- 1 Problem formulation
  - Fraud Detection: Real-time Anomaly Detection
    - Key Performance Indicators (KPIs)
    - Core Challenges
  - Plan Ranking: Personalized Recommendation
    - Key Performance Indicators (KPIs)
    - Core Challenges
  - 2 Data-preparation plan
    - 2.1 Raw Data "Mock Sheet"
      - Fraud Detection Data
        - payments_raw
        - customer_profile (Fraud-related fields)
      - Plan Ranking Data
        - customer_profile (Insurance-related fields)
        - plan_catalogue
        - quote_logs
    - 2.2 ETL / Feature Store Pipeline
      - Fraud Detection Features
      - Plan Ranking Features
      - The Core Problem: Train-Serve Skew
      - How a Feature Store Solves This
  - 3 Modelling strategy
    - 3.1 Fraud Detection
      - Real-time serving path
    - 3.2 Plan Ranking
  - 4 Evaluation & testing
    - Fraud Model Evaluation
      - Offline Evaluation
      - Online Evaluation
    - Ranking Model Evaluation
      - Offline Evaluation
      - Online Evaluation
    - Automated Model Quality & Safety Gates
      - Offline Pre-deployment Gates
      - Online Post-deployment Gates
  - 5 GenAI / LLM Integration Ideas
  - 6 Bonus: Code from implementation notebooks

- Fraud Detection: Feature Engineering (Pandas)
- Fraud Detection: Cost-Sensitive Model Training (XGBoost)
- Fraud Detection: Unsupervised Anomaly Detection (Isolation Forest)
- Plan Ranking: Learning-to-Rank Model Training (LightGBM)
- For Production Scale: Feature Engineering (PySpark)

1 Problem formulation

Fraud Detection: Real-time Anomaly Detection

The Business Problem: We need to build a system that can analyze incoming payment transactions and accurately identify which ones are fraudulent. The primary objective is to block these fraudulent payments before they are processed, thereby preventing financial loss. A crucial constraint is to achieve this with minimal disruption to legitimate customers, meaning we must keep the rate of incorrectly flagged transactions (false positives) as low as possible.

Key Performance Indicators (KPIs)

- **Area Under the Precision-Recall Curve (AUPRC):** Given the severe class imbalance (very few fraudulent transactions), AUPRC is a more informative metric than simple accuracy or AUROC. It measures the trade-off between precision and recall for the positive (fraud) class.
- **Cost-weighted F_1 Score:** A custom metric that assigns different costs to false positives (blocking a real customer) and false negatives (letting fraud through). This directly aligns the model's performance with business impact, often expressed as "net dollars saved per 1,000 transactions."
- **Mean Time to Detect (MTTD):** Measures how quickly the system can identify and adapt to new types of fraudulent activity.

Core Challenges

- **Extreme Class Imbalance:** Fraudulent events are rare, often accounting for less than 0.1% of all transactions. This makes it difficult for a model to learn their patterns without being overwhelmed by the majority class.

- **Concept Drift:** Fraudsters constantly change their tactics (Methods of Operation or "MOs"). A model trained on historical data may quickly become outdated. The system must be able to adapt to these evolving patterns.
- **Real-time Latency Constraints:** To be effective, the fraud check must happen in real-time within the payment processing flow. This imposes a strict latency budget, typically under 150 milliseconds for the entire feature lookup and scoring process.

Plan Ranking: Personalized Recommendation

The Business Problem: When a customer requests a health insurance quote, we present them with a list of available plans. The goal is to rank this list in a way that maximizes the likelihood of the customer purchasing a plan they are satisfied with. A successful ranking system will increase conversion rates and customer lifetime value.

This is a classic **learning-to-rank** problem. For a given query (the customer's profile), we need to sort a list of documents (the insurance plans) to optimize for a specific outcome (purchase and satisfaction).

Key Performance Indicators (KPIs)

- **Ranking Quality Metrics:**
  - **NDCG@K (Normalized Discounted Cumulative Gain):** The primary offline metric, as it handles graded relevance signals (e.g., `purchase = 5`, `click = 1`, `no interaction = 0`) and rewards placing better plans higher up the list.
  - **MAP@K (Mean Average Precision) & MRR (Mean Reciprocal Rank):** Excellent secondary metrics that are perfect for scenarios where we simplify relevance to a binary signal (`relevant` vs. `non-relevant`). MRR is particularly useful for measuring how high the first relevant plan is ranked.
- **Business Impact Metrics:**
  - **Conversion Uplift:** The key online business outcome, measured via A/B testing. We must show a statistically significant lift in the quote-to-purchase rate.
  - **Post-purchase Value:** Long-term success is evaluated by tracking customer **Lifetime Value (LTV)**, churn rate, and the frequency of support calls post-purchase.

Core Challenges

- **Defining Plan "Relevance":** The relevance or score of a policy is not absolute; it is defined by implicit user feedback. A purchase is a strong positive signal, and a click is a weaker one. Formulating a robust mapping from these user actions to a numeric score for model training is a key challenge.
- **Heterogeneous Features & Privacy:** The system needs to effectively combine diverse data types: customer demographics (age, location), lifestyle data, sensitive medical history, and detailed plan attributes (premiums, coverage vectors). All of this must be handled with strict adherence to data privacy regulations.
- **The Cold-Start Problem:** The system will struggle to provide personalized rankings for two key groups: new customers with no prior interaction history, and newly introduced insurance plans that have not yet accumulated any performance data.

---

2 Data-preparation plan

2.1 Raw Data "Mock Sheet"

For the two problems, we would source data from the following key tables:

Fraud Detection Data

`payments_raw`

- **Volume / Cadence:** 2 M rows / day, via stream
- **Source:** Internal payment processing logs, generated in real-time as customers transact via payment gateways (e.g., Stripe, Adyen).
- `txn_id` (string): Unique transaction identifier.
- `cust_id` (string): Unique customer identifier.
- `amount` (float): Transaction value in a standard currency.
- `timestamp` (datetime): Time of the transaction.
- `geo_ip` (string): IP address of the device used for the transaction.
- `device_fingerprint` (string): Hash representing the user's device and browser combination.
- `mcc_code` (integer): Merchant Category Code, indicating the type of business (e.g., restaurant, airline).
- `label_is_fraud` (boolean): The ground truth label, available post-facto after investigation.

`customer_profile` **(Fraud-related fields)**

- **Volume / Cadence:** 6 M active customers, daily snapshot
- **Source:** A combination of the internal CRM database and data enriched from external credit bureaus (e.g., Experian, Equifax).
- `cust_id` (string): Unique customer identifier, links to `payments_raw`.
- `account_age_days` (integer): How long the customer has been with the company. A very new account can be a risk signal.
- `credit_score` (integer): A numerical score representing the customer's creditworthiness.
- `avg_txn_amount` (float): The customer's average transaction amount over their lifetime.
- `country_of_residence` (string): The customer's registered country.

Plan Ranking Data

`customer_profile` **(Insurance-related fields)**

- **Volume / Cadence:** 6 M active customers, daily snapshot
- **Source:** User-provided information stored in the internal CRM system, collected during account creation or the quoting process.
- `cust_id` (string): Unique customer identifier, links to `quote_logs`.
- `age` (integer): Customer's age.
- `gender` (string): Customer's self-reported gender.
- `postcode` (string): Customer's postal code, for geo-specific eligibility.
- `occupation` (string): The customer's job role, which can be mapped to a risk category.
- `family_structure` (object): Details on dependents (e.g., `{ "marital_status": "married", "num_children": 2 }`).
- `lifestyle_factors` (object): Self-reported habits, (e.g., `{ "smoker": false, "activity_level": "active" }`).
- `medical_history` (object): Anonymized summary of past health events (e.g., `{ "previous_diagnoses": ["asthma"], "surgeries": ["appendectomy"] }`). This is highly sensitive data (PHI) and requires strict handling.

`plan_catalogue`

- **Volume / Cadence:** ~150 plans, weekly refresh

- **Source:** Direct data feeds from partner insurance companies, or via a specialized health plan data aggregator (e.g., Vericred).
- `plan_id` (string): Unique plan identifier.
- `premium_monthly` (float): The monthly cost of the plan.
- `coverage_vector` (array[float]): An embedding representing the plan's coverage details across various benefit categories.
- `deductible` (float): The out-of-pocket amount the customer must pay before the insurance plan starts to pay.
- `insurer_rating` (float): A rating of the insurer's reputation (e.g., 1-5).
- `available_postcodes` (array[string]): List of postcodes where this plan is offered.

`quote_logs`

- **Volume / Cadence:** 200 k quotes / day, via stream
- **Source:** First-party user interaction logs captured from the company's frontend application (website or mobile app).
- `session_id` (string): Identifier for a user's browsing session, grouping multiple quote views.
- `quote_id` (string): Unique identifier for a specific quote request event.
- `cust_id` (string): Unique customer identifier.
- `plan_id` (string): The identifier of the plan being shown.
- `timestamp` (datetime): Time the quote was shown.
- `rank_shown` (integer): The position on the list where the plan was shown (1, 2, 3...).
- `quoted_premium` (float): The specific premium calculated for this user and quote, which may differ from the base plan premium.
- `has_existing_policy` (boolean): Whether the user indicated they already have an insurance policy.
- `click_flag` (boolean): True if the customer clicked on this plan for more details.
- `purchase_flag` (boolean): True if the customer ultimately purchased this plan.

2.2 ETL / Feature Store Pipeline

1. **Ingestion & Staging:**
    - **Multi-source Ingestion:** Data arrives from diverse sources and formats. We'll need connectors for Kafka streams (`payments_raw`, `quote_logs`), SFTP drops for batch files (CSVs, Excel files from partners), and REST APIs (for

fetching data from sources like `plan_catalogue` aggregators which might use JSON or XML).

- **Raw Data Lake:** All incoming data is first landed in its raw format into a staging area in a data lake (e.g., AWS S3, Google Cloud Storage). This provides a durable, auditable source of truth.
- **Initial Structuring:** From the data lake, initial processing jobs parse the various formats and load the data into a more structured, but still raw, format (e.g., Parquet tables in the lake). This is where we decide on the best storage: non-relational/columnar stores (like Mongo or Elasticsearch) are ideal for high-volume event logs, while relational databases (like PostgreSQL) are well-suited for customer profile and catalogue data.

2. **Cleaning & Normalisation:**

- **Schema Enforcement & Type Consistency:** Apply strict schemas to all datasets. This includes casting fields to their correct data types (e.g., ensuring all monetary values are `float`, IDs are `string`). A critical step is standardizing inconsistent formats, such as converting all date strings (`MM/DD/YYYY`, `YYYY-DD-MM`) to the ISO 8601 standard (`YYYY-MM-DDTHH:MM:SSZ`).
- **Contextual Data Validation:** Instead of blindly removing rows, we apply business-rule-based validation. For example:
  - A negative `amount` in `payments_raw` could be a valid refund, so it should be flagged but not necessarily dropped. However, a negative `quoted_premium` is an error.
  - A `timestamp` in the near future might be valid for a scheduled payment or policy start date. We would define a reasonable lookahead window (e.g., 90 days) and flag anything beyond that as a potential error.
  - Similarly, a `timestamp` too far in the past (e.g., before the customer's account existed or older than a reasonable transaction history, say 2 years) is likely an error and should also be flagged.
- **Value Unification:** Standardise categorical features (e.g., mapping `"USA"`, `"U.S."`, and `"United States"` to a single representation). This also includes unifying geo-resolution from IP addresses and postcodes.

3. **Enrichment / Feature Engineering:**

This step is where we create the predictive signals for our models by combining and transforming the cleaned data. We'll generate features tailored to each specific problem.

Fraud Detection Features

- **Transactional Aggregates (Velocity Features):** These capture the pace and pattern of spending. We'll compute these over various rolling time windows (e.g., last 1h, 6h, 24h, 7d).
  - `txn_count_window`: Number of transactions.
  - `txn_amount_sum_window`: Total amount spent.
  - `unique_merchants_window`: Number of distinct merchants visited.
  - `avg_time_between_txns_window`: Average time between transactions.
- **Behavioral Profile Comparison:** We'll compare the current transaction to the customer's historical behavior.
  - `amount_zscore`: How many standard deviations is the current transaction amount from the customer's historical average?
  - `is_new_merchant`: Has the customer ever used this merchant before?
  - `is_new_country`: Is this the first transaction from this country for this customer?
  - `balance_depletion_speed`: A feature that tracks how quickly a series of transactions is depleting the account's available balance, flagging rapid "cash-out" patterns common in account takeovers.
- **Graph-based Network Features:** We will model relationships between entities to uncover sophisticated fraud rings and patterns, drawing inspiration from anti-money laundering (AML) techniques. This requires a batch process using a graph database (e.g., Neo4j) or library (e.g., NetworkX) to pre-compute scores that are then served from the feature store.
  - We will construct a graph where customers, devices, IP addresses, and merchants are nodes, and transactions are edges.
  - `device_fraud_ring_score`: A score based on how many other accounts have used the same `device_fingerprint` and if any of them have confirmed fraud.
  - **Transaction Cycle Detection:** We will run algorithms to detect circular money movements (e.g., Account A pays B, B pays C, and C pays A back). Such cycles are highly anomalous and a strong indicator of synthetic activity.
  - **"Smurfing" Pattern Detection:** We will identify patterns of "structuring," where a single entity uses multiple accounts to make many small transactions. This includes flagging repeated suspicious amounts (e.g., values like $1,999 or amounts just under a legal reporting threshold) to

stay below detection thresholds. A feature could be `fan_in_ratio` or `fan_out_ratio` for an account over a specific period.

- **Geo-Spatial Enrichment:**
  - By joining the `geo_ip` with a database like MaxMind, we can derive `ip_risk_score` or check if it's from an anonymous proxy.
  - Joining `postcode` with public census data can provide `postcode_deprivation_index` or `regional_risk_score`.

Plan Ranking Features

- **Customer-Plan Interaction Features:** These features directly measure the fit between the customer and the plan.
  - `affordability_ratio`: `quoted_premium` / estimated income (derived from occupation or postcode data).
  - `coverage_match_score`: Cosine similarity between an embedding of the customer's `medical_history` and the plan's `coverage_vector`.
  - `deductible_to_premium_ratio`: The ratio of the plan's deductible to its monthly premium.
- **Semantic & Text-based Features:**
  - We will use a pre-trained language model (like a Sentence-BERT variant) to create embeddings for free-text fields in the plan catalogue. This allows for semantic matching against user needs, even if keywords don't align perfectly.
- **User State Features:**
  - `is_switching`: A binary flag derived from `has_existing_policy`. Switchers may have different preferences than first-time buyers.
  - `family_plan_suitability`: A score based on how well a plan's family coverage options match the user's `family_structure`.

4. **Missing Values & Outliers:**

- **Continuous:** Winsorise numerical features at the 1st and 99th percentiles to handle extreme outliers. Impute missing values using the median, or create a distinct *missing* category if absence is a signal.
- **Categorical:** Introduce a dedicated "UNK" (Unknown) category for unseen values during inference.
- The outlier detection process itself can generate features for the unsupervised fraud model.

5. **Feature Store: The Bridge Between Data and Models** A crucial question is whether we need a dedicated Feature Store or if storing engineered features in a standard database is sufficient. For our use cases, especially real-time fraud detection, a feature store is highly recommended.

   The Core Problem: Train-Serve Skew

   The primary challenge a feature store solves is preventing **train-serve skew**. This occurs when the features used to train a model are different from the features used to make predictions in production. This can happen for several reasons:

   - **Dual Logic:** Without a feature store, teams often write one pipeline for batch-generating features for training (e.g., a Spark job) and a separate, independent implementation for the real-time serving path (e.g., a microservice). It is extremely difficult to keep the logic of these two systems perfectly identical. A subtle difference in how nulls are handled or a floating-point calculation can degrade model performance.
   - **Data Availability:** The online system needs to generate features with very low latency (<150ms for fraud). A standard analytical database is not built for such rapid key-value lookups, whereas a feature store has a specialized low-latency online component.
   - **Point-in-Time Correctness:** For training, it's critical to use the feature values as they existed at the exact moment of the event. Querying a standard database to get this "point-in-time" view for millions of historical events is complex and slow.

   How a Feature Store Solves This

   A feature store (like Feast, Tecton, or Hopsworks) provides a unified framework to solve these issues:

   - **Define Once, Use Anywhere:** You define the feature transformation logic a single time. The feature store then manages both the batch generation of historical features for training and the low-latency serving of online features for inference. This guarantees consistency.
   - **Online/Offline Architecture:** It automatically manages two storage layers:
     - An **Offline Store** (e.g., data warehouse tables) for large-scale training data.

        - An **Online Store** (e.g., Redis, DynamoDB) for fast, key-value lookups at serving time.
    - **Discovery and Governance:** It acts as a central registry for features, allowing teams to discover, share, and reuse features, which improves efficiency and governance.

    For our project, the real-time, high-stakes nature of **Fraud Detection** makes a feature store almost essential. For **Plan Ranking**, while simpler architectures are possible, a feature store still provides significant benefits by ensuring user profile features are consistent and readily available for ranking models.

---

3 Modelling strategy

3.1 Fraud Detection

Our fraud detection strategy is a multi-layered approach, combining a powerful supervised model with techniques to handle the specific challenges of this domain.

- **Baseline Model: Gradient-Boosted Trees**
    - **Technique:** We will use a Gradient-Boosted Tree model, specifically XGBoost or LightGBM.
    - **Rationale:** These models are the industry standard for tabular data. They excel at handling mixed feature types (numerical and categorical), are highly performant for real-time inference, and have built-in mechanisms like `scale_pos_weight` to handle class imbalance effectively.
- **Unsupervised Supplement for Novel Fraud**
    - **Technique:** We will deploy an unsupervised model, such as a Deep Autoencoder or an Isolation Forest, in parallel.
    - **Rationale:** This model is not trained on fraud labels but on learning the structure of "normal" transactions. It helps us catch fraud patterns our main model hasn't seen before. The anomaly score it generates becomes a powerful feature for the main GBDT model.
        - A **Deep Autoencoder** is a neural network trained to reconstruct its input. When trained on normal transactions, it fails to accurately reconstruct

anomalous ones, resulting in a high "reconstruction error" which we use as an anomaly score.
        - An **Isolation Forest** builds a forest of random decision trees. It isolates observations by randomly selecting a feature and then randomly selecting a split value. Anomalies are "few and different" and are therefore easier to isolate, leading to shorter paths in the trees and a lower isolation score.

- **Advanced Imbalance Handling**
    - **Technique:** Beyond simple model weighting, we will implement dynamic class weighting where the weight is proportional to the financial cost of an error. We will also experiment with advanced sampling techniques like SMOTE-NC during training.
    - **Rationale:** This directly aligns the model's optimization function with the business impact of its errors. By creating synthetic fraud examples, we give the model more data to learn from, improving its ability to recognize fraudulent patterns.
        - **SMOTE-NC (Synthetic Minority Over-sampling Technique for Nominal and Continuous features):** This is a powerful technique for creating "synthetic" examples of the minority class (fraud). Unlike standard SMOTE, it can correctly handle datasets containing both numerical and categorical features, which is exactly our case. It generates new fraud samples by interpolating between existing fraud cases for numerical features and using the most frequent category for categorical features among nearest neighbors.

- **Explainability for Compliance**
    - **Technique:** We will use the SHAP (SHapley Additive exPlanations) framework to generate multiple types of visual explanations:
        - **Global Feature Importance:** SHAP summary plots will show the top 20 most influential features across all predictions.
        - **Feature Dependence Plots:** These plots will help us understand how the value of a single feature (e.g., transaction amount) affects the model's output, capturing non-linear relationships.
        - **Local Explanations:** For individual transaction reviews, we will use LIME or SHAP force plots for case-by-case analysis.
    - **Rationale:** This multi-faceted approach to explainability is crucial. Global importance plots provide a high-level overview for model validation.

Dependence plots allow data scientists to debug feature behavior. And local explanations are required by compliance teams and fraud analysts to justify individual decisions to block a transaction.

Real-time serving path

The real-time inference pipeline will be simple and fast:

```
(Payment JSON) → FeatureLookups → GBDT model → If AnomalyScore > τ₂ OR
GBDT_proba > τ₁ → block & queue for manual review
```

3.2 Plan Ranking

Our plan ranking system is designed as a multi-stage funnel to efficiently filter and rank a large catalogue of plans, ensuring relevance and performance.

1. **Feature Design**
    - *Query (user) features*: Embedded medical history, demographics, latent lifestyle cluster.
    - *Document (plan) features*: Coverage vector, cost parameters, insurer reputation.
    - *Query-document features*: Dot-product similarity, price-to-income ratio, "coverage gap" distance.
2. **Model Stack: A Multi-Stage Ranking Funnel** Our model is structured in phases to balance complexity and performance.
    - **Phase 1: Candidate Generation & Filtering**
        - **Model:** A simple, rule-based system.
        - **Notes:** This initial stage quickly removes plans that are clearly unsuitable for the user. The rules will check for hard constraints like geo-eligibility (`available_postcodes`) and user-defined price caps. This dramatically reduces the number of candidates to a manageable set (e.g., < 20).
    - **Phase 2: Fine-Grained Ranking**

- **Model:** A powerful Learning-to-Rank (LTR) model, such as LambdaMART or XGBoost-LTR.
- **Why an LTR Model?** While simpler methods exist, a formal LTR model is chosen for this critical ranking stage for several reasons:
  - **It Directly Optimizes Ranking:** Unlike other methods, listwise LTR models like LambdaMART are explicitly designed to optimize ranking metrics like NDCG. The model learns to answer the question, "*Is plan A better than plan B for this user?*" rather than just, "*What is the absolute score for plan A?*" This is more robust and directly aligned with the business goal of producing the best ordered list.
  - **Contrast with Regression + Sort:** A common alternative is to train a standard classification or regression model to predict the probability of purchase for each plan, then simply sort the results. This "pointwise" approach is weaker because the model never sees the full list during training; it evaluates each plan in isolation. Its loss function (e.g., LogLoss) isn't directly related to ranking quality.
  - **Contrast with Embedding Similarity:** Ranking by pure embedding similarity (as our two-tower model does) is excellent for *candidate generation* or solving the cold-start problem by finding a broad set of semantically relevant items. However, for the final, fine-grained re-ranking of the top candidates, LTR models are superior because they can incorporate a much richer set of explicit interaction features (e.g., `affordability_ratio`, `coverage_match_score`) that go beyond latent similarity.
- **Notes:** This is the core of our ranker. It's trained on implicit user feedback from `quote_logs` (purchases and clicks). A key challenge is correcting for **position bias**: users are far more likely to click items at the top of a list regardless of relevance. We will use Inverse Propensity Weighting (IPW) to handle this.
  - **Calculating Propensities:** We can compute the propensity `P(click | position = k)` for each rank `k` directly from our historical `quote_logs`. This is simply the historical click-through rate for each rank slot: `(total clicks at rank k) / (total impressions at rank k)`. For example, we might find that position 1 has a 20% click rate, while position 5 has a 2% click rate.

        - **Applying Weights:** When training the LTR model, each positive sample (a click or purchase) will be weighted by the inverse of its propensity score: `weight = 1 / P(click | position = k)`. This gives more importance to a click on a lower-ranked item (which was unlikely to be seen) and discounts clicks on top-ranked items (which were likely to be clicked anyway), helping the model learn the plan's true relevance rather than just its position.
    - **Cold-Start & Semantic Matching Fallback**
        - **Model:** A two-tower neural network.
        - **Notes:** This model addresses the cold-start problem for new users or new plans. One tower encodes user features into a vector, and the other tower encodes plan features into a vector in the same space. We can then use Approximate Nearest Neighbour (ANN) search to find semantically relevant plans even without direct interaction data.
3. **Post-processing: Promoting Diversity** A final step after ranking is to apply a diversity penalty. This addresses a common issue where a pure relevance-based ranker might show a list of near-identical plans at the top.
    - **The Problem:** Imagine our LTR model is highly confident that a "Bronze-level plan with a $5,000 deductible" is the best fit. It might rank three such plans from different insurers, all with nearly identical relevance scores, as the top 3 results. While technically correct, this provides a poor user experience as it doesn't offer any real choice.
    - **The Rationale for Diversity:** The goal of diversity is not to show *irrelevant* plans, but to ensure the top-ranked results represent a good *variety* of the available high-quality options. By penalizing for excessive similarity among the top items, we can swap out a redundant plan for a slightly different one (e.g., a Silver-level plan or one with a lower deductible) that still has a high relevance score. This increases the chance that the user will discover the plan that best suits their nuanced needs and improves overall satisfaction with the results.
    - **Technique:** We can implement this using a re-ranking algorithm like **Maximal Marginal Relevance (MMR)**, which iteratively selects items that are both relevant to the user's query and dissimilar to the items already selected for the final list.

---

4 Evaluation & testing

Our evaluation strategy is two-pronged, with distinct offline and online testing plans for each model to ensure both statistical rigor and real-world business impact.

Fraud Model Evaluation

Offline Evaluation

- **Method:** We will use a time-sensitive, stratified 5-fold cross-validation. This approach simulates a realistic production scenario by always training on older data and validating on more recent data (e.g., train on days `t-60` to `t-7`, validate on days `t-7` to `t`). Stratification ensures the rare fraud class is appropriately represented in each fold.
- **Key Metrics:** Our primary offline metric will be **AUPRC (Area Under the Precision-Recall Curve)**, which is ideal for imbalanced datasets. We will also plot a **net-dollar-saved curve** to directly translate model performance into business value.

Online Evaluation

- **Method:** The model will first be deployed in **shadow mode**, where it makes predictions on live traffic without taking any blocking action. This allows us to safely monitor its decisions and alert volumes. After validation, we will perform a **progressive rollout**, starting with 1% of traffic and gradually increasing to 100%, all managed via feature flags for quick rollback if needed.
- **Key Metrics:** In production, we will monitor core business KPIs: the **chargeback rate** (our primary measure of success), the **false-positive cost** (from reviewing incorrectly blocked transactions), and the model's average **prediction latency**.

Ranking Model Evaluation

Offline Evaluation

- **Method:** We will use a temporal hold-out set, training the model on all data up to the last two weeks and using the final two weeks for validation. This prevents data

leakage from future user interactions and properly tests the model's ability to generalize.
- **Key Metrics:** Our core ranking metrics will be **NDCG@3/5**, **Recall@10**, and the **Expected Revenue** generated by the model's proposed ranking.

Online Evaluation

- **Method:** We will run live **A/B tests** (at the cookie or user ID level) to compare the new ranking model against the current production model. We may also explore using a **Bayesian bandit** approach for a more efficient test.
    - **A/B Test vs. Bandit:** A standard A/B test is purely about exploration: traffic is split fixedly (e.g., 50/50) for the duration of the test. A bandit algorithm, however, balances exploration with exploitation: it starts by exploring, but as it sees one version performing better, it dynamically shifts more traffic to the winning variant, thus minimizing the "cost" of showing users an inferior version.
    - **The Bayesian Approach:** A Bayesian bandit models the conversion rate of each version not as a single number, but as a probability distribution (our "belief"). As more data arrives, it uses Bayesian inference to update this belief. This allows the system to make smarter decisions about traffic allocation even with limited data, converging on the best option more quickly. A common algorithm for this is Thompson Sampling.
- **Key Metrics:** The ultimate success metric is a statistically significant **conversion uplift** of over 2%. Crucially, this must be achieved with a non-inferior rate of subsequent customer service calls, ensuring we haven't negatively impacted user satisfaction.

Automated Model Quality & Safety Gates

A crucial part of our MLOps strategy is building automated gates to ensure that only high-quality, safe models make it to production.

Offline Pre-deployment Gates

- **What it is:** This is an automated check within our CI/CD pipeline (e.g., using GitHub Actions) that runs before a model can be deployed.
- **Mechanism:** The pipeline automatically trains the new model candidate and evaluates it against our held-out test set. We then compare its key offline metrics

(e.g., AUPRC for fraud, NDCG for ranking) against the currently deployed production model's scores on the same test set.
- **The Gate:** If the new model shows a regression (namely performance decrease) greater than a pre-defined tolerance (e.g., 0.5 percentage points), the pipeline fails. This automatically blocks the model from being promoted, preventing a worse model from reaching users.

Online Post-deployment Gates

- **What it is:** This is a live monitoring and automated rollback system that operates during the progressive rollout phase.
- **Mechanism:** As the new model serves a small percentage of live traffic (e.g., the first 1% or 5%), an automated monitoring system will track its real-time performance on key business KPIs.
- **The Gate:** We will set predefined safety thresholds for critical online metrics (e.g., a >10% increase in the false positive rate, a >20ms increase in median latency, or a significant drop in click-through rate). If the new model variant breaches any of these thresholds for a sustained period, the system will automatically trigger an alert and can be configured to halt the rollout and revert traffic back to the stable production model, ensuring minimal negative impact.

---

5 GenAI / LLM Integration Ideas

Here are several ways we can leverage Large Language Models (LLMs) to enhance our project:

- **Synthetic Data Generation for Robustness Testing** LLMs can create high-quality synthetic data to augment our training sets and improve model robustness across both problems.
    - **For Fraud Detection:** By prompting a model like GPT-4.1 with scenarios like, "*Generate 20 realistic fraudulent transaction sequences conditioned on a stolen credit card,*" we can create diverse, hard-to-detect examples to augment our rare minority class and stress-test the model's resilience.
    - **For Plan Ranking:** We can generate synthetic user profiles and their likely plan choices, especially for rare but critical segments (e.g., users with specific

chronic conditions). This helps address the cold-start problem and allows us to evaluate model fairness and performance for underrepresented groups.

- **Advanced Feature Extraction from Text** LLMs are excellent at converting unstructured text into meaningful numerical representations (embeddings). We can use a powerful embedding model (e.g., OpenAI's `text-embedding-3-large`) to process text fields like merchant descriptors or insurance claim notes. These rich vector embeddings can then be fed as features into our downstream models.

- **Semantic Fraud Signal Detection** This is a more targeted application of text analysis for fraud. An LLM can be trained or fine-tuned to specifically look for signs of fraudulent intent within text fields. For example, it could analyze recipient names, payment descriptions, or account holder information to flag semantically suspicious patterns (e.g., a recipient named "Quick Cash Ltd.", variations of the same name to avoid limits, or descriptions that sound like money laundering). The LLM's output (e.g., a "suspiciousness score") would be a powerful feature for the main fraud model.

- **Plain-English Plan Explanations** After our ranking model has produced a list of plans, we can use an LLM to provide a personalized, plain-English summary for the top recommendations. For example, a call to the LLM could explain, "*Why this plan fits you,*" based on the user's profile and the plan's attributes, with appropriate guardrails to ensure accuracy and prevent hallucinations.

- **Interactive Analyst Tooling** We can build a chat-based interface that allows data analysts to query our data and model results using natural language. This would enable an analyst to ask questions like, "*Why has the fraud block rate increased for users in segment X over the last week?*" The LLM-powered tool would then translate this question into the appropriate queries against our feature store and SHAP explanation dashboards.

- **Automated Documentation and Lineage** An LLM can be configured to watch our project's Git repository. When a data scientist commits a change to a feature engineering pipeline, the LLM can automatically draft documentation for the new feature, describe its lineage, and even update the README for the corresponding Airflow DAG.

- **AI-assisted Development** The entire development lifecycle can be accelerated by using AI coding assistants (e.g., Cursor, GitHub Copilot). These tools can write boilerplate code for data processing, generate unit tests for feature transformations,

help debug complex models, and explain unfamiliar parts of the codebase, freeing up data scientists to focus on higher-level problem-solving.

- **Multimodal Document Processing** For tasks like customer onboarding or claim processing, we can use multimodal models that understand both images and text. A user could upload a photo of their ID or a PDF of a medical bill. The model could then extract the structured information, check for signs of tampering, and automatically populate the user's profile, streamlining data entry and verification.

- **Reasoning for Complex Analysis** We can use the advanced reasoning capabilities of LLMs to automate complex analysis. For example, a "fraud investigation assistant" could be built. When a complex case is flagged, the model would receive all linked data and generate a step-by-step "chain-of-thought" hypothesis about the fraud method (e.g., "Hypothesis: Account Takeover. Step 1: Login from a new, high-risk IP. Step 2..."). This would drastically speed up the work of human analysts.

---

6 Bonus: Code from implementation notebooks

A selection of Python snippets from the accompanying notebooks (`fraud_detection_notebook.ipynb`, `ranking_model_notebook.ipynb`) that demonstrate key parts of the implementation.

Fraud Detection: Feature Engineering (Pandas)

This snippet from `fraud_detection_notebook.ipynb` shows the creation of time-windowed "velocity" features, which are highly predictive for fraud.

```
# Create time-windowed velocity features using pandas
# (from fraud_detection_notebook.ipynb)
for window_hours in [1, 6, 24, 168]:  # 1h, 6h, 1d, 1w
    window_str = f'{window_hours}H'
    data_indexed = data.set_index('timestamp')

    # Count of transactions in window
```

```
    data[f'txn_count_{window_hours}h'] = (
        data_indexed.groupby('cust_id')
        .rolling(window_str, closed='both')
        .size()
        .reset_index(level=0, drop=True)
    )

    # Sum of amounts in window
    data[f'amount_sum_{window_hours}h'] = (
        data_indexed.groupby('cust_id')['amount']
        .rolling(window_str, closed='both')
        .sum()
        .reset_index(level=0, drop=True)
    )
```

Fraud Detection: Cost-Sensitive Model Training (XGBoost)

Here we configure an XGBoost classifier, paying special attention to the `scale_pos_weight` parameter. This tells the model to penalize misclassifying a rare "fraud" instance much more heavily than a "legitimate" one, aligning the model with business costs.

```
# Calculate class weights and train a cost-sensitive XGBoost model
# (from fraud_detection_notebook.ipynb)
n_legitimate = (y_train == 0).sum()
n_fraud = (y_train == 1).sum()

# Give more weight to catching fraud based on business cost
fraud_cost = 100
false_positive_cost = 1
scale_pos_weight = (n_legitimate * fraud_cost) / (n_fraud *
false_positive_cost)

xgb_model = xgb.XGBClassifier(
    objective='binary:logistic',
    eval_metric='aucpr', # Area under PR curve is better for imbalance
    scale_pos_weight=scale_pos_weight,
    max_depth=6,
    n_estimators=200,
    random_state=42
)
# Train using data balanced with SMOTE-NC for better performance
xgb_model.fit(X_train_balanced, y_train_balanced)
```

Fraud Detection: Unsupervised Anomaly Detection (Isolation Forest)

To catch novel fraud patterns not seen in the training labels, we supplement our main model with an unsupervised anomaly detector like Isolation Forest. Its score can be a powerful feature.

```
# Train an Isolation Forest on legitimate data to find novel anomalies
# (from fraud_detection_notebook.ipynb)
from sklearn.ensemble import IsolationForest

iso_forest = IsolationForest(
    contamination=0.001,  # Expected proportion of anomalies
    random_state=42
)

# Train only on normal transactions
X_train_legitimate = X_train[y_train == 0]
iso_forest.fit(X_train_legitimate)

# Anomaly scores can now be generated for all data
anomaly_scores = iso_forest.decision_function(X_test)
```

Plan Ranking: Learning-to-Rank Model Training (LightGBM)

The ranking problem requires a specialized Learning-to-Rank model. Here, we use LightGBM's `LGBMRanker` with the `lambdarank` objective, which is designed to directly optimize ranking metrics like NDCG.

```
# Train a LambdaMART model for ranking using LightGBM
# (from ranking_model_notebook.ipynb)
import lightgbm as lgb

# qid_train is an array specifying group boundaries for ranking lists
ltr = lgb.LGBMRanker(
    objective='lambdarank',
    metric='ndcg',
    ndcg_eval_at=[3, 5]
)

ltr.fit(
    X_train, y_train, group=qid_train,
```

```
    eval_set=[(X_val, y_val)], eval_group=[qid_val]
)
```

For Production Scale: Feature Engineering (PySpark)

While the notebooks use pandas for rapid prototyping, a production system handling millions of daily events would use a distributed framework like Spark. This snippet illustrates how similar velocity features would be generated at scale.

```
# === Fraud features (PySpark) ===
from pyspark.sql import functions as F, Window

# Define window for last 6 hours of activity per card
window_6h =
Window.partitionBy('cust_id').orderBy('timestamp').rangeBetween(-21600, 0) #
seconds in 6h

txn_feats = (
    payments_raw
    .withColumn('amt_zscore', (F.col('amount') - mean_amt) / std_amt)
    .withColumn('txn_count_6h', F.count('*').over(window_6h))
    .withColumn('unique_mcc_24h',

F.approx_count_distinct('mcc_code').over(window_6h.rangeBetween(-86400, 0)))
)
```