The Ultimate Guide to Machine Learning System Design

Introduction

Machine learning (ML) has transcended academic research to become a cornerstone of modern technology, powering a vast array of applications from personalized recommendations to sophisticated fraud detection systems. Consequently, the ability to design, build, and deploy robust ML systems is no longer a niche skill but a fundamental requirement for many engineering and data science roles. This shift is profoundly reflected in the interview processes at leading technology companies.

The Rise of the ML System Design Interview

Machine Learning System Design interviews have emerged as a critical component in the evaluation of ML engineers and data scientists, particularly for mid-to-senior level positions. Their primary purpose extends beyond assessing knowledge of specific algorithms; they aim to evaluate a candidate's proficiency in architecting, designing, and reasoning about complex, scalable, and robust end-to-end ML solutions within a practical, real-world context.[1] These interviews probe a candidate's understanding of the entire ML lifecycle, from initial problem formulation and data ingestion through model deployment, monitoring, and continuous iteration, emphasizing the practical trade-offs and constraints inherent in such endeavors.[1] Success demands not only technical depth but also strong system thinking, the ability to translate ambiguous business problems into concrete ML solutions, and effective communication skills.[2]

This evolution in interview focus signifies a maturation in the field of machine learning. No longer is isolated model building sufficient. Companies seek engineers who can deliver holistic, working systems that solve tangible business problems and integrate seamlessly into production environments. The emphasis is on the entire ecosystem

within which an ML model operates, demanding a comprehensive understanding of data pipelines, deployment strategies, monitoring mechanisms, and the iterative nature of ML development.

Purpose of This Guide

This guide is meticulously crafted to provide a structured, end-to-end framework for mastering ML system design interviews. It aims to equip readers with a repeatable methodology and the conceptual depth required to excel in interviews at top-tier technology companies.[1] The focus throughout is on understanding the "why" behind design decisions, not merely the "what." By synthesizing information from established design principles, common problem archetypes, and advanced considerations, this guide serves as an exhaustive resource for candidates preparing for these challenging evaluations.

What to Expect

This guide will navigate the reader through a comprehensive journey, beginning with the foundational principles of ML system design and progressing to practical applications and advanced topics.
Chapter 1 will introduce "The Machine Learning System Design Framework," a systematic five-step approach to tackling any ML design problem.
Chapter 2 will delve into "Common Machine Learning System Design Archetypes," providing detailed walkthroughs and illustrative diagrams for prevalent systems like recommendation engines, search ranking, feed-based platforms, and fraud detection.
Chapter 3 will explore "Advanced Topics & Considerations," covering essential modern practices such as MLOps, ethical AI development, designing for scale, and the critical role of human-in-the-loop processes.
By the end of this guide, readers will possess a robust understanding of how to approach ML system design problems with confidence, clarity, and strategic insight.

Chapter 1: The Machine Learning System Design Framework

Developing products powered by machine learning presents unique challenges compared to traditional software engineering, primarily due to the inherent uncertainty surrounding model performance, critical dependence on data, and complex ethical considerations.[1] A systematic framework is therefore essential to navigate this ambiguity and mitigate risks. This chapter introduces a five-step framework that provides essential guideposts for designing ML systems, emphasizing that this process is often an iterative cycle rather than a rigid linear sequence. Discoveries made in later stages frequently necessitate revisiting and refining earlier assumptions and decisions.[1]

1.1. Step 1: Problem Formulation & Scope Definition

Conceptual Overview: This initial stage is paramount and focuses on deeply defining the 'why' and 'what' of the ML system.[1] It involves translating a potentially vague business problem into a concrete, measurable ML objective, clarifying scope, scale, latency, and other operational constraints.[1]

The Art of Asking Clarifying Questions:
Asking insightful clarifying questions at the outset is crucial for understanding the problem's boundaries and success criteria.1 These questions should not be random but targeted to uncover essential constraints and objectives that will shape the entire design. For example, understanding the scale (e.g., millions of daily active users, vast product catalogs) and latency requirements (e.g., sub-200ms for real-time recommendations) directly influences model complexity and infrastructure choices.1 Similarly, data availability—including user data, item metadata, interaction logs, and the presence of real-time event streams—dictates feasible modeling approaches.1 Other critical areas to probe include the desired level of personalization, strategies for handling new users or items (the cold-start problem), and any specific requirements related to diversity, serendipity, or fairness in the system's output.1
A structured approach to questioning can be highly effective. The following table provides an actionable checklist for interviewees, ensuring comprehensive coverage of critical areas early in the design discussion.

Table 1.1.1: Key Clarifying Questions for Problem Formulation

Category	Key Question Examples
Business Objective	What is the primary business goal (e.g.,

	increase engagement, revenue, reduce fraud, improve user satisfaction)? Are there secondary objectives?
	How will the success of this system be measured from a business perspective?
Scale & Performance	What is the expected scale (e.g., Daily Active Users (DAUs), Queries Per Second (QPS), data volume, catalog size)?
	What are the latency requirements (e.g., p50, p90, p95, p99 response times)? Are there throughput targets?
Data Availability	What data sources are available (e.g., user profiles, item metadata, interaction logs, text, images)? Are real-time data streams available?
	What is the quality and quantity of the available data? Is historical labeled data available?
User Experience	How will users interact with this system? What does a successful outcome look like for the end-user?
	How should new users or new items (cold start) be handled?
	Are there specific considerations for fairness, bias, diversity, or serendipity in the results?
Specific Constraints	Are there any computational resource constraints (CPU, GPU, memory)? Are there budget constraints for development or operation?
	Are there any regulatory, privacy (e.g., GDPR, CCPA), or ethical constraints to consider?
	How frequently does the underlying data or user behavior change? Does the system need to adapt quickly?
Success Metrics (ML)	What are the initial thoughts on ML evaluation metrics (e.g., accuracy, precision, recall, NDCG, MAE)?

Defining the Business Objective:
A core task in this step is to translate a potentially vague problem statement into a concrete business goal. For instance, when designing an e-commerce recommendation system, the primary business objective is typically to increase user engagement (measured by metrics like Click-Through Rate (CTR) or Add-to-Cart Rate) and ultimately drive sales (Conversion Rate, Revenue Lift).1 In the context of fraud detection, the objective is often twofold: to minimize financial losses from fraudulent transactions while simultaneously minimizing the disruption to legitimate customers caused by incorrectly blocked transactions.1 For a news feed, the primary goal is usually to maximize user engagement, which can encompass clicks, likes, comments, shares, or time spent viewing a post.1
It is critical to connect the ML solution directly to its intended business impact. The ML system is a means to an end, and design choices should consistently be tied back to the business value they aim to deliver. If an interviewer presents an underspecified problem, the candidate should proactively probe for these underlying business objectives, as they will dictate the subsequent ML objectives and the metrics used to measure success.[1]

Scoping the Problem:
Once the business objective is clear, the problem must be scoped by defining specific constraints and requirements. This includes quantifying the scale (e.g., QPS, DAUs, data volume), latency targets, and overall performance expectations.1 For example, a system requiring real-time predictions for millions of users with sub-second latency will necessitate different architectural and modeling choices than a batch processing system with more relaxed constraints.7 Understanding these parameters early is crucial as they significantly influence design trade-offs throughout the process.
Choosing the Right ML Paradigm:
The initial problem understanding and defined objectives guide the selection of an appropriate ML paradigm.4 This involves determining whether the core task is best framed as classification (e.g., fraud vs. not fraud), regression (e.g., predicting travel time), ranking (e.g., ordering search results or feed items), clustering, generation, or another ML approach. For example, fraud detection is often formulated as a binary classification or an anomaly detection problem.1 ETA prediction for a ride-sharing app is a regression task 1, while visual search primarily involves image similarity retrieval.1
The choice of ML paradigm acts as a translation layer, converting the business need into a specific type of prediction or inference required to achieve the desired outcome.[2] Candidates should be able to articulate

why a particular paradigm is the most suitable for the stated objectives and constraints. For example, for a news feed, one might consider whether simple classification of posts (e.g., "interesting" vs. "not interesting") is sufficient, or if a more nuanced ranking approach is needed to optimize the order of diverse content types.

1.2. Step 2: Data Deep Dive

Conceptual Overview: Data is the lifeblood of any machine learning system. This step involves a thorough examination of data requirements, potential sources, quality assessment, and strategies for processing and management.[2] Ensuring high-quality data is paramount, as the performance of the ML model is fundamentally dependent on the data it is trained on.[4]

Data Acquisition and Sources:
The first activity is to identify all relevant data sources. Common sources include user profiles (often stored in relational or NoSQL databases), product or content catalogs (from databases or APIs), and user interaction data (such as clickstream logs, purchase histories, ratings, views, and search queries).1 For systems requiring up-to-the-minute information, real-time event streams from platforms like Apache Kafka or AWS Kinesis are crucial for ingesting recent user activity or transaction data.1 In some cases, third-party data, such as IP reputation scores for fraud detection or social trend data for recommendation systems, may also be incorporated.1 Historical logs, often stored in data lakes or data warehouses, provide the basis for model training.1
The types of data encountered vary widely and can include numerical data (e.g., transaction amounts, age), categorical data (e.g., product categories, user segments), text data (e.g., product descriptions, reviews), image data (e.g., product photos, user-uploaded images), and multimodal data combining several of these types.[4] Understanding the specific data sources and types available for a given ML archetype is essential. For instance, a recommendation system will heavily rely on user-item interaction data and item metadata, while a fraud detection system will focus on transaction details and user account history.

Table 1.2.1: Common Data Sources & Types in ML Systems

ML Archetype	Typical Data Sources	Key Data Types	Collection Methods/Considerations
Recommendation Systems	User profiles, item catalogs, user-item interactions (views, clicks, purchases, ratings, watch time), event streams	User IDs, item IDs, categorical features, numerical ratings, text, image embeddings	Logging user activity, database dumps, real-time event ingestion (e.g., Kafka, Kinesis) [1]

Search & Ranking	Query logs, document/product corpus, user interaction with search results (clicks, conversions), user profiles	Text (queries, documents), numerical features (relevance scores), user IDs	Logging search queries and interactions, web crawling, catalog databases [1]
Feed-Based Systems	User profiles, social graph, content items (posts, articles, videos), user interactions with feed items, event streams	Text, images, videos, user IDs, timestamps, interaction types (likes, shares)	User-generated content platforms, interaction logging, real-time event ingestion [1]
Fraud/Anomaly Detection	Transaction logs, account information, device information, third-party risk data, historical labeled fraud cases	Numerical transaction data, categorical features, IP addresses, device IDs	Real-time transaction streams, database queries, external API integrations [1]
ETA Prediction	GPS location streams (drivers, riders), road network data, real-time traffic data, historical ride logs, weather data	Geospatial coordinates, timestamps, numerical speeds, categorical road types	Real-time GPS ingestion, map data APIs, traffic data feeds, historical databases [1]
Visual Search	Product catalog images, user-uploaded query images	Images, item IDs, categorical metadata	Image databases, real-time image uploads [1]

Data Exploration and Analysis:
Once data sources are identified, a critical step is data exploration and analysis (EDA). This involves understanding the data's characteristics, uncovering potential biases, and assessing its limitations.1 Data profiling techniques are used to understand the content and structure of the data, while data validation helps to spot errors or inconsistencies.8 EDA is crucial for identifying data quality issues such as missing values, outliers, or incorrect entries. It also helps in understanding data distributions and the relationships between different variables.
A particularly important aspect of EDA is the identification of potential biases within the data.[1] For example, historical data used for training a loan application model might reflect past discriminatory lending practices; using such data without awareness or

mitigation can lead to an ML system that perpetuates these biases.[9] Similarly, if training data for a content moderation system contains spurious correlations (e.g., certain neutral keywords being accidentally associated with harmful content in the training set), the model may learn incorrect patterns and perform poorly or unfairly in production.[1] Therefore, EDA is not just about preparing features for modeling but also about proactively identifying ethical risks and data limitations that could undermine the system's fairness and effectiveness.

Feature Engineering:
Feature engineering is often described as both an art and a science, involving the creation of powerful, informative features from raw data.1 This process typically requires domain knowledge and expertise in handling various data types.1 The goal is to transform raw data inputs into a format that ML models can effectively learn from.
Common feature engineering techniques include:

- **For Numerical Data:** Normalization (scaling values to a range like) or standardization (scaling to have zero mean and unit variance) to ensure features with larger magnitudes do not dominate the learning process.[4] Handling missing values through imputation (e.g., mean, median, or model-based imputation) is also common.
- **For Categorical Data:** One-hot encoding (creating binary columns for each category) or embedding (learning dense vector representations) are standard approaches.[5]
- **For Text Data:** Techniques range from simple Bag-of-Words (BoW) or TF-IDF (Term Frequency-Inverse Document Frequency) representations to more sophisticated word embeddings (e.g., Word2Vec, GloVe, FastText) or contextual embeddings from transformer models like BERT.[1]
- **For Image Data:** Pre-trained Convolutional Neural Networks (CNNs) are often used to extract dense embedding vectors that capture visual features.[1]
- **For Time-Series Data:** Lagged variables (past values of the series), rolling window statistics (e.g., moving averages), and time-based features (e.g., day of week, month) are frequently created.
- **Interaction Features:** Creating new features by combining or transforming existing ones (e.g., ratios, products) to capture non-linear relationships.

The specific features engineered are highly dependent on the problem domain. For instance, in fraud detection, "velocity features" (e.g., number of transactions by a user in the last hour) and graph-based features (identifying connections in fraud rings) can be highly predictive.[1] In news feed ranking, features like post age, author statistics, and content embeddings are critical.[1] This domain-specificity means that feature

engineering is not a one-time step but an iterative process of hypothesis generation, feature creation, and testing, often benefiting from collaboration with domain experts.

Table 1.2.2: Feature Engineering Techniques Overview

Data Type	Common Techniques	Examples	Considerations/Trade-offs
Categorical	One-Hot Encoding, Label Encoding, Target Encoding, Embeddings	User segment, product category, payment type	Dimensionality (OHE), information loss (Label), target leakage (Target), complexity (Embeddings)
Numerical	Normalization (Min-Max), Standardization (Z-score), Binning, Log Transformation	Age, price, transaction amount, sensor readings	Algorithm sensitivity to scale, distribution assumptions, handling outliers
Text	Bag-of-Words (BoW), TF-IDF, N-grams, Word Embeddings (Word2Vec, GloVe), Transformers	Product descriptions, user reviews, social media posts	Loss of sequence (BoW/TF-IDF), vocabulary size, computational cost (Transformers), pre-trained vs. custom embeddings
Image	Pixel Intensities, Histogram of Oriented Gradients (HOG), CNN-based Embeddings	Product images, user-uploaded photos, medical scans	Dimensionality, computational cost, need for pre-trained models (CNNs), robustness to variations
Time-Series	Lag Features, Rolling Statistics (Mean, Std), Seasonal Decomposition, Date/Time Parts	Stock prices, sales data, sensor data over time	Choice of lag/window size, handling non-stationarity, feature leakage from future
Graph	Node Degree, Centrality Measures, Community	Social networks, transaction networks, knowledge graphs	Computational complexity, scalability for large graphs,

	Detection, Graph Embeddings (Node2Vec)		choice of embedding method
Interaction	Feature Crossing (e.g., product of two features), Polynomial Features	Combining user location and time of day, user tenure * purchase frequency	Can capture non-linearities, risk of overfitting, increases feature space significantly

Data Labeling and Annotation:
For supervised learning tasks, creating high-quality ground truth data through labeling or annotation is essential.8 This involves assigning labels (e.g., "fraudulent," "positive sentiment," "relevant item") to raw data instances. Strategies for obtaining labels include manual human annotation, programmatic labeling (using rules or heuristics), and synthetic data augmentation.4
Particularly for complex or subjective tasks like content moderation or nuanced sentiment analysis, human annotators are often indispensable.[1] This necessitates robust data labeling pipelines, clear and consistent annotation guidelines, and training for annotators to minimize subjectivity and ensure inter-annotator agreement.[1] The quality of these labels is a major determinant of model performance; models trained on noisy or inconsistently labeled data, or data with spurious correlations, can learn incorrect patterns and perform poorly or unfairly.[1] Therefore, investing in high-quality labeling processes, potentially incorporating human-in-the-loop mechanisms for continuous refinement, is critical.[12]

Data Storage and Pipelines:
Efficient data storage and processing pipelines are fundamental to the ML system. This includes systems for both offline (batch) and online (real-time) data handling.

- **Batch Processing:** Typically used for model training and periodic feature computation. Raw data (e.g., historical interaction logs) is often stored in data lakes (e.g., AWS S3, Google Cloud Storage). ETL (Extract, Transform, Load) jobs, frequently implemented using distributed processing frameworks like Apache Spark, are run on this historical data to clean it, aggregate user/item behavior, engineer features, and generate labeled training datasets. The processed data and features may be stored back in the data lake or loaded into a data warehouse (e.g., Amazon Redshift, Google BigQuery) for analysis and model training.[1]
- **Stream Processing:** Essential for real-time feature updates and reacting to immediate user actions. Event streaming platforms like Apache Kafka or AWS Kinesis ingest high-volume data (e.g., clicks, views, transactions). Stream processing engines (e.g., Apache Flink, Spark Streaming, AWS Lambda) consume these events to update user profiles, compute near real-time features (e.g.,

"items viewed by the user in the last 5 minutes"), or trigger alerts.[1]

A key component in modern ML data infrastructure, especially for systems requiring real-time predictions, is the **Feature Store**. A Feature Store is a centralized repository for storing, managing, versioning, and serving ML features. It ensures consistency in feature definitions and computations between training (batch) and serving (online) environments, which helps prevent training-serving skew. Feature Stores also facilitate low-latency feature retrieval during real-time inference, which is crucial for applications like personalized recommendations or fraud detection.[1]

1.3. Step 3: Modeling

Conceptual Overview: The modeling step focuses on selecting appropriate model architectures, defining a comprehensive training strategy, and establishing robust methods for evaluating model performance.[1] A guiding principle is to start with simpler models and iterate towards more complex solutions as needed, based on performance and system requirements.

Model Selection:
The choice of an ML model is a critical decision, guided by the nature of the learning problem (classification, regression, ranking, etc.), the specific use case, data characteristics, and practical constraints such as latency, interpretability, computational resources, and business requirements.4 It is generally advisable to begin with a simple baseline model to establish initial performance benchmarks and then iteratively explore more complex alternatives.1
Common model types for different ML tasks include:

- **Recommendation Systems:**
  - *Candidate Generation:* Collaborative Filtering techniques (User-based CF, Item-based CF, Matrix Factorization methods like SVD or ALS), Content-Based Filtering, and Two-Tower neural network models are prevalent.[1]
  - *Ranking:* Learning-to-Rank (LTR) models are standard. These include Pointwise approaches (e.g., Logistic Regression, Gradient Boosted Trees (GBTs) like XGBoost or LightGBM, Neural Networks), Pairwise approaches (e.g., RankNet, LambdaRank), and Listwise approaches (e.g., LambdaMART). Deep Learning models such as Wide & Deep, DeepFM, and DLRM are also widely used for their ability to model complex interactions.[1]
- **Fraud Detection:**
  - GBTs (XGBoost, LightGBM, CatBoost) are highly effective for the structured,

tabular data common in fraud detection. Deep Learning models (MLPs, RNNs for sequential transaction data, GNNs for relational data) can capture more complex patterns. Rule-based systems often complement ML models, and ensemble methods combining diverse models can enhance robustness.[1] Unsupervised anomaly detection methods like Autoencoders or Isolation Forests are also valuable, especially for novel fraud patterns.[1]

- **News Feed Ranking:**
  - LTR is the primary task. Linear models (e.g., Logistic Regression) can serve as baselines. GBTs are strong performers for ranking with diverse features. Deep Learning models (MLPs, Wide & Deep, DeepFM, DLRM, custom neural networks) are commonly employed in large-scale systems to handle complex interactions and multimodal content (text, image embeddings).[1]
- **ETA Prediction:**
  - This is a regression task. Linear Regression can provide a baseline. GBTs are often highly effective. Deep Learning approaches include sequence models (RNNs, LSTMs, GRUs, Transformers) to model route sequences and traffic dynamics, and GNNs to operate on road network graphs.[1]
- **Visual Search:**
  - The core is image similarity search. Convolutional Neural Networks (CNNs), such as ResNet, EfficientNet, or Vision Transformers (ViT), pre-trained on large datasets like ImageNet, are used as backbones to generate image embeddings. The output layer is modified to produce these dense vector representations.[1]

The "No Free Lunch" theorem applies to model selection: no single model is universally optimal. The choice always involves trade-offs between accuracy, complexity, interpretability, training time, and inference latency.[4] Candidates should not default to the most complex model but should justify their selection based on these trade-offs and often propose an iterative approach starting with a simpler baseline.[1] Furthermore, hybrid models, which combine the strengths of different approaches (e.g., collaborative filtering with content-based features for recommendations, or unsupervised anomaly scores as input to a supervised fraud classifier), are often employed in practice to achieve more robust and comprehensive solutions.[1]

Training the Model:
Effective model training is more than just calling a fit function; it's a strategic process involving careful data preparation, selection of appropriate loss functions, optimization algorithms, regularization techniques, and validation strategies.1

Key aspects of model training include:

- **Loss Functions:** The choice of loss function is dictated by the ML task. Examples include Cross-Entropy for classification, Mean Squared Error (MSE) or Mean Absolute Error (MAE) for regression, specialized LTR loss functions like LambdaRank for ranking tasks, and metric learning losses like Triplet Loss or Contrastive Loss for learning similarity embeddings in visual search.[1]
- **Optimization Algorithms:** Common optimizers include Stochastic Gradient Descent (SGD) and its variants (e.g., Adam, Adagrad). Hyperparameters like learning rate and batch size need careful tuning.
- **Regularization:** Techniques such as L1/L2 regularization, dropout, and early stopping are employed to prevent overfitting and improve model generalization.[5]
- **Handling Data Imbalance:** This is particularly critical in domains like fraud detection where the target class (fraudulent transactions) is rare.[1] Strategies include:
    - *Data-level approaches:* Undersampling the majority class, oversampling the minority class (e.g., using SMOTE - Synthetic Minority Over-sampling Technique).
    - *Algorithmic approaches:* Cost-sensitive learning (assigning higher misclassification costs to the minority class) or adjusting the classification threshold of the model's output.
- **Negative Sampling:** For recommendation systems trained on implicit feedback (e.g., clicks, views), where only positive interactions are observed, negative examples (items the user did not interact with) must be strategically sampled for training.[1]
- **Data Splitting and Validation:** For time-dependent data (common in recommendations, fraud, ETA prediction), time-based splits (e.g., training on past data, validating on more recent data, testing on future data) are essential to avoid lookahead bias and realistically evaluate the model's ability to predict future behavior.[1] Stratified K-fold cross-validation is often used for other scenarios.
- **Hyperparameter Tuning:** Systematically searching for optimal model hyperparameters (e.g., learning rate, number of layers, regularization strength, embedding dimensions) using techniques like grid search, random search, or Bayesian optimization, guided by performance on a validation set.[1]
- **Distributed Training:** For very large datasets or complex models, training may need to be distributed across multiple machines or GPUs using frameworks like TensorFlow Distributed, PyTorch Distributed Data Parallel, or Horovod.[1]

Evaluating the Model:
Model evaluation involves choosing appropriate offline metrics and designing a robust

strategy to assess performance before deployment, and subsequently validating impact through online testing.1

- **Offline Evaluation Metrics:** These are calculated on a held-out test set using historical data.
    - *For Classification (e.g., Fraud Detection, Sentiment Analysis):* Precision, Recall, F1-Score (especially for the minority class in imbalanced scenarios), AUC-ROC, AUC-PR (Area Under the Precision-Recall Curve, preferred for highly imbalanced data as it focuses on minority class performance), and the Confusion Matrix are key.[1] Accuracy alone can be highly misleading.[1]
    - *For Ranking (e.g., Recommendations, Search, Feeds):* Precision@k, Recall@k, Mean Average Precision (MAP@k), Mean Reciprocal Rank (MRR), and Normalized Discounted Cumulative Gain (NDCG@k) are standard for evaluating the quality of ranked lists.[1]
    - *For Regression (e.g., ETA Prediction):* MAE, RMSE, Mean Absolute Percentage Error (MAPE), and R-squared are common. Analyzing the distribution of prediction errors is also crucial.[1]
    - *For Embedding Quality (e.g., Visual Search):* Metric learning losses (Triplet/Contrastive) during training, and qualitative assessment (t-SNE/UMAP visualizations) or quantitative checks of embedding space structure post-training.[1]
    - *Beyond Accuracy Metrics:* For systems like recommendations or feeds, it's important to evaluate aspects like diversity (variety in recommendations), novelty (how unknown the recommendations are), serendipity (surprising yet relevant suggestions), and coverage (percentage of catalog items that can be recommended).[1]
    - *Fairness Metrics:* Evaluating performance disparities across different user subgroups using metrics like False Positive Rate Parity or Equalized Odds is critical for responsible AI.[1]
- **Online Evaluation (A/B Testing):** This is the gold standard for measuring real-world impact.
    - The new model or system variant is exposed to a segment of live users, and its performance on key business metrics (e.g., CTR, conversion rate, revenue, user engagement, fraud loss reduction, task completion rates) is compared against a control group or the existing system.[1]
    - Online A/B testing provides the ground truth for a model's effectiveness because offline metrics, while useful for rapid iteration, can sometimes be poor predictors of online performance due to biases present in historical data (e.g., presentation bias, selection bias).[1]

A balanced scorecard of metrics, including primary business objectives, model performance indicators, and guardrail metrics (to monitor unintended negative consequences), is essential for a holistic evaluation.[1]

Table 1.3.1: Offline vs. Online Evaluation Metrics: Pros & Cons

Metric Type	Specific Metrics Examples	Description	Pros	Cons	When to Use
Offline	NDCG@k, MAP@k, AUC-PR, F1-Score, MAE, RMSE, Precision@k, Recall@k	Calculated on historical, held-out data before deployment.	Fast iteration, cost-effective, good for model selection and hyperparameter tuning, helps debug.	May not correlate well with online business impact due to data biases, doesn't capture real user behavior.	During model development, for comparing candidate models, for initial performance assessment. [1]
Online	CTR, Conversion Rate, Revenue, Engagement Time, Fraud Loss Prevented, Task Success Rate	Measured via live A/B tests on actual users in the production environment.	Reflects true business impact and user behavior, gold standard for validation.	Slower, resource-intensive, risk of negative user impact, harder to isolate causal effects.	For final model validation before full rollout, for measuring actual impact on business KPIs. [1]

1.4. Step 4: System Architecture & Deployment

Conceptual Overview: This step involves designing the overall system that orchestrates everything from data ingestion to prediction serving. Key considerations include choosing between online (real-time) and offline (batch) serving paradigms, ensuring the system is scalable and reliable, and selecting appropriate deployment

strategies for introducing new models or updates into production.[1]

High-Level Architecture:
ML systems, particularly for complex applications like recommendation or feed ranking, often employ a multi-stage pipeline architecture.1 A common pattern is the adoption of a microservices architecture, where different functionalities are encapsulated in independent, deployable services, enhancing modularity and scalability.32
Typical components can be broadly categorized into offline and online:

- **Offline Components:** These handle tasks that are not time-sensitive and are often performed in batch.
  - *Data Sources:* Databases (user profiles, product catalogs), Data Lakes (raw logs).[1]
  - *Batch ETL Pipeline:* Using tools like Apache Spark for data cleaning, transformation, feature engineering, and generating training datasets.[1]
  - *Feature Store (Offline Storage):* Stores features for model training, ensuring consistency.[1]
  - *Training Pipeline:* Orchestrates model training using frameworks like TensorFlow, PyTorch, or Spark ML, often managed by tools like Airflow or Kubeflow Pipelines.[1]
  - *Model Registry:* Versions and stores trained models and their associated metadata.[1]
  - *ANN Index Builder & Storage (for applicable systems):* Creates and stores Approximate Nearest Neighbor indexes from item embeddings for fast retrieval in candidate generation or visual search.[1]
- **Online Components:** These handle real-time requests and must operate with low latency.
  - *API Gateway:* Serves as the entry point for all client requests, handling routing, authentication, and rate limiting.[1]
  - *Real-time Event Stream & Stream Processor:* Platforms like Kafka and engines like Flink or Spark Streaming process incoming events (e.g., user clicks, transactions) to update real-time features or trigger actions.[1]
  - *Feature Store (Online Serving):* Provides low-latency access to features (often via a cache like Redis combined with persistent storage) needed for real-time predictions.[1]
  - *Candidate Generation Service (e.g., for Recommendations):* Queries an ANN index or other sources to retrieve an initial set of candidates.[1]
  - *Ranking/Prediction Service:* Hosts the trained ML model (e.g., LTR model, fraud detection model) using model serving frameworks (TensorFlow Serving, TorchServe, NVIDIA Triton) to score candidates or make predictions.[1]

- *Re-ranking Logic (Optional):* Applies business rules, diversity constraints, or fairness adjustments to the model's output before presenting it to the user.[1]
- *Monitoring System:* Collects logs and metrics from all online services for health checks and performance tracking.[1]

The two-stage architecture (candidate generation followed by ranking) is a particularly prevalent pattern in large-scale recommendation and feed systems. This design efficiently narrows down a vast corpus of items to a manageable subset for more computationally intensive, personalized ranking, thereby balancing comprehensiveness with low latency.[1]

Serving Predictions: Online (real-time) vs. Offline (batch) serving.
The choice between online and offline serving depends heavily on the application's latency and freshness requirements.

- **Online (Real-time) Serving:** Predictions are made on-demand as requests arrive. This is essential for applications like fraud detection (decision within milliseconds) [1] or dynamically updated ETAs.[1] Models are deployed as services, and feature fetching must also be low-latency.
- **Offline (Batch) Serving:** Predictions are pre-computed periodically (e.g., nightly) for all users or items and stored for later retrieval. This is suitable when real-time updates are not critical and can reduce online computation load.
- **Hybrid Serving:** Often, a combination is used. For instance, in recommendation systems, item embeddings and initial candidate sets might be generated offline (batch), while the final personalized ranking of these candidates happens online in real-time.[1]

The serving strategy is a direct consequence of the problem's requirements. Systems demanding immediate responses to fresh inputs necessitate online serving, while others can leverage batch precomputation for efficiency.

Scalability and Reliability:
Ensuring the system can handle increasing load (users, data, requests) and remain fault-tolerant is paramount.1

- **Scalability Techniques:**
  - *Horizontal Scaling:* Designing stateless services that can be replicated across multiple machines, with load balancers distributing traffic.[1] Microservices architectures inherently support this.[32]
  - *Distributed Data Processing:* Using frameworks like Apache Spark for large-scale ETL and feature engineering.[1]
  - *Efficient Indexing and Search:* For systems with large catalogs (e.g.,

recommendations, visual search), optimizing ANN search (e.g., FAISS, ScaNN) by tuning parameters or sharding the index is crucial.[1]
    - *Caching:* Employing caches (e.g., Redis, Memcached) for frequently accessed data like user embeddings, item features, or even pre-computed recommendations can significantly reduce latency and database load.[1]
- **Reliability Techniques:**
    - *Fault Tolerance:* Designing systems with redundancy, failover mechanisms, and robust error handling.
    - *Monitoring and Alerting:* Continuous monitoring of system health and performance to detect and address issues proactively.

Deployment Strategies:
Safely introducing new models or system updates into production requires careful deployment strategies to minimize risk and validate performance.1

- **Containerization:** Packaging models and their dependencies into containers (e.g., Docker) ensures consistent environments across development, testing, and production.[1]
- **Orchestration:** Using platforms like Kubernetes to automate the deployment, scaling, and management of containerized applications.[1]
- **Common Deployment Patterns:**
    - *Shadow Mode:* Deploying the new model alongside the current production model. The new model receives live traffic and makes predictions, but its outputs are only logged and compared against the production model, not shown to users. This allows for performance validation without impacting users.[1]
    - *Canary Releases (Gradual Rollout):* Routing a small percentage of live traffic (e.g., 1%, 5%) to the new model. If it performs well according to online metrics, the traffic percentage is gradually increased until all traffic is on the new model. This limits the blast radius of potential issues.[1]
    - *A/B Testing:* Exposing different user segments to different model versions simultaneously and comparing their performance on key online metrics. This is the standard for statistically validating the impact of changes.[1]
    - *Blue/Green Deployment:* Maintaining two identical production environments: Blue (current live) and Green (new version). Traffic is switched from Blue to Green after the Green environment is fully tested. Allows for quick rollback if issues arise.[1]
- **Model Registry:** A centralized system for versioning trained models, storing their metadata (e.g., training data, hyperparameters, evaluation metrics), and managing their lifecycle. This is crucial for reproducibility, governance, and

facilitating rollbacks if needed.[1]

Table 1.4.1: Comparison of Deployment Strategies

Strategy	Description	Pros	Cons	Use Cases
Shadow Mode	New model runs in parallel with the old, predictions logged but not served. [1]	No user impact, allows direct comparison of predictions, good for catching bugs, performance testing.	Doubles inference cost, doesn't measure actual user interaction with new model.	Initial validation of a new model's stability and basic performance before exposing it to users.
Canary / Gradual Rollout	Gradually shift a percentage of traffic to the new model. [1]	Limits risk/blast radius, allows monitoring impact on a small user base, enables rollback.	Slower rollout process, requires robust monitoring to detect issues early.	Safely introducing new models or significant changes, especially when uncertainty about impact is high.
A/B Testing	Expose different user segments to different models/features simultaneously and compare metrics. [1]	Statistically rigorous comparison of variants, measures actual impact on user behavior and business KPIs.	Requires careful experimental design, sufficient traffic for statistical significance, can be complex to manage.	Validating the effectiveness of new models, features, or UI changes; optimizing for specific online metrics.
Blue/Green Deployment	Deploy new version to a separate identical environment, then switch traffic. [1]	Instant rollout/rollback, minimizes downtime, full environment testing before switch.	Requires double the infrastructure resources, potential issues if environments aren't perfectly identical.	Deploying well-tested new versions where quick rollback capability is critical. Less common for iterative ML model updates than canary/A/B.

1.5. Step 5: Monitoring & Iteration

Conceptual Overview: Once an ML system is deployed, the work is far from over. This phase focuses on continuously tracking the system's performance in production, monitoring model accuracy, detecting data and concept drift, and establishing a robust feedback loop for ongoing improvement and model retraining.[1] ML systems are dynamic and can degrade over time if not actively maintained.

Monitoring in Production:
Comprehensive monitoring is essential to ensure the system operates reliably and effectively. Key areas to monitor include:

- **System Health Metrics:** These track the operational status and efficiency of the deployed infrastructure.
  - *Latency:* End-to-end response time for predictions, as well as latency of individual components (e.g., feature fetching, model inference). Monitoring tail latencies (P95, P99) is crucial as averages can hide issues affecting a subset of users.[1]
  - *Throughput:* Queries Per Second (QPS) or Transactions Per Second (TPS) handled by the services.[1]
  - *Error Rates:* HTTP error codes (e.g., 5xx server errors, 4xx client errors) from services, indicating failures or problems.[1]
  - *Resource Utilization:* CPU, memory, network I/O, and GPU utilization of serving instances and data processing jobs to identify bottlenecks and optimize costs.[1]
- **Online Model Performance Metrics:** These directly measure the real-world impact of the ML model on business objectives and user experience, typically tracked via A/B testing dashboards and live monitoring tools.
  - *Business/Engagement Metrics:* Metrics specific to the application, such as CTR, conversion rate, revenue, session duration for e-commerce [1]; fraud detection rate, false positive rate, financial impact for fraud systems [1]; like/comment/share rates, time spent for feeds [1]; MAE, RMSE for ETA predictions.[1]
  - *Guardrail/User Satisfaction Metrics:* Indicators of negative user experience, such as rates of users hiding recommendations, reporting content as irrelevant, high customer complaint volumes related to blocked transactions, or increased unsubscribe rates.[1]
- **Drift Detection:** Real-world data distributions and underlying concepts can change over time, causing model performance to degrade. This phenomenon,

known as drift, must be proactively monitored.

- *Data Drift:* Changes in the statistical properties of input features (e.g., shifts in the distribution of transaction amounts, changes in the popularity of certain product categories). This can be detected by comparing current data distributions to a reference window (e.g., training data) using statistical tests (Kolmogorov-Smirnov, Chi-Square) or distance metrics (Population Stability Index - PSI, Wasserstein distance).[1]
- *Concept Drift:* Changes in the underlying relationship between input features and the target variable (e.g., user preferences evolve, new fraud tactics emerge). This is often harder to detect directly without fresh labels. Proxy indicators include shifts in the model's output score distribution (prediction drift) or a sustained degradation in online business metrics after accounting for seasonality or external factors.[1]
- *Embedding Drift:* For systems using embeddings (e.g., recommendations, visual search), monitoring the distribution of learned user and item embeddings can indicate shifts in user preferences or item characteristics not captured by other features.[1]

  Models are not "set and forget" systems; drift is an inevitable consequence of operating in a dynamic environment. Proactive monitoring for various types of drift is essential for knowing when to retrain models or intervene to maintain performance.

- **Logging and Alerting:**
  - Implement comprehensive logging for all requests, predictions, errors, and key system events.[1]
  - Set up automated alerts for critical issues, such as system downtime, high error rates, latency spikes exceeding Service Level Objectives (SLOs), significant drops in online business metrics, or detection of substantial data or concept drift.[1] Tools like Prometheus/Grafana, Datadog, or specialized ML monitoring platforms (e.g., Evidently AI, WhyLabs, Arize) are commonly used.[1]

Table 1.5.1: Key Monitoring Areas & Metrics

Monitoring Area	Key Metrics	Examples	Tools/Techniques
System Health	Latency (P95/P99), Throughput (QPS/TPS), Error Rates (5xx, 4xx), Resource Utilization	API response time, requests processed per second, server error percentage, CPU load	Prometheus, Grafana, Datadog, CloudWatch, infrastructure monitoring tools [1]

	(CPU, Memory, GPU)		
Online Model Performance	Business KPIs (CTR, Conversion, Revenue), Engagement Metrics, Task-specific metrics (Fraud Rate, MAE), Guardrails	Click-through rate on recommendations, sales from search, user session length, false positive rate in fraud detection, rate of hidden feed items	A/B testing platforms, internal dashboards, analytics tools [1]
Data Drift	Feature distribution shifts, PSI, KS-test, Chi-Square test, Wasserstein distance	Change in average transaction amount, shift in distribution of user age, new categories appearing in product data	Evidently AI, WhyLabs, Arize, custom statistical monitoring scripts [1]
Concept Drift	Prediction score distribution shifts, online KPI degradation, offline metrics on recent labeled data	Model starts predicting consistently higher/lower scores, unexplained drop in CTR, decrease in offline NDCG on new data	ML monitoring platforms, A/B testing dashboards, periodic offline evaluation [1]
Fairness/Bias	Disparity in error rates (FP/FN) across groups, differences in prediction outcomes for protected attributes	Higher false positive rate for a specific demographic in fraud detection, lower recommendation relevance for a minority user group	Fairness Indicators (Google), Aequitas, custom dashboards tracking fairness metrics [1]

The Feedback Loop:
A robust feedback loop is essential for continuous improvement. This involves systematically collecting data on how the system performs in production and how users interact with it, and then using this information to inform subsequent iterations.1 User interactions—such as clicks, likes, purchases, shares, skips, hides, reports, or dwell time—are continuously logged and fed back into the data pipeline.1 This new data, especially when it includes labeled outcomes (e.g., confirmed fraudulent transactions, actual arrival times for ETA, items purchased after a recommendation), is invaluable for:

- Understanding real-world model performance.
- Identifying areas where the model is underperforming or where user needs are

not being met.
- Generating fresh, relevant data for model retraining.
  For instance, in fraud detection, establishing a fast and reliable mechanism to obtain labels for recent transactions (from chargeback data, analyst reviews, or customer reports) and incorporate them quickly into training data is paramount for adapting to new fraud schemes.1

Model Retraining and Updating:
ML models are not static artifacts; they require regular updates to maintain their relevance and accuracy in the face of changing data distributions, evolving user behaviors, new content or products, and shifting external factors.1

- **Frequency:** The optimal retraining frequency varies significantly depending on the specific application and the rate of change in the underlying data patterns.
  - Ranking models for dynamic environments like e-commerce recommendations or news feeds are often retrained daily or weekly.[1]
  - Candidate generation models (e.g., for embeddings) might be updated slightly less frequently.
  - Fraud detection models typically require very frequent retraining (daily, weekly, or even intra-day) due to the rapid evolution of fraud tactics and high concept drift.[1]
  - ETA prediction models also benefit from frequent batch retraining (e.g., daily or multiple times per day) to incorporate the latest traffic dynamics and road network changes.[1]
  - Visual search embedding models might be retrained less often (e.g., quarterly), but their associated ANN indexes need frequent updates (e.g., daily or multiple times a day) to reflect inventory changes.[1]
- **Method:**
  - *Batch Learning:* This is the standard approach. Models are retrained from scratch or fine-tuned using a recent window of historical data (e.g., the last 30-90 days). Batch learning ensures models learn from comprehensive data and are generally more stable.[1]
  - *Online Learning:* While full online retraining of complex models can be challenging and may introduce instability, some components (like user embeddings or real-time features in a feature store) might benefit from incremental online updates. This allows for faster adaptation to immediate user actions but is often used to supplement, rather than replace, periodic batch retraining.[1]
- **Triggers for Retraining:** Retraining can be initiated based on:
  - *Scheduled intervals:* The most common approach (e.g., daily, weekly).

    - *Performance degradation:* When monitoring detects a significant drop in online business metrics or offline evaluation metrics below a predefined threshold.[1]
    - *Drift detection:* When significant data drift or concept drift is identified.[1]
    - *Major system changes:* Such as the introduction of new product categories, significant platform feature updates, or changes in business strategy.
- **Automation (CI/CD for ML / MLOps):** The entire ML lifecycle, including data extraction, preprocessing, feature engineering, model training, evaluation, model registration, and deployment, should be automated using CI/CD pipelines (e.g., Jenkins, GitLab CI, AWS CodePipeline, Azure DevOps, SageMaker Pipelines). This automation, a core tenet of MLOps, ensures consistency, reliability, reproducibility, and enables rapid iteration.[1]

The retraining strategy adopted is ultimately a balance between the need for model freshness (to capture the latest patterns), model stability (to avoid erratic behavior), and operational cost (computational resources for retraining). This strategy must be justified based on the specific problem's dynamics and constraints.

Chapter 2: Common Machine Learning System Design Archetypes

This chapter delves into common archetypes of machine learning systems frequently encountered in interviews and real-world applications. For each archetype, a detailed walkthrough will be provided, applying the five-step framework discussed in Chapter 1. This includes understanding the specific problem nuances, typical data requirements, common modeling choices, architectural considerations, and monitoring strategies. Each section will also include a conceptual high-level design diagram to visually represent the system components and data flow. The discussions will draw heavily upon the examples and principles outlined in the provided research materials.[1]

2.1. Recommendation Systems (e.g., "Design a Netflix movie recommendation system")

Recommendation systems are ubiquitous, aiming to predict user preferences and suggest relevant items, such as movies, products, articles, or music.

Problem Formulation & Scope Definition:

- **Business Objective:** The primary goal is typically to increase user engagement (e.g., click-through rate (CTR) on recommendations, watch time, session length, items added to cart) and user retention.[1] Secondary goals might include increasing content diversity, promoting new or niche items, and ensuring fairness to content creators or sellers.[1] For a Netflix-like system, key objectives would be to maximize content consumption, improve user satisfaction, and reduce churn.
- **ML Task:** This is fundamentally a personalized ranking and recommendation problem. A common and effective approach, especially for large catalogs, is a **two-stage system**:
  1. **Candidate Generation (Retrieval):** Efficiently retrieve a manageable subset (e.g., hundreds or thousands) of potentially relevant items from the entire catalog for a given user. Speed is paramount here.[1]
  2. **Ranking (Scoring):** Take the smaller set of candidate items and precisely score/rank them based on predicted relevance or engagement likelihood for the specific user and context. This stage can afford more complex models and features.[1]
- **Clarifying Questions:**
  - *Scale:* How many users and items (e.g., millions of users, millions of movies/products)? What is the expected QPS for recommendations?.[1]
  - *Latency:* What are the latency requirements for serving recommendations (e.g., <200ms for real-time display)?.[1]
  - *Data Availability:* What user data (profiles, demographics), item data (metadata like genre, actors, descriptions, product attributes), and user-item interaction data (views, ratings, purchases, watch history, likes, shares) are available? Are real-time event streams of user activity accessible?.[1]
  - *Cold Start:* How should recommendations be handled for new users (no interaction history) and newly added items (no interaction data)?.[1]
  - *Diversity, Serendipity, Fairness:* Are there requirements to promote diverse content, avoid filter bubbles, introduce unexpected but relevant items (serendipity), or ensure fair exposure for different types of content/creators?.[1]
  - *Context:* Should recommendations consider user context like time of day, device, or current activity (e.g., recently watched movie)?

Data Deep Dive:

- **Data Sources:**
  - *User Profiles:* Demographics, preferences, subscription tier (for Netflix).
  - *Item Catalog:* Movie/show metadata (title, genre, cast, director, synopsis, release year, MPAA rating, image posters, trailers). Product metadata (name, category, brand, price, description, images).
  - *User-Item Interactions:* Explicit feedback (star ratings, thumbs up/down) and implicit feedback (watch history, percentage of movie watched, searches, clicks on recommended items, adding to watchlist, time spent on product page, purchases).[1]
  - *Real-time Event Streams:* Capturing current user activity (e.g., what the user is watching now, items just added to cart).
- **Feature Engineering:** A User-Item-Context framework is often used.[1]
  - *User Features:* User ID (for embeddings), historical engagement patterns (preferred genres, actors, directors), average watch time, explicit ratings history, demographics (age, location if available and ethical), user embeddings learned from past interactions. Real-time features like "genres watched in current session."
  - *Item Features:* Item ID (for embeddings), content metadata (genre embeddings, actor embeddings, textual features from synopsis/description processed with NLP, image features from posters), item popularity (global view counts, recent trending scores), item embeddings.
  - *Context Features:* Time of day, day of week, device type, current page (e.g., homepage vs. post-watch screen).
  - *Interaction Features (for Ranking):* User's historical interaction frequency/recency with item's genre/actors, similarity scores between user and item embeddings (e.g., dot product).
- **Feature Store:** Highly recommended for managing and serving these features with low latency, especially real-time features, and ensuring consistency between training and serving.[1]

Modeling:

- **Candidate Generation Models:**
  - *Collaborative Filtering (CF):*
    - *Item-Based CF:* "Users who watched X also watched Y." Based on co-occurrence patterns.[1]
    - *User-Based CF:* "Users similar to you also watched Z." Less scalable for many users.[1]

        - *Matrix Factorization (MF):* Techniques like SVD, ALS learn latent user and item embeddings from the interaction matrix. Handles sparse data well but suffers from cold start.[1]
    - *Content-Based Filtering:* Recommends items similar to what the user liked based on item metadata (e.g., recommend movies with the same genre/actors). Good for cold-start items and niche tastes.[1]
    - *Two-Tower Neural Networks:* Learn user and item embeddings in separate towers, optimized to bring embeddings of positive pairs closer. Efficient for retrieval using Approximate Nearest Neighbor (ANN) search.[1]
    - *Popularity-Based:* "Trending Now" or "Most Popular" sections. Simple, good for new users.
    - *Negative Sampling:* Since most feedback is implicit (views, not explicit dislikes for every unviewed item), strategies are needed to select or generate negative examples for training models like MF or Two-Tower.[1]
- **Ranking Models:**
    - *Learning-to-Rank (LTR):*
        - *Pointwise:* Predict an absolute score (e.g., P(watch), predicted rating) for each candidate item. Models: Logistic Regression, GBTs (XGBoost, LightGBM), Neural Networks.[1]
        - *Pairwise:* Predict which of two items is more relevant. Models: RankNet, LambdaRank.[1]
        - *Listwise:* Directly optimize a list-based metric like NDCG. Models: LambdaMART.[1]
    - *Deep Learning Models:* Architectures like Wide & Deep (combining memorization and generalization), DeepFM, DLRM can model complex feature interactions, often using embeddings from candidate generation or learning them end-to-end.[1]
- **Cold Start Handling:**
    - *New Users:* Start with popular items, content-based recommendations based on initial preferences (e.g., genre selection during sign-up), or demographic-based suggestions. Gradually incorporate collaborative signals as interaction data grows. Multi-Armed Bandit (MAB) strategies can balance exploration and exploitation.[1]
    - *New Items:* Rely on content-based features (genre, actors, synopsis for movies; category, description for products). Boost visibility to gather initial interactions.
- **Hybrid Models:** Combine CF and content-based approaches (e.g., using content features to initialize item embeddings in MF or as features in the ranking model) to leverage the strengths of both.[1]

- **Offline Evaluation Metrics:**
  - *Candidate Generation:* Recall@k (proportion of truly relevant items in the retrieved set).[1]
  - *Ranking:* Precision@k, Recall@k, MAP@k, NDCG@k. For classification tasks (e.g., P(click)), AUC-ROC, AUC-PR, LogLoss. For regression (rating prediction), MSE, MAE.[1]
  - *Beyond Relevance:* Metrics for Novelty, Diversity, Serendipity, Coverage.[1]

System Architecture & Deployment:

- **Architecture:** A hybrid deployment is common.[1]
  - *Offline Batch Processing:* For training candidate generation models (embedding learning), training ranking models, and building ANN indexes.
  - *Online Real-time Serving:*
    1. User request arrives.
    2. Fetch user embedding (from cache/DB) or compute real-time user features.
    3. **Candidate Generation Service:** Query ANN index (e.g., FAISS, ScaNN, Milvus) with user embedding to get top-K candidate item IDs.[1] Other sources like "most popular" or "recently added" can also contribute candidates.
    4. **Feature Fetching Service:** Retrieve features for the user and candidate items from a low-latency Feature Store.
    5. **Ranking Service:** The ranking model (deployed as a microservice using TF Serving, TorchServe, etc.) scores each candidate.[1]
    6. **Re-ranking Stage (Optional):** Apply business rules (filter watched items, ensure diversity, boost promoted content, de-duplicate).[1]
    7. Return final ranked list.
- **Scalability:** Horizontal scaling of stateless services (API gateway, ranking service, feature service). Optimized and potentially sharded ANN indexes. Caching for user embeddings, item features, or even full recommendation lists.[1]
- **Deployment Strategies:** Shadow deployment to compare new models with old ones without user impact. A/B testing (canary/gradual rollout) to validate online performance. Model Registry for versioning and managing models.[1]

Monitoring & Iteration:

- **System Health Metrics:** End-to-end recommendation latency (P95/P99), throughput (QPS), error rates of services, resource utilization.[1]
- **Online Model Performance Metrics:**
  - *Business/Engagement:* CTR on recommendations, conversion rate (e.g.,

starting to watch a movie, purchasing a product), revenue per user, session duration, retention rate.[1]
    - *Guardrails:* Rate of users hiding/downvoting recommendations, content diversity index, fairness metrics (e.g., exposure for different genres/creators).[1]
- **Drift Detection:**
    - *Data Drift:* Monitor distributions of key input features (e.g., user activity levels, popularity of genres).
    - *Concept Drift:* Track changes in the relationship between features and user engagement (e.g., are certain genres becoming more/less popular?). Monitor prediction score distributions. A sustained drop in online business metrics is a key indicator.
    - *Embedding Drift:* Monitor distributions of user and item embeddings for significant shifts.[1]
- **Feedback Loop:** Continuously log user interactions (views, ratings, clicks, skips, watch time) and feed this data back into the system for retraining and feature updates.[1]
- **Retraining Strategy:**
    - *Frequency:* Ranking models often retrained daily or weekly. Candidate generation models (embeddings) might be retrained weekly or bi-weekly. ANN indexes updated frequently (e.g., daily or even hourly) to reflect catalog changes (new movies/products, availability).[1]
    - *Method:* Primarily batch learning using recent historical data. Online learning might be used for near real-time updates to some features or user embeddings.[1]
    - *Triggers:* Scheduled intervals, significant performance degradation, detected drift, major catalog changes.[1]

Design Diagram: Recommendation System

Code snippet

```
graph TD
  subgraph Offline Pipeline
    UDB --> BATCH_ETL
    IDB --> BATCH_ETL
    LOGS[(Interaction Logs)] --> BATCH_ETL
    BATCH_ETL --> FEAT_OFFLINE(Feature Store - Offline)
```

```
    FEAT_OFFLINE --> TRAIN_CAND
    FEAT_OFFLINE --> TRAIN_RANK
    TRAIN_CAND --> MR_CAND(Model Registry - Candidate)
    TRAIN_RANK --> MR_RANK(Model Registry - Ranker)
    MR_CAND --> ANN_BUILD
    ANN_BUILD --> ANN_IDX(ANN Index Storage)
  end

  subgraph Online Serving
    USER_REQ --> API_GW{API Gateway}

    subgraph Candidate Generation Stage
      API_GW --> FEAT_USER_FETCH(Fetch User Features/Embedding)
      FEAT_USER_FETCH --> FEAT_ONLINE_USER(Feature Store - Online User Cache)
      FEAT_ONLINE_USER --> CAND_GEN_SVC
      ANN_IDX --> CAND_GEN_SVC
      OTHER_CANDS --> CAND_GEN_SVC
      CAND_GEN_SVC --> CANDIDATES((Candidate Items))
    end

    subgraph Ranking Stage
      CANDIDATES --> FEAT_ITEM_FETCH(Fetch Item Features for Candidates)
      FEAT_ITEM_FETCH --> FEAT_ONLINE_ITEM(Feature Store - Online Item Cache)
      FEAT_ONLINE_ITEM --> RANK_SVC
      MR_RANK --> RANK_SVC
      CANDIDATES --> RANK_SVC
      RANK_SVC --> RANKED_ITEMS((Ranked Items))
    end

    RANKED_ITEMS --> RERANK_FILTER
    RERANK_FILTER --> FINAL_RECS((Final Recommendations))
    FINAL_RECS --> API_GW
    API_GW --> USER_RESP
  end

  subgraph Real-time Updates
    USER_INTERACTION[User Interaction e.g., Click, View] --> EVENT_STREAM
    EVENT_STREAM --> STREAM_PROC
    STREAM_PROC --> FEAT_ONLINE_USER
```

```
    STREAM_PROC --> LOGS
  end

  subgraph Monitoring & Alerting
    CAND_GEN_SVC --> MONITOR
    RANK_SVC --> MONITOR
    RERANK_FILTER --> MONITOR
    MONITOR --> DASHBOARD
  end
```

Table 2.1.1: Key Design Choices for Recommendation Systems

Aspect	Choices/Techniques	Rationale/Trade-offs
Candidate Generation	Collaborative Filtering (Item-Item, User-User, Matrix Factorization), Content-Based, Two-Tower NN, Popularity-based	CF captures user preferences but cold start. Content handles cold start but risks overspecialization. Two-Tower is scalable for embedding-based retrieval. Popularity is a simple fallback. [1]
Ranking Model	Pointwise (LogReg, GBT, NN), Pairwise (RankNet), Listwise (LambdaMART), Deep Learning (Wide & Deep, DLRM)	Pointwise is simpler. Pairwise/Listwise better model ranking nature but more complex. Deep Learning captures complex interactions but needs more data/compute. [1]
Cold Start Strategy	Popularity, Content-based from initial preferences/metadata, Hybrid models, Exploration (e.g., MAB)	Balances providing some recommendation vs. learning new user/item signals. MAB helps manage exploration-exploitation. [1]
Key Evaluation Metrics	Offline: NDCG@k, MAP@k, Recall@k (candidates). Online: CTR, Conversion, Engagement Time, Retention, Diversity.	Offline metrics guide development. Online metrics are ground truth for business impact. Diversity/Novelty important for long-term satisfaction. [1]

Scalability Approach	Two-stage architecture, ANN for candidate retrieval, Distributed training/serving, Feature Store, Caching.	Manages large item/user scale. ANN for fast lookup. Feature Store for low-latency. Caching reduces load. [1]
Real-time Features	User activity in current session, recently trending items.	Improves responsiveness to immediate user intent but adds complexity to data pipelines and feature serving. [1]
Exploration vs. Exploitation	UCB, Epsilon-greedy, dedicated exploration slots, diversity in re-ranking.	Exploitation drives short-term metrics but can lead to filter bubbles. Exploration is key for discovery and long-term satisfaction but might temporarily lower engagement. [1]

The exploration-exploitation dilemma is central to recommendation systems. Continuously exploiting known user preferences by showing similar content reliably drives short-term engagement metrics but risks trapping users in "filter bubbles" and leading to monotony.[1] Conversely, exploring novel, diverse, or less popular content is vital for long-term user satisfaction, discovery of new interests, and platform health, even if it might temporarily decrease immediate engagement metrics.[1] System designers must therefore incorporate strategies to balance this, such as using algorithms like Upper Confidence Bound (UCB) or epsilon-greedy, dedicating specific slots in the UI for exploratory content, or enforcing diversity constraints during the re-ranking phase.

The two-stage architecture (candidate generation followed by ranking) is a fundamental pattern for building scalable and efficient recommendation systems, especially when dealing with massive item catalogs.[1] The first stage rapidly filters the vast item space down to a few hundred or thousand potentially relevant candidates. This allows the second stage to apply more computationally expensive and highly personalized ranking models to this smaller set, without being overwhelmed by the entire catalog. This separation of concerns is key to meeting low-latency requirements in large-scale deployments.

Finally, the quality and timeliness of the feedback loop significantly drive the personalization capabilities of a recommendation system. Rich interaction data, such as views, likes, shares, and precise watch/dwell times, directly fuels the learning

process for collaborative filtering and learning-to-rank models.[1] Incorporating real-time feature updates based on immediate user activity further enhances the system's responsiveness and relevance.[1] Consequently, the design of robust data ingestion pipelines and efficient feature engineering processes is as critical to the success of a recommendation system as the choice of the modeling algorithms themselves.

2.2. Search & Ranking (e.g., "Design a Yelp search ranking system")

Search and ranking systems are fundamental to information retrieval, enabling users to find relevant documents, products, or information from vast collections based on their queries.

Problem Formulation & Scope Definition:

- **Business Objective:** The primary goal is to enable users to find the products or information they are looking for quickly, easily, and effectively.[1] A successful search experience leads to increased product discovery, higher conversion rates (for e-commerce), greater revenue, and improved customer satisfaction and loyalty.[1] For a Yelp-like system, this means helping users find relevant local businesses, leading to user engagement (e.g., clicks on business profiles, calls, direction requests) and potentially advertising revenue. It's also crucial to balance user needs with various business objectives, such as maximizing profitability, managing inventory (for e-commerce), or promoting specific brands/listings.[1]
- **ML Task:** While basic keyword matching is a start, modern search heavily relies on Machine Learning, particularly **Learning-to-Rank (LTR)** techniques.[1] The task is, given a user's search query and their context, to rank the retrieved set of potentially relevant items (businesses, products, documents) in an order that optimizes for a combination of user relevance and business objectives.[1]
- **Clarifying Questions:**
    - *Query Types:* What types of queries will the system handle? For Yelp: specific business names ("Starbucks near me"), categories ("Italian restaurants in downtown"), symptom-based ("places for a birthday dinner"). For e-commerce: exact product names, general categories, problem-solving queries ("sunburn relief").[1]
    - *Scale:* How many queries per second? How large is the corpus of documents/businesses/products to search?

    - *Latency:* What is the acceptable search response time? (Typically very low, e.g., <200-500ms).
    - *Data Availability:* What data is available? Query logs (queries, clicks, conversions), document/business/product catalog (text descriptions, categories, attributes, location, images, reviews), user profiles (location, search history, preferences).[1]
    - *Personalization:* Should search results be personalized based on user history, location, or other context?
    - *Business Objectives:* How should relevance be balanced with business goals like promoting sponsored listings or higher-margin products?.[1]
    - *Ranking Factors:* Beyond textual relevance, what other factors should influence ranking (e.g., ratings, reviews, price, distance for local search, popularity)?

Data Deep Dive:

- **Data Sources:**
    - *Query Logs:* User search queries, clicked results, session information, conversions (e.g., booking a table, purchasing a product).[1]
    - *Document/Business/Product Corpus:* Detailed information about each item that can be searched. For Yelp: business name, category, address, phone, hours, textual description, user reviews, ratings, photos. For e-commerce: product titles, descriptions, specifications, categories, brand, price, images.[1]
    - *User Profiles:* User location (critical for local search), search history, interaction history, potentially demographics (used ethically).[1]
    - *User-Generated Content:* Reviews and ratings are vital signals for ranking in platforms like Yelp.[1]
- **Feature Engineering:** This is arguably the most critical component of a successful LTR system.[1] Features capture different aspects of relevance and desirability:
    - *Query Features:* Properties of the query itself (length, terms used, identified entities like brands or categories, predicted semantic intent, query type - navigational, informational, transactional).[1]
    - *Document/Item Features (Content-Based):*
        - *Text Relevance:* Scores from matching query terms to item titles, descriptions, reviews (e.g., TF-IDF, BM25 scores).[1]
        - *Attributes:* Category, brand, price, location (for businesses), specific attributes (size, color, cuisine type), image features, structured data markup.[1]
        - *Quality/Authority:* For businesses: average rating, number of reviews,

recency of reviews. For web documents: PageRank-like scores.
    - *User Features (Contextual/Personalization):* User's current location, device type, time of day, past search history, overall purchase/interaction history, inferred interests.[1]
    - *User-Item Interaction Features (Behavioral/Collaborative):*
        - Historical CTR or conversion rate for a specific item given the query (or for the user).
        - Add-to-cart actions, dwell time on the item page, user ratings/reviews for the item.[1]
    - *Popularity/Business Features:* Overall sales volume or visit frequency, recent trends (velocity), profit margin, inventory level, promotional status, advertising bids (for sponsored results).[1] For Yelp: check-ins, views of business page.
- **Feature Store:** Beneficial for serving features, especially real-time user context or interaction features, with low latency for personalized ranking.[1]

Modeling:

- **Retrieval (Candidate Generation):**
    - The first step is to retrieve a set of potentially relevant documents/items from the corpus that match the query.
    - *Traditional Methods:* Inverted indexes with keyword matching using algorithms like TF-IDF or BM25.[1]
    - *Semantic Search:* Incorporating techniques like word embeddings (Word2Vec, FastText) or transformer models (e.g., BERT-based sentence embeddings) to understand query intent beyond literal keyword matching, handling synonyms, related concepts, and natural language phrasing.[1] This is crucial for better query understanding.
- **Ranking (LTR):**
    - The retrieved candidates are then scored and ranked by an LTR model.
    - *LTR Algorithms:*
        - *Pointwise:* Predicts a relevance score for each query-item pair (e.g., Logistic Regression, GBTs).
        - *Pairwise:* Predicts which of two items is more relevant for a query (e.g., RankNet, LambdaRank).
        - Listwise: Directly optimizes a list-based metric like NDCG (e.g., LambdaMART, ListNet).
          Gradient Boosted Decision Trees (GBDTs), such as XGBoost and LightGBM, are widely used and highly effective in LTR due to their ability to handle diverse features and non-linear relationships.1 Deep learning models, potentially using frameworks like TensorFlow Ranking, offer

another powerful alternative.1

- **Training Data for LTR:**
  - Typically consists of (query, item, label) tuples. Labels are often derived from historical user interactions: e.g., item purchased = highest relevance, item clicked = medium relevance, item impressed but not clicked = low/no relevance.[1] For Yelp, a click on a business, a call, or a request for directions could be positive signals.
- **Offline Evaluation Metrics:**
  - *Ranking Metrics:* NDCG@k is the primary metric for assessing the quality of the ranked list. MAP@k and MRR are also commonly used.[1] Relevance judgments are derived from historical user interactions (judgment lists).[1]

System Architecture & Deployment:

- **Architecture:** A typical search system includes several key components [6]:
  - *Crawler/Indexer (for web search or large, dynamic content):* Collects and processes documents/items to build a searchable index. For platforms like Yelp or e-commerce, this involves indexing the business/product catalog.
  - *Query Processor:* Parses the user query, performs spelling correction, query expansion (synonyms, related terms), and intent understanding.
  - *Retrieval Engine:* Uses the processed query to fetch an initial set of candidate documents/items from the index.
  - *Feature Service:* Gathers features for the query, user, and candidate items.
  - *Ranking Engine:* Applies the LTR model to score and rank the candidates.
  - *(Optional) Re-ranking Stage:* Applies business rules, diversification, or promotion logic.
- **Serving:** Real-time ranking of retrieved candidates is essential for a good user experience.
- **Scalability:** Distributed indexes (e.g., using Elasticsearch, Solr, or custom solutions). Scalable retrieval and ranking services. Caching of popular queries or results.
- **Index Updates:** The search index needs to be updated regularly to reflect new or changed content (new businesses, updated product details, new reviews).

Monitoring & Iteration:

- **System Health Metrics:** Query latency (average and tail), indexing speed and freshness, error rates of services, throughput (QPS).[1]
- **Online Model Performance Metrics:**
  - *Relevance & Engagement:* CTR on search results, conversion rate (e.g., purchase, booking), time spent on SERP before clicking, query reformulation

rate (lower is better), SERP bounce rate (users leaving immediately).[1]
    - *Business Outcomes:* Revenue per search session, average order value from search.[1]
    - *User Experience (Guardrails):* Zero Result Rate (frequency of searches returning no results), search latency, customer satisfaction feedback (surveys).[1]
- **Drift Detection:**
    - *Query Drift:* Monitor changes in query distributions, popular search terms, query length/complexity.
    - *Document/Item Drift:* Monitor changes in the characteristics of the content being searched (e.g., new types of businesses, shifts in product attributes).
    - *Concept Drift:* Track changes in what users consider relevant for given queries (e.g., by monitoring CTR for stable queries over time).
- **Feedback Loop:** User interactions with search results (clicks, conversions, dwell time, query reformulations) are critical feedback for retraining LTR models and improving relevance signals.
- **Retraining Strategy:** LTR models are typically retrained regularly (e.g., daily or weekly) using recent query logs and interaction data. The search index needs frequent updates.

Design Diagram: Search & Ranking System

Code snippet

```
graph TD
    USER[User] -- Query --> QUERY_PROC
    QUERY_PROC -- Processed Query --> RETRIEVAL
    INDEX --> RETRIEVAL
    RETRIEVAL -- Candidate Items --> FEATURE_ENG

    subgraph Data Sources for Features
        USER_PROFILE_DB --> FEATURE_ENG
        ITEM_CATALOG_DB --> FEATURE_ENG
        INTERACTION_LOGS[(Interaction Logs)] --> FEATURE_ENG
        REALTIME_CONTEXT --> FEATURE_ENG
        FEATURE_STORE --> FEATURE_ENG
    end
```

```
  FEATURE_ENG -- Features for Query, User, Candidates --> RANKING_MODEL
  MODEL_REGISTRY --> RANKING_MODEL
  RANKING_MODEL -- Ranked Scores --> RERANK_FILTER
  RERANK_FILTER -- Final Ranked List --> RESULTS_FORMAT
  RESULTS_FORMAT --> USER

  subgraph Offline Training & Indexing
    RAW_DATA --> ETL
    ETL --> TRAINING_DATA
    TRAINING_DATA --> MODEL_TRAINING
    MODEL_TRAINING --> MODEL_REGISTRY
    RAW_DATA --> INDEXING_PIPELINE[Indexing Pipeline]
    INDEXING_PIPELINE --> INDEX
  end

  subgraph Monitoring
    QUERY_PROC --> MONITOR
    RETRIEVAL --> MONITOR
    RANKING_MODEL --> MONITOR
    RERANK_FILTER --> MONITOR
    MONITOR --> DASHBOARDS_ALERTS
  end

  USER -- Interaction Feedback --> INTERACTION_LOGS
```

Table 2.2.1: Key Design Choices for Search & Ranking Systems

Aspect	Choices/Techniques	Rationale/Trade-offs
Retrieval Strategy	Keyword-based (TF-IDF, BM25), Semantic Search (Embeddings, Transformers), Hybrid	Keyword is fast but misses semantic intent. Semantic improves relevance but is more complex/costly. Hybrid balances. [1]
Ranking Model (LTR)	Pointwise (GBTs), Pairwise (RankNet), Listwise (LambdaMART), Deep Learning models	GBTs are strong for tabular features. DL can capture complex non-linearities. Choice depends on data, scale, and desired complexity.

		[1]
Key Feature Types	Query features, Document/Item content features, User features (personalization), User-Item interaction features, Business features	A rich feature set is crucial for LTR. Balancing feature richness with computation cost and latency. [1]
Relevance Definition	Derived from clicks, conversions, dwell time, explicit ratings/reviews.	Implicit signals are abundant but noisy. Explicit signals are clearer but sparser. Defining relevance labels for LTR is key. [1]
Personalization	Incorporate user history, location, preferences as features in LTR.	Improves relevance for individual users but can create filter bubbles or privacy concerns if not handled carefully. [1]
Business Objectives Integration	Include business metrics (profit margin, promotion status) as features in LTR, or use multi-objective optimization, or re-ranking rules.	Directly optimizing for relevance might not align with business goals. Requires careful balancing to avoid user frustration. [1]
Query Understanding	Spelling correction, synonym expansion, intent classification, named entity recognition, semantic parsing.	Crucial for interpreting user needs accurately, especially for ambiguous or natural language queries. [1]

A core tension in e-commerce search, and often in other search domains, is balancing pure relevance (what the user is most likely looking for) against business objectives like profitability or inventory management.[1] Ranking solely on predicted relevance might not maximize profit or help clear overstocked items. Conversely, ranking solely on business metrics will likely frustrate users. LTR models address this by allowing business goals to be incorporated as features in the ranking function, enabling a tunable balance that can be optimized through experimentation.[1]

Query understanding is foundational to any effective search system. Accurately interpreting user intent requires sophisticated Natural Language Processing (NLP) capabilities to handle misspellings, synonyms, complex natural language queries, and

the different types of search intents (e.g., navigational, informational, transactional).[1] Implementing robust semantic search capabilities, which go beyond simple keyword matching, adds technical complexity but is often necessary for high-quality results.[1]

A potential unintended consequence of heavily relying on behavioral signals like CTR and conversions in LTR models is the creation of popularity feedback loops, often termed the "rich get richer" dynamic.[1] Products or items that are already popular tend to get ranked higher because their strong behavioral signals are highly predictive of future engagement. This higher ranking leads to even more visibility and interactions, further solidifying their top positions. Consequently, new products or items catering to niche tastes may struggle to gain the initial visibility needed to generate positive behavioral signals, making it difficult for them to surface even if they are highly relevant to certain queries. This can stifle product discovery and reduce the diversity of the accessible catalog over time. Strategies to inject diversity, promote new items, or explicitly mitigate popularity bias in the ranking algorithm are therefore important considerations.

2.3. Feed-based Systems (e.g., "Design the Twitter timeline")

Feed-based systems, such as social media timelines (Facebook, Twitter/X, Instagram, LinkedIn) or news aggregators, present users with a dynamic, ranked stream of content items.

Problem Formulation & Scope Definition:

- **Business Objective:** The primary goal is typically to maximize user engagement. Engagement can be defined by various actions like clicks, likes, comments, shares, re-posts, time spent viewing a post, or a weighted combination of these.[1] Secondary goals often include promoting content diversity, ensuring freshness of content, maintaining fairness in content exposure (e.g., for different creators or friends vs. pages), upholding content quality (reducing clickbait or misinformation), and fostering user well-being (avoiding echo chambers or excessive negativity).[1]
- **ML Task:** The core ML task is personalized ranking of feed items. Given a user, the system needs to rank a pool of candidate content items (e.g., posts, images, videos, links) to maximize the likelihood of engagement. This usually involves a multi-stage pipeline [1]:

1. **Candidate Retrieval (Source Selection):** Fetch potential feed items from various sources based on the user's connections (friends, followed pages/people), group memberships, and potentially recommended content from outside their network.
2. **Ranking (Scoring):** Score each candidate post based on its predicted relevance and engagement likelihood for the specific user in the current context.
3. **Re-ranking/Filtering:** Adjust the ranked list based on business rules, diversity requirements, fairness constraints, freshness boosts, removal of unwanted content, and impression capping (avoiding showing the same post repeatedly).

- **Clarifying Questions:**
  - *Define "Engagement":* Which specific actions constitute engagement? How are they weighted?.[1]
  - *Secondary Goals:* How important are diversity, freshness, fairness, content quality, and user well-being? How will these be measured?.[1]
  - *Scale:* Number of users (DAUs/MAUs), number of posts generated daily, QPS for feed generation?.[1]
  - *Latency:* How quickly must the feed load (e.g., <500ms)?.[1]
  - *Content Types:* Text posts, images, videos, links, stories, live streams? Do different types require different handling?.[1]
  - *Content Sources:* Posts from friends, followed entities, groups, algorithmic recommendations? How should these be balanced?.[1]
  - *Data Availability:* Access to user profiles (demographics, interests), social/follow graph, content metadata, historical user-post interaction data (impressions, clicks, likes, comments, shares, dwell time), real-time user context (time, location, device)?.[1]

Data Deep Dive:

- **Data Sources:**
  - *User Profile Database:* Demographics, user-stated interests, inferred interests from activity.
  - *Social Graph Database:* Information on user connections (friends, followers, following), group memberships.
  - *Content Database:* Stores all posts, including text, image/video metadata, URLs, creation timestamps, author information.
  - *Real-time Interaction Streams (e.g., Kafka, Kinesis):* Logs impressions, clicks, likes, comments, shares, hides, reports, dwell time on posts.[1]
  - *External Signals:* Trending topics, news events that might influence content

relevance.

- **Feature Engineering:** Features capture characteristics of the user, the post, their interaction history, and the current context.[1]
  - *User Features:* User ID, demographics (age, location), inferred topics of interest (from past engagement), overall activity level (e.g., posting frequency, comment rate), network size, user embeddings.
  - *Post Features:* Post ID, Author ID, Author Type (friend, page, group), Content Type (text, image, video), Content Embeddings (text embeddings from BERT/Word2Vec; image embeddings from CNNs like ResNet; video features), extracted topics/entities from content, post age (time since creation – crucial for freshness), post statistics (historical engagement rate like likes/impressions, recent engagement velocity – virality signal).
  - *Author Features:* Author's follower count, historical post performance, relationship strength/type to the viewing user (e.g., close friend, followed celebrity).
  - *User-Post Interaction Features (Critical for Personalization):* User's historical interaction with the author (frequency, recency, type of interaction – e.g., liked many posts from this author), user's historical interaction with similar content (same topic, same content type), similarity between user's inferred interests and post content (e.g., cosine similarity of embeddings), predicted probabilities of specific actions (P(like), P(comment)) from simpler baseline models.
  - *Contextual Features:* Time of day, day of week, device type, user location, network speed (relevant for video).
  - *Real-time Features:* User's activity in the current session (e.g., topics engaged with recently), number of times this post has already been shown to the user in recent sessions (for impression fatigue/capping).
- **Feature Store:** Highly beneficial for managing features, especially real-time user activity and post engagement signals, ensuring low-latency access for ranking and consistency.[1]

Modeling:

- **Candidate Retrieval:**
  - Candidates are sourced from various pools: recent posts from friends, posts from followed pages/groups, and potentially algorithmically recommended "discovery" content.[1]
  - This stage often uses heuristics (e.g., time cutoffs for freshness), simple rules, or lightweight models to efficiently gather a pool of hundreds or thousands of candidate posts.[1]

- **Ranking (LTR):**
  - *Models:* Similar to other ranking tasks, GBTs (XGBoost, LightGBM) are strong performers. Deep Learning models (MLPs, architectures like Wide & Deep, DeepFM, DLRM, or custom networks) are commonly used in large-scale feed ranking systems due to their ability to learn complex, non-linear interactions between high-dimensional sparse features (like user/post IDs) and dense features (like embeddings).[1]
  - *Training Data:* Examples are typically (user, post, features, label), where labels represent engagement events (e.g., click, like, comment, share, significant dwell time). Implicit feedback (impressions without positive engagement) requires careful handling, often through negative sampling strategies.[1]
- **Cold Start Handling:**
  - *New Users:* Initially rely on popular content within their network (if any connections exist), content popular in their region/demographic, or content related to interests explicitly stated during onboarding. Gradually explore user preferences.[1]
  - *New Posts:* Rely heavily on content features (text/image analysis), author features (e.g., author popularity), and initial reactions from the first few viewers. An "exploration" phase might show new posts to a small, diverse set of users to quickly gather engagement signals.[1]
- **Offline Evaluation Metrics:**
  - *Ranking Metrics:* NDCG@k, MAP@k, Precision@k, Recall@k (where relevance is based on historical engagement).[1]
  - *Classification Metrics (for pointwise models predicting specific interactions):* AUC-ROC, LogLoss, calibration metrics.[1]

System Architecture & Deployment:

- **Architecture:** A multi-stage request flow is typical.[1]
  1. User requests feed (e.g., opens app, pulls to refresh).
  2. **Candidate Retrieval Service(s):** Backend services fetch candidate posts from multiple sources (e.g., querying social graph DB for friends' posts, follow graph DB for pages' posts, group DBs, recommendation engine). This can be a significant "fan-out" challenge.
  3. **Feature Fetching Service:** Retrieves precomputed features (from Feature Store's online serving) and computes necessary real-time features for the user and all candidate posts. This must be very fast.
  4. **Ranking Service:** Receives user context and candidate posts with features. Uses the trained LTR model (hosted via TF Serving, TorchServe, etc.) to score each post.

5. **Re-ranking/Filtering Service:** Takes the scored list and applies post-processing: filters blocked/reported content, applies diversity rules (e.g., don't show too many posts from the same author consecutively), boosts fresh content, ensures fairness constraints are met, applies impression capping/fatigue rules.[1]
6. The final, ordered list of post IDs is returned to the client, which then fetches full content for display.

- **Scalability:** Addressing the "fan-out" challenge in candidate retrieval is key. Horizontal scaling of stateless microservices (candidate retrieval, feature fetching, ranking, re-ranking). Aggressive caching where possible, though personalization limits its effectiveness. High QPS demands on ranking and feature services.[1]
- **Deployment Strategies:** Rigorous A/B testing is mandatory for any changes. Gradual rollouts (canary releases) to minimize risk.[1]

Monitoring & Iteration:

- **System Health Metrics:** End-to-end feed load latency, latency of each stage (candidate retrieval, feature fetching, ranking, re-ranking), service availability and error rates, resource utilization.[1]
- **Online Model Performance Metrics:**
  - *Primary Engagement:* CTR, Like Rate, Comment Rate, Share Rate, Time Spent per session/post, Scroll Depth.[1]
  - *Secondary Goals/Guardrails:* Content diversity scores, freshness metrics (average age of content seen), fairness metrics (exposure/engagement for different creator types), negative feedback rates (hides, reports, unfollows, quick session abandonment).[1]
- **Drift Detection:**
  - *Data Drift:* Monitor distributions of user features (activity levels, interest distribution), post features (prevalence of video vs. image, topic trends), and interaction types (shift from likes to shares).[1]
  - *Concept Drift:* Track changes in the relationship between features and engagement outcomes. Monitor distribution of predicted ranking scores. Significant, unexplained changes in online engagement metrics are key indicators.[1]
- **Feedback Loop:** User interactions (positive and negative) are continuously logged and fed back into the data pipeline for subsequent training runs and feature updates.[1]
- **Retraining Strategy:** Feed ranking models require frequent updates. Regular batch retraining, often daily, is necessary to incorporate new interaction data, learn about new users/content, and adapt to evolving engagement patterns and

trends.[1]

Design Diagram: Feed-Based System

Code snippet

```
graph TD
    USER_APP[User Application e.g., Mobile App] -- Feed Request --> API_GW{API Gateway}

    subgraph Candidate Generation
        API_GW --> CAND_RETRIEVAL_SVC
        SOCIAL_GRAPH_DB --> CAND_RETRIEVAL_SVC
        FOLLOW_GRAPH_DB --> CAND_RETRIEVAL_SVC
        GROUP_DB --> CAND_RETRIEVAL_SVC
        REC_ENGINE --> CAND_RETRIEVAL_SVC
        CAND_RETRIEVAL_SVC -- Raw Candidates --> FEATURE_SVC
    end

    subgraph Feature Engineering & Storage
        FEATURE_SVC -- Needs Features for User & Candidates --> FEATURE_STORE_ONLINE(Feature Store - Online Cache e.g., Redis)
        USER_PROFILE_DB --> FEATURE_STORE_ONLINE
        CONTENT_DB --> FEATURE_STORE_ONLINE
        FEATURE_STORE_OFFLINE --> FEATURE_STORE_ONLINE
    end

    FEATURE_SVC -- Enriched Candidates with Features --> RANKING_SVC
    MODEL_REGISTRY --> RANKING_SVC
    RANKING_SVC -- Scored & Ranked Candidates --> RERANK_FILTER_SVC

    subgraph Business Logic & Policies
        DIVERSITY_RULES --> RERANK_FILTER_SVC
        FRESHNESS_RULES --> RERANK_FILTER_SVC
        FAIRNESS_RULES[Fairness Constraints] --> RERANK_FILTER_SVC
        IMPRESSION_CAPPING[Impression Capping] --> RERANK_FILTER_SVC
        CONTENT_POLICY[Content Policy Filters] --> RERANK_FILTER_SVC
```

```
    end

    RERANK_FILTER_SVC -- Final Ordered Feed Item IDs --> API_GW
    API_GW -- Feed Response --> USER_APP

    subgraph Offline Processing
        INTERACTION_LOGS[(User Interaction Logs)] --> ETL_PIPELINE
        ETL_PIPELINE --> FEATURE_STORE_OFFLINE
        ETL_PIPELINE --> TRAINING_DATA
        TRAINING_DATA --> MODEL_TRAINING
        MODEL_TRAINING --> MODEL_REGISTRY
    end

    subgraph Real-time Updates
        USER_APP -- Real-time Interactions --> EVENT_STREAM
        EVENT_STREAM --> STREAM_PROC
        STREAM_PROC --> FEATURE_STORE_ONLINE
        STREAM_PROC --> INTERACTION_LOGS
    end

    subgraph Monitoring
        CAND_RETRIEVAL_SVC --> MONITOR_SYS
        RANKING_SVC --> MONITOR_SYS
        RERANK_FILTER_SVC --> MONITOR_SYS
        MONITOR_SYS --> DASHBOARDS
    end
```

Table 2.3.1: Key Design Choices for Feed-Based Systems

Aspect	Choices/Techniques	Rationale/Trade-offs
Candidate Sources	Friends' posts, Followed entities, Group content, Algorithmic recommendations (discovery).	Balancing content from known connections with discovery. Managing fan-out for users with many connections. [1]
Ranking Objective(s)	Maximize specific engagement (likes, comments, shares, time spent), or a composite score.	Simple engagement can be gamed. Composite or multi-objective approaches better align with long-term

	Multi-objective ranking.	user satisfaction and platform health. [1]
Key Features	User history, post content (text/image/video embeddings), author features, user-post interactions, real-time context.	Rich, diverse features are needed for personalization. Real-time features improve responsiveness but add complexity. [1]
Diversity/Freshness Strategy	Re-ranking rules, explicit boosting of new content, ensuring variety in sources/topics.	Prevents filter bubbles and stale feeds. May slightly reduce immediate relevance for some users but improves overall experience. [1]
Fairness Considerations	Monitoring exposure for different creator types, re-ranking to mitigate biases.	Ensures equitable visibility, prevents algorithmic marginalization. May involve trade-offs with pure engagement optimization. [1]
Real-time Update Mechanism	Stream processing for new content and interactions, updating caches/feature stores. WebSockets for client updates.	Ensures feed reflects latest activity quickly. Adds infrastructural complexity. [1]
Impression Capping/Fatigue	Tracking impressions per user per post, down-ranking or filtering recently seen items.	Prevents users from seeing the same content repeatedly, improving experience. Requires state management. [1]

Perhaps the most significant challenge in designing feed-based systems is the tension between maximizing simple, short-term engagement metrics and fostering a healthy, responsible, and diverse content ecosystem.[1] Optimizing solely for clicks or immediate interactions can inadvertently amplify clickbait, sensationalism, or polarizing content, potentially leading to echo chambers, reduced content quality, and a decline in long-term user trust and satisfaction.[1] Therefore, a critical design consideration is the incorporation of secondary objectives—such as content diversity, freshness, fairness to creators, and the promotion of high-quality content—into the ranking function. This often requires defining complex, multi-objective ranking functions or applying carefully designed constraints and heuristics during the re-ranking stage. It's an ongoing area of research and ethical deliberation for platform

designers.

The "fan-out" problem is a notable scalability challenge in the candidate generation stage for social feeds.[1] When a user has thousands of friends or follows numerous pages, efficiently retrieving all potentially relevant recent posts from these myriad sources in real-time can be very demanding on backend systems. This necessitates optimized data storage for the social graph and follow relationships, efficient querying mechanisms, and potentially pre-computation or caching strategies for parts of the candidate set.

Real-time feature engineering plays a vital role in making feeds dynamic and highly relevant to a user's immediate context. Features such as "topics the user has engaged with in the current session" or "recent engagement velocity of a post" (indicating virality) allow the ranking algorithm to adapt quickly.[1] However, implementing these real-time features requires robust stream processing infrastructure and low-latency feature stores, adding to the system's architectural complexity and operational overhead. The benefits in terms of improved engagement and relevance must be weighed against these costs.

2.4. Fraud & Anomaly Detection (e.g., "Design a system to detect fraudulent credit card transactions")

Fraud and anomaly detection systems are critical for identifying and preventing malicious activities, unauthorized access, or unusual behaviors that deviate from normal patterns, thereby protecting users and businesses.

Problem Formulation & Scope Definition:

- **Business Objective:** The primary goal is twofold: 1) Minimize financial losses or other damages resulting from fraudulent activities (e.g., unauthorized transactions, account takeovers). 2) Minimize the disruption and negative experience for legitimate users caused by incorrectly flagging their activities as fraudulent (false positives).[1] Understanding the relative business cost of a False Negative (fraud missed) versus a False Positive (legitimate transaction blocked or user challenged) is crucial for tuning the system.[1]
- **ML Task:** This is most commonly framed as a **Binary Classification** problem: given the features associated with an event (e.g., a transaction, a login attempt), the model predicts one of two classes: "Fraudulent/Anomalous" or

"Legitimate/Normal".[1] Alternatively, especially when labeled fraud data is scarce or novel fraud patterns are expected, it can be approached as an **Anomaly Detection** (unsupervised learning) task, aiming to identify events that deviate significantly from established normal behavior patterns.[1] Often, a risk scoring output is desired, where the model provides a continuous score indicating the likelihood of fraud, allowing for flexible thresholding.[1]

- **Clarifying Questions:**
  - *Type of Fraud/Anomaly:* What specific type of fraud is being targeted (e.g., credit card transaction fraud, account takeover, insurance claim fraud, fake account creation)? The features and patterns will differ.
  - *Scale:* What is the volume of events to be processed (e.g., transactions per second, login attempts per day)?.[1]
  - *Latency:* Is real-time detection required (e.g., for transaction authorization, typically tens to hundreds of milliseconds)? Or can detection be done in near real-time or batch (e.g., for post-event analysis)?.[1]
  - *Data Availability:* What data is available for each event in real-time (e.g., transaction details, user account history, device fingerprint, IP address)? Is there historical labeled data (fraud/not fraud), and how reliable are these labels?.[1]
  - *Interpretability:* How important is it to explain why an event was flagged as fraudulent (e.g., for regulatory compliance, customer service, analyst review)?.[1]
  - *Adaptability:* Fraudsters constantly evolve their tactics. How quickly does the system need to adapt to new and emerging fraud patterns?.[1]

Data Deep Dive:

- **Data Sources:**
  - *Real-time Event Stream:* Transaction data from payment gateways, login attempts from authentication systems, user activity logs (often via Kafka or similar streaming platforms).[1]
  - *User/Account Database:* Cardholder profiles, historical spending/login patterns, account tenure, security settings.[1]
  - *Merchant/Entity Database:* Information about merchants, payees, or other entities involved in transactions.
  - *Device/Contextual Data:* IP address, device fingerprint (OS, browser, hardware identifiers), User-Agent string, geolocation data.[1]
  - *Third-Party Data Providers:* Services offering IP reputation scores, device risk scores, geolocation validation, known fraudulent identifiers.[1]
  - *Historical Labeled Data:* Logs of past events definitively identified as

fraudulent or legitimate (e.g., from chargebacks, confirmed ATO incidents, analyst investigations).[1]

- **Feature Engineering:** This is crucial for fraud detection, as features aim to capture deviations from normal behavior or known fraudulent patterns.[1]
  - *Event-Specific Features:* Transaction amount, currency, merchant category, time of day, day of week, type of transaction (online, POS), login success/failure.
  - *User/Account Historical Features:* Average spending frequency/amount, typical merchant categories used, typical login times/locations, account age, time since last password change, number of devices used historically.
  - *Behavioral Analytics Features:* Deviations from established baseline behavior for a specific user (e.g., unusually large transaction, login from a new country/device).[1]
  - *Device Fingerprinting Features:* Uniqueness of device, known association with fraud, consistency with user's typical devices.[1]
  - *Session/Contextual Features (Real-time):* IP address geolocation, IP reputation, consistency of IP with user's history, time since last transaction/login from this user/device, number of failed login attempts prior to success.
  - *Velocity Features (Real-time Aggregates):* Number/amount of transactions by user/card in the last N minutes/hours/days; number of distinct merchants/countries transacted with recently; transaction amount compared to user's short-term average.[1]
  - *Graph Features (Advanced):* Constructing graphs connecting users, devices, IP addresses, merchants, payment instruments. Features derived from graph analysis (e.g., community detection to find fraud rings, centrality measures, shared neighbors) can capture sophisticated, coordinated fraud patterns but add complexity.[1]
- **Feature Store:** A low-latency online feature store is almost mandatory for real-time fraud detection. It allows the system to quickly retrieve precomputed historical aggregates (e.g., user's average spend) and access recently updated real-time features (like velocity counts) during inference, minimizing lookup latency which is critical for decision-making.[1]

Behavioral analytics and device fingerprinting are particularly important. Behavioral analytics involves establishing a baseline profile of typical behavior for each user (e.g., common login times, geographic locations, transaction types) and flagging significant deviations as potentially risky.[1] Device fingerprinting analyzes various attributes of the device used for access (OS, browser, screen resolution, IP address) to create a

relatively unique "fingerprint"; changes in this fingerprint or connections from devices known to be associated with fraud can indicate an attack.[1]

Modeling:

- **Model Selection:**
  - *Supervised Models (if labeled data is sufficient):*
    - *Tree-based Models:* Gradient Boosted Trees (XGBoost, LightGBM, CatBoost) are very popular and often achieve state-of-the-art performance on the structured, tabular data typical in fraud detection. They handle mixed feature types well and can provide feature importance.[1]
    - *Deep Learning:* Neural Networks (MLPs) can capture complex non-linear interactions. RNNs might be used for sequences of transactions or user actions. Graph Neural Networks (GNNs) can learn directly from relational structures if graph features are central to the strategy.[1]
    - *Logistic Regression, SVMs:* Can serve as baselines or components in ensembles.[1]
  - *Unsupervised Models (Anomaly Detection - for novel fraud or scarce labels):*
    - *Autoencoders:* Trained to reconstruct normal behavior; high reconstruction error flags anomalies.[1]
    - *Isolation Forests:* Efficiently identify outliers.[1]
    - *One-Class SVMs:* Learn a boundary around normal data.[1]
  - *Rule-Based Systems:* Often used in conjunction with ML models. Simple, interpretable rules can catch obvious fraud patterns, act as overrides, or form a first layer of defense.[1]
  - *Ensemble Methods:* Combining predictions from multiple diverse models (e.g., GBT + MLP + Rules, or supervised + unsupervised scores) can often improve robustness and overall performance.[1]
- **Handling Data Imbalance:** Fraudulent transactions are typically rare, leading to highly imbalanced datasets. This is a critical challenge.[1]
  - *Data-level Approaches (Resampling):*
    - *Undersampling:* Randomly remove samples from the majority (non-fraud) class. Risk: may discard useful information.[1]
    - *Oversampling:* Duplicate samples from the minority (fraud) class or generate synthetic minority samples using techniques like SMOTE (Synthetic Minority Over-sampling Technique). Risk: may lead to overfitting on the minority class.[1]
    - Apply resampling carefully, usually only to the training set, not validation/test sets.[1]

    - *Algorithmic Approaches:*
        - *Cost-Sensitive Learning:* Assign a higher misclassification cost to the minority (fraud) class during model training, forcing the model to pay more attention to correctly identifying fraud.[1]
        - *Threshold Moving:* Train the model normally, then adjust the classification threshold (default is often 0.5) on the output probability to achieve the desired balance between precision and recall based on the validation set or cost analysis.[1]
- **Offline Evaluation Metrics:** Standard accuracy is highly misleading.[1]
    - *Key Metrics:* Precision (for the Fraud class), Recall (Sensitivity, True Positive Rate for Fraud), F1-Score (for Fraud). These focus on the performance related to the minority (fraud) class.[1]
    - *AUC-PR (Area Under the Precision-Recall Curve):* Generally more informative than AUC-ROC for highly imbalanced datasets as it focuses on the performance of the minority class.[1]
    - *Confusion Matrix:* Detailed breakdown of True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN).[1]
    - *Cost-Based Metric:* A custom metric incorporating the estimated financial cost of a False Negative and the cost of a False Positive can guide optimization.[1]
    - *Fairness Metrics:* False Positive Rate Parity and False Negative Rate Parity across different user groups to monitor for biased error rates.[1]

System Architecture & Deployment:

- **Architecture:** Real-time, synchronous prediction service is typically required, as the transaction authorization process waits for the fraud score.[1]
    1. Transaction request received by authorization system.
    2. Request forwarded to Fraud Detection System (often via an API Gateway).
    3. **Feature Enrichment Service:** Gathers/computes features in real-time (using stream processing on incoming events and lookups to the online Feature Store).
    4. **Prediction Service:** Hosts the trained ML model(s) and generates a fraud score/probability.
    5. **Decision Engine:** Applies the classification threshold, potentially combines the ML score with business rules, anomaly scores, or outputs from other models.
    6. Decision (Approve, Deny, Step-up Authentication/Review) returned to the authorization system.[1]
- **Infrastructure:** Highly available and fault-tolerant infrastructure is

non-negotiable. Optimized model serving containers (e.g., using C++ implementations, ONNX runtime, TensorRT for NNs) deployed on scalable platforms like Kubernetes. Low-latency access to the Feature Store is critical.[1]

- **Deployment Strategies:**
  - *Shadow Mode:* Essential for running new models in parallel with the production model, logging predictions without affecting decisions, to compare performance and behavior extensively before switching traffic.[1]
  - *Blue/Green or Canary Deployments:* Used to roll out new model versions cautiously.[1]
  - *A/B Testing:* Might be used to compare different models or thresholds on a subset of traffic, carefully monitoring business metrics.[1]

Monitoring & Iteration:

- **System Health Metrics:** End-to-end Latency (P95, P99 are critical), Throughput (TPS), Service Error Rates, Resource Utilization. Monitor latency contributions of each step (feature enrichment, model inference).[1]
- **Online Model Performance Metrics:**
  - Track Fraud Detection Rate (Recall) and False Positive Rate as close to real-time as possible (requires a feedback loop for labels from chargebacks, manual reviews, user reports, which may have a delay).[1]
  - Monitor the distribution of fraud scores generated by the model for both flagged and non-flagged transactions. Shifts can indicate problems.
  - Track financial impact ($ saved vs. $ lost).
  - Monitor fairness guardrails like FP/FN Rate Disparity across sensitive groups.[1]
- **Drift Detection:** Fraud patterns change rapidly.
  - *Data Drift:* Monitor distributions of critical input features (e.g., transaction amount distribution, merchant category frequencies, IP geolocation patterns, velocity feature distributions). Sudden shifts can signal new legitimate behavior patterns or new fraud attack vectors.[1]
  - *Concept Drift:* Monitor the relationship between features and the likelihood of fraud. Track changes in performance metrics (Precision, Recall, AUC-PR) over time on recent data. Watch for increases in fraud rates for transaction types previously considered low-risk.[1]
  - *Adversarial Monitoring:* Look for unusual or suspicious patterns in input data that might indicate attempts to deliberately fool the model.
- **Feedback Loop:** A crucial component is establishing a fast and reliable feedback loop to get labels for recent transactions (from chargebacks, analyst reviews, customer reports of fraud or false positives) and incorporate them into the training data quickly.[1]

- **Retraining Strategy:** Frequent model updates are necessary to combat evolving fraud tactics.
    - *Frequency:* Due to the high rate of concept drift in fraud, models often need frequent retraining – potentially daily, weekly, or even intra-day in highly dynamic environments.[1]
    - *Method:* Regular batch learning using the latest available labeled data is most common. Online learning techniques could offer faster adaptation but can be less stable; a hybrid approach (periodic full batch retraining with more frequent online fine-tuning) might be used.[1]
    - *Champion/Challenger Framework:* Continuously train and evaluate new candidate models (challengers) against the current production model (champion) using offline metrics and shadow deployment before promotion.[1]

Design Diagram: Fraud Detection System

Code snippet

```
graph TD
    subgraph Real-time Transaction Flow
        TXN_SOURCE -- Transaction Event --> EVENT_STREAM
        EVENT_STREAM --> STREAM_PROC

        subgraph Data Sources for Enrichment
            USER_DB --> STREAM_PROC
            MERCHANT_DB --> STREAM_PROC
            THIRDPARTY_API --> STREAM_PROC
            FEATURE_STORE_ONLINE(Feature Store - Online e.g., Redis) --> STREAM_PROC
        end

        STREAM_PROC -- Enriched Transaction + Features --> PREDICTION_SVC
        MODEL_REGISTRY --> PREDICTION_SVC
        PREDICTION_SVC -- Fraud Score/Probability --> DECISION_ENGINE

        subgraph Business Rules & Thresholds
            RULES_DB --> DECISION_ENGINE
            THRESHOLDS --> DECISION_ENGINE
```

```
    end

    DECISION_ENGINE -- Decision (Approve/Deny/Review) --> AUTH_SYSTEM
    AUTH_SYSTEM -- Response --> TXN_SOURCE
  end

  subgraph Offline Model Training & Feedback
    HISTORICAL_LOGS --> BATCH_ETL
    USER_DB_BATCH --> BATCH_ETL
    MERCHANT_DB_BATCH --> BATCH_ETL
    FEATURE_STORE_OFFLINE --> BATCH_ETL

    BATCH_ETL --> TRAINING_DATA
    TRAINING_DATA --> MODEL_TRAINING
    MODEL_TRAINING -- Trained Model --> MODEL_REGISTRY

    MANUAL_REVIEW -- Fraud Labels --> LABEL_STORE
    LABEL_STORE --> HISTORICAL_LOGS
    LABEL_STORE --> BATCH_ETL
  end

  subgraph Monitoring & Alerting
    EVENT_STREAM --> MONITOR_SYS
    PREDICTION_SVC --> MONITOR_SYS
    DECISION_ENGINE --> MONITOR_SYS
    MONITOR_SYS --> DASHBOARDS
  end
```

Table 2.4.1: Key Design Choices for Fraud/Anomaly Detection Systems

Aspect	Choices/Techniques	Rationale/Trade-offs
Model Type	Supervised (GBTs, DL, Ensembles), Unsupervised (Autoencoders, Isolation Forest), Rule-based, Hybrid.	Supervised needs labels but good for known patterns. Unsupervised for novel fraud. Rules for clarity. Hybrid often best. [1]
Imbalance Handling	Resampling (Undersampling, SMOTE), Cost-Sensitive	Crucial due to rare fraud class. Resampling alters data.

	Learning, Threshold Moving.	Cost-sensitive/thresholding adjusts model focus. [1]
Key Features	Transaction details, User history, Device/IP info, Velocity counters, Behavioral biometrics, Graph features.	Rich, diverse features are essential. Velocity and behavioral features capture dynamic risk. Graph features for complex networks. [1]
Latency Target	Hard Real-time (tens to hundreds of ms) for transaction authorization.	Drives model complexity, feature engineering choices, and infrastructure. Simpler models or optimized inference needed. [1]
Interpretability Need	SHAP/LIME for complex models, or use inherently interpretable models (Rules, Logistic Regression, Decision Trees).	Required for analysts, customer service, regulatory compliance. Trade-off with predictive power of complex models. [1]
Adaptability Strategy	Frequent retraining, online learning components, robust drift monitoring, champion/challenger.	Fraud patterns evolve rapidly. System must adapt quickly. Fast feedback loop for labels is critical. [1]
False Positive vs. False Negative Cost	Business-defined trade-off, reflected in metric choice (Precision vs. Recall) and threshold tuning.	High FPs frustrate users. High FNs mean financial loss. Optimal balance depends on business risk appetite. [1]

The paramount trade-off in real-time fraud detection is often between prediction latency and model complexity/accuracy.[1] More sophisticated models (e.g., deep GNNs) or features requiring extensive real-time computation (e.g., complex aggregations, graph traversals) might improve detection accuracy but risk violating the strict millisecond latency budget required for synchronous transaction authorization. This necessitates careful feature selection, model optimization for speed (e.g., quantization, pruning), and highly efficient serving infrastructure.

Adaptability to evolving fraud tactics is non-negotiable.[1] Fraudsters are constantly innovating, meaning models trained on historical data can quickly become outdated (concept drift). The system must therefore be designed for rapid iteration, including frequent model retraining with the latest labeled fraud data, continuous monitoring for

drift, and potentially the use of unsupervised anomaly detection methods to flag entirely new or unseen fraud patterns.[1] A fast and reliable feedback loop to obtain labels for recent transactions (e.g., from chargebacks, analyst reviews) is critical for this adaptive capability.

Finally, the business decision regarding the relative costs of false positives versus false negatives profoundly influences the system's design and tuning.[1] Incorrectly blocking a legitimate transaction (a false positive) can lead to significant customer dissatisfaction and lost business. Conversely, failing to detect a fraudulent transaction (a false negative) results in direct financial loss. The acceptable balance between these two types of errors, often reflected in the choice of primary evaluation metric (e.g., prioritizing recall to catch more fraud vs. precision to reduce false alarms) and the setting of the model's decision threshold, must be explicitly defined in collaboration with business stakeholders. Different thresholds or even different models might be employed for various customer segments or transaction types based on their inherent risk profiles.

Chapter 3: Advanced Topics & Considerations

Beyond the core framework and common archetypes, designing modern, production-grade machine learning systems requires a deep understanding of several advanced topics. These include the operational discipline of MLOps, the critical importance of ethics, fairness, and bias mitigation, techniques for building systems that can scale to handle massive loads, and the increasingly recognized role of human intelligence in augmenting and refining ML capabilities. This chapter explores these vital considerations.

3.1. MLOps: A practical guide to the tools and best practices for automating the machine learning lifecycle.

Conceptual Overview:
MLOps (Machine Learning Operations) is a set of practices, principles, and a culture that aims to deploy and maintain machine learning models in production reliably, efficiently, and at scale.39 Inspired by DevOps principles from software engineering, MLOps addresses the

unique challenges of the ML lifecycle, such as managing complex data dependencies, ensuring model reproducibility, handling model and data drift, and scaling training and serving infrastructure.39 It fosters collaboration and bridges the gap between data scientists (who primarily develop models), ML engineers (who productionize models), and IT/operations teams (who manage infrastructure).34

Key Pillars/Components of MLOps:

A robust MLOps strategy incorporates several key components that work together to streamline the end-to-end ML workflow:

- **Version Control:** This is fundamental for reproducibility and traceability. It extends beyond just code (typically managed with Git) to include:
  - *Data Versioning:* Tools like DVC (Data Version Control), Pachyderm, or lakeFS allow tracking changes to datasets, ensuring that experiments and model training runs can be reproduced with the exact data they used [[20],

Works cited

1. Product Design Questions.pdf
2. ML System Design in a Hurry - Hello Interview, accessed June 11, 2025, https://www.hellointerview.com/learn/ml-system-design
3. How to Answer ML System Design Questions - Exponent, accessed June 11, 2025, https://www.tryexponent.com/courses/ml-system-design/mlsd-framework
4. Machine Learning System Design Interview (2025 Guide) - Exponent, accessed June 11, 2025, https://www.tryexponent.com/blog/machine-learning-system-design-interview-guide
5. System Design Delivery Framework - Hello Interview, accessed June 11, 2025, https://www.hellointerview.com/learn/ml-system-design/in-a-hurry/delivery
6. Aman's AI Journal • Intro to ML Design, accessed June 11, 2025, https://aman.ai/sysdes/intro/
7. ML Systems Design Interview Guide - Patrick Halina, accessed June 11, 2025, http://patrickhalina.com/posts/ml-systems-design-interview-guide/
8. End-to-end Machine Learning Workflow - ML-Ops.org, accessed June 11, 2025, https://ml-ops.org/content/end-to-end-ml-workflow
9. Bias in AI - Chapman University, accessed June 11, 2025, https://www.chapman.edu/ai/bias-in-ai.aspx
10. What is AI bias? Causes, effects, and mitigation strategies - SAP, accessed June 11, 2025, https://www.sap.com/resources/what-is-ai-bias
11. Feature Engineering: Scaling, Normalization, and Standardization - GeeksforGeeks, accessed June 11, 2025, https://www.geeksforgeeks.org/ml-feature-scaling-part-2/
12. What is Human-in-the-Loop (HITL) in AI & ML? - Google Cloud, accessed June 11, 2025, https://cloud.google.com/discover/human-in-the-loop
13. What is Human-in-the-loop? | TELUS Digital, accessed June 11, 2025, https://www.telusdigital.com/glossary/human-in-the-loop

14. Human-in-the-Loop Machine Learning (HITL) Explained - Encord, accessed June 11, 2025, https://encord.com/blog/human-in-the-loop-ai/
15. What is Human-in-the-Loop Annotation, & Why Does It Matter? - SoftAge AI, accessed June 11, 2025, https://softage.ai/blog/what-is-human-in-the-loop-annotation-and-why-it-matters-explained/
16. How Human-in-the-Loop Boosts Performance of AI-driven Data Annotation? | Infosys BPM, accessed June 11, 2025, https://www.infosysbpm.com/blogs/annotation-services/how-human-in-the-loop-boosts-performance-of-ai-driven-data-annotation.html
17. Recommendation Systems in Machine Learning - Appinventiv, accessed June 11, 2025, https://appinventiv.com/blog/recommendation-system-machine-learning/
18. Fairness - AI Ethics Lab, accessed June 11, 2025, https://aiethicslab.rutgers.edu/e-floating-buttons/fairness/
19. Addressing AI Bias and Fairness: Challenges, Implications, and Strategies for Ethical AI, accessed June 11, 2025, https://smartdev.com/addressing-ai-bias-and-fairness-challenges-implications-and-strategies-for-ethical-ai/
20. Best Data Versioning Tools for MLOps| Coralogix Blog, accessed June 11, 2025, https://coralogix.com/ai-blog/best-data-versioning-tools-for-mlops/
21. Experiment Tracking in Machine Learning - Everything You Need to Know - viso.ai, accessed June 11, 2025, https://viso.ai/deep-learning/experiment-tracking/
22. What is Model Registry in Machine Learning? The Ultimate Guide in 2024 - Qwak, accessed June 11, 2025, https://www.qwak.com/post/what-is-model-registry
23. Machine Learning for Fraud Detection: An In-Depth Overview - Itransition, accessed June 11, 2025, https://www.itransition.com/machine-learning/fraud-detection
24. How machine learning works for payment fraud detection and prevention - Stripe, accessed June 11, 2025, https://stripe.com/resources/more/how-machine-learning-works-for-payment-fraud-detection-and-prevention
25. ml-system-design/ranking.md at main - GitHub, accessed June 11, 2025, https://github.com/ifding/ml-system-design/blob/main/ranking.md
26. Distributed Training: Guide for Data Scientists - neptune.ai, accessed June 11, 2025, https://neptune.ai/blog/distributed-training
27. Scaling Deep Learning with Distributed Training: Data Parallelism to Ring AllReduce, accessed June 11, 2025, https://mirza.im/posts/2024-08-11-distributed-training-data-parallelism/
28. What Is Model Parallelism? | Pure Storage, accessed June 11, 2025, https://www.purestorage.com/knowledge/what-is-model-parallelism.html
29. Paradigms of Parallelism | Colossal-AI, accessed June 11, 2025, https://colossalai.org/docs/concepts/paradigms_of_parallelism/
30. Recommendation systems overview | Machine Learning - Google for Developers, accessed June 11, 2025, https://developers.google.com/machine-learning/recommendation/overview/type

s
31. Blueprints for recommender system architectures: 10th anniversary edition, accessed June 11, 2025, http://amatria.in/blog/RecsysArchitectures
32. When ML Meets Microservices: Engineering for Scalability and Performance - HackerNoon, accessed June 11, 2025, https://hackernoon.com/when-ml-meets-microservices-engineering-for-scalability-and-performance
33. Microservices-based architecture: Scaling enterprise ML models - Sigmoid, accessed June 11, 2025, https://www.sigmoid.com/blogs/microservices-based-architecture-key-to-scaling-enterprise-ml-models/
34. Mastering MLOps: Key Benefits and Practices for 2025 - Appquipo, accessed June 11, 2025, https://appquipo.com/blog/a-complete-guide-to-mlops/
35. What is a ML Model Registry? - JFrog, accessed June 11, 2025, https://jfrog.com/learn/mlops/model-registry/
36. The Measure and Mismeasure of Fairness: A Critical Review of Fair Machine Learning | Columbia | CPRC, accessed June 11, 2025, https://cprc.columbia.edu/events/measure-and-mismeasure-fairness-critical-review-fair-machine-learning
37. Fairness (machine learning) - Wikipedia, accessed June 11, 2025, https://en.wikipedia.org/wiki/Fairness_(machine_learning)
38. System Design of Google Search Engine | Deep Notes - Deepak's Wiki, accessed June 11, 2025, https://deepaksood619.github.io/computer-science/interview-question/system-design-google-search/
39. MLOps in 2025: What You Need to Know to Stay Competitive - HatchWorks, accessed June 11, 2025, https://hatchworks.com/blog/gen-ai/mlops-what-you-need-to-know/
40. What is CI/CD for Machine Learning | Iguazio, accessed June 11, 2025, https://www.iguazio.com/glossary/ci-cd-for-machine-learning/
41. What is (CI/CD) for Machine Learning? - JFrog, accessed June 11, 2025, https://jfrog.com/learn/mlops/cicd-for-machine-learning/
42. Fairness: Mitigating bias | Machine Learning - Google for Developers, accessed June 11, 2025, https://developers.google.com/machine-learning/crash-course/fairness/mitigating-bias
43. The Seven Pillars Of MLops - Cyber Security Intelligence, accessed June 11, 2025, https://www.cybersecurityintelligence.com/blog/the-seven-pillars-of-mlops-8408.html
44. Design Patterns for Machine Learning Based Systems with Human-in-the-Loop | Request PDF - ResearchGate, accessed June 11, 2025, https://www.researchgate.net/publication/376365306_Design_Patterns_for_Machine_Learning_Based_Systems_with_Human-in-the-Loop
45. Search Engine Design | Google Search Architecture and Ranking - YouTube, accessed June 11, 2025, https://www.youtube.com/watch?v=MXLMQ5yWlwk

46. High Level Architecture - Wikipedia, accessed June 11, 2025, https://en.wikipedia.org/wiki/High_Level_Architecture
47. System Design of Twitter Feed - NamasteDev Blogs, accessed June 11, 2025, https://namastedev.com/blog/system-design-of-twitter-feed/
48. Architecture of fraud detection | Download Scientific Diagram - ResearchGate, accessed June 11, 2025, https://www.researchgate.net/figure/Architecture-of-fraud-detection_fig1_332049054
49. US11240372B2 - System architecture for fraud detection - Google Patents, accessed June 11, 2025, https://patents.google.com/patent/US11240372B2/en