Faire Technical Interview Preparation Guide

Complete Prep for Software Engineer Roles

Interview Focus: Algorithms, System Design (Search), ML Basics

October 21, 2025

Contents

1 Executive Summary: About Faire

1.1 Company Overview

Faire is an online wholesale marketplace that connects independent retailers with unique brands. Founded in 2017 and based in San Francisco, Faire uses machine learning to match local retailers with brands and products that uniquely fit their stores.

Key Value Propositions:

- **For Retailers:** Discover unique products, net 60 payment terms, free returns on opening orders
- **For Brands:** Access to 700,000+ retailers globally, data-driven insights, streamlined operations
- **Two-Sided Marketplace:** Balance retailer needs (relevance, conversion, personalization) with brand needs (order volume, exposure)

1.2 Core Technical Challenges

Based on Faire's product and engineering blog, their main technical challenges include:

1. **Search & Discovery:** Help retailers find relevant products among millions of SKUs
2. **Recommendations:** Inspire retailers in "browsing mode" with personalized suggestions
3. **Ranking:** Balance multiple stakeholders (retailers, brands, platform) in ranking algorithms
4. **Two-Sided Optimization:** Optimize for both retailer satisfaction and brand success
5. **Cold Start:** New retailers and new brands need quality recommendations immediately
6. **Real-Time Features:** Low-latency feature serving for ranking models

2 Faire Tech Stack & Architecture

Understanding Faire's actual tech stack will help you tailor your answers and ask informed questions.

2.1 Backend & Languages

Primary Language: Kotlin

- Entire backend monolith written in Kotlin (migrated from Java)
- Handles 1,000+ requests per second
- Android app also in Kotlin
- Allows for concise, type-safe code with null safety

Why Kotlin?

- Interoperable with existing Java libraries (Hibernate for ORM, Guice for DI)
- Modern language features (coroutines for async, data classes)
- Strong community and tooling support

2.2 Core Infrastructure

Data Storage:

- **MySQL:** Primary relational database
- **Redis:** Caching layer and feature store for ML models
- **NoSQL:** For specific use cases

Search:

- **ElasticSearch:** Product search with custom scoring functions
- **Lucene:** Underlying search library

Message Queues & Streaming:

- **Amazon SQS:** Asynchronous job processing
- **Kinesis Firehose:** Event streaming to data warehouse

Compute & Orchestration:

- **Kubernetes:** Container orchestration
- **Nginx:** Load balancing and routing
- **AWS ELB:** External load balancing

2.3 Machine Learning Stack

Models:

- **XGBoost:** Primary model for ranking (search and recommendations)
- **Embedding Models:** Learn representations from user behavior (clicks, add-to-cart, orders)

Feature Engineering:

- **Redis Feature Store:** Hundreds of features retrieved in real-time per product
- Features include: retailer-product similarity, engagement signals, brand metadata, product attributes

Data Infrastructure:

- **DBT:** Data transformation and modeling
- **Tableau:** Business intelligence and analytics

2.4 Two-Stage Ranking Architecture

Faire uses a classic retrieval + ranking pipeline:

Stage 1: Retrieval (Candidate Generation)

- ElasticSearch retrieves top 1,000 candidate products
- Uses custom scoring function balancing precision/recall
- Fast: optimized for low latency

Stage 2: Ranking (Re-ranking)

- Retrieve hundreds of features per product from Redis
- Score 1,000 candidates with XGBoost model
- Rank products by predicted engagement/conversion

3 Interview Process Overview

3.1 Typical Interview Timeline

Based on candidate reports:

1. **Recruiter Screen (30 min):** Align on basics, discuss role expectations
2. **Online Coding Assessment:** CodeSignal or similar (4 questions in 70 min)
    - 2 easy LeetCode-level questions
    - 2 medium to hard LeetCode-level questions
    - Score 700+ typically required to proceed
3. **Technical Interviews (2-3 rounds):**
    - 2 rounds: DSA/LeetCode-style coding (CoderPad)
    - 1 round: System Design (focus on backend/service scaling, 90% backend)
    - Possibly 1 round: SQL/data modeling
4. **Manager Match (1 round):** Behavioral, culture fit, team alignment
5. **Timeline:** 2-4 weeks total

3.2 Difficulty Rating

Glassdoor Rating: 3.14/5.0

Coding Questions: Straightforward medium/hard, not ”insanely tricky”

System Design: Backend-focused, emphasis on scalability and tradeoffs

3.3 What the Hiring Manager Said

Based on your conversation, expect:

- **Algorithms:** LeetCode-style DSA questions
- **System Design:** Main focus on *search* systems (Faire's core product)
- **ML Basics:** Evaluation metrics (precision, recall, NDCG, etc.)
- **Culture Fit:** Alignment with Faire's values and mission

4 Part 1: Algorithm & Coding Questions

4.1 Expected Difficulty

Online Assessment: 2 Easy + 2 Medium-Hard LeetCode
Live Coding Rounds: Medium LeetCode problems (some harder variants)
Focus Areas: Arrays, hashmaps, two pointers, trees, graphs, dynamic programming

4.2 Common LeetCode Patterns for Faire

Based on e-commerce/marketplace context, expect questions related to:

4.2.1 Pattern 1: Arrays & Hashing

Sample Problem: Product Inventory Management
Problem: Given an array of product IDs representing items in a retailer's cart, find the most frequently purchased product. If there's a tie, return the product with the smallest ID.

```python
def most_frequent_product(cart):
    """
    Time: O(n), Space: O(n)
    """
    from collections import Counter

    if not cart:
        return -1

    # Count frequencies
    freq = Counter(cart)

    # Find max frequency
    max_freq = max(freq.values())

    # Among products with max freq, return smallest ID
    candidates = [pid for pid, count in freq.items()
                  if count == max_freq]

    return min(candidates)

# Example
cart = [101, 203, 101, 305, 203, 101, 203]
print(most_frequent_product(cart))  # 101 (freq=3, but smaller ID)
```

Follow-up: What if we need to return top K products? Use a heap.

4.2.2 Pattern 2: Two Pointers

Sample Problem: Merge Brand Catalogs

Problem: Two brands have sorted arrays of product prices. Merge them into one sorted array.

```python
def merge_catalogs(prices1, prices2):
    """
    Two pointers to merge sorted arrays.
    Time: O(m + n), Space: O(m + n)
    """
    result = []
    i, j = 0, 0

    while i < len(prices1) and j < len(prices2):
        if prices1[i] <= prices2[j]:
            result.append(prices1[i])
            i += 1
        else:
            result.append(prices2[j])
            j += 1

    # Append remaining
    result.extend(prices1[i:])
    result.extend(prices2[j:])

    return result
```

4.2.3 Pattern 3: Binary Search

Sample Problem: Find Minimum Price in Rotated Catalog

Problem: A sorted price list was rotated. Find the minimum price.

```python
def find_min_price(prices):
    """
    Binary search on rotated sorted array.
    Time: O(log n), Space: O(1)
    """
    left, right = 0, len(prices) - 1

    while left < right:
        mid = (left + right) // 2

        # Right half is unsorted, min is in right half
        if prices[mid] > prices[right]:
            left = mid + 1
        else:
            # Left half is unsorted or mid is min
            right = mid

    return prices[left]

# Example
```

```python
prices = [4.99, 5.99, 6.99, 1.99, 2.99, 3.99]
print(find_min_price(prices))  # 1.99
```

4.2.4 Pattern 4: Trees (Product Taxonomy)

Sample Problem: Category Tree Traversal

Problem: Faire has product categories organized in a tree. Count total products in a category and all subcategories.

```python
class CategoryNode:
    def __init__(self, name, product_count):
        self.name = name
        self.product_count = product_count
        self.children = []

def count_products(root):
    """
    DFS to count products in category tree.
    Time: O(n), Space: O(h) for recursion stack
    """
    if not root:
        return 0

    total = root.product_count

    for child in root.children:
        total += count_products(child)

    return total

# Example tree: Home Decor -> (Furniture, Lighting)
root = CategoryNode("Home Decor", 100)
furniture = CategoryNode("Furniture", 50)
lighting = CategoryNode("Lighting", 30)
root.children = [furniture, lighting]

print(count_products(root))  # 180
```

4.2.5 Pattern 5: Graphs (Retailer-Brand Network)

Sample Problem: Find Connected Retailers

Problem: Given a graph where nodes are retailers and edges connect retailers who order from the same brands, find the number of connected components (retailer communities).

```python
def count_communities(n, edges):
    """
    Union-Find to count connected components.
    Time: O(n + e * alpha(n)), Space: O(n)
    """
    parent = list(range(n))

    def find(x):
        if parent[x] != x:
                parent[x] = find(parent[x])  # Path compression
            return parent[x]

    def union(x, y):
        root_x, root_y = find(x), find(y)
        if root_x != root_y:
            parent[root_x] = root_y

    # Build connections
    for u, v in edges:
        union(u, v)

    # Count unique roots
    return len(set(find(i) for i in range(n)))

# Example: 5 retailers, edges: [(0,1), (1,2), (3,4)]
print(count_communities(5, [(0,1), (1,2), (3,4)]))  # 2 communities
```

4.2.6 Pattern 6: Dynamic Programming

Sample Problem: Maximum Order Value (Knapsack Variant)

Problem: A retailer has a budget. Given products with prices and profit margins, maximize total profit without exceeding budget.

```python
def max_profit(budget, prices, profits):
    """
    0/1 Knapsack problem.
    Time: O(n * budget), Space: O(budget)
    """
    n = len(prices)
    dp = [0] * (budget + 1)

    for i in range(n):
        price = prices[i]
        profit = profits[i]

        # Traverse backwards to avoid using same item twice
        for b in range(budget, price - 1, -1):
            dp[b] = max(dp[b], dp[b - price] + profit)

    return dp[budget]

# Example
budget = 100
prices = [30, 40, 50, 60]
profits = [10, 20, 25, 30]
print(max_profit(budget, prices, profits))  # 50 (items 1 and 2)
```

4.3 E-commerce Specific Coding Questions

These are deduced based on Faire's product:

4.3.1 Question 1: Product Search Ranking

Problem: Given a search query and a list of products (each with name, description, price), return the top K most relevant products. Relevance is based on keyword matches (more matches = higher relevance). If tied, prioritize lower price.

```python
def search_products(query, products, k):
    """
    Time: O(n * m + n log n), where n = products, m = avg words
    Space: O(n)
    """
    query_words = set(query.lower().split())

    scored_products = []
    for product in products:
        text = (product['name'] + ' ' + product['description']).lower()
        words = set(text.split())

        # Count keyword matches
        matches = len(query_words & words)

        # Score: (matches, -price) for sorting
        scored_products.append((matches, -product['price'], product))

    # Sort by matches (desc), then price (asc)
    scored_products.sort(key=lambda x: (x[0], -x[1]), reverse=True)

    return [p[2] for p in scored_products[:k]]

# Example
products = [
    {"name": "Vase", "description": "ceramic blue vase", "price": 25},
    {"name": "Blue Pot", "description": "ceramic pot", "price": 20},
    {"name": "Ceramic Bowl", "description": "handmade ceramic", "price":
        30}
]
query = "blue ceramic"
print(search_products(query, products, 2))
# Returns: Blue Pot (2 matches, $20), Vase (2 matches, $25)
```

4.3.2 Question 2: Order Fulfillment

Problem: Faire processes orders from multiple retailers. Given an array of order timestamps (Unix time), find the maximum number of orders processed in any 1-hour window.

```python
def max_orders_in_hour(timestamps):
    """
    Sliding window approach.
    Time: O(n log n + n), Space: O(1)
    """
    if not timestamps:
        return 0

    timestamps.sort()
    max_count = 0

    for i in range(len(timestamps)):
        window_end = timestamps[i] + 3600  # 1 hour in seconds

        # Binary search for end of window
        left, right = i, len(timestamps) - 1
        while left < right:
            mid = (left + right + 1) // 2
            if timestamps[mid] <= window_end:
                left = mid
            else:
                right = mid - 1

        count = left - i + 1
        max_count = max(max_count, count)

    return max_count

# Example
timestamps = [1000, 2000, 3000, 4500, 5000, 6000]
print(max_orders_in_hour(timestamps))  # 4 orders
```

4.4 SQL Questions

Faire emphasizes SQL and data modeling. Expect questions like:

4.4.1 Question 1: Top Brands by Revenue

Problem: Given tables `orders(order_id, brand_id, amount, date)` and `brands(brand_id, name)`, find the top 5 brands by total revenue in the last 30 days.

```sql
SELECT
    b.name,
    SUM(o.amount) AS total_revenue
FROM orders o
JOIN brands b ON o.brand_id = b.brand_id
WHERE o.date >= CURRENT_DATE - INTERVAL 30 DAY
GROUP BY b.brand_id, b.name
ORDER BY total_revenue DESC
LIMIT 5;
```

4.4.2 Question 2: Retailer Churn Rate

Problem: Calculate monthly churn rate. Tables: `retailers(retailer_id, signup_date)`, `orders(order_id, retailer_id, order_date)`.

Definition: Churn = retailers who placed orders in month N but not in month N+1.

```sql
WITH monthly_active AS (
    SELECT
        retailer_id,
        DATE_TRUNC('month', order_date) AS month
    FROM orders
    GROUP BY retailer_id, month
),
churned AS (
    SELECT
        m1.month,
        m1.retailer_id
    FROM monthly_active m1
    LEFT JOIN monthly_active m2
        ON m1.retailer_id = m2.retailer_id
        AND m2.month = m1.month + INTERVAL '1 month'
    WHERE m2.retailer_id IS NULL
)
SELECT
    month,
    COUNT(DISTINCT retailer_id) AS churned_retailers
FROM churned
GROUP BY month
ORDER BY month;
```

5 Part 2: System Design - Search & Recommendations

5.1 Why Search is Critical for Faire

The hiring manager specifically mentioned **search** as the main system design focus. This aligns with Faire's core value proposition:

- **700K+ retailers** need to discover products from **100K+ brands**
- **Millions of SKUs** across diverse categories (home decor, apparel, food, etc.)
- **Two shopping modes:**
  - *Focused search:* "I need ceramic mugs for my coffee shop"
  - *Inspirational browsing:* "Show me trending home decor"

5.2 System Design Problem 1: Design Faire's Product Search

5.2.1 Problem Statement

Design a search system for Faire's wholesale marketplace. Retailers should be able to search for products across millions of SKUs from 100K+ brands. The system should support:

- Keyword search with autocomplete
- Filters (price, category, brand, location, minimum order, etc.)
- Relevance ranking
- Low latency (¡ 200ms p99)
- Personalization based on retailer's store type and past orders
- Handle 10,000 QPS at peak

5.2.2 Clarifying Questions

Ask the interviewer:

1. **Scale:** How many products? How many searches/day?
2. **Latency:** What's acceptable p50, p95, p99?
3. **Ranking:** Should we optimize for clicks? Add-to-cart? Orders?
4. **Personalization:** How much personalization vs. global relevance?
5. **Consistency:** Strong consistency needed or eventual consistency OK?
6. **Internationalization:** Multiple languages? Multiple currencies?

5.2.3 Requirements & Scale Estimation

Functional Requirements:

- Search products by keywords
- Filter by attributes (price, category, brand, etc.)
- Autocomplete search queries
- Rank results by relevance
- Personalize results per retailer

Non-Functional Requirements:

- Low latency: ¡ 200ms p99
- High availability: 99.9% uptime
- Scalability: 10K QPS peak
- Consistency: Eventual consistency OK for search index

Scale Estimation:

- **Products:** 10M SKUs
- **Brands:** 100K brands
- **Retailers:** 700K active retailers
- **Searches:** 1M searches/day = 12 QPS average, 120 QPS peak (10x)
- **Index size:** Assume 10KB per product → 100GB index

5.2.4 High-Level Architecture

Components:

1. **API Gateway:** Load balancing, rate limiting, authentication
2. **Search Service:** Handles search requests, queries ElasticSearch
3. **ElasticSearch Cluster:** Inverted index for fast keyword search
4. **Ranking Service:** ML-based ranking (XGBoost)
5. **Feature Store (Redis):** Real-time features for ranking
6. **Autocomplete Service:** Trie-based prefix matching
7. **Personalization Service:** Retriever embeddings, user preferences
8. **Data Ingestion Pipeline:** Update search index from product catalog

Data Flow:

```
Retailer -> API Gateway -> Search Service
                                |
                                v
                          ElasticSearch (retrieve top 1000)
                                |
                                v
                          Ranking Service (+ Redis features)
                                |
                                v
                          XGBoost Model (re-rank to top 50)
                                |
                                v
                          Personalization Layer
                                |
                                v
                          Return ranked results
```

5.2.5 Detailed Component Design

1. ElasticSearch Cluster

Index Schema:

```
{
  "product_id": "12345",
  "name": "Ceramic Mug Set",
  "description": "Handmade ceramic mugs, set of 4",
  "category": "Home & Kitchen > Drinkware",
  "brand_id": "brand_789",
  "brand_name": "ArtisanCraft Co",
  "price": 45.99,
  "tags": ["ceramic", "handmade", "eco-friendly"],
  "min_order_quantity": 6,
  "inventory_count": 150,
  "brand_location": "Portland, OR",
  "created_at": "2024-01-15T10:00:00Z",
  "popularity_score": 8.5
}
```

Scoring Function:
ElasticSearch uses BM25 for text relevance, but we can customize:

$$\text{score} = \alpha \cdot \text{BM25}(q, d) + \beta \cdot \text{popularity}(d) + \gamma \cdot \text{recency}(d) \quad (1)$$

Where:

- BM25(q, d): keyword match score
- popularity(d): click-through rate, order rate
- recency(d): boost newer products
- α, β, γ: learned weights

Sharding Strategy:

- Shard by product ID hash (uniform distribution)
- 10 shards for 10M products = 1M products/shard
- 3 replicas per shard for high availability

2. Ranking Service with XGBoost
After ElasticSearch retrieves 1,000 candidates, re-rank with ML model.
Features (hundreds per product):

- **Text relevance:** BM25 score, title match, description match
- **Retailer-product similarity:** Embedding cosine similarity
- **Engagement signals:** CTR, add-to-cart rate, purchase rate
- **Product attributes:** Price, category, brand popularity, inventory
- **Personalization:** Retailer's past category preferences, price sensitivity

Model Training:

- **Objective:** Predict probability of order given search query
- **Training data:** Search queries + clicked/ordered products (positive), random products (negative)
- **Model:** XGBoost (gradient boosted trees)
- **Offline evaluation:** NDCG@10, Precision@10, Recall@50
- **Online evaluation:** A/B testing (CTR, conversion rate, revenue)

Feature Retrieval:

- Store features in Redis (key: product_id, value: feature vector)
- Batch fetch 1,000 products' features in parallel
- Latency: ¡ 10ms for Redis lookup

3. Autocomplete Service
Data Structure: Trie (Prefix Tree)

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
        self.suggestions = []  # Top 5 completions

class Autocomplete:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, query, frequency):
        node = self.root
        for char in query.lower():
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]

        node.is_end_of_word = True
        # Update top suggestions at this node
        self.update_suggestions(node, query, frequency)

    def search(self, prefix):
        node = self.root
        for char in prefix.lower():
            if char not in node.children:
                return []
            node = node.children[char]

        return node.suggestions[:5]
```

Populating Autocomplete:

- Mine frequent queries from search logs
- Use product names, brand names, category names
- Update daily from offline job
- Store in Redis for low-latency lookup

4. Personalization Layer
Use **embedding-based retrieval** to personalize results.
Embeddings:

- **Retailer embeddings:** Learned from order history, clicks, store category

- **Product embeddings:** Learned from co-purchase patterns, attributes
- **Similarity:** Cosine similarity in embedding space

Model: Two-tower neural network (similar to YouTube's DNN)

- **Retailer tower:** Encode retailer features → retailer embedding
- **Product tower:** Encode product features → product embedding
- **Training:** Maximize dot product for positive pairs (retailer ordered product)

Personalization at Serving:

```python
def personalize(ranked_products, retailer_embedding):
    """
    Re-weight top 50 products by retailer-product similarity.
    """
    for product in ranked_products:
        similarity = cosine_similarity(
            retailer_embedding,
            product.embedding
        )
        product.score = 0.7 * product.score + 0.3 * similarity

    return sorted(ranked_products, key=lambda p: p.score, reverse=True)
```

5.2.6 Database Schema

Products Table:

```sql
CREATE TABLE products (
    product_id BIGINT PRIMARY KEY,
    brand_id BIGINT,
    name VARCHAR(255),
    description TEXT,
    category_id INT,
    price DECIMAL(10, 2),
    min_order_qty INT,
    inventory_count INT,
    created_at TIMESTAMP,
    updated_at TIMESTAMP,
    INDEX (brand_id),
    INDEX (category_id)
);
```

Search Logs Table:

```sql
CREATE TABLE search_logs (
    log_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    retailer_id BIGINT,
    query VARCHAR(500),
    clicked_product_id BIGINT,
    position INT,
    timestamp TIMESTAMP,
    INDEX (retailer_id, timestamp),
    INDEX (query)
);
```

5.2.7 API Design

Search API:

```
GET /api/v1/search?q=ceramic+mugs&filters={"category":"drinkware","
    price_max":50}&page=1&size=20

Response:
{
  "query": "ceramic mugs",
  "total_results": 1523,
  "page": 1,
  "size": 20,
  "products": [
    {
      "product_id": 12345,
      "name": "Ceramic Mug Set",
      "brand": "ArtisanCraft Co",
      "price": 45.99,
      "image_url": "https://...",
      "min_order": 6
    },
    ...
  ],
  "facets": {
    "categories": {"drinkware": 800, "home_decor": 300},
    "price_ranges": {"0-25": 500, "25-50": 700, "50-100": 323}
  }
}
```

Autocomplete API:

```
GET /api/v1/autocomplete?q=cera

Response:
{
  "suggestions": [
    "ceramic mugs",
    "ceramic vases",
    "ceramic plates",
    "ceramic bowls",
    "ceramic decor"
  ]
}
```

5.2.8 Optimizations & Tradeoffs

Caching Strategy:

- **CDN:** Cache popular search results (top 1000 queries)

- **Redis:** Cache recent searches per retailer
- **TTL:** 5 minutes for search results (balance freshness vs. performance)

Cold Start Problem:

- **New retailers:** Use store category to initialize preferences
- **New products:** Boost in ranking for visibility, collect data quickly

Handling Spikes:

- Auto-scaling ElasticSearch cluster
- Rate limiting per retailer (prevent abuse)
- Degrade gracefully: skip personalization if Redis is slow

Consistency Tradeoffs:

- Product updates: eventual consistency OK (few minutes delay)
- Critical updates (out-of-stock): real-time update via pub/sub

5.2.9 Metrics & Monitoring

System Metrics:

- Latency: p50, p95, p99
- QPS, error rate, cache hit rate
- ElasticSearch shard health

Business Metrics:

- CTR (click-through rate) per search
- Conversion rate (search $\rightarrow$ order)
- Revenue per search
- Zero-result rate (searches with no results)

ML Metrics:

- NDCG@10 (offline)
- Precision@5, Recall@20 (offline)
- Online A/B testing (CTR, conversion)

5.3 System Design Problem 2: Design Faire's Recommendation System

5.3.1 Problem Statement

Design a recommendation system for Faire's homepage and product pages. When a retailer logs in, show personalized product recommendations. Also show "You may also like" on product detail pages.

Requirements:

- Personalized homepage recommendations (50 products)
- "Similar products" on product pages (10 products)
- Handle cold start (new retailers, new products)
- Low latency (¡ 100ms)
- Optimize for order conversion, not just clicks

5.3.2 Approach: Two-Stage Retrieval + Ranking

This mirrors Faire's actual architecture (based on their engineering blog).

Stage 1: Candidate Generation

- Retrieve 1,000 candidate products from multiple sources:
  1. **Collaborative filtering:** Retailers similar to you ordered these
  2. **Content-based:** Products similar to what you've ordered
  3. **Trending:** Popular products in your category
  4. **New arrivals:** Recently added products

Stage 2: Ranking

- Score 1,000 candidates with XGBoost
- Features: retailer-product similarity, engagement, diversity
- Return top 50

5.3.3 Collaborative Filtering with Embeddings

Faire uses **embedding-based CF** learned from implicit feedback.

Training Data:

- **Positive:** (retailer, product) pairs from orders, add-to-cart, favorites
- **Negative:** Random products (or hard negatives: clicked but not ordered)

Model: Two-Tower DNN

```python
import torch
import torch.nn as nn

class TwoTowerModel(nn.Module):
    def __init__(self, num_retailers, num_products, embedding_dim=128):
        super().__init__()

        # Retailer tower
        self.retailer_embedding = nn.Embedding(num_retailers,
           embedding_dim)
        self.retailer_fc = nn.Sequential(
            nn.Linear(embedding_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 128)
        )

        # Product tower
        self.product_embedding = nn.Embedding(num_products, embedding_dim)
        self.product_fc = nn.Sequential(
            nn.Linear(embedding_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 128)
        )

    def forward(self, retailer_ids, product_ids):
        # Encode retailer
        retailer_emb = self.retailer_embedding(retailer_ids)
        retailer_vec = self.retailer_fc(retailer_emb)

        # Encode product
        product_emb = self.product_embedding(product_ids)
        product_vec = self.product_fc(product_emb)

        # Dot product (similarity)
        scores = (retailer_vec * product_vec).sum(dim=1)

        return torch.sigmoid(scores)
```

Loss Function:

$$\mathcal{L} = -\sum_{(r,p)\in\text{positive}} \log \sigma(e_r \cdot e_p) - \sum_{(r,p')\in\text{negative}} \log(1 - \sigma(e_r \cdot e_{p'})) \quad (2)$$

Serving:

- Store product embeddings in FAISS (Approximate Nearest Neighbors)
- Given retailer embedding, find top K nearest products
- Latency: ¡ 10ms for ANN search

5.3.4 Product-to-Product Recommendations

For ”You may also like” on product pages:

Approach 1: Co-Purchase Matrix

- Build matrix: product A $\times$ product B = count of co-purchases
- Use collaborative filtering (item-item similarity)
- Cosine similarity in co-purchase space

Approach 2: Embedding Similarity

- Use product embeddings from two-tower model
- For product P, find K nearest neighbors in embedding space
- Pre-compute and cache in Redis

5.3.5 Diversity & Exploration

Avoid showing only similar products (filter bubble).

Diversity Technique: MMR (Maximal Marginal Relevance)

$$\text{MMR}(p) = \lambda \cdot \text{relevance}(p) - (1 - \lambda) \cdot \max_{p' \in S} \text{similarity}(p, p') \tag{3}$$

Where:

- S: already selected products
- λ: tradeoff between relevance and diversity (e.g., 0.7)

Exploration: Thompson Sampling

- Occasionally show new/underexplored products
- Use multi-armed bandit to balance exploration vs. exploitation

5.3.6 Two-Sided Optimization

Faire must balance:

- **Retailer utility:** Relevant, high-quality products
- **Brand utility:** Sufficient exposure, especially for new brands

Approach:

- Add **brand diversity** constraint: limit products per brand
- Boost **new brands** in ranking (time decay)
- Monitor brand-level metrics: impression share, order share

5.4 Follow-Up Questions

Be prepared for these follow-ups:

1. **How would you handle multi-language search?**
    - Use language-specific analyzers in ElasticSearch
    - Cross-lingual embeddings for semantic search
    - Translate queries to English, then search
2. **How to prevent spam/fake products in search results?**
    - ML classifier to detect spam (based on title, description patterns)
    - Human review queue for flagged products
    - Downrank products with low engagement or high return rate
3. **How to A/B test ranking algorithms?**
    - Split traffic by retailer ID hash
    - Track metrics: CTR, conversion, revenue per search
    - Use statistical significance testing (t-test, chi-square)
    - Gradual rollout: 5% → 10% → 50% → 100%
4. **What if ElasticSearch goes down?**
    - Fallback to cached popular queries
    - Degrade to simpler search (MySQL full-text search)
    - Show trending products as fallback
    - Alert on-call engineer, auto-restart nodes
5. **How to scale to 100K QPS?**
    - Horizontal scaling: more ElasticSearch nodes
    - Read replicas for feature store (Redis)
    - CDN for static content and popular searches
    - Database sharding by geography or retailer segment

6 Part 3: Machine Learning Basics & Evaluation Metrics

6.1 Why ML Metrics Matter for Faire

Faire's search and recommendations are ML-powered. You need to:

- Understand common ML evaluation metrics
- Know when to use each metric
- Articulate tradeoffs (precision vs. recall, etc.)
- Relate metrics to business goals

6.2 Classification Metrics

6.2.1 Confusion Matrix

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

6.2.2 Core Metrics

1. Accuracy

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{4}$$

When to use: Balanced classes
Limitation: Misleading for imbalanced datasets (e.g., fraud detection: 99% non-fraud)
2. Precision

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5}$$

Interpretation: Of all predicted positives, how many are correct?
When to use: Cost of FP is high (e.g., spam detection - don't mark legitimate emails as spam)
Faire Example: Predicting "retailer will order this product"

- High precision: most recommended products are relevant
- Low precision: many irrelevant products shown

3. Recall (Sensitivity, True Positive Rate)

$$\text{Recall} = \frac{TP}{TP + FN} \tag{6}$$

Interpretation: Of all actual positives, how many did we catch?
When to use: Cost of FN is high (e.g., disease detection - don't miss sick patients)
Faire Example: Search results

- High recall: retrieve all relevant products
- Low recall: miss many relevant products

4. F1 Score

$$F1 = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2TP}{2TP + FP + FN} \tag{7}$$

When to use: Balance between precision and recall, imbalanced classes
Limitation: Treats precision and recall equally (sometimes one matters more)
5. ROC-AUC
ROC Curve: Plot of True Positive Rate (Recall) vs. False Positive Rate at various thresholds

$$\text{FPR} = \frac{FP}{FP + TN} \tag{8}$$

AUC (Area Under Curve): Probability that model ranks random positive higher than random negative
When to use: Compare models, threshold-agnostic evaluation
Limitation: Optimistic for imbalanced datasets (use PR-AUC instead)

6.2.3 Example Interview Question

Question: Faire wants to predict which products a retailer will order. You build a model with 90% accuracy. Is this good?

Answer:

- **Need more context:** What's the class distribution?
- If only 5% of shown products are ordered, a naive model (always predict "no order") has 95% accuracy but is useless.
- **Better metrics:** Precision@K, Recall@K (see ranking metrics below)
- **Business metric:** Revenue per recommended product

6.3 Ranking Metrics

For search and recommendations, we care about **order** of results, not just classification.

6.3.1 Precision@K and Recall@K

Precision@K:

$$\text{Precision@K} = \frac{\text{\# relevant items in top K}}{K} \tag{9}$$

Recall@K:

$$\text{Recall@K} = \frac{\text{\# relevant items in top K}}{\text{total relevant items}} \tag{10}$$

Example: Search for "ceramic mugs", 100 total relevant products
Top 10 results: 7 relevant, 3 irrelevant

- Precision@10 = 7/10 = 0.7
- Recall@10 = 7/100 = 0.07

Faire Use Case: Precision@10 for search (top 10 results matter most)

6.3.2 Mean Average Precision (MAP)

Average Precision (AP): Average of precision values at each relevant position

$$AP = \frac{1}{|\text{relevant}|} \sum_{k=1}^{n} \text{Precision@k} \times \text{rel}(k) \tag{11}$$

Where rel(k) = 1 if item at position k is relevant, else 0.
MAP: Average AP across multiple queries
Example:
Ranked list: [R, NR, R, R, NR, R] (R = relevant, NR = not relevant)
Relevant positions: 1, 3, 4, 6
Precision at each:

- P@1 = 1/1 = 1.0
- P@3 = 2/3 = 0.67

- P@4 = 3/4 = 0.75
- P@6 = 4/6 = 0.67

$AP = \frac{1}{4}(1.0 + 0.67 + 0.75 + 0.67) = 0.77$
Limitation: Doesn't consider position (item at rank 1 vs. rank 10 weighted equally)

6.3.3 Normalized Discounted Cumulative Gain (NDCG)

Why NDCG? Addresses two issues:

1. **Graded relevance:** Items can be highly relevant, somewhat relevant, or not relevant (not just binary)
2. **Position matters:** Higher-ranked items should be more relevant

Cumulative Gain (CG):

$$CG@K = \sum_{i=1}^{K} \text{rel}_i \tag{12}$$

Where rel_i is relevance score of item at position i (e.g., 0, 1, 2, 3)
Discounted Cumulative Gain (DCG):

$$DCG@K = \sum_{i=1}^{K} \frac{\text{rel}_i}{\log_2(i+1)} \tag{13}$$

Position discount: items at lower ranks contribute less
Ideal DCG (IDCG):
DCG if items were perfectly ranked by relevance
Normalized DCG (NDCG):

$$NDCG@K = \frac{DCG@K}{IDCG@K} \tag{14}$$

Range: [0, 1], where 1 = perfect ranking
Example:
Ranked list with relevance scores: [3, 2, 0, 1, 2]
$DCG@5 = \frac{3}{\log_2 2} + \frac{2}{\log_2 3} + \frac{0}{\log_2 4} + \frac{1}{\log_2 5} + \frac{2}{\log_2 6}$
$DCG@5 = 3.0 + 1.26 + 0 + 0.43 + 0.77 = 5.46$
Ideal ranking: [3, 2, 2, 1, 0]
$IDCG@5 = 3.0 + 1.26 + 0.77 + 0.43 + 0 = 5.46$
$NDCG@5 = 5.46/5.46 = 1.0$ (if we had perfect ranking)
For actual ranking: $NDCG@5 = 5.46/5.46 = 1.0$ (happens to be perfect in this case)
Faire Use Case: NDCG@10 for evaluating search ranking offline

6.3.4 Hit Rate (Recall@K for Recommendations)

Definition: Proportion of users who found at least one relevant item in top K recommendations

$$\text{Hit Rate@K} = \frac{\text{\# users with } \geq 1 \text{ relevant item in top K}}{\text{total users}} \tag{15}$$

Faire Example:
100 retailers, show top 10 recommendations

- 80 retailers ordered at least one recommended product
- Hit Rate@10 = 80/100 = 0.8

6.4 Regression Metrics

If predicting continuous values (e.g., estimated order value):

6.4.1 Mean Absolute Error (MAE)

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{16}$$

Pros: Easy to interpret (average error in same units as target)
Cons: Treats all errors equally (large errors not penalized more)

6.4.2 Mean Squared Error (MSE)

$$MSE = \frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{17}$$

Pros: Penalizes large errors more heavily
Cons: Units are squared (harder to interpret)

6.4.3 Root Mean Squared Error (RMSE)

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2} \tag{18}$$

Pros: Same units as target, penalizes large errors

6.4.4 R-Squared (R^2)

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \tag{19}$$

Interpretation: Proportion of variance explained by model
Range: $(-\infty, 1]$, where 1 = perfect predictions, 0 = model is as good as mean baseline

6.5 A/B Testing & Online Metrics

Offline metrics (NDCG, Precision@K) don't always correlate with business outcomes. Must A/B test.

6.5.1 Online Metrics for Search/Recommendations

Engagement Metrics:

- **Click-Through Rate (CTR):** clicks / impressions
- **Add-to-Cart Rate:** adds / impressions
- **Conversion Rate:** orders / impressions

Business Metrics:

- **Revenue per search**
- **Average order value (AOV)**
- **Customer lifetime value (LTV)**

User Experience:

- **Session duration**
- **Bounce rate**
- **Zero-result rate** (searches with no results)

6.5.2 A/B Test Design

Hypothesis: New ranking model increases conversion rate
Setup:

- **Control group:** 50% of retailers see current ranking
- **Treatment group:** 50% see new ranking
- **Randomization:** Hash retailer ID to assign group

Duration: Run for 2 weeks (capture weekly patterns)
Sample Size: Calculate needed sample for statistical power

$$n = \frac{2(Z_{\alpha/2} + Z_{\beta})^2\sigma^2}{(\mu_1 - \mu_0)^2} \tag{20}$$

Statistical Test:

- **Metric:** Conversion rate (proportion)
- **Test:** Two-proportion z-test
- **Significance level:** $\alpha = 0.05$

Decision:

- If $p < 0.05$ and treatment is better: ship new model
- If not significant: keep investigating (longer test, larger sample)
- Monitor for novelty effect (users like new things initially)

6.6 Common ML Interview Questions

6.6.1 Question 1: Precision vs. Recall Tradeoff

You're building a spam product detector for Faire. Would you optimize for precision or recall? Why?

Answer:

Optimize for precision (minimize false positives).

Reasoning:

- **FP (false positive):** Legitimate product marked as spam
  - Harm: Brand loses sales, may leave platform
  - High cost to business
- **FN (false negative):** Spam product not caught
  - Harm: Some low-quality products shown
  - Lower cost (can be caught by user reports)

Approach:

- Set high threshold for spam classification (e.g., 0.95 probability)
- Human review queue for borderline cases
- Monitor precision/recall in production

6.6.2 Question 2: Imbalanced Dataset

Only 2% of shown products are actually ordered. How do you handle this in training?

Answer:

1. **Resampling:**
   - **Oversample** minority class (ordered products)
   - **Undersample** majority class (not ordered)
   - Use SMOTE (Synthetic Minority Over-sampling)
2. **Class Weights:**
   - Assign higher weight to minority class in loss function
   - XGBoost: `scale_pos_weight` parameter
3. **Evaluation Metrics:**
   - Don't use accuracy (misleading)
   - Use Precision@K, Recall@K, F1, AUC-PR
4. **Hard Negative Mining:**
   - Sample negatives intelligently (products that were clicked but not ordered)
   - Harder negatives improve model discrimination

6.6.3 Question 3: NDCG vs. MAP

When would you use NDCG instead of MAP for search evaluation?

Answer:

Use NDCG when:

1. **Graded relevance:** Products have different levels of relevance (highly relevant, somewhat relevant, not relevant)
2. **Position matters a lot:** Top results should be weighted much more than lower results
3. **Industry standard:** NDCG is more common in industry (Google, Bing, etc.)

Use MAP when:

1. **Binary relevance:** Items are either relevant or not (no grades)
2. **All relevant items matter:** Not just top K

For Faire search: Use NDCG@10 (graded relevance, top 10 results critical)

6.7 ML System Design: Click Prediction Model

Design an ML system to predict click probability for search results at Faire.

6.7.1 Problem Formulation

Input: (retailer, query, product)

Output: P(click — retailer, query, product)

Training Data: Search logs (query, shown products, clicked products)

6.7.2 Feature Engineering

Query Features:

- Query length, word count
- Query category (intent classification)
- Historical CTR for this query

Product Features:

- Price, category, brand
- Popularity (orders, clicks, add-to-carts)
- Image quality score
- Inventory level

Retailer Features:

- Store category, location
- Past order history (categories, price range)

- Account age, activity level

Query-Product Features:

- Text match score (BM25)
- Embedding similarity (query embedding · product embedding)
- Position in search results (strong predictor!)

Retailer-Product Features:

- Cosine similarity in embedding space
- Did retailer order this category before?
- Price relative to retailer's avg. order value

6.7.3 Model Choice

Option 1: XGBoost (Faire's choice)

- **Pros:** Fast training, handles tabular features well, interpretable
- **Cons:** Doesn't capture complex interactions automatically

Option 2: Deep Neural Network

- **Pros:** Learns feature interactions, works well with embeddings
- **Cons:** Slower, harder to interpret, needs more data

Hybrid: Use embeddings from DNN as features in XGBoost

6.7.4 Training Pipeline

1. **Data Collection:** Stream search logs to data warehouse (Kinesis)
2. **Feature Engineering:** Daily batch job (Spark) to compute features
3. **Training:** Weekly re-train XGBoost on last 30 days of data
4. **Evaluation:** Offline: AUC, LogLoss; Online: A/B test CTR
5. **Deployment:** Export model to PMML, load in ranking service

6.7.5 Serving

```python
def rank_search_results(retailer_id, query, candidates):
    # Get retailer features from cache
    retailer_features = redis.get(f"retailer:{retailer_id}")

    # Compute query features
    query_features = compute_query_features(query)

    # For each candidate product
    scored_products = []
    for product in candidates:
        # Get product features from cache
        product_features = redis.get(f"product:{product.id}")

        # Compute interaction features
        interaction_features = compute_interaction_features(
            retailer_features, query_features, product_features
        )

        # Combine all features
        feature_vector = concatenate([
            retailer_features,
            query_features,
            product_features,
            interaction_features
        ])

        # Predict click probability
        click_prob = xgboost_model.predict(feature_vector)

        scored_products.append((product, click_prob))

    # Sort by predicted click probability
    scored_products.sort(key=lambda x: x[1], reverse=True)

    return [p for p, score in scored_products]
```

6.7.6 Position Bias Problem

Issue: Items at top positions get more clicks simply due to position, not relevance.
Solutions:

1. **Position feature:** Include position as feature (naive, doesn't fix bias)
2. **Inverse Propensity Weighting:** Weight training examples by $1/P(\text{position})$
3. **Randomization:** Randomly shuffle top results for small % of traffic to collect unbiased data
4. **Examination hypothesis:** Model P(click) = P(relevant) $\times$ P(examined)

7 Part 4: Behavioral & Culture Fit

7.1 Faire's Mission & Values

Mission: Empower entrepreneurs to chase their dreams
Vision: Build the future of commerce where anyone can start and grow a business
Values (inferred from job descriptions and culture):

- **Customer obsession:** Retailers and brands are customers
- **Ownership:** Engineers lead projects end-to-end

- **Data-driven:** Decisions backed by metrics and experiments
- **Collaboration:** Work with cross-functional teams
- **Impact:** Ship features that matter to real businesses

7.2 Common Behavioral Questions

7.2.1 Question 1: Why Faire?

Structure your answer:

1. **Mission alignment:** Why you care about supporting small businesses
2. **Technical challenge:** Excited about search/ML/two-sided marketplace problems
3. **Growth stage:** Opportunity to make impact at a scaling startup
4. **Personal connection:** Maybe you have friends who run small shops

Example Answer:
"I'm drawn to Faire because of the mission to empower independent retailers. My aunt runs a small boutique, and I've seen firsthand how challenging wholesale sourcing can be—long payment terms, high minimums, limited product discovery. Faire solves these pain points beautifully.

From a technical perspective, I'm excited about the search and recommendation challenges. Building ML systems that balance retailer preferences with brand visibility is a fascinating two-sided optimization problem. And as someone passionate about data-driven product development, Faire's focus on experimentation and metrics really resonates with me."

7.2.2 Question 2: Tell me about a time you led a project

Use STAR method:

- **Situation:** Set the context
- **Task:** What needed to be done?
- **Action:** What did YOU do? (focus on your contributions)
- **Result:** Quantifiable impact

Prepare examples for:

- Leading a technical project
- Solving a challenging bug
- Collaborating with non-engineers (PM, design)
- Making a data-driven decision
- Handling ambiguity or changing requirements
- Dealing with a difficult teammate or situation

7.2.3 Question 3: What metrics would you track for Faire?

This was mentioned as a common behavioral question. Show business acumen.

Retailer Metrics:

- **Acquisition:** New retailer signups, activation rate
- **Engagement:** Monthly active retailers, searches per session
- **Conversion:** Order conversion rate, avg. order value
- **Retention:** Repeat purchase rate, churn rate, LTV

Brand Metrics:

- **Acquisition:** New brand signups, catalog size
- **Engagement:** Order volume, revenue
- **Retention:** Brand churn, % brands with repeat orders

Platform Metrics:

- **GMV:** Gross Merchandise Value
- **Take rate:** Platform revenue as % of GMV
- **Search quality:** Zero-result rate, CTR, conversion

Two-Sided Health:

- **Liquidity:** % retailers who find products to order
- **Balance:** New brands getting orders vs. established brands

7.3 Questions to Ask Interviewers

About the Role:

- What does success look like in the first 3/6/12 months?
- What are the biggest technical challenges the team is facing?
- How does the team balance new features vs. tech debt?

About the Team:

- How is the search team structured? (Frontend, backend, ML, data science?)
- How do you collaborate with PM, design, and data science?
- What's the on-call rotation like?

About the Company:

- How has Faire's strategy evolved with the market?
- What are the company's top priorities for the next year?

- How do you balance retailer needs vs. brand needs in product decisions?

Technical:

- What ML models are you currently using for search ranking?
- How do you evaluate new ranking algorithms? (A/B testing process?)
- What's the data infrastructure like? (Streaming vs. batch?)
- How do you handle cold start for new retailers/brands?

8 Part 5: Preparation Strategy

8.1 4-Week Study Plan

8.1.1 Week 1: Algorithms & Data Structures

Goal: Solve 30 LeetCode problems (focus on medium difficulty)
Daily: 2 hours

- Day 1-2: Arrays & Hashing (10 problems)
- Day 3-4: Two Pointers, Sliding Window (10 problems)
- Day 5: Binary Search (5 problems)
- Day 6: Linked Lists (5 problems)
- Day 7: Review & mock interview

Resources:

- LeetCode Blind 75
- NeetCode (YouTube + roadmap)
- Tech Interview Handbook

8.1.2 Week 2: More Algorithms + SQL

Goal: 25 more problems + 10 SQL problems
Daily: 2 hours

- Day 1-2: Trees (BFS, DFS, BST) - 10 problems
- Day 3: Graphs (BFS, DFS, Union-Find) - 5 problems
- Day 4: Dynamic Programming (5 problems)
- Day 5: SQL (10 problems on LeetCode)
- Day 6: Heaps, Stacks, Queues (5 problems)
- Day 7: Review & mock interview

8.1.3 Week 3: System Design

Goal: Master 4 system design problems + read about Faire's stack
Daily: 2-3 hours

- Day 1: Read Faire's engineering blog (craft.faire.com)
- Day 2: Design search system (practice this guide's Problem 1)
- Day 3: Design recommendation system (practice Problem 2)
- Day 4: Design URL shortener (classic problem)
- Day 5: Design Instagram/Twitter feed
- Day 6: Review all designs, focus on tradeoffs
- Day 7: Mock system design interview with friend

Resources:

- Grokking the System Design Interview
- System Design Primer (GitHub)
- ByteByteGo (YouTube)

8.1.4 Week 4: ML + Behavioral + Mock Interviews

Goal: Solidify ML metrics, prepare stories, simulate interviews
Daily: 2-3 hours

- Day 1: Study ML metrics (this guide's Part 3)
- Day 2: Practice ML system design (click prediction, ranking)
- Day 3: Prepare behavioral stories (STAR method, 5 stories)
- Day 4: Mock coding interview (Pramp or friend)
- Day 5: Mock system design interview
- Day 6: Review all weak areas
- Day 7: Rest, light review, prepare questions to ask

8.2 Day Before Interview

- **Light review:** Skim your notes, don't cram
- **Re-read this guide:** Sections 1, 2, 3 summaries
- **Prepare questions:** Write down 5 questions to ask
- **Logistics:** Test video/audio, charge laptop, have pen & paper ready
- **Mindset:** Get good sleep, stay confident!

8.3 During the Interview

Coding Interview:

1. **Clarify:** Ask questions, confirm input/output, edge cases
2. **Think aloud:** Explain your approach before coding
3. **Start simple:** Brute force first, then optimize
4. **Test:** Walk through examples, catch bugs
5. **Communicate:** If stuck, ask for hints (shows collaboration)

System Design Interview:

1. **Clarify requirements:** Functional, non-functional, scale
2. **High-level design first:** Draw boxes (API, DB, cache, etc.)
3. **Deep dive:** Pick 2-3 components to detail (interviewer will guide)
4. **Discuss tradeoffs:** SQL vs. NoSQL, sync vs. async, etc.
5. **Time management:** Don't spend 40 min on API design

Behavioral Interview:

1. **STAR method:** Structure answers clearly
2. **Quantify impact:** "Reduced latency by 40%", "Increased CTR by 15%"
3. **Show learning:** "This taught me to...", "I'd do X differently next time"
4. **Ask questions:** Show genuine interest in the role

9 Appendix A: Faire-Specific Questions (To Be Added)

This section will be populated with actual interview questions from Glassdoor, Prepfully, and InterviewQuery after you provide the content.

9.1 Placeholder for Glassdoor Questions

[To be filled in after user provides Glassdoor content]

9.2 Placeholder for Prepfully Questions

[To be filled in after user provides Prepfully content]

9.3 Placeholder for InterviewQuery Content

[To be filled in after user provides InterviewQuery content]

10 Appendix B: Faire Engineering Blog Insights

10.1 Key Articles to Read

1. **"How data and machine learning shape Faire's marketplace"**
    - URL: craft.faire.com/how-data-and-machine-learning-shape-faires-marketplace-510855c4a9bc
    - **Key takeaways:**
        - Recommendation embeddings from clicks, add-to-cart, orders
        - Hybrid recommender system for cold start
        - XGBoost models for ranking
        - Redis feature store for real-time features
        - Balancing retailer and brand stakeholder needs
2. **"Real-time ranking at Faire part 2: the feature store"**
    - URL: craft.faire.com/real-time-ranking-at-faire-part-2-the-feature-store-3f1013d3fe5d
    - **Key takeaways:**
        - Feature store architecture with Redis
        - Handling hundreds of features per product
        - Low-latency feature retrieval (¡ 10ms)

10.2 Tech Stack Summary

Languages: Kotlin (backend), JavaScript/TypeScript (frontend)
Data: MySQL, Redis, NoSQL, ElasticSearch, Kinesis, DBT
ML: XGBoost, embeddings, collaborative filtering
Infrastructure: Kubernetes, AWS (ELB, SQS), Nginx

11 Appendix C: Additional Resources

11.1 Coding Practice

- **LeetCode:** leetcode.com (Blind 75, NeetCode 150)
- **NeetCode:** neetcode.io (video solutions + roadmap)
- **AlgoExpert:** algoexpert.io (paid, high quality)
- **HackerRank:** hackerrank.com (SQL practice)

11.2 System Design

- **Grokking the System Design Interview:** educative.io
- **System Design Primer:** github.com/donnemartin/system-design-primer
- **ByteByteGo:** bytebytego.com (Alex Xu's course)
- **Designing Data-Intensive Applications:** Book by Martin Kleppmann

11.3 Machine Learning

- **StatQuest:** youtube.com/@statquest (ML concepts)
- **Google ML Crash Course:** developers.google.com/machine-learning/crash-course
- **Chip Huyen's ML Interviews Book:** huyenchip.com/ml-interviews-book

11.4 Behavioral

- **Tech Interview Handbook:** techinterviewhandbook.org/behavioral-interview
- **Grokking the Behavioral Interview:** educative.io

12 Summary & Final Tips

12.1 Key Takeaways

1. **Faire is a two-sided marketplace** connecting retailers and brands via search and recommendations
2. **Tech stack:** Kotlin, ElasticSearch, XGBoost, Redis, MySQL, Kubernetes
3. **Interview focus:**
    - Algorithms: Medium LeetCode, 2 rounds DSA
    - System Design: Search systems (your main focus)
    - ML: Evaluation metrics (precision, recall, NDCG)
    - Behavioral: Culture fit, why Faire
4. **Search system design** is critical - know two-stage retrieval + ranking
5. **ML metrics:** Master NDCG, Precision@K, Recall@K for ranking problems
6. **Tradeoffs:** Be ready to discuss latency vs. accuracy, precision vs. recall, complexity vs. maintainability

12.2 The Night Before

- Review this guide's summaries (sections 2, 4, 5)
- Re-read Faire's mission and engineering blog posts
- Prepare 5 questions to ask
- Get 8 hours of sleep

12.3 You've Got This!

You've prepared thoroughly. Remember:

- **Communication ¿ perfection:** Interviewers want to see your thought process
- **Ask questions:** Shows curiosity and collaboration
- **Stay calm:** If stuck, take a breath, think aloud, ask for hints
- **Be yourself:** Faire values authenticity and passion

Good luck! You're going to do great!