Cohort-Level Feature Set

To predict how an entire cohort of users will behave over the first year, I distilled every cohort into a small set of statistical summaries that capture both central tendency and heterogeneity.

At the heart is **cohort_size**: the count of users who made their first event on a given date. A larger cohort tends to produce more stable aggregate metrics; smaller cohorts can exhibit high variance purely by chance.

Because platform choice drives both engagement and monetization patterns, I include the **percentage of iOS** and **iPadOS** users. These two columns (the result of pivoting the "operating_system" distribution) tell us whether the cohort skews mobile (iOS) or tablet (iPadOS), which can correlate with purchase friction or session length.

Next come a family of **"avg"** and **"std"** features for each user-level behavior, both averaged across the cohort:

- **auto_renew_off** (how often users disable auto-renew),
- **free_trial** starts,
- **paywall** encounters,
- **refund** requests,
- **renewal** events, and
- **subscribe** events.

The *mean* of each behavior gauges overall cohort engagement or churn propensity, while the *standard deviation* captures dispersion—high variability may signal a mixed user base (some highly active, others barely active), which often complicates forecasting.

I also summarize temporal patterns via **mean_event_hour** and **std_event_hour**: the average and spread of the hour-of-day when cohort members act. A cohort whose activity is tightly clustered (low std) around business hours will likely convert differently than one spread evenly through nights and weekends.

Finally, to ground our model in past performance, I include the cohort's **mean_revenue_1y** and **std_revenue_1y**: the cohort average of the historical first-year revenue and its user-level variability accross the cohort. Although these correlate strongly with future revenue, by combining them with behavioral and platform features, the model can learn when cohorts with similar past returns diverge (e.g., high-variance cohorts may be riskier).

Seasonality enters via **season_vec**, a one-hot encoding of the quarter or "season" in which the cohort began. This captures macro trends—holiday sign-ups, summer lulls, end-of-year push—that aren't evident in day-of-week or hour-of-day alone.

Together, these 20 features give a compact, yet multi-dimensional view of each cohort—size, platform mix, engagement patterns, revenue history, and season—so that the gradient-boosted tree can learn which combinations predict strong versus weak cohort performance.

User-Level Feature Set

To forecast individual user revenue, I distilled each user's first 15 days of data into a concise vector that balances raw counts, timing patterns, and categorical context.

I started with **raw event counts**:

- **auto_renew_off**, **free_trial**, **paywall**, **refund**, **renewal**, and **subscribe**. These counts directly measure how often a user turned off renewal (a churn signal), used a free trial (a trial-to-paid

friction proxy), hit a paywall, asked for refunds, renewed subscription, or subscribed anew. A user with many paywall hits but few subscriptions may be price-sensitive; many renewals suggest stickiness.

I capture time-of-day behavior with **avg_event_hour**, the average hour when this user was active. Early-morning versus late-night users often show different engagement rhythms, which can affect purchase timing and the slope of future revenue curves.

Monetary signals appear in **total_revenue** (revenue generated in the 15-day window) and **first_year_revenue** (the known first-year revenue). While total_revenue over 15 days is the immediate "hook", having the true first-year figure on hand lets us calibrate the model during training—learning how short-term revenue maps onto the full year.

I include **stickiness_ratio** (days active ÷ days observed) as a normalized retention metric. Two users might both have three renewal events, but one who was active 14 out of 15 days is demonstrably more engaged than one who only logged in twice.

Finally, three one-hot vectors encode categorical context:

- **season_vec** (user's *unique* cohort season),
- **country_vec**, and
- **operating_system_vec**.

These let the model adjust for macro factors (e.g. summer holiday vs. Christmas during winter), geographic differences (some markets convert faster), and platform effects (iPadOS vs. iOS pricing).

By combining raw counts, timing summaries, normalized retention, early revenue signals, and categorical context into a 13-dimensional feature vector, I give the tree-based learner a comprehensive mix of behavioral, temporal, financial, and demographic inputs to forecast each user's 1-year spending on the application.

ANSWERS TO OPEN ENDED QUESTİONS

1st Question:

Requiring that every training record carry a full twelve-month revenue history meant that, by design, our model could only learn from cohorts and users whose "year-one" labels were already complete. This introduced two key difficulties. First, it created a **recency gap**: any shifts in user behavior or in the product (new features, pricing changes, marketing campaigns) that happened in the past year simply never appeared in our training set, leaving the model vulnerable to **concept drift** when it saw brand-new cohorts. Second, it **shrank and aged our sample**: by dropping all users who joined *in the last 364 days*, we lost both volume (fewer examples) and coverage (fewer recent seasons, OS versions or country mixes), which can degrade generalization and amplify biases towards "legacy" patterns.

I tried to overcome these constraints with three complementary strategies.

1. **Explicit time-and-segment features**
   By one-hot encoding season, country and operating system, the model could internally adjust for macro trends—holidays versus summer lulls, emerging markets versus mature ones, iPadOS vs. iOS—that otherwise lived only in recent data.
2. **Proxy training on short-horizon labels**
   Rather than wait a full year for every new user, my **user-level pipeline** was trained on the 15-day revenue I did have. At scoring time I simply multiply the 15-day forecast by 365/15 to produce an annualized LTV. This lets us learn from the freshest behaviors the moment users exceed 15 days of history, cutting the

label-delay from 365 days down to 15. The obvious assumption is that these 15 days are sufficiently representative of the whole year, which is arguable.

3. **Robust, time-aware validation and regular retraining**
   I always hold out the **next contiguous time window** for testing (never mixing "future" cohorts into training), so my performance estimates mirror real-world rollout. Further, at a production environment, as soon as another month's worth of cohorts matures to 1 year, I would retrain—keeping the model's "view" from trailing reality by more than a few weeks.

Together, these steps close the recency gap, bolster sample size with high-quality proxy labels, and ensure the LTV forecasts stay both accurate and up-to-date.

2nd Question:

Our cohort and user pipelines deliberately distilled raw behavior into two modestly sized vectors—on the order of a dozen to a few dozen features each—so I never confronted a "wide" feature matrix of hundreds or thousands of columns. Each attribute was hand-picked for its business meaning (e.g. counts of renewal events, mean event hour, one-hot encoded seasons), which kept dimensionality low and interpretability high. Because of that design, I relied on straightforward validation: sanity-checking distributions and summary statistics after preprocessing, confirming schemas and row counts, and inspecting cross-validated performance to ensure no feature was introducing noise or leakage.

In settings where true high-dimensionality arises (e.g., from text embeddings, hundreds of one-hot categories, or thousands of automated feature-generation), you'd need more rigorous vetting. First, integrity checks at the pipeline level ensure each column is present, non-null where expected, and within plausible ranges. Next, exploratory analysis of variance and pairwise correlation can surface flat or collinear features, which add little or even degrade model fit. Beyond univariate filters, model-based selection methods such as L1 regularization or tree-ensemble feature importances help prune irrelevant dimensions. Finally, thorough cross-validation with separate hold-out windows guards against overfitting: one tracks how performance changes when subsets of features are removed or when dimensionality reduction (for example, via PCA) is applied. Only by combining data-quality checks, statistical filtering, and empirical model evaluation can one be confident that a high-dimensional feature set is both clean and genuinely beneficial.

3rd Question:

I did indeed run multiple "experiments" in the sense that for both our cohort and user pipelines I swept over a grid of gradient-boosted-tree hyperparameters: various max depth, iteration count and step size. Then, I evaluated each candidate with cross-validation so that the system could automatically pick the best combination. In practice (i.e., in the Interactive Python Notebooks): define a ParamGridBuilder in Spark, wrap it in a CrossValidator configured to optimize on validation MAE, and then call fit() on the training splits. By design, Spark's CrossValidator carried out all of the train-validate cycles, tracked the metrics for each hyperparameter combination, and surfaced the bestModel for downstream scoring.

Beyond hyperparameter sweeps, however, I did not use a dedicated experiment-tracking framework (for example MLflow or TensorBoard) to catalog dozens of runs, log artifacts, or compare variants side by side in a dashboard. Instead, my notebooks served as the primary record: each time I ran a new parameter grid or toggled a feature, I captured the resulting metrics in markdown cells immediately following the execution. In that way every experiment had its own code block, its grid definition, and its printed evaluation on the hold-out.

This approach gave quick visibility—when I reran the notebook I could scroll to the latest results and see which hyperparameters or feature combinations improved our validation scores—but it meant trade-offs. Without a persistent experiment store I couldn't easily roll back to an intermediary model version or

compare run metadata in bulk. To manage that risk I relied on two conventions: I always tagged my best performing model with a clear name (i.e., "bestModel" in the code) before persisting, and I committed each notebook revision to Git right after tuning. That way the exact code, grid settings and evaluation outputs remained versioned alongside our source.

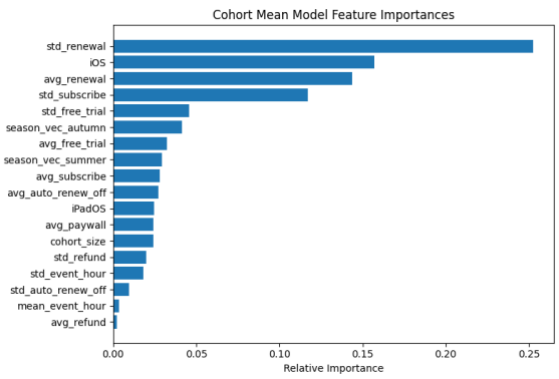
In a more mature production setting I would augment this manual process with an automated tracker. Logging every hyperparameter set, feature list and metric to MLflow would let us compare dozens of runs without cluttering the notebook. I could register my best models by stage (Staging, Production) and automatically deploy only after passing a *promotion gate* (for example, improved test-set MAPE and *non-regression* on key segments). Nevertheless, even in its notebook-centric form our practice of clear code sections, grid-based cross-validation and consistent Git commits gave us a systematic, reproducible path to select and promote the strongest model.

4 Model Evaluation and Interpretation

4.1 Cohort-Level Models

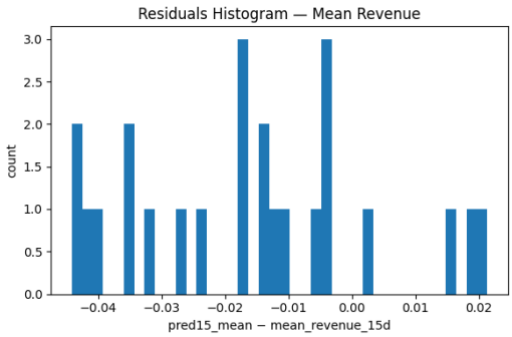
4.1.1 Mean-Revenue Predictor

Figure 4.1. Cohort Mean Model Feature Importances



The dominant drivers of 15-day mean revenue at the cohort level are (1) the standard deviation of renewals, (2) share of users on iOS, and (3) average renewal rate. Together these three account for over 50% of the model's relative importance.

Figure 4.2. Residuals Histogram – Mean Revenue

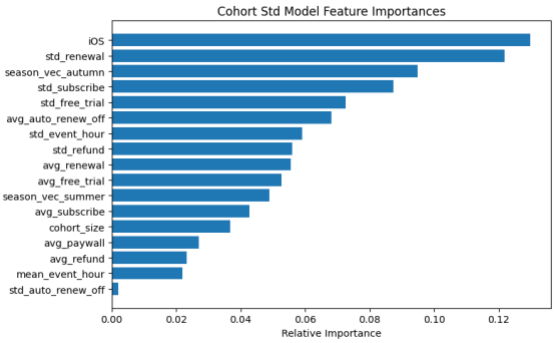


Residuals (predicted – actual) show a slight negative bias (mean ≈ –0.0159), with a standard deviation of ≈ 0.0191 and a median of ≈ –0.0167. Errors are tightly clustered within ±0.03, indicating low bias and high precision in mean-revenue predictions.

Statistic	Value
Mean residual	–0.0159
Median residual	–0.0167
Std. dev. residual	0.0191

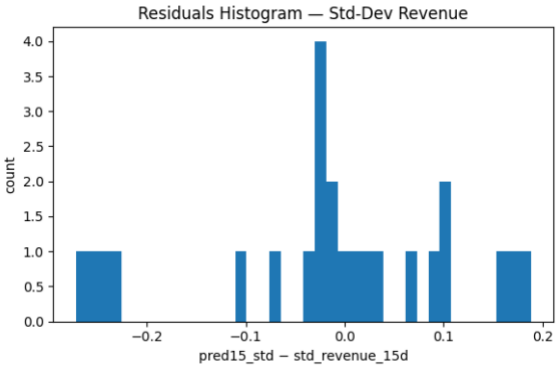
4.1.2 Std-Dev-Revenue Predictor

Figure 4.3. Cohort Std Model Feature Importances



Predicting within-cohort revenue volatility relies most on (1) iOS share, (2) renewal variability, and (3) autumn cohort seasonality—together about 35% of importance.

Figure 4.4. Residuals Histogram – Std-Dev Revenue

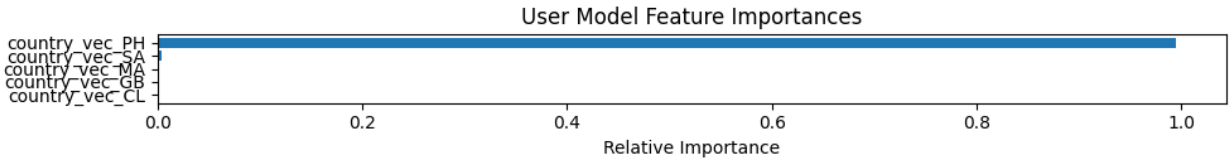


These residuals also have a small negative bias (mean ≈ –0.0163; median ≈ –0.0154) but exhibit a wider spread (σ ≈ 0.1313), reflecting greater heteroskedasticity when modeling cohort volatility.

Statistic	Value
Mean residual	–0.0163
Median residual	–0.0154
Std. dev. residual	0.1313

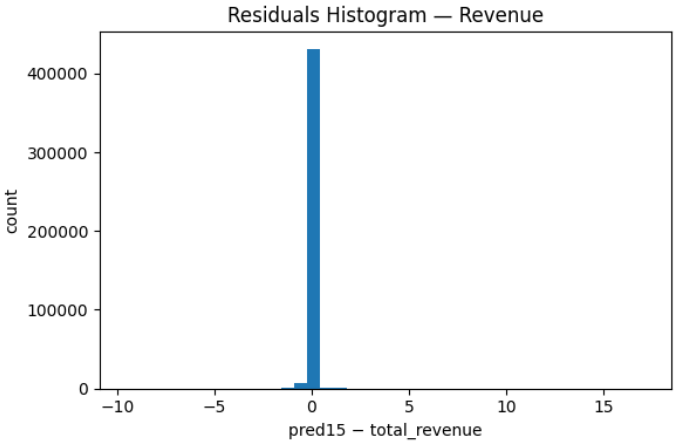
4.2 User-Level Model

Figure 4.5. User Model Feature Importances



The per-user spend model is overwhelmingly driven by the Philippines country vector (nearly 100% of importance), with all other country vectors collectively contributing the remaining ~0%.

Figure 4.6. Residuals Histogram – Revenue



User-level residuals are essentially unbiased (mean ≈ –0.00134; median ≈ –0.00292) but have a heavy tail (σ ≈ 0.109), indicating that while most users are predicted very accurately, a small subset incur large errors.

Statistic	Value
Mean residual	–0.00134
Median residual	–0.00292
Std. dev. residual	0.1090