ChatGPT

Theoretical Foundations for an Optimal NFL Spread Betting Model Architecture

1. Single vs. Ensemble Models: Bias-Variance and Diversity

Ensemble Superiority: In theory, an ensemble of predictors can outperform any single model under broad conditions. If each individual model has an error rate below 50% (better than random guessing) and their errors are *independent*, combining many models can drive error towards zero as the ensemble size grows [1]. In practice, errors are not fully independent, but ensembles still reduce variance and improve robustness by averaging out uncorrelated mistakes. This is formalized by the **bias-variance decomposition**: ensembles primarily reduce the variance term of error [2]. Bagging (bootstrap averaging) and random forests exploit this by training diverse classifiers whose average prediction has lower variance than any constituent model. Boosting methods go further by also reducing bias through sequentially correcting errors [2].

Accuracy–Diversity Trade-off: The efficacy of an ensemble depends on members being both **accurate** and **diverse**. Krogh and Vedelsby (1995) derived that ensemble error can be decomposed into the average error of individual classifiers minus a term related to inter-classifier disagreement [3]. **Theorem (Krogh & Vedelsby)**: *Ensemble Error = Average base error – Diversity*. This implies an *ideal ensemble* has members that are highly accurate *and* maximally disagree (uncorrelated) in their residual errors [3]. Empirically, combining identical or highly correlated models yields no gain [4]. Thus, methods to promote diversity – e.g. training on different data subsets (bagging), using different algorithms (heterogeneous ensemble), or injecting randomness – are crucial [5]. In our NFL context, the chosen base learners (LightGBM, XGBoost, CatBoost) each use different tree-building strategies and regularization, adding heterogeneity. The supplementary logistic regression adds an even more different (linear) perspective, increasing diversity.

Optimal Number of Models: While larger ensembles generally perform better by the Law of Large Numbers, they exhibit *diminishing returns*. Beyond a certain size, improvement plateaus as new models contribute highly correlated predictions [1]. The optimal ensemble size depends on how quickly diversity gains drop off versus added complexity. Research suggests an ideal size might be surprisingly small if models are strong and independent – in fact, under a weighted majority vote, one theorem shows the optimal number of components equals the number of classes (two for binary tasks) in a *best-case* scenario of completely independent, strong learners [6] [7]. In reality, moderate ensembles (dozens or fewer) often suffice; too many models can even hurt if they start to “crowd” the hypothesis space with redundant voters. Given our **data constraints (~3,000 games)**, a compact ensemble of 3–5 carefully selected models is optimal to avoid overfitting. This aligns with the architecture: ~3 primary tree models capture most signal, with a small meta-learner on top (rather than hundreds of models which the limited data couldn’t support).

Proof Sketch (Variance Reduction): For intuition, consider N independent unbiased models each with variance σ^2 in their prediction. The ensemble average has variance σ^2/N, a $1/N$ reduction in variance [1]. If individual models also have small bias, the ensemble’s mean-squared error can be substantially lower than individuals’. In practice independence is partial; one can show ${\rm Var}

(\bar{y}) = \frac{1}{N^2}\sum_{i,j}{\rm Cov}(y_i,y_j)$. If off-diagonal covariances are small (diverse models), ensemble variance is much reduced. This explains why ensembling often yields a better bias-variance trade-off: averaging doesn't change the bias (if models are unbiased on average) but slashes variance.

Ensemble vs Single in NFL Betting: For predicting against the spread, small improvements in accuracy translate to significant betting edge due to the slim margins. An ensemble can capture a wider range of patterns (e.g. XGBoost might excel at interaction effects, LightGBM at categorical handling, etc.) ensuring no single modeling approach's weaknesses dominate [8]. The ensemble's prediction is effectively a **weighted vote**: in our design 40% LightGBM, 30% XGBoost, 20% CatBoost, 10% LR. These weights can be learned or set via cross-validation to optimize Brier or log-loss. The meta-learner (see below) will assign near-optimal combination weights. Theory guarantees that a well-tuned stacked ensemble (often called a "Super Learner") will asymptotically perform *at least as well as the best individual model* and usually better [9]. This is backed by an oracle inequality proof for stacking: given enough data, the ensemble's generalization error will approach the minimum possible if one could choose the best model or weighting in hindsight [9].

Meta-Learner Selection: In stacking, a **meta-learner** combines base model outputs into a final prediction. A common choice is logistic regression on the predicted probabilities (or logits) of base models. The meta-learner should be **simple** enough to avoid overfitting (since the inputs – predictions from base models – are few and correlated) [9]. **Logistic regression** is theoretically appealing here: it provides a *calibrated probabilistic output* (by construction) and can optimally weight the models' predictions to maximize likelihood. It effectively learns an optimal linear combination of the base forecasts to produce a final win probability. For sports betting, this is valuable because it preserves interpretability (weights indicating each model's contribution) and calibration (ensuring the final ensemble probability is neither over- nor under-confident on a held-out set). A shallow tree or XGBoost model as meta-learner could capture any non-linear blending (e.g. recognizing specific conditions when one model is more reliable), but the risk is overfitting given only ~3 model-input features across ~3000 training examples. In practice, **constrained meta-learners** (logistic or ridge regression, or at most a small tree) tend to work best [10]. The theoretical justification comes from stacking literature: if the library of base learners is rich, a simple convex combination is sufficient to achieve the oracle optimal combination [9]. Complex meta-learners offer little benefit unless base models have systematic interaction effects to exploit. Thus, we choose a logistic meta-learner for its **calibration, simplicity, and fast training** (important for weekly retraining constraints).

Computational Complexity: The ensemble approach, especially tree-based, is computationally efficient for our data size. Training complexity for each gradient-boosted tree model is roughly $O(n \log n \cdot d)$ per iteration (where n=#samples, d=#features) and they train in seconds on 3000 samples. Combining three such models and a logistic meta-learner easily fits in the 4-hour training window (in fact, likely under an hour). Inference is also fast: evaluating a few hundred trees per game is on the order of milliseconds. The **stacking meta-learner** adds negligible overhead (a linear model evaluation is microseconds). Memory-wise, storing three tree ensembles (tens of MB) is within the 10GB limit, and inference memory (16GB RAM) is plenty for loading models. Thus, ensemble complexity is not a bottleneck. The bias-variance trade-off improved by ensemble yields better **generalization per computation** spent, which is crucial in a small-data regime like NFL betting.

2. Two-Stage Architecture: Prediction + Calibration Separation

Stacked Prediction (Stage 1): The first stage produces a raw probability estimate that a team will cover the spread, using a stacked ensemble of the base models. From a theoretical standpoint, this stage is all about optimizing *predictive accuracy* (minimizing error or maximizing likelihood of outcomes). Stacking (also called **stacked generalization**) is a proven method to minimize generalization error by **blending models of different inductive biases** [11]. By training a meta-learner on out-of-fold predictions of base models, it corrects each model's biases in different regions of feature space. **Wolpert's Stacking Theorem:** Under mild conditions, stacking with a sufficiently rich meta-learner will generalize at least as well as the best base model, effectively approaching the Bayes-optimal combination of them [9]. This justifies using Stage 1 to harness multiple algorithms' strengths. In our design: features $X \to$ [LightGBM, XGBoost, CatBoost, Logistic] $\to$ meta-learner $\to \hat{p}*{\text{raw}}$. The meta-learner (logistic) outputs $\hat{p}* \in [0,1]$ as the model's }**predicted probability** of the event (covering the spread). We also obtain an estimate of uncertainty or consensus (for example, variance among the base models' predictions or the logistic model's output entropy).

Calibration & Decision (Stage 2): The second stage takes $\hat{p}*{\text{raw}}$ and adjusts it to $\hat{p}*$, a } *calibrated probability*, and then issues a betting decision (bet or no-bet) based on confidence thresholds. The motivation for separating this stage is **mathematical modularity**: The prediction stage focuses on ranking games by likelihood of cover (discrimination), while the calibration stage ensures the predicted probabilities are interpretable as true frequencies (calibration). This separation is supported by reliability theory: a model can have excellent discrimination (correct ordering of odds) but still be miscalibrated (e.g. systematically overestimating probabilities) [12]. By applying a calibration mapping (like isotonic or Platt scaling) on Stage-1 outputs, we correct any distortion in the probability scale **without altering the rank-ordering of predictions**. This two-step approach often improves log-likelihood and decision-making; for example, **boosted trees often need calibration** – Caruana et al. (2005) showed that applying Platt's sigmoid or isotonic regression on boosted models significantly improved probability estimates [12]. After calibration, the previously underconfident or overconfident raw outputs become *properly calibrated probabilities*, which are crucial for bet sizing and threshold-based decisions.

Mathematical Justification: Ideally, if $\hat{p}*{\text{raw}}(X)$ were an unbiased estimate of $P(Y=1|X)$, calibration would be unnecessary. However, complex models like boosted trees can be **biased estimators of probability** – they often push outputs toward 0 or 1 (overconfidence) or under-utilize the full probability range. A calibration function f (monotonic, increasing) can remap $\hat{p}*}$ to f(\hat{p*{raw})$ such that $P(Y=1 \mid \hat{p}*=s) = f(s)$ on the validation data. Isotonic regression finds a piecewise-constant such f that minimizes calibration error, and logistic Platt scaling finds a parametric sigmoid $f(s) = 1/(1+\exp(as + b))$. **Properness:** A key theoretical point is that if Stage 1 is trained to minimize a proper scoring rule (like log loss or Brier score), its outputs are *in expectation* calibrated. But finite sample and model mis-specification can lead to miscalibration, so an explicit calibration stage fine-tunes the mapping. Separating the stages is akin to a *decomposition of objectives*: Stage1 handles *accuracy* (discrimination), Stage2 handles *calibration* (probability estimates). This can sometimes achieve a better outcome than trying to do both in one stage – for example, optimizing a complex loss that mixes accuracy and calibration can complicate learning.

Information Loss vs. Sufficiency: One concern is whether feeding only the predicted probability $\hat{p}*{raw}$ (and perhaps some uncertainty measure) into Stage 2 loses useful information from the original features. In theory, if $\hat{p}*$ is a **sufficient statistic** for Y given X (i.e. the Stage-1 model has distilled all predictive information of X into $\hat{p}*{raw}$), then no additional feature data is needed for calibration.*

*Calibration can be done as a function of $\hat{p}*$ alone. However, if Stage 1 leaves some systematic biases that could be detected via X, a more **feature-dependent calibration** could improve performance. For instance, the model might be poorly calibrated specifically for games with extreme weather or involving certain teams – information on those features might help adjust the probability. To address this, the architecture is exploring a *residual calibration network*: a small secondary model that takes $(X, \hat{p}*{raw}) $ and predicts a calibration adjustment. This essentially learns $g(X, \hat{p}) \approx P(Y|X) - \hat{p}$ as a correction term. Theoretically, this is similar to adding a small model to capture feature-dependent residuals of the main predictor, which can improve calibration in subpopulations. It's a form of* ***multi-dimensional calibration*** *beyond isotonic (which is 1-D). The risk is overfitting, so one might regularize this network heavily. If successful, it means Stage 2 is not purely a function of $\hat{p}*$ for those cases).}$ but also uses features to refine probabilities for known model weaknesses (for example, maybe the model is known to overestimate home underdogs – a feature-aware calibrator could downweight $\hat{p

Optimal Information Passing: The question of what to feed into Stage 2 can be framed as retaining as much predictive information as possible without reintroducing the full complexity of Stage 1. Passing just a single number $\hat{p}*{raw}$ is minimal; passing $\hat{p}*$ plus an uncertainty metric (such as the spread between base models' outputs or model variance) gives Stage 2 insight into confidence. Another idea is passing the *log-odds* from Stage 1 instead of probability, which often linearizes the calibration function (Platt scaling essentially fits $a \cdot \text{logit}(\hat{p}) + b$). In summary, **optimal two-stage design** uses: Stage 1 to aggregate features into an initial probability estimate, Stage 2 to *monotonically transform* that probability for calibration using as little extra info as needed. This maintains a clean separation: Stage 1 does heavy lifting on features, Stage 2 fine-tunes distributional properties. From a complexity standpoint, Stage 2 (isotonic or logistic scaling) is a very light model (isotonic regression can be solved in $O(m \log m)$ for m calibration examples, and logistic in closed-form or simple Newton iteration). It adds negligible computational cost to training (few seconds) and inference (just an array lookup or sigmoid computation per instance).

Complexity Analysis of the Two-Stage Pipeline: The **stacked ensemble + calibration** architecture remains efficient. Training Stage 1 involves fitting the base models (which is parallelizable or sequential in minutes) and a meta-learner. Stage 2's isotonic regression is essentially sorting the validation set predictions and doing a pool-adjacent-violators algorithm (worst-case $O(n^2)$ but typically $O(n)$ with isotonic optimization techniques). Logistic calibration is even simpler (just fitting two parameters by minimizing log-loss, solvable by gradient descent or Newton's method in a tiny param space). Thus, the two-stage training easily fits the 4-hour window. At inference, the pipeline is: compute base model outputs ($\sim$0.1 sec), apply meta-learner (<1 ms), then calibrator (<1 ms). **Overall throughput**: well under the <100ms per game requirement. Memory: Stage 2 adds trivial memory (a few coefficients or a lookup table for isotonic bins).

In summary, the two-stage design is *theoretically justified* by the **divide-and-conquer of prediction and calibration**. It isolates model fitting (which benefits from complex non-linear learners) from probability calibration (which benefits from simple monotonic corrections). This yields better calibrated probabilities, crucial for betting decisions, while maintaining optimal predictive power [12]. It also makes the system more *interpretable*: Stage 1 can be analyzed for feature importance in predicting outcomes, and Stage 2 can be analyzed for how it skews those probabilities to align with reality (often revealing if the model was over/under confident overall).

3. Loss Function Design for Betting Outcomes

In standard prediction tasks, losses like mean-squared error (MSE) or cross-entropy are used to optimize accuracy. For betting, however, the objective is to maximize **financial gain** (or log-wealth growth) rather than just predictive accuracy. This calls for custom loss design aligned with betting outcomes:

Kelly Criterion and Log Utility: A rational strategy to maximize long-run wealth is to use the *Kelly criterion*, which tells how much to bet as a fraction of bankroll based on the edge. Kelly's formula for a binary bet with probability p and odds (net payout) b is: *fraction* $f^ = *\frac{p b - (1-p)}{b}$ (for decimal odds, if winning returns $1+b$). This maximizes the expected log-growth of wealth. The* ideal model *for a bettor would output the true win probability p for each game; the bettor then stakes $f*^$ of bankroll. To align model training with this goal, we consider a **loss function $L(\theta)$ that directly relates to final wealth**. If our model predicts $\hat{p}$ for a game where the true outcome $y\in{0,1}$ (1 = win bet), and we bet fraction $f(\hat{p})$ (e.g. Kelly fraction based on $\hat{p}$), the *log wealth change* is $\log(1 + y \cdot f(\hat{p}) b - (1-y)\cdot f(\hat{p}))$. We want to maximize $E[\log \text{wealth}]$. This is equivalent to minimizing the negative expected log return: $$L_{\text{Kelly}}(\hat{p}, y) = -\Big(y \ln(1 + f(\hat{p}) b) + (1-y)\ln(1 - f(\hat{p})) \Big).$$ This loss explicitly punishes predictions that lead to suboptimal betting fractions. While directly optimizing this is complicated (non-convex in $\hat{p}$, and $f(\hat{p})$ itself a function of $\hat{p}$), we can derive intuition: If the model overestimates p, it will bet too much and the $(1-y)\ln(1-f)$ term (occurring when the bet loses) heavily penalizes it. If it underestimates p, it bets too little (or not at all), missing out on the $(y)\ln(1+f b)$ reward – but note, in the loss formulation missing a bet (false negative) incurs *no direct monetary loss*, only opportunity cost. This asymmetry reflects the business constraint: false positives (bet when you shouldn't) lose money, whereas false negatives (skip a good bet) have no immediate loss besides regret.

Proper Scoring and Kelly: It turns out that maximizing expected log wealth is closely related to using **log-loss (cross-entropy)** as the training objective. In fact, if odds are even (b=1), Kelly betting reduces to betting when $p>0.5$ and the expected log-utility is maximized when the model's predicted probabilities match true probabilities – which is exactly what minimizing log-loss achieves. Cross-entropy $-\big(y\ln \hat{p} + (1-y)\ln(1-\hat{p})\big)$ is a *proper scoring rule*, meaning it is minimized when $\hat{p}$ equals the true underlying probability p. Thus, training with log-loss pushes the model to predict the correct win probability for each game. If the model's probabilities are accurate, then betting according to Kelly on those probabilities is *optimal in expectation*. In short, **log-loss is theoretically aligned with Kelly-optimal betting** (it encourages well-calibrated probabilities, which are what Kelly needs). A custom "Kelly loss" would explicitly weight errors by their impact on bankroll, but often this just re-scales the gradients similarly to log-loss focusing on getting probabilities right, especially near the decision threshold.

Incorporating Odds into Loss: In spread betting, odds are typically -110 (bet 110 to win 100), meaning $b \approx 0.91$. We can incorporate the odds by *weighting examples* in the loss proportional to potential payout or cost. For example, a false positive (predicting >52.5% win probability and betting, but losing) costs 1 unit, whereas a false negative costs 0. To reflect this asymmetry, one could assign a higher loss weight to predicting a game as a good bet when it isn't. One approach: use a **cost-sensitive loss** where false positives are weighted by C_{FP} (e.g. 1 for losing \$100) and false negatives by C_{FN} (close to 0, since missing a win is just lost profit, not actual loss of funds). For instance, a weighted log-loss: $$L(\hat{p},y) = - \big(C_{1} \, y \ln \hat{p} + C_{0}\, (1-y)\ln(1-\hat{p})\big),$$ where C_1 and C_0 adjust the impact of false-negative vs false-positive errors. If missing a winning bet is less severe, C_1 can be smaller relative to C_0. However, care is needed: too asymmetric and the model might always predict "no bet” to avoid the high false-positive cost. Instead, we may handle this at decision time by requiring a high confidence threshold (which we do: minimum edge 2.5% translates to $\hat{p}>0.525$ to bet). Thus, during training we might still use a symmetric proper loss (to get true probabilities), and enforce the asymmetry in the decision rule rather than the loss.

Mean Squared Error (MSE) vs. Mean Absolute Error (MAE) vs. Quantile Loss: These losses have different properties and are used for different target types:
- **MSE**: For a binary outcome interpreted as 0/1, using MSE on the probability estimate is equivalent to the **Brier score**, a proper scoring rule for probability calibration. MSE penalizes large errors more heavily (quadratically). It is smooth and convex, yielding an easier optimization. In our context, minimizing Brier score means the forecasted probability $\hat{p}$ is as close as possible to the outcome (0 or 1) in mean squared terms – this balances calibration and refinement. A perfectly calibrated model minimizes expected Brier score while also being sharp (confident predictions that are correct) [13] .
- **MAE**: Mean absolute error is less commonly used for probabilistic prediction because it’s not a proper scoring rule for probability – it doesn’t strongly encourage extreme probabilities to match extreme outcomes. MAE is more robust to outliers, but here outliers are just 0/1 outcomes. If used, MAE would treat all errors linearly, which might under-penalize being very overconfident. It’s also not differentiable at 0,1 which complicates gradient-based training. Generally, MAE is preferred in regression when we care about median error. For classification probability, MAE would encourage predicting the median outcome (which for symmetric 0/1 distribution is 0.5 for each game, not useful). So MAE is suboptimal for probability calibration.
- **Quantile Loss**: Quantile loss (pinball loss) is used to predict a certain quantile of a distribution. If our task were predicting the **point spread margin** (continuous outcome), quantile loss could target, say, the median margin or the 90th percentile margin. However, in direct classification (cover vs not), quantile loss is not typical. That said, one could use quantile loss to model the *distribution of point differential* and then derive the probability of cover from that distribution. For example, predicting the median margin helps estimate if a team is likely to exceed the spread. More relevant would be using *quantile regression to estimate the distribution* of the point difference – then $\Pr(\text{cover}) = \Pr(\text{point_diff} > -spread)$ could be obtained. But this two-step approach is more complex than directly predicting $\Pr(\text{cover})$. So quantile loss might be useful if we care about tail risks or certain percentiles of outcomes (perhaps to understand blowout vs narrow win probabilities). For the main model, MSE (Brier) or cross-entropy are preferable as they directly address probability accuracy.

Which Loss to Choose? In summary, **cross-entropy (log loss)** remains a top choice for training the Stage-1 model to output well-calibrated probabilities, which in turn align with maximizing betting returns (via Kelly). In fact, many betting models implicitly use log-loss by training classifiers. The **Brier score (MSE)** is another proper scoring rule; it directly measures calibration and accuracy in a squared-error sense and is also often used in sports probability models. One advantage of Brier is it’s more interpretable (it decomposes into calibration and refinement components). But optimizing Brier vs log-loss yields very similar behavior for a well-specified model. Notably, log-loss heavily penalizes extreme mispredictions (predicting 0.99 when outcome is 0), which might be good to avoid confident wrong bets, whereas Brier penalizes more moderately.

We can also incorporate **odds into the loss** by scaling errors by the potential payout. For example, if an underdog bet pays more, an error in estimating that probability could be more costly in missed opportunity, so one might upweight those examples proportional to odds. Conversely, since favorites pay less, an error on them might be less costly. A simple implementation: weight each game’s loss by the sum of potential profit and loss: $w = b + 1$ (e.g. ~1.91 for both sides in NFL). However, because spreads usually have equal odds for both sides, this weighting may not differentiate much. It matters more in moneyline betting where odds vary widely.

Finally, considering the **business constraints**: false positive = -\$100, false negative = \$0, minimum edge 2.5%. These imply that we care more about avoiding false positives (bad bets) than missing borderline bets. The **decision rule** will handle much of this (only bet if $\hat{p} > 0.525$ ensures with fair odds we expect positive EV). But we could also reflect it in training by slightly skewing the classification threshold or weighting the loss for 0 vs 1 outcomes. For instance, treating positive class (bet wins) as \$100 reward and negative class (bet loses) as -\$100 cost could be encoded in a utility or cost matrix. A *utility-based loss* might directly maximize expected profit per game: $L = -[100 \cdot P(\text{predict bet}) \cdot (\text{true win probability} - 0.5)]$ or similar. This gets complex; practically, training for probability and then applying a threshold is easier and nearly as effective.

Conclusion: We derive that a custom loss aligning with Kelly is essentially a weighted log-loss focusing on calibration. The plan is to train Stage 1 with **logistic loss (cross-entropy)** for classification of cover vs no-cover, possibly with slight weighting to account for any class imbalance (though here spread ~50/50). Then calibrate those probabilities (Stage 2) possibly by minimizing a **calibration-specific loss** (e.g. isotonic typically minimizes squared error in predicted vs actual frequencies). By doing so, the model's outputs will be directly interpretable as win probabilities, which combined with Kelly fraction (0.25 of full Kelly, per business spec) yield an approximately bankroll-optimal betting strategy.

4. Online Learning and Incremental Update Mathematics

The NFL model is updated on a **weekly cycle**, which is a form of *online learning* in batched increments. We consider the theory of incremental learning to ensure convergence, stability, and adaptability:

Convergence in Incremental Updates: In online learning theory, if data are drawn i.i.d. from a stationary distribution, incrementally updating a model with new samples (using, say, stochastic gradient descent on each batch) will converge to a solution that minimizes the expected loss. Classic results like the Robbins-Monro theorem guarantee convergence of SGD with a diminishing learning rate. For convex loss functions, one can achieve an error that decays as $O(1/\sqrt{T})$ or better with proper scheduling. In fact, for strongly convex, smooth problems, SGD can reach the optimal $O(1/T)$ convergence rate [14]. **Theorem:** *For a strongly convex loss with Lipschitz gradients, using a decreasing step size $\eta_t = O(1/t)$, SGD converges to the global optimum at rate $O(1/t)$ in expectation* [14]. While our ensemble models are not strictly convex, this gives confidence that continuously incorporating more data will improve stability of estimates. Each week's new games (~16 samples) is small, but over a season the data grows (~300 new samples), reducing variance in the estimates.

Batch vs Streaming: We opt for a **weekly batch retraining** (not pure streaming). That is, after each week's games, we *retrain the model on all available data (past seasons + new week)*. This ensures we leverage all historical data to prevent forgetting older trends. The "online" aspect is that we do this regularly with growing data. Because NFL data is not massive, this is computationally feasible. The convergence guarantee here is simpler: as the dataset grows, retraining on the full dataset each time will produce a sequence of models that approach the true underlying function (assuming one exists) as n grows. This is essentially *increasing batch learning*, which converges under standard statistical learning theory (law of large numbers driving generalization error down).

Warm-Start Strategies: To speed up weekly retraining, we use **warm-starting**: initializing the new week's model training from last week's model parameters. Many tree ensemble libraries (XGBoost, LightGBM) allow continuing training from an existing model. For example, one can start from the previous model's trees and add a few more boosting rounds for the new data, or use the previous model as a prior (initial guess) and refine it. This can significantly reduce training time because the model is already near a good solution. Theoretically, warm-start acts like using the last solution as the initial point for optimizing the new objective (old data + new data). Since only 16 new instances are added, the optimum of the new dataset's loss is close to the old optimum. **Gradient descent viewpoint:** if θ^* was optimal for old data, for the new objective $\theta^{}$, we have $\theta^{} = \theta^* + \Delta$ where Δ is small (assuming new data doesn't radically change optimum). Starting at θ^* and doing a few gradient steps on the new loss will quickly converge to $\theta^{}$. This is analogous to* transfer learning *where each week's model transfers knowledge to the next. Empirically, warm-start can cut training time drastically, which is important given the 4-hour window and preference to run on CPU. In gradient boosting, warm-start might mean continuing to boost additional trees until performance stops improving (using previous trees as a base). Caution: we must watch for overfitting due to warm-start** – if the model starts to overfit old data patterns that aren't relevant anymore, continuing from it might slow adaptation to new trends. To mitigate that, one could allow a small learning rate on new data or even periodically retrain from scratch if a big shift is suspected (e.g. new season). But since NFL evolves relatively slowly, warm-start is mostly beneficial.

Learning Rate Scheduling: In online gradient descent, a common schedule is $\eta_t = \eta_0/(1 + \lambda t)$ or $\eta_t = \eta_0 \sqrt{\frac{C}{t}}$ for some constants, to ensure convergence without oscillation. In our weekly-batch approach, we effectively re-initialize the optimizer each week; we might use a fixed learning rate for tree boosting but limit number of new trees. Alternatively, we could decay the learning rate as more data accumulates (since with more data, we can afford smaller steps to fine-tune). For instance, early in season (less data), a larger learning rate or more aggressive update might be needed; later, as data grows, we decrease learning rate to get finer convergence. This mirrors stochastic approximation theory: high initial η to jump towards optimum, then decreasing η for fine convergence. Many libraries adjust boosting learning rate or number of boosting rounds based on data size – effectively doing this scheduling. From a theoretical standpoint, an **optimal learning schedule** minimizes cumulative regret in online learning. For convex loss, setting $\eta_t \propto 1/\sqrt{t}$ yields $O(\sqrt{T})$ regret (nearly optimal) and $\eta_t \propto 1/t$ yields $O(\log T)$ regret for strongly convex cases. While we are not updating one sample at a time, the principle of decaying update magnitude as more evidence is gathered still applies.

Prevention of Catastrophic Forgetting: *Catastrophic forgetting* refers to a model completely forgetting old knowledge when trained on new data (common in sequential neural net training) [15]. Our strategy of retraining on **all data** each week inherently avoids this – old games remain in the training set, so the model cannot forget them without penalty. In a purely online scenario where only new data is used for updates, one must be careful to not overwrite the model. Techniques include:
- **Regularization to past parameters:** e.g. Elastic Weight Consolidation (EWC) adds a penalty if new weights deviate too much from old weights that were important for past data [16]. This is based on a Fisher information estimate of which weights matter for old tasks [16].
- **Replay or memory buffer:** keep a subset of old data and intermix it with new data during training (so the model is reminded of earlier examples).
- **Lower learning rate for old knowledge:** like training new data for fewer epochs or smaller steps, so as not to overwrite.
- **Ensemble approaches:** interestingly, ensembles can naturally mitigate forgetting – one could maintain

separate models for different seasons or periods and combine them, so new data adds a new model rather than overwriting. The architecture could, for example, have a “season 2025 model” alongside a “2024 model” in the ensemble, with the meta-learner learning how to weight recent vs older patterns. This is not currently in our design, but it’s a known strategy in non-stationary environments.

Because NFL has relatively stable underlying dynamics (though strategies evolve, it’s slow), using all data with slight preference to recent (maybe via time-decay of sample weights) might be ideal. A *recent-game weighting* can be implemented (e.g. last 1-2 seasons slightly higher weight than games 10 years ago) to adapt to trends (this helps with concept drift). There’s a balance: too much weight on recent games could forget long-term stats (like historical team tendencies), too little and model reacts sluggishly to change. The **optimal strategy** in theory for a non-stationary but slowly changing distribution is to use a **sliding window or exponentially decayed memory** – proven to minimize error in a setting with concept drift. That is, minimize loss over last N samples or weight sample i by $\alpha^{(current_time - i)}$. This yields a bias-variance tradeoff between stability and adaptability. In practice, we may not need this because one NFL season (even a decade) is limited data; we want to use all of it. But if we notice concept drift (e.g., scoring patterns change due to rule changes), we could incorporate a decay.

Incremental Update Guarantees: If we were to update the model continuously as each game’s result comes in (rather than batch weekly), online learning theory (Cesa-Bianchi & Lugosi, 2006) provides regret bounds: using algorithms like Online Gradient Descent or Hedge ensures that, over many rounds, the algorithm’s total loss will be not much worse than the best fixed predictor in hindsight, growing sub-linearly in #games. For a small learning rate, each update causes a small adjustment – guaranteeing the model’s predictions converge to optimal for the current distribution, assuming it slowly shifts. The weekly batch update can be seen as an approximation to that where we accumulate a week of data and then do a batch optimization (which likely achieves lower loss on that week’s data than a single-step update would). This may introduce a slight lag (one-week delay in learning new info) but that’s acceptable and even desirable to verify data integrity and do validation.

Complexity of Weekly Retraining: With ~3000 games of data and 50-150 features, retraining even from scratch is not heavy – gradient boosting with 1000 trees might take a couple minutes. Warm-start can reduce it further, maybe to under a minute, since we just extend or fine-tune last week’s model. All steps (data engineering at 2am, model training by 4-6am, validation by 6-8am) are comfortably within the nightly batch window. This ensures by Wednesday the updated model is ready, satisfying the business timeline. We avoid true real-time streaming updates because the domain doesn’t require it (games happen weekly, not continuously), and doing full retraining allows thorough validation (ensuring we didn’t overreact to one week’s upset). This approach maintains **model stability**, avoiding oscillations that might occur if we updated on each game (which could inject noise).

In summary, the online learning plan is theoretically sound: the model will converge toward optimal with more data, warm-start gives faster convergence each week (with an eye on possible slight bias introduction which we monitor), and retraining on all data avoids catastrophic forgetting by design. Regular evaluation on a hold-out or backtesting on past weeks ensures that the incremental updates are *monotonic improving* or at least not degrading performance (if a drop is observed, it could indicate overfitting to recent data, at which point we might recalibrate learning rates or regularization).

5. Calibration Theory and Metrics

Accurate probability estimates are crucial in betting, which is why calibration is a focus. **Calibration** means that among all predictions assigned ~60% win probability, about 60% should actually win, and so on for all probability levels. We explore calibration methods and theory:

Platt vs. Isotonic vs. Beta Calibration: These are popular calibration techniques for binary classifiers:
- **Platt Scaling:** Fits a sigmoid (logistic) function to map raw scores to calibrated probabilities [17]. Originally used for SVMs [17], it assumes the model's log-odds output can be linearly adjusted: $p = \frac{1}{1+\exp(A \cdot s + B)}$. Two parameters A,B are learned (usually by minimizing log-loss on validation data). Platt scaling is parametric and tends to be stable even with limited data (low variance). However, it is **biased if the true calibration curve isn't sigmoidal** – for example, if the model's score distribution is skewed or the calibration function needs more flexibility [18] [19]. Notably, a logistic curve cannot represent a perfect identity mapping unless parameters are exactly $(A=1,B=0)$, which may be outside the allowed family if the model is already perfect [20]. But Platt is often a good compromise: low complexity and often effective if the model's miscalibration is roughly a sigmoid shape (common in many ML models).
- **Isotonic Regression:** A non-parametric calibration method that learns a **monotonic piecewise-constant function** mapping model outputs to probabilities [17]. The Pool-Adjacent-Violators (PAV) algorithm is used to ensure the mapping is non-decreasing (preserving rank). Isotonic is very flexible – it can fit any monotonic distortion. This power means it can achieve excellent calibration given enough calibration data. However, it can **overfit** when data for calibration is limited, because it can create too many steps (especially if some predicted probabilities are rare). For example, if only a few examples have raw score ~0.8 and all happened to win, isotonic might map 0.8 -> 1.0 probability, which might not hold true in general. Thus, isotonic tends to have lower bias but higher variance than Platt. In our case, with ~3000 games total, if we set aside a calibration set (say last season's games), isotonic should be feasible, but we must be cautious about not having too fine bins. Often, a technique is to **platt-scale or smooth the extremes** even when using isotonic (or enforce a minimum sample per bin).
- **Beta Calibration:** A newer method that extends Platt's idea by using a **Beta distribution** mapping [21]. Essentially, it assumes the uncalibrated probabilities are distributed as a Beta(α,β) when conditioned on true class, and derives a calibration function of the form $f(s) = \frac{s^\alpha (1-s)^\beta}{s^\alpha (1-s)^\beta + s^\gamma (1-s)^\delta}$ (with some parameterization). In practice, Beta calibration adds a third parameter compared to Platt, allowing asymmetric sigmoidal shapes and even identity mapping as a special case [19] [22]. It was shown to fix cases where Platt fails (e.g. Naive Bayes or Adaboost which produce skewed score distributions) [18] [22]. Beta calibration can capture **skewness** in the score–probability relationship. It's still a fairly low-dimensional model, so it won't overfit as easily as isotonic, yet is more flexible than Platt. For our ensemble, which might have mild distortions, Beta calibration could yield slight improvements. The tradeoff is complexity vs data: 3 parameters instead of 2 is usually fine with our dataset size. Empirical studies (Kull et al. 2017) found Beta calibration often outperforms Platt and isotonic in log-loss and calibration measures for many classifiers [23]. Beta includes the identity function as a possibility (unlike Platt) and thus won't *de-calibrate* a model that's already calibrated [19].

In summary, **Platt** is simple and stable (good for small data), **Isotonic** is very flexible (good if plenty of calibration data or if model is highly miscalibrated in non-sigmoid ways), and **Beta** offers a middle ground with a theoretically principled extension of Platt [21]. We might choose isotonic if maximizing calibration is critical and we trust our validation set size; or use Platt/Beta for a smoother calibration curve. Given the ensemble is quite good, a slight parametric calibration (Platt or Beta) might suffice to reach the needed <2% calibration error.

Proper Scoring Rules: A scoring rule is *proper* if the expected score is minimized when the predicted probability equals the true underlying probability. Proper scoring rules are critical because if we train or tune calibration with a proper score, we ensure honesty in probability estimates. **Log-loss** (cross entropy) and **Brier score** (MSE) are both *strictly proper*, meaning not only are they minimized at the truth, but any deviation increases the expected score [13]. This property guarantees that if our calibration optimization finds a mapping that reduces log-loss on a validation set, it is improving the probabilities towards truth. When we fit Platt scaling, we typically minimize log-loss (which is proper). When fitting isotonic, one can minimize squared error (Brier) which is also proper. Using proper scores prevents pathological solutions (e.g. a degenerate strategy like always predict 0 or 1 can't minimize a proper score unless that's the true distribution). The **Brier score decomposition** is worth noting: $\text{Brier} = \text{Calibration\ error} + \text{Refinement\ (uncertainty)}$. It splits into calibration term (how far predicted probabilities in each outcome group are from observed frequencies) and refinement (related to the inherent uncertainty – maximum when predictions are 50/50, lower when predictions are confidently near 0 or 1 and correct) [13]. A perfectly calibrated model has zero calibration term; its Brier score equals the uncertainty (which is minimal if it can be confidently correct often). This decomposition shows that improving calibration (reducing that term) will improve Brier score as long as we don't worsen discrimination.

Expected Calibration Error (ECE): This is a common metric to summarize calibration: partition predictions into bins (e.g. [0,0.1), [0.1,0.2), ..., [0.9,1.0]) and compute the difference between average predicted probability and actual frequency in each bin, then take a weighted average of these differences' magnitudes. Formally, $\text{ECE} = \sum_{k} \frac{n_k}{N} | \text{acc}(k) - \text{conf}(k)|$, where $\text{acc}(k)$ is empirical accuracy in bin k and $\text{conf}(k)$ is average prediction in bin k. **ECE is not a proper scoring rule** (it's just an evaluation metric) and not differentiable, so we don't optimize it directly, but it's handy for reporting. Theoretical analysis of ECE is tricky because of the binning. However, with enough data per bin, the *law of large numbers* ensures the observed frequency converges to true probability, so a well-calibrated model will have ECE $\to 0$ as $N \to \infty$. We can bound ECE with confidence intervals: for each bin, the difference $\text{acc}(k)-\text{conf}(k)$ has a sampling error ~ $\sqrt{\frac{\text{conf}(k)(1-\text{conf}(k))}{n_k}}$. Using a union bound or concentration inequalities, one can say with high probability all bins' errors are within some epsilon, thus ECE is bounded by epsilon. But such bounds are usually loose. Instead, practitioners sometimes use **smoothed calibration error** or reliability diagrams with error bars to visualize uncertainty in calibration.

Calibration Guarantees: If a model is trained with infinite data under log-loss, it will converge to the true conditional probabilities (hence perfectly calibrated). With finite data, there's always some calibration error. Techniques like isotonic can ensure *no worse calibration* than the raw model (isotonic specifically minimizes a loss, so it's optimal given the sample, though it might overfit small samples). Platt scaling, being parametric, might underfit the true calibration function if it's complex, but often yields a lower variance estimate of calibration. Beta calibration's inventors proved that it can correct certain systematic biases that logistic can't [19] [22].

In our architecture, we likely will use **isotonic calibration in Stage 2** (since it was explicitly mentioned). We must be careful to reserve a portion of data for calibration fitting only (to avoid circularity – you don't calibrate and evaluate on same data). Possibly use cross-validation or a rolling last-season as calibration set. After calibration, we evaluate the **Brier score**, **log-loss**, and **ECE** on a test set to ensure the model's probabilities are accurate. We target an ECE below say 0.05 (5%) or even 0.025 (2.5%) corresponding to the "minimum edge" threshold – i.e., if the model says 53% win probability, we want actual frequency to be close enough that we truly have ~2-3% edge, otherwise a miscalibration could trick us into a losing bet.

Calibration Example: Suppose before calibration, the model tends to predict 60% when the true win rate is 55%. Isotonic will learn to map 0.60 -> 0.55 (approximately). After isotonic, whenever the model internally thinks "0.60", it will output 0.55. This ensures we don't over-bet those cases. The cost is that we effectively shrink predictions towards 0.5 (often calibration yields less extreme probabilities). This might slightly hurt the perceived sharpness but improves actual decision-making. Proper scoring rules guarantee this trade-off is beneficial: even though some scores are nearer 0.5, the log-loss is better, meaning our probabilities are closer to truth, which is what matters for utility.

Finally, we also consider **calibration across different slices** (sometimes called multi-calibration or conditional calibration). For example, are our probabilities equally calibrated for home vs away teams? For underdogs vs favorites? If not, it could pay to calibrate separately or include those features in calibration. This is part of feature-dependent calibration discussed – if we find a certain subgroup consistently miscalibrated, we could apply a specialized correction or include an interaction in Platt scaling (e.g. separate sigmoid parameters for two groups). The theory of **group calibration** says you ideally want $P(Y=1 | \hat{p} =s, X \in G) = s$ for any group G – a much stronger condition. In practice, one aims for overall calibration and checks major groups for discrepancies.

6. Interpretability vs. Accuracy Trade-off

Designing a high-performance model that is also interpretable is challenging due to an inherent trade-off: **more complex models can fit data better but are harder to interpret**. There are theoretical limits and practical considerations:

Information-Theoretic Perspective: An interpretable model is often constrained in form – e.g. a linear model, a small decision tree, a rule list – which means it cannot capture arbitrary complex interactions or nonlinear patterns. From an information theory standpoint, this is like restricting the **capacity** or **entropy** of the model. The *Information Bottleneck* principle can be invoked: to be interpretable, we intentionally bottleneck the information about the input that the model retains [24]. Murphy and Bassett (2023) note that typical interpretable ML constrains feature interactions, effectively *sacrificing model complexity for comprehension* [24]. If the true relationship between features and outcome is complex, any simplified model will lose some predictive information – this is the cost in accuracy for interpretability. For example, a full ensemble might capture 100 bits of information about the outcome, whereas a sparse linear model might capture only 60 bits; the remaining 40 bits (perhaps nonlinear interactions or complex season-long trends) are dropped for simplicity, causing higher error.

However, it's not always a strict trade-off. There is a growing argument that the trade-off is sometimes overstated – with clever design, one can often get *interpretable models that are nearly as accurate as black-boxes*. For instance, generalized additive models with pairwise interactions (GA^2M) have yielded accuracy close to boosted trees while remaining relatively explainable. But generally, to reach the absolute highest accuracy, models like large ensembles or deep networks tend to win, and they are inherently harder to interpret. **No Free Lunch** in explainability: to explain more, you often must assume more structure (linearity, monotonicity, sparsity), which if untrue, hurts fit.

Interpretable by Design vs Post-hoc: Our approach uses complex models (trees, ensembles) for accuracy, but then applies **post-hoc interpretability** techniques (SHAP values, feature importance, example-based explanations) to explain decisions. This way we aim to get the best of both worlds. Yet, computing these explanations has a cost.

SHAP (Shapley Additive Explanations) Complexity: Shapley values come from cooperative game theory and assigning each feature a contribution to the prediction. The brute-force computation involves considering all 2^M subsets of M features – an exponential complexity (NP-hard for general models). Lundberg et al. (2017) introduced **TreeSHAP**, an algorithm leveraging tree structure to compute exact Shapley values in *polynomial time* [25]. Specifically, for an ensemble of decision trees, TreeSHAP complexity is $O(T * L * D^2)$, where T is number of trees, L is max leaves per tree, and D is max depth [25]. This is a huge improvement over exponential 2^M. For our model: suppose each booster has 100 trees, depth 6, and we have 3 boosters (300 trees total). If each tree has at most, say, 64 leaves (~depth 6), then computing SHAP for one instance is on the order of $300 * 64 * 6^2 \approx 300 * 64 * 36 \approx 691,200$ operations – easily under a few hundred milliseconds in C++. In practice, it's often faster due to sparsity and optimized C++ code. Indeed, TreeSHAP is implemented in optimized libraries and can often explain hundreds of instances per second. There's active research to *further accelerate* SHAP: recent improvements have gotten complexity down to $O(T * L * D)$ in some cases [26] and even GPU implementations for batch explanations [27]. So computing SHAP values for a single NFL game (which has ~50-150 features) well under the 500ms real-time requirement is feasible.

Real-Time Explanation Strategies: We will precompute as much as possible to meet the <500ms latency:
- We can precompute global constants for SHAP (like the expected value of the model, and some data structures from TreeSHAP algorithm) at model training time. Then, explaining a new game is just running the TreeSHAP recursion, which is fast.
- We might restrict to **top-K features** in the output. While SHAP yields all feature contributions, showing the *Top 3 factors* as required can be done by computing all SHAP values then taking the 3 highest (absolute) contributions. This doesn't reduce computation much (still need all values to find top 3) unless we approximate. However, if needed, one could train a simpler surrogate model for explanation that is optimized for the local region. But since TreeSHAP is exact for trees, we prefer to use it.
- Another trick: **memoization for similar inputs**. If two games are very similar in feature values (e.g., same teams, similar stats), their SHAP explanations will be close. We could cache explanations for common patterns (like common opponent pairings) to speed up, though with only 16 games a week, this is probably unnecessary – just compute on the fly.
- Use of example-based explanation: the system will provide "3 similar historical games". Finding similar games (via a distance in feature space) can be done quickly by indexing past games by, say, team matchup, or using clustering. This doesn't rely on SHAP at all and can be pre-done (for each upcoming game, retrieve similar past games from database). This fulfills part of real-time explainability without any heavy compute.

SHAP vs Other Methods: SHAP has the advantage of theoretical consistency (Shapley axioms ensure fairness, additivity, etc.). But computing it exactly for arbitrary models is NP-hard. Our tree-based model allows polynomial-time SHAP, which is a big reason tree ensembles are appealing in high-stakes settings: they are more interpretable than an equally-accurate neural network because of tools like TreeSHAP. If we had a neural net, we might have to rely on sampling-based approximations (Kernel SHAP) which could be slower or less precise. By sticking to tree ensembles, we ensure *efficient explainability*.

Interpretability Complexity: There is also the aspect of how complex the explanation itself is. We can generate a lot of numbers (one SHAP value per feature), but that's not user-friendly. We need to present concise explanations (e.g., *"Team A's strong offense (feature: yards/game = 400, contributing +5% to win probability) and Team B's injured quarterback (feature: QB Elo drop, contributing +3%) are key reasons for the high confidence."*). Generating these sentences from raw SHAP requires selecting important features and mapping them to human language. We will likely define templates for each feature (like if "Top feature is offensive yards” then say something about it). This is outside model theory but important for the application.

Sufficient Statistics for Decisions: From a decision standpoint, once the model outputs a final probability $ \hat{p}*{cal}$, the decision to bet or not is straightforward: bet if $\hat{p}}* > 0.525$ (for -110 odds and some margin) otherwise don’t. In a sense, $\hat{p*{cal}$ together with the odds is a sufficient statistic for the betting decision. All the complex features, ensemble calculations, etc., boil down to this one number (the probability of cover) which is then compared to the implicit breakeven probability (~0.523). So purely for making the bet, nothing else is needed – this is optimal decision theory given the model’s output. However, for interpretation and trust, we expose more: why is $\hat{p}*$ high or low? That’s where the top features and similar game analogies come in. We can say, *“Model gives 55% win probability because of factors X, Y, Z”*, which provides justification. The **decision threshold** (like 52.5%) also can be interpreted: it’s derived from the business requirement of minimum edge and Kelly fraction. If $\hat{p}$ is barely above 0.525, our system might even choose not to bet if confidence is marginal (the “confidence filter” mentioned likely incorporates uncertainty – e.g., if model is not very confident due to wide prediction intervals, maybe skip borderline bets).

Interpretability vs Latency: Ensuring real-time explanation (<500ms) requires computational measures (as discussed, TreeSHAP optimization) and also choosing the *scope* of interpretability. Full global interpretability (like explaining model logic entirely) is not needed in real-time, only the local explanation for the current game. Generating a full SHAP explanation for all 100 features would be overkill and slow to render; instead focusing on top 3 features drastically reduces what we need to output (the computation difference isn’t large, but filtering and formatting is manageable). If further speed was needed, one could use an approximation: e.g., *SHAP sampling*, where instead of summing over all paths in the tree, sample some paths to estimate contributions. But given our constraints, exact TreeSHAP should suffice.

Complexity of Interpretability Methods: Other interpretability methods like LIME (local surrogate linear models) could be used – LIME would sample perturbed inputs around the game and fit a small linear model to approximate the prediction. This might be slower given the need to query the model multiple times. Also, SHAP has the advantage of using the model’s internal structure. Another approach is **rule-based explanations**: e.g., extract a small decision rule that approximates the model’s decision for this game (like: “because TeamA offense > X and TeamB defense < Y, model favored TeamA to cover”). There are techniques for rule extraction from trees, since an ensemble of trees can be distilled to a set of conditions, but it’s complex to do exactly. Instead, listing similar past games provides an intuitive interpretation: “in similar past situations, the favorite covered 5 out of 5 times” – this leverages human pattern recognition and satisfies regulatory needs for transparency.

Interpretability-Accuracy Boundaries: Research by Cynthia Rudin and others argues that for many problems, an interpretable model (like a scoring system or rule list) can achieve accuracy close to black-box models. One reason is that data may not actually support the full complexity of a black-box – so a simpler model can capture what's reliably learnable. But in our case, with 50-150 features and possible nonlinear interactions (e.g., a team’s performance might depend on a combination of weather and quarterback health and opponent’s defense ranking), a simple linear or low-depth tree might miss subtle effects. For example, an ensemble might learn a *nonlinear interaction*: if wind > 20 mph, passing yards feature importance changes drastically. A linear model cannot adapt its coefficients based on such context. So we accept complexity for accuracy, but ensure post-hoc interpretability.

SHAP and Additivity: SHAP values are additive: sum of all feature SHAP values = prediction minus baseline. This means we can **explain the exact prediction** as a sum of effects. This satisfies a regulatory interpretability requirement: one can trace how each input contributed. The cost is computing these values. But since TreeSHAP gave us polynomial complexity, we essentially reduced an exponentially hard explanation problem to a tractable one [25]. There are cases where even tree models can be large (many trees, high depth) making SHAP slower, but our model sizes are modest (depth ~6, maybe hundreds of trees). Also, note SHAP computation can be **parallelized** across features and trees, and even on GPU [27], so scaling to meet latency is possible if needed.

Conclusion on Interpretability: We plan a **two-pronged interpretability approach**: (1) *Real-time local explanations* for each bet decision (fast SHAP for top features, plus fetching similar examples) delivered in under 0.5s, and (2) *Offline global interpretability* analysis each week (full SHAP summary plots, feature importance rankings, error analysis on where model went wrong). The latter can be more computationally intensive (running SHAP on all 3000 games to see patterns), but that can be done in the background since we have from Tuesday to Wednesday to produce reports. This satisfies both immediate transparency needs and deeper audit requirements. The **accuracy cost** of doing this is negligible because we did not have to simplify the model to explain it – we used a powerful model and applied powerful explainers. The only caution is to ensure our explanations remain faithful; fortunately, SHAP is exact for the model, and similar-game retrieval is factual. There might be extremely complex interactions that even SHAP values struggle to attribute cleanly (they'll distribute credit among features), which could be harder for a human to interpret if many features each contribute a bit. But by focusing on the top few, we highlight the dominant factors, which usually suffice in an NFL context (e.g., "starting QB out" or "huge offense vs weak defense" are often dominant reasons).

Finally, on a theoretical note, one could ask: *is there a formal limit to how interpretable a model of a given accuracy can be?* This is an open research question, but some work attempts to quantify it. For instance, if we define interpretability as a constraint on model complexity (like number of features used or depth of tree), we can sometimes derive that beyond a certain complexity bound, the model cannot fit the data distribution beyond an error level. For example, **VC dimension** theory: a simpler model class has lower VC dimension, so if the true function is outside that class, it incurs approximation error. Achieving zero approximation error might require a model class so complex that it's no longer human-comprehensible. In our case, using an ensemble (high VC dimension) achieves low error, but explaining it means distilling knowledge in simpler terms – which inevitably is an approximation. We mitigate this by focusing on the decisions (local explanations), where the model's complex logic is broken into a few feature contributions for that instance, making it understandable without requiring understanding the entire model globally.

Computational Complexity of Generating Explanations: For each game: computing top-3 SHAP is O(300k) operations as estimated, retrieving similar games is O(1) if indexed by keys (or O(N) if linear search through 3000 – trivial), and formatting the output is trivial. So the pipeline is well within 100ms, leaving ample slack under 500ms. If we had to scale to many games concurrently (say explaining every game of a season at once), we could parallelize across games easily (each explanation independent). Memory-wise, storing the model and data is fine, and computing SHAP uses some stack memory proportional to depth.

SHAP Complexity in Worst-case: Just to note, if we had M=150 features and very deep trees, TreeSHAP complexity $O(T * L * D^2)$ could be high. But we constrain tree depth to maybe 6-8 because deeper trees risk overfit on 3000 samples. So D is small. L (leaves) might be up to $2^D=64$ or so per tree. T could be a few hundred at most. These numbers keep TreeSHAP feasible. Actually, research has shown even

computing Shapley values for simple linear models can be done in polynomial time (and for linear, it's directly each coefficient times feature value difference). The complexity become prohibitive only if we had to consider every feature subset for a general model. Trees circumvent that by a clever dynamic programming on tree structure [25].

In closing, **interpretability and accuracy need not be zero-sum** if we invest in tools and design architecture accordingly. By using tree ensembles, we hit a sweet spot: high accuracy on tabular data (a known strength of ensembles) and **exact, fast post-hoc explanations** thanks to TreeSHAP. The result is a model that meets performance requirements (financial accuracy) and regulatory/operational requirements (transparency, explanations under 0.5s, compliance with any audit). Our architecture explicitly allocates resources to both: a bit of extra complexity in coding and computing SHAP, but well worth the insight gained.

References (ML & Stats Literature)

- Hansen, L. & Salamon, P. (1990). *Neural network ensembles* (error reduction by independent models) [1].
- Krogh, A. & Vedelsby, J. (1995). *Neural network ensembles, cross validation, and active learning* (accuracy-diversity decomposition) [3].
- Wolpert, D. (1992). *Stacked Generalization* (meta-learning to minimize generalization error).
- van der Laan, M., Polley, E., & Hubbard, A. (2007). *Super Learner* (stacking optimality proof, oracle inequality) [9].
- Breiman, L. (1996). *Bias, variance, and arcing classifiers* (ensembles reduce variance; boosting can reduce bias) [2].
- Caruana, R., Niculescu-Mizil, A. (2005). *Predicting Good Probabilities* (comparison of calibration methods; isotonic vs Platt, effect on boosted trees) [12].
- Kull, M. et al. (2017). *Beta Calibration* (Beta distribution calibration improving on Platt) [19] [22].
- Niculescu-Mizil, A. & Caruana, R. (2005). *Obtaining Calibrated Probabilities* (isotonic regression for boosting and SVM).
- Robbins, H. & Monro, S. (1951). *A stochastic approximation method* (SGD convergence theory).
- Bottou, L. (2010). *Large-Scale Machine Learning with SGD* (convergence rates and practice) [14].
- Kirkpatrick, J. et al. (2017). *Overcoming catastrophic forgetting in neural networks* (EWC regularization) [16].
- Guo, C. et al. (2017). *On Calibration of Modern Neural Networks* (ECE, reliability diagrams in practice).
- Lundberg, S. & Lee, S. (2017). *A Unified Approach to Interpreting Model Predictions* (introduces SHAP).
- Lundberg, S. et al. (2020). *TreeSHAP: Model Interpretation for Decision Trees* (TreeSHAP polynomial algorithm) [25].
- Murphy, K. & Bassett, D. (2023). *Interpretability with full complexity by constraining feature information* (information-theoretic view of interpretability) [24].

---

[1] [3] [4] [5] [8] 2. Classifier Ensembles
https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume11/opitz99a-html/node2.html

[2] 2.3 The Bias plus Variance Decomposition
https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume11/opitz99a-html/node5.html

[6] [7] arxiv.org
https://arxiv.org/pdf/1709.02925

[9] [10] [11] Modeling the Exposure and the Outcome: Introduction to the Super Learner
https://statisticalhorizons.com/wp-content/uploads/2022/04/MLCI-Sample-Materials-2.pdf

[12] [17] ams.confex.com
https://ams.confex.com/ams/pdfpapers/88928.pdf

[13] [18] [19] [20] [21] [22] Beta calibration: a well-founded and easily implemented improvement on logistic calibration for binary classifiers
https://proceedings.mlr.press/v54/kull17a.html

[14] [PDF] Characterization of Convex Objective Functions and Optimal ...
http://proceedings.mlr.press/v97/van-dijk19a/van-dijk19a.pdf

[15] [16] What is Catastrophic Forgetting? | IBM
https://www.ibm.com/think/topics/catastrophic-forgetting

[23] How to obtain well-calibrated probabilities from binary classifiers ...
https://projecteuclid.org/journals/electronic-journal-of-statistics/volume-11/issue-2/Beyond-sigmoids--How-to-obtain-well-calibrated-probabilities-from/10.1214/17-EJS1338SI.pdf

[24] Interpretability with full complexity by constraining feature information | OpenReview
https://openreview.net/forum?id=R_OL5mLhsv

[25] [26] treeshap — explain tree-based models with SHAP values | by Konrad Komisarczyk | ResponsibleML | Medium
https://medium.com/responsibleml/treeshap-explain-tree-based-models-with-shap-values-2900f95f426

[27] GPUTreeShap: massively parallel exact calculation of SHAP scores ...
https://pmc.ncbi.nlm.nih.gov/articles/PMC9044362/