```
from sklearn.cluster import KMeans
from app.core.base import PreprocessorModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@register_component
class KMeansTeamClusterer(PreprocessorModel):
    """
    Clusters teams/matches into style clusters (attack-heavy, defense-heavy, etc.).
    Used mostly offline or as feature transformer: cluster id as feature.
    """
    def __init__(self, n_clusters: int = 6) -> None:
        self.name = "KMeansTeamClusterer"
        self.task_type = TaskType.PREPROCESSOR
        self.mode = Mode.BOTH
        self.model = KMeans(n_clusters=n_clusters, random_state=42)

    def fit_transform(self, X: np.ndarray) -> np.ndarray:
        return self.model.fit_predict(X).reshape(-1, 1)

    def transform(self, X: np.ndarray) -> np.ndarray:
        return self.model.predict(X).reshape(-1, 1)
```

app/models/ml/dim_reduction/pca_reducer.py:

```
import numpy as np
from sklearn.decomposition import PCA
from app.core.base import PreprocessorModel
```

```
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@register_component
class PCAMatchReducer(PreprocessorModel):
    def __init__(self, n_components: int = 10) -> None:
        self.name = "PCAMatchReducer"
        self.task_type = TaskType.PREPROCESSOR
        self.mode = Mode.BOTH
        self.model = PCA(n_components=n_components)


    def fit_transform(self, X: np.ndarray) -> np.ndarray:
        return self.model.fit_transform(X)


    def transform(self, X: np.ndarray) -> np.ndarray:
        return self.model.transform(X)
```

GMM / DBSCAN / Hierarchical clustering implement the same interface.

4.10. #32–34 & #36 – deep & sequence models (LSTM/Transformer)

We'll sketch an LSTM model using PyTorch (you can swap for Keras if you prefer):

app/models/ml/deep/lstm_seq.py

```
import numpy as np
from typing import Any
import torch
from torch import nn
from app.core.base import SequenceModel
from app.core.registry import register_component
```

```python
from app.core.tasks import TaskType, Mode


class _LSTMNet(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int = 64, num_layers: int = 1, num_classes: int = 3):
        super().__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        # x: (batch, seq_len, input_dim)
        out, _ = self.lstm(x)
        # take last time step
        out = out[:, -1, :]
        out = self.fc(out)
        return out


@register_component
class LSTMOutcomeSequenceModel(SequenceModel):
    """
    Sequence model for outcome prediction based on last N matches or in-play event streams.
    """
    def __init__(self, input_dim: int = 16, hidden_dim: int = 64) -> None:
        self.name = "LSTMOutcomeSequenceModel"
        self.task_type = TaskType.SEQUENCE
        self.mode = Mode.BOTH
```

```
    self.input_dim = input_dim
    self.net = _LSTMNet(input_dim, hidden_dim)
    self.loss_fn = nn.CrossEntropyLoss()
    self.optimizer = torch.optim.Adam(self.net.parameters(), lr=1e-3)

def fit_sequences(self, sequences: Any, targets: np.ndarray, epochs: int = 10) -> None:
    """
    sequences: list/array of shape (n_samples, seq_len, input_dim)
    targets: class labels (0=home,1=draw,2=away)
    """
    X = torch.tensor(np.array(sequences), dtype=torch.float32)
    y = torch.tensor(targets, dtype=torch.long)
    dataset = torch.utils.data.TensorDataset(X, y)
    loader = torch.utils.data.DataLoader(dataset, batch_size=32, shuffle=True)

    self.net.train()
    for _ in range(epochs):
        for xb, yb in loader:
            self.optimizer.zero_grad()
            logits = self.net(xb)
            loss = self.loss_fn(logits, yb)
            loss.backward()
            self.optimizer.step()

def predict_sequence(self, sequence: Any) -> np.ndarray:
    self.net.eval()
```

```python
    x = torch.tensor(np.array(sequence)[None, ...], dtype=torch.float32)
    with torch.no_grad():
        logits = self.net(x)
        probs = torch.softmax(logits, dim=1).cpu().numpy()[0]
    return probs
```

MLP and CNN can be implemented with either scikit-learn MLP or PyTorch, but they still implement either SupervisedModel or SequenceModel.

Transformer-based models would have a similar pattern, just with a different network architecture.

---

5. Training orchestrator (works for any model)

We refactor training so it can handle *any* component type, based on its task_type and mode.

app/services/training.py

```python
import os
import numpy as np
from typing import List, Any
from sqlalchemy.orm import Session
from app.core.registry import get_component_class
from app.core.base import (
    SupervisedModel,
    ScorelineModel,
    TotalsModel,
    SurvivalModel,
    RatingModel,
    SequenceModel,
    PreprocessorModel,
)
```

```
from app.features import prematch, shot_level, inplay, sequences
from app.config import MODEL_STORE_DIR, DEFAULT_LEAGUE_ID

os.makedirs(MODEL_STORE_DIR, exist_ok=True)

class TrainingService:
    def __init__(self, session: Session, league_id: int = DEFAULT_LEAGUE_ID):
        self.session = session
        self.league_id = league_id

    # --- Main entry point ---

    def train_component(self, component_name: str) -> None:
        cls = get_component_class(component_name)
        comp = cls()  # type: ignore
        print(f"Training {component_name} for league {self.league_id}")

        if isinstance(comp, SupervisedModel):
            X, y = prematch.build_outcome_dataset(self.session, self.league_id)
            comp.fit(X, y)

        elif isinstance(comp, ScorelineModel):
            matches = prematch.get_match_objects(self.session, self.league_id)
            comp.fit(matches)

        elif isinstance(comp, TotalsModel):
```

```
        X, y = prematch.build_totals_dataset(self.session, self.league_id, target="goals")
        comp.fit(X, y)

    elif isinstance(comp, SurvivalModel):
        X, durations, events = inplay.build_survival_dataset(self.session, self.league_id)
        comp.fit(X, durations, events)

    elif isinstance(comp, RatingModel):
        matches = prematch.get_match_objects(self.session, self.league_id)
        comp.update_from_matches(matches, self.session)

    elif isinstance(comp, SequenceModel):
        seqs, targets = sequences.build_outcome_sequences(self.session, self.league_id)
        comp.fit_sequences(seqs, targets)

    elif isinstance(comp, PreprocessorModel):
        X = prematch.build_feature_matrix(self.session, self.league_id)
        comp.fit_transform(X)

    else:
        raise NotImplementedError(f"No training path for {component_name}")

    out_path = os.path.join(MODEL_STORE_DIR,
f"{component_name}_L{self.league_id}.joblib")
    comp.save(out_path)
    print(f"Saved {component_name} to {out_path}")
```

You'd implement the helper functions in app/features/prematch.py, inplay.py, etc., to return appropriate X,y / sequences / durations.

train_all_models.py can just loop over a list of component names:

```
# scripts/train_all_models.py

from app.db.session import SessionLocal

from app.services.training import TrainingService

from app.core.registry import list_components


if __name__ == "__main__":

    session = SessionLocal()

    svc = TrainingService(session)

    for name in list_components().keys():

        # You might filter which ones to train for now:

        if name.endswith("OutcomeModel") or name.endswith("GoalModel"):

            svc.train_component(name)

    session.close()
```

---

6. Prediction service + FastAPI endpoints (unchanged on the surface)

From the API's point of view, the user still wants:

- /predictions/prematch – W/D/L, scoreline, totals, etc.
- /predictions/inplay – live W/D/L, next goal, etc.

The prediction service just has to know *which* components to load and which feature builder to call.

app/services/prediction.py (simplified for OUTCOME & SCORELINE only):

```
import os

import numpy as np

from typing import Dict, Any, List
```

```
from sqlalchemy.orm import Session
from app.core.registry import get_component_class
from app.core.base import SupervisedModel, ScorelineModel
from app.features import prematch, inplay
from app.config import MODEL_STORE_DIR, DEFAULT_LEAGUE_ID


class PredictionService:
    def __init__(self, session: Session, league_id: int = DEFAULT_LEAGUE_ID):
        self.session = session
        self.league_id = league_id


    def _load(self, name: str):
        cls = get_component_class(name)
        path = os.path.join(MODEL_STORE_DIR, f"{name}_L{self.league_id}.joblib")
        return cls.load(path)


    def prematch_outcome(self, match_id: int, model_names: List[str]) -> Dict[str, Any]:
        X_match = prematch.build_single_match_features(self.session, match_id, self.league_id)
        X = X_match.reshape(1, -1)
        result: Dict[str, Any] = {}
        for name in model_names:
            model = self._load(name)
            if isinstance(model, SupervisedModel):
                probs = model.predict_proba(X)[0]
                result[name] = {
                    "home_win": float(probs[0]),
```

```python
            "draw": float(probs[1]),
            "away_win": float(probs[2]),
        }
    elif isinstance(model, ScorelineModel):
        state = prematch.build_match_state(self.session, match_id, self.league_id)
        matrix = model.predict_score_matrix(state)
        odds = model.outcome_probs_from_matrix(matrix)
        result[name] = odds
    else:
        # could add support for SimulationModel or others
        continue
return result


def inplay_outcome(self, live_event_json: dict, model_names: List[str]) -> Dict[str, Any]:
    snap = inplay.build_snapshot_from_event(live_event_json)
    X = snap.X.reshape(1, -1)
    result: Dict[str, Any] = {"minute": snap.minute}
    for name in model_names:
        model = self._load(name)
        if isinstance(model, SupervisedModel):
            probs = model.predict_proba(X)[0]
            result[name] = {
                "home_win": float(probs[0]),
                "draw": float(probs[1]),
                "away_win": float(probs[2]),
            }
```

```
    return result
```

FastAPI endpoints from the previous answer remain very similar; they only call PredictionService with a list of component names.

---

7. How this refactor supports *all 36 models*

- Every row in your table maps to a **concrete class** inheriting from one of the base types.
- Training is **unified** via TrainingService.train_component, which chooses the right data based on the component type.
- Prediction is **unified** via PredictionService, which loads components and calls the correct method.
- Unsupervised models (KMeans, PCA, GMM, DBSCAN) are used as **preprocessors** or for **analysis endpoints** (e.g. “cluster my teams” in /analysis routes).
- Sequence models (LSTM, Transformer) have their own SequenceModel path, trained on sequences of matches or in-play events built by features/sequences.py.
- Rating systems (Elo, Glicko, TrueSkill, Bradley–Terry) are all RatingModels but can also feed into feature builders (as rating features) and baseline outcome predictions.
- Monte Carlo simulator wraps *any* ScorelineModel (Poisson, DC, Bivariate, Copula, Zero-inflated) and provides richer simulation-based forecasts for both pre-match and in-play.