```
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()
    )
```

```
    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
```

```python
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

```python
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]),
                }
```

```
            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:
```