Nexgate Recommendation System - Architecture Documentation

Nexgate Recommendation System - Architecture Documentation

Version 1.0 | Social Commerce Platform

? Table of Contents

? Executive Summary

What is Nexgate?

Nexgate is a **social commerce platform** that combines:

- E-commerce marketplace
- Social media feed experience

- Group buying (collective purchasing)
- Installment payment options
- Community engagement

Core Philosophy

"Discovery over Search"

- Users discover products through personalized feed (primary)
- Search is secondary enhancement feature
- Social proof drives purchasing decisions
- Affordability through group buying and installments

Recommendation System Goals

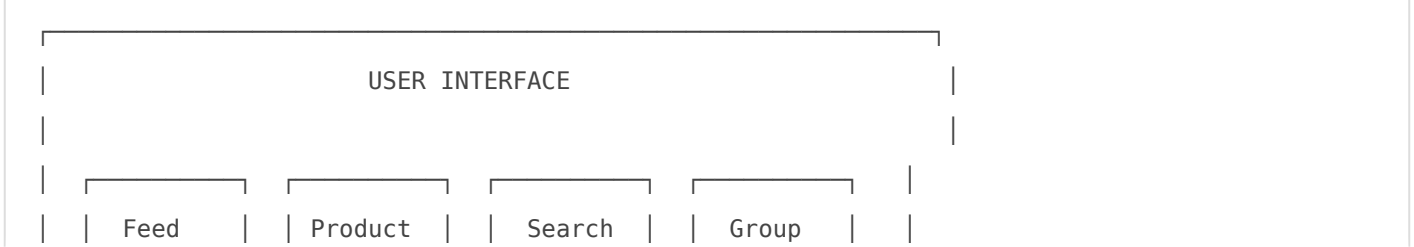
1. **Maximize Engagement:** Keep users scrolling and interacting
2. **Drive Conversions:** Turn views into purchases
3. **Build Community:** Connect buyers through group purchases
4. **Enable Affordability:** Help users buy premium products through installments
5. **Provide Value:** Show relevant products at the right time

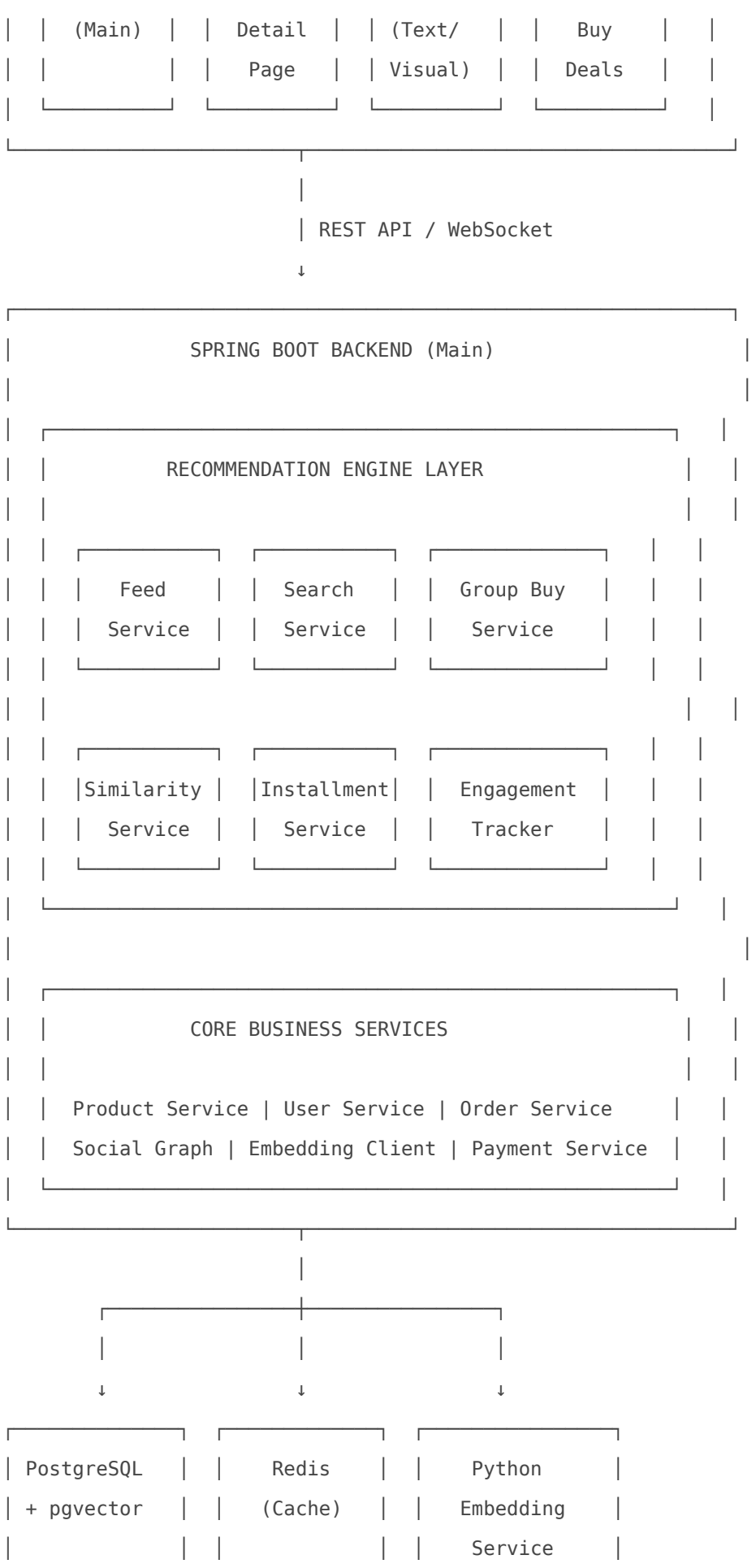
Key Success Metrics

- **Feed Engagement Rate:** Target > 5% (likes, comments, shares per view)
- **Click-Through Rate:** Target > 8% (feed → product page)
- **Conversion Rate:** Target > 3% (view → purchase)
- **Average Session Time:** Target > 10 minutes
- **Group Buy Completion Rate:** Target > 70% of groups reach goal
- **Installment Adoption:** Target > 40% of purchases over $500

?? System Overview

High-Level Architecture




(Main)
Detail Page
(Text/ Visual)
Buy Deals
REST API / WebSocket
SPRING BOOT BACKEND (Main)
RECOMMENDATION ENGINE LAYER
Feed Service
Search Service
Group Buy Service
Similarity Service
Installment Service
Engagement Tracker
CORE BUSINESS SERVICES
Product Service | User Service | Order Service
Social Graph | Embedding Client | Payment Service
PostgreSQL + pgvector
Redis (Cache)
Python Embedding Service

```
| • Products   |  | • Sessions  |  |               |
| • Users      |  | • Feed      |  | • Sentence    |
| • Orders     |  | • Seen Posts|  |   Transformers |
| • Posts      |  | • Hot Data  |  | • CLIP Model  |
| • Embeddings |  |             |  |               |
└──────────────┘  └─────────────┘  └───────────────┘
```

Component Responsibilities

Frontend (Mobile/Web):

- Display personalized feed
- Handle user interactions (like, share, comment)
- Upload images for visual search
- Show group buy status and timers
- Display installment options

Spring Boot Backend:

- Core business logic
- Recommendation engine orchestration
- API endpoints
- Real-time notifications
- Payment processing

PostgreSQL + pgvector:

- Primary data storage
- Vector similarity search
- Transactional data
- User/product relationships

Redis:

- Session management
- Feed caching (short-lived)
- Seen posts tracking
- Real-time counters
- Rate limiting

Python Embedding Service:

- Text embedding generation
- Image embedding generation
- Runs as separate microservice

- Models stay loaded in memory

? Recommendation Strategies Overview

Strategy Matrix

Feature	Strategy	Uses Embeddings	Complexity	Priority
Main Feed	Hybrid Scoring	No	Medium	P0 (Critical)
Product Similarity	Vector Similarity	Yes (Text + Image)	High	P1 (Important)
Text Search	Semantic Matching	Yes (Text)	High	P1 (Important)
Visual Search	Image Matching	Yes (Image)	High	P2 (Nice to have)
Group Buy Match	Interest + Urgency	Partial	Medium	P0 (Critical)
Group Invites	Social Graph	No	Low	P1 (Important)
Installment Suggest	Budget Analysis	No	Low	P0 (Critical)
Trending	Engagement Metrics	No	Low	P0 (Critical)
Collaborative	Purchase Patterns	No	Medium	P1 (Important)

When to Use What

Use Scoring System (No Embeddings):

- Main feed personalization
- Group buy recommendations
- Installment suggestions
- Trending content
- Social recommendations (follows, friends)

Use Embeddings:

- "Similar products" feature
- Semantic text search
- Visual search (upload image)
- Style/aesthetic matching

Use Collaborative Filtering:

- "Frequently bought together"
- "Customers also viewed"
- Cross-sell recommendations

Use Social Graph:

- Friend activity
- Network purchasing patterns
- Group buy invitations
- Influencer recommendations

? Detailed Recommendation Strategies

1. Main Feed Algorithm (Primary Experience)

Purpose: Personalized social commerce feed (like TikTok Shop/Instagram Shopping)

Composition Strategy:

```
Feed Mix (Per 20 posts):
├── 60% Posts from Following (12 posts)
│   • Sellers user follows
│   • Friends' posts
│   • Brands user engaged with
│
├── 25% Trending in User's Categories (5 posts)
│   • Hot products in categories user likes
│   • Viral posts with high engagement
│   • New products gaining traction
│
├── 10% Local Sellers (2 posts)
│   • Sellers in same city/region
│   • Faster delivery options
│   • Support local businesses
│
└── 5% Sponsored/Promoted (1 post)
    • Paid advertising
    • Featured products
    • Platform partnerships
```

Scoring Formula:

Each post receives a score from 0-100 based on weighted factors:

```
Post Score =
    (35% × Social Score) +
    (25% × Engagement Prediction Score) +
    (20% × Personalization Score) +
    (15% × Recency Score) +
    (5% × Quality Score)
```

Factor Breakdown:

A. Social Score (35 points max)

```
Following Seller: +15 points
Friends Engaged (per friend): +2 points (max 10)
Seller Verified: +5 points
Seller Rating (0-5 stars): +0 to 5 points
```

B. Engagement Prediction Score (25 points max)

```
Category Match (user's top 3 categories): +10 points
Price in User's Range: +8 points
Similar to Past Likes: +7 points
```

C. Personalization Score (20 points max)

```
Matches Recent Searches (per keyword): +2 points (max 8)
Local Seller (same city): +7 points
Matches User Preferences: +5 points
```

D. Recency Score (15 points max)

```
< 1 hour old: 15 points
< 6 hours old: 12 points
< 24 hours old: 8 points
< 3 days old: 4 points
Older: 1 point
```

E. Quality Score (5 points max)

```
3+ Images: +2 points
Detailed Description (>100 chars): +1 point
High Engagement Rate (>10%): +2 points
```

Feed Generation Process:

```
Step 1: Gather Candidates
├── Query posts from following (last 7 days)
├── Query trending posts in user's categories
├── Query local sellers' posts
└── Query sponsored posts
→ Result: ~100-200 candidate posts

Step 2: Filter
├── Remove posts user already saw (Redis cache)
├── Remove out-of-stock products
├── Remove blocked/reported sellers
└── Remove duplicate products
→ Result: ~80-150 eligible posts

Step 3: Score Each Post
├── Calculate social score
├── Calculate engagement prediction
├── Calculate personalization
├── Calculate recency
└── Calculate quality
→ Result: Each post has score 0-100

Step 4: Rank & Mix
├── Sort by score (highest first)
├── Apply diversity rules:
│   • Max 3 consecutive posts from same seller
│   • Max 5 posts from same category in 20
│   • Insert 1 sponsored post at position 7-10
└── Apply freshness shuffle (boost 2-3 recent high-quality posts)
→ Result: Final ranked list

Step 5: Pagination
├── Return top 20 posts
├── Generate cursor (ID of 20th post)
```

```
├── Cache seen post IDs (Redis, 30 days TTL)
└── Track impressions for analytics
```

Optimization Strategies:

Caching:

- User's following list: Redis, 1 hour TTL
- User's interested categories: Redis, 6 hours TTL
- Trending posts: Redis, 5 minutes TTL
- User's seen posts: Redis, 30 days TTL

Pre-computation:

- Trending posts calculated every 5 minutes (background job)
- User interest profiles updated daily (background job)
- Seller reputation scores updated hourly

Feed Diversity:

```
Diversity Rules:
• No more than 3 consecutive posts from same seller
• Category distribution: At least 3 different categories in top 10
• Price variance: Mix of low/mid/high price points
• Content type mix: Products, tutorials, reviews, testimonials
```

---

2. Product Similarity (Vector Embeddings)

Purpose: Show similar products on product detail page

Where Used:

- Product detail page "Similar Products" section
- "You might also like" recommendations
- Alternative suggestions when out of stock

How It Works:

Text-Based Similarity:

```
Product: "Red Nike Running Shoes"
↓
Generate text embedding from:
```

```
"Red Nike Running Shoes. Comfortable athletic footwear
 designed for daily training and jogging. Category: Footwear."
↓
Embedding: [0.23, 0.87, 0.12, ... 384 numbers]
↓
Store in product.text_embedding column
↓
When user views product:
→ Compare this embedding with all other products
→ Find 10 closest matches using cosine similarity
→ Display as "Similar Products"
```

Image-Based Similarity:

```
Product Image: [actual shoe photo]
↓
Download image and process pixels
↓
Generate image embedding using CLIP:
[0.91, 0.12, 0.45, ... 512 numbers]
↓
Store in product.image_embedding column
↓
When user views product:
→ Compare this embedding with all other products
→ Find 10 visually similar matches
→ Display as "Visually Similar"
```

Combined Similarity:

```
Option to blend both:

Combined Score = (0.6 × Text Similarity) + (0.4 × Image Similarity)

Example:
Product A vs Product B:
• Text similarity: 0.85 (both describe athletic shoes)
• Image similarity: 0.72 (both look sporty)
• Combined: (0.6 × 0.85) + (0.4 × 0.72) = 0.51 + 0.29 = 0.80
```

```
Result: 80% similarity → Good recommendation!
```

When to Use Which:

Text Embedding (Semantic Similarity):

- User needs products with similar function
- Finding alternatives/substitutes
- Matching by features and description
- Example: "wireless earbuds" → finds AirPods, Galaxy Buds, etc.

Image Embedding (Visual Similarity):

- User cares about style/appearance
- Fashion and design-focused products
- Color and aesthetic matching
- Example: Minimalist white sneakers → finds all similar looking white shoes

Performance Considerations:

```
Similarity Search Performance:
├── Without index: ~2-5 seconds (scan all products)
├── With ivfflat index: ~50-200ms (much faster!)
└── With caching: ~10-30ms (best!)

Optimization:
• Create pgvector index on embedding columns
• Cache similar products per product (24 hour TTL)
• Pre-compute similarities for popular products
• Limit search to same category first (faster)
```

---

3. Search System

Purpose: Help users find products through text or image queries

A. Text Search (Semantic)

Traditional Keyword Search (What we're improving):

```
User searches: "comfortable work shoes"
↓
System finds: Products containing words "comfortable" AND "work" AND "shoes"
```

```
↓
Problems:
• Misses "office footwear" (different words, same meaning)
• Misses "business casual sneakers" (related but no exact match)
• Can't understand intent or context
```

Semantic Search (With Embeddings):

```
User searches: "comfortable work shoes"
↓
Generate embedding for query:
"comfortable work shoes" → [0.45, 0.78, 0.23, ...]
↓
Compare with ALL product text embeddings
↓
Find closest matches:
• "Business casual sneakers" □ (similarity: 0.89)
• "Office-appropriate footwear" □ (similarity: 0.86)
• "Professional dress shoes" □ (similarity: 0.82)
↓
Understands meaning, not just keywords!
```

Search Process:

```
Step 1: User Types Query
"gift for dad who likes tech"
↓
Step 2: Generate Query Embedding
Call Python service: "gift for dad who likes tech"
→ Returns: [0.34, 0.67, 0.91, ... 384 numbers]
↓
Step 3: Similarity Search
Compare query embedding with product text embeddings
Use cosine similarity to find closest matches
↓
Step 4: Ranking & Filtering
• Base ranking: Similarity score
• Boost: In stock items (+10%)
• Boost: Popular items (+5%)
• Boost: New arrivals (+5%)
```

```
• Filter: Remove out of budget (if known)
↓
Step 5: Return Results
Top 20 most relevant products
```

Search Enhancements:

```
Query Understanding:
├── Intent detection: "gift" → Show gift-appropriate items
├── Context: "for dad" → Skew towards masculine products
├── Category hint: "tech" → Filter to electronics/gadgets
└── Budget inference: Previous searches/purchases

Ranking Factors:
├── Semantic relevance (primary): 60%
├── Popularity/sales: 20%
├── Recency (new products): 10%
├── User's past preferences: 10%
```

B. Visual Search (Image Upload)

Purpose: Find products by uploading a photo

How It Works:

```
Step 1: User Sees Product in Real Life
User sees cool shoes on Instagram/street
Takes photo or saves image
↓
Step 2: Upload to Nexgate
User opens app → Camera icon → Upload photo
↓
Step 3: Generate Image Embedding
Send image bytes to Python CLIP service
Process actual pixels (not URL or filename!)
→ Returns: [0.91, 0.23, 0.67, ... 512 numbers]
↓
Step 4: Find Visually Similar Products
Compare upload embedding with all product image embeddings
Use cosine similarity
↓
```

```
Step 5: Return Results
"Found 20 similar items"
Show visually matching products
Include similarity % for transparency
```

Visual Search Use Cases:

```
Use Case 1: "Find This Exact Product"
User uploads: Photo of specific Nike shoe
System finds: That exact shoe (if in catalog)
Or: Visually identical alternatives

Use Case 2: "Find Similar Style"
User uploads: Minimalist white sneakers
System finds: All minimal white shoe styles
From different brands

Use Case 3: "Shop the Look"
User uploads: Full outfit photo
System identifies: Individual items
Shows: Matching products for each piece

Use Case 4: "Find Better Price"
User uploads: Expensive designer bag
System finds: Similar looking bags
At various price points
```

Performance:

```
Visual Search Speed:
├── Image upload: ~1 second
├── Embedding generation: ~200-300ms
├── Similarity search: ~100-200ms (with index)
└── Total: ~1.5 seconds (acceptable!)

Optimization:
• Compress uploaded images (reduce transfer time)
• Process images at consistent size (224x224 for CLIP)
• Cache embeddings for popular uploaded images
• Use GPU for embedding generation (5x faster)
```

4. Group Buy Recommendations

Purpose: Connect users with active group purchases and maximize group completion rates

A. Find Groups to Join

Scenario:

```
User viewing: iPhone 15 ($999)
User thinking: "Too expensive..."
↓
System shows:
"□□Join a group buy and save $150!
 □□3 active iPhone groups:
 • Group A: 8/10 people, 3 hours left → JOIN
 • Group B: 5/10 people, 24 hours left → JOIN
 • Group C: 2/10 people, 48 hours left → JOIN"
```

Recommendation Logic:

```
Finding Relevant Groups:

Step 1: Identify User's Interest
User action: Viewed/searched/added to cart iPhone
Interest level: High
↓
Step 2: Find Active Groups
Query all groups:
• Product: iPhone 15 (exact match)
• Status: Active (not expired/completed)
• Not full yet (has open spots)
↓
Step 3: Score Each Group
Group Score =
    (0.40 × Product Match) +
    (0.30 × Urgency/Time Left) +
    (0.20 × Fill Rate) +
    (0.10 × Discount Size)

Example:
```

```
Group A:
• Product Match: 100% (exact iPhone)
• Urgency: 90% (3 hours left = high urgency)
• Fill Rate: 80% (8/10 = almost there!)
• Discount: 15% ($150 savings)
Score = (0.40×100) + (0.30×90) + (0.20×80) + (0.10×15)
     = 40 + 27 + 16 + 1.5 = 84.5 □ TOP RECOMMENDATION!

Group C:
• Product Match: 100%
• Urgency: 20% (48 hours = low urgency)
• Fill Rate: 20% (2/10 = just started)
• Discount: 15%
Score = (0.40×100) + (0.30×20) + (0.20×20) + (0.10×15)
     = 40 + 6 + 4 + 1.5 = 51.5

↓
Step 4: Rank & Display
Show Group A first (highest score + urgency)
Display countdown timer
Show social proof (8 people already joined!)
```

Similar Product Groups (Using Embeddings):

```
Scenario: User wants iPhone but no active iPhone groups

System finds:
"No active iPhone groups, but:
 □□Samsung S24 Group: 6/10, 5 hours left → JOIN
 □□Google Pixel Group: 4/8, 12 hours left → JOIN"

How: Use text embeddings to find similar products
iPhone embedding ≈ Samsung/Google phone embeddings
Show groups for similar high-end smartphones
```

Urgency Triggers:

```
Smart Notifications:

When group is 80% full:
```

```
→ "⚡ Only 2 spots left in iPhone group! Join now!"

When group has < 6 hours left:
→ "⏰ Group closes in 5 hours! Don't miss 15% off!"

When user's friend joins:
→ "👥 Sarah just joined iPhone group! Join her?"

When price drops to user's budget:
→ "💰 Now $849 with group buy - fits your budget!"
```

B. Invite Friends to Your Group

Scenario:

```
User creates group: "MacBook Pro - Need 10 people"
Current: 3/10 members (user + 2 others)
↓
System suggests who to invite
```

Invitation Recommendation Logic:

```
Finding Best Invitees:

Step 1: Candidate Pool
├── User's followers
├── User's friends (mutual follows)
├── People user previously bought with
└── Active users who viewed similar products

Step 2: Score Each Potential Invitee
Invite Score =
    (0.35 × Product Interest) +
    (0.25 × Social Connection) +
    (0.20 × Past Group Activity) +
    (0.15 × Budget Match) +
    (0.05 × Platform Activity)

Example - Inviting John:
• Product Interest: 90% (viewed MacBooks 3 times this month)
• Social Connection: 80% (mutual friend, engaged with user's posts)
```

```
• Past Group Activity: 70% (joined 2 groups before, completed both)
• Budget Match: 85% (bought items in $800-1500 range)
• Platform Activity: 95% (logs in daily, active buyer)

Score = (0.35×90) + (0.25×80) + (0.20×70) + (0.15×85) + (0.05×95)
      = 31.5 + 20 + 14 + 12.75 + 4.75 = 83 □ EXCELLENT MATCH!

↓
Step 3: Personalized Invitation Message
"Invite John:
 □□He viewed MacBook Pro recently
 □□Mutual friend with Sarah (already in group)
 □ Reliable (completed 2 past groups)"

[SEND INVITE]
```

Social Proof in Invitations:

```
Smart Messaging:

To close friend:
"Hey! I'm buying a MacBook with 9 others. Join us?
 We save $200 each! Only 1 spot left."

To network connection:
"Group buy for MacBook Pro
 3 of your friends already joined
 Save 15% | 6 spots left"

To stranger (similar interests):
"MacBook Pro group buy
 Popular in tech community
 Join 9 others | Save $200"
```

C. Complete the Group (Urgency Marketing)

Scenario:

```
Group status: 8/10 members, 4 hours left
Need: 2 more people to unlock deal
Risk: Group expires without completion
```

Completion Strategy:

```
System Actions:

Action 1: Broadcast to High-Intent Users
Target:
├── Users who viewed this product
├── Users with product in cart
├── Users who joined similar groups
└── Friends of current members

Notification:
"⚡ URGENT: 2 spots left!
 iPhone group closes in 4 hours
 Join now and save $150"

Action 2: Discount Boost (If needed)
If group stuck at 80% for 2+ hours:
→ Platform adds extra 5% discount
→ "New deal: Save $150 → Save $170!"
→ Creates momentum

Action 3: Social Pressure
Show current members:
"Your group needs 2 more people
 Share with friends to complete deal
 [SHARE ON WHATSAPP] [SHARE ON INSTAGRAM]"

Action 4: Fallback Options
If group fails:
→ "Sorry, group didn't complete
   □□Join this similar group instead: [...]
   OR: Get notified when new iPhone group starts"
```

Group Recommendation Priorities:

```
Priority Matrix:

Urgent + Almost Full = Highest Priority
├── 90% full + < 6 hours left
```

```
├── Show on feed prominently
├── Push notifications
└── Email reminders

High Interest Match = High Priority
├── Exact product user wants
├── Similar products (embeddings)
├── Category match
└── Show in search results

Social Connection = Medium Priority
├── Friends in group
├── Followed sellers
├── Past group partners
└── Show in "Friends' Activity"

General Discovery = Low Priority
├── Trending groups
├── Category exploration
├── New group announcements
└── Show in feed occasionally
```

---

5. Installment Recommendations

Purpose: Make expensive products affordable through payment plans

Key Principle: Show the right payment option at the right moment

A. "You Can Afford This!"

Scenario:

```
User profile:
• Average purchase: $50-200
• Max purchase: $400
• Never bought > $600

User views: MacBook Pro ($1,299)
Typical reaction: "Too expensive, can't afford"
```

Installment Intervention:

```
System detects:
Product price ($1,299) > User max purchase ($400) × 3
↓
Calculate monthly payment:
$1,299 ÷ 12 months = $108/month
↓
Check affordability:
$108/month < User max purchase ($400)? □ YES
↓
Show prominent banner:
┌─────────────────────────────────────┐
│ □□You can afford this!              │
│ Pay just $108/month for 12 months   │
│ • No interest                       │
│ • Pay ahead anytime                 │
│ • Cancel anytime                    │
│ [SEE PAYMENT OPTIONS]               │
└─────────────────────────────────────┘
```

Affordability Logic:

```
Trigger Rules:

Show Installment When:
1. Product price > (User's average purchase × 3)
2. Monthly payment < (User's max purchase × 1.5)
3. Product in user's interested categories
4. User has good payment history (if existing customer)

Example:
Product: $1,299
User avg: $150
User max: $400

Check 1: $1,299 > ($150 × 3) = $1,299 > $450 □
Check 2: $108/month < ($400 × 1.5) = $108 < $600 □
→ SHOW INSTALLMENT OPTION!
```

```
Don't Show If:
• Product too cheap (< $200)
• User can afford full price easily
• Monthly payment still too high for user
```

Payment Plan Recommendations:

```
Personalized Plans:

For Budget-Conscious User:
"12 months at $108/month" (smallest monthly)

For Established User:
"6 months at $216/month" (faster payoff)

For Premium User:
"3 months at $433/month" (minimize interest duration)

Dynamic Suggestion:
Based on user's typical payment speed:
• Pays bills early → Suggest shorter term
• Budget constrained → Suggest longer term
• First-time → Suggest flexible middle option
```

B. Premium Product Feed

Scenario:

```
User interested in: Laptops
User budget: $300-500
Problem: Premium laptops cost $1000+
```

Solution - Installment Feed Section:

```
┌─────────────────────────────────────┐
│ □□Premium Products - Easy Payments │
├─────────────────────────────────────┤
│ MacBook Pro                         │
│ $1,299 → Just $108/month            │
│ □ 4.9 stars | □□Hot deal           │
│ "Fits your monthly budget!"         │
```

```
├──────────────────────────────────────┤
│ Dell XPS 15                          │
│ $1,199 → Just $100/month             │
│ □ 4.8 stars | □□Professional      │
│ "Popular in your network"            │
└──────────────────────────────────────┘
```

Feed Scoring for Installment Products:

```
Installment Product Score =
    (0.40 × Category Interest) +
    (0.30 × Affordability with Installment) +
    (0.20 × Product Quality) +
    (0.10 × Social Proof)

Example - MacBook for User:
• Category: 95% (loves tech, views laptops often)
• Affordability: 90% ($108/month fits budget)
• Quality: 98% (4.9 star rating, premium brand)
• Social: 75% (3 friends own MacBooks)

Score = (0.40×95) + (0.30×90) + (0.20×98) + (0.10×75)
     = 38 + 27 + 19.6 + 7.5 = 92.1 □□□

This scores higher than cheaper laptops because:
• Installments make it affordable
• Matches user's aspirations
• Higher quality product
```

C. Upgrade Suggestions

Scenario:

```
User adding to cart: iPhone 15 ($799)
Better option exists: iPhone 15 Pro ($999)
Difference: $200
```

Upgrade Recommendation:

```
┌──────────────────────────────────────┐
│ □□Upgrade to iPhone 15 Pro?        │
```

```
│                                    │
│ Current choice: $799               │
│ iPhone 15 Pro: $999 (+$200)        │
│                                    │
│ With 12-month plan:                │
│ • Standard: $67/month              │
│ • Pro: $83/month (+$16)            │
│                                    │
│ Pro benefits:                      │
│ • Better camera system             │
│ • Titanium design                  │
│ • More storage                     │
│                                    │
│ [UPGRADE FOR JUST $16/MONTH]       │
└────────────────────────────────────┘
```

Upgrade Logic:

```
When to Suggest Upgrade:

Condition 1: Premium version exists
iPhone 15 → iPhone 15 Pro □

Condition 2: Monthly difference is small
Standard: $67/month
Pro: $83/month
Difference: $16/month (< $20) □

Condition 3: User can afford it
User's budget allows +$16/month □

Condition 4: Meaningful upgrade
Pro version has substantial improvements □

→ SHOW UPGRADE SUGGESTION!

Don't Suggest If:
• Difference > $30/month (too much)
• Upgrade is minimal (not worth it)
• User explicitly chose budget option
```

```
• User on tightest plan already
```

D. Bundle Recommendations

Scenario:

```
User purchased: MacBook Pro ($1,299)
Payment plan: $108/month for 12 months
```

Bundle Suggestion:

```
┌─────────────────────────────────────┐
│ □□Complete Your Setup               │
│                                     │
│ Add to your payment plan:           │
├─────────────────────────────────────┤
│ □□□Magic Mouse                      │
│ $79 → +$7/month                     │
│ New total: $115/month               │
├─────────────────────────────────────┤
│ ⌨□ Magic Keyboard                   │
│ $99 → +$8/month                     │
│ New total: $123/month               │
├─────────────────────────────────────┤
│ □□Laptop Sleeve                     │
│ $49 → +$4/month                     │
│ New total: $112/month               │
│                                     │
│ □□Still within your budget!         │
└─────────────────────────────────────┘
```

Bundle Logic:

```
Accessory Recommendation Rules:

Rule 1: Complementary Products
Main: MacBook
Suggest: Mouse, keyboard, bag, USB-C accessories
(Use collaborative filtering: "bought together")

Rule 2: Affordable Addition
```

```
Current: $108/month
User budget: Up to $200/month
Available: $92/month for accessories
→ Suggest items totaling < $90/month

Rule 3: Prioritize by Value
High priority:
• Essential (mouse, charger)
• High satisfaction (4.5+ stars)
• Popular (many bought together)

Low priority:
• Optional accessories
• Lower rated items
• Expensive add-ons

Rule 4: Convenience Factor
"Add all 3 accessories for +$19/month
 Save time, get complete setup!"
```

E. Payment Flexibility Highlights

Key Feature: Users can pay ahead on installments

Recommendation Angle:

```
┌──────────────────────────────────────┐
│ □□Flexible Payments                  │
│                                      │
│ Start at $108/month                  │
│ • Got a bonus? Pay extra anytime     │
│ • Finish early, save on duration     │
│ • No penalties for early payment     │
│ • Pause if needed (1 month grace)    │
│                                      │
│ "Pay at your own pace!"              │
└──────────────────────────────────────┘
```

When to Highlight:

```
For New Users:
"Flexible payment plans available
 Pay early, pay later - you choose"

For Hesitant Users:
"Not sure? Start with small payments
 You can always pay more when ready"

For Seasonal Workers:
"Pay extra during busy season
 Reduce payments during slow months"

For Goal-Oriented Users:
"Finish your payments early
 Track progress in dashboard"
```

---

6. Combined Recommendations (Group Buy + Installments)

The Ultimate Deal Strategy

Scenario:

```
Product: MacBook Pro
Regular price: $1,299
Group buy discount: $200
Installment option: Available
```

Combined Offer:

```
┌─────────────────────────────────────┐
│ □□BEST DEAL COMBO!                  │
│                                     │
│ MacBook Pro                         │
│ Regular: $1,299                     │
│                                     │
│ □□Join Group: $1,099 (Save $200!)   │
│ □□+ Pay Monthly: $92/month          │
│                                     │
```

```
| □□7/10 people | □ 6 hours left    |
|                                    |
| □ Best combo deal:                 |
| • Save $200 with group             |
| • Just $92/month affordable        |
| • No interest                      |
|                                    |
| [JOIN GROUP + START PAYMENTS]      |
└────────────────────────────────────┘
```

Combined Score Formula:

```
Ultimate Deal Score =
    (0.25 × Category Interest) +
    (0.25 × Affordability Boost) +
    (0.20 × Group Discount Size) +
    (0.15 × Social Proof) +
    (0.10 × Urgency) +
    (0.05 × Past Group Success)

Example - MacBook for User:
• Category: 95% (loves tech)
• Affordability: 95% ($92/month from $108!)
• Discount: 100% ($200 is huge savings)
• Social: 70% (7 people joined)
• Urgency: 90% (6 hours left!)
• Past: 80% (completed 2 groups)

Score = (0.25×95) + (0.25×95) + (0.20×100) +
        (0.15×70) + (0.10×90) + (0.05×80)
     = 23.75 + 23.75 + 20 + 10.5 + 9 + 4
     = 91 □□□ MAXIMUM APPEAL!
```

Why Combined Works:

```
Psychological Triggers:

1. Double Savings:
   "Save $200 + Pay less monthly = Win-Win"
```

```
2. Urgency:
   "Group closes in 6 hours! Act now!"

3. Social Proof:
   "7 others already in! Don't miss out!"

4. Affordability:
   "$92/month fits your budget easily"

5. Low Risk:
   "Flexible payments + Group guarantee"

Conversion Multiplier:
• Group buy alone: 8% conversion
• Installment alone: 15% conversion
• COMBINED: 25-30% conversion! □□
```

Feed Placement:

```
Priority Positioning:

Top of Feed:
• Active group + installment combos
• Closing soon (<6 hours)
• User's interested categories

Mid-Feed:
• New group + installment deals
• Popular combos (high join rate)
• Category exploration

Bottom Feed:
• General group buy awareness
• Installment education
• Success stories
```

? Embedding System Deep Dive

What Are Embeddings? (Conceptual Understanding)

Simple Analogy:

```
Think of embeddings as "coordinates" for meaning:

Words/Images → Machine Learning Model → Numbers (coordinates)

Similar meanings → Similar coordinates → Close together in space

Example in 2D (real embeddings are 384 or 512 dimensions!):

"Red Shoes"      → Point (0.8, 0.2)
"Crimson Sneakers" → Point (0.82, 0.18)  ← Close!
"Blue Laptop"    → Point (0.1, 0.9)      ← Far away!

Distance between "Red Shoes" and "Crimson Sneakers" is small
→ They're similar!
```

Architecture of Embedding System

```
┌─────────────────────────────────────────────┐
│          SPRING BOOT BACKEND                │
│                                             │
│  Need embedding for product/search query?   │
│              ↓                              │
│  HTTP REST Call to Python Service           │
│  POST /embed-text or POST /embed-image      │
│  Body: {text: "..."} or {image_bytes: "..."} │
└──────────────┬──────────────────────────────┘
               │
               │ HTTP Request
               ↓
┌─────────────────────────────────────────────┐
│       PYTHON EMBEDDING SERVICE              │
│       (Flask Microservice)                  │
│                                             │
```

```
│  ┌──────────────────────────────────────────┐  │
│  │  Models Loaded in Memory (at startup):│  │
│  │                                          │  │
│  │  • Sentence Transformers              │  │
│  │    (all-MiniLM-L6-v2)                 │  │
│  │    → Text embeddings (384 dims)       │  │
│  │                                          │  │
│  │  • CLIP                                  │  │
│  │    (openai/clip-vit-base-patch32)     │  │
│  │    → Image embeddings (512 dims)      │  │
│  └──────────────────────────────────────────┘  │
│                                                │
│  Receives request → Processes → Returns array│
└──────────────────┬─────────────────────────────┘
                   │
                   │ Response: [0.23, 0.87, ...]
                   ↓
┌──────────────────────────────────────────────┐
│          SPRING BOOT BACKEND                 │
│                                              │
│  Receives embedding array                    │
│              ↓                               │
│  Stores in PostgreSQL (vector column)        │
│  OR: Uses for similarity search              │
└──────────────────────────────────────────────┘
```

When Embeddings Are Generated

Trigger 1: Product Creation/Update

```
Flow:
Admin adds product
→ Product saved to database (ID assigned)
→ Background async job triggered
→ Generate text embedding (name + description)
→ Generate image embedding (from image URL)
→ Update product record with both embeddings
→ Product now searchable by similarity!

Timeline:
```

• Product creation: Immediate (< 100ms)
• Embedding generation: Background (2-5 seconds)
• Total user-facing delay: None (async)

Trigger 2: Search Query

Flow:
User types search query
→ Frontend sends to backend
→ Backend calls Python service
→ Python generates query embedding
→ Backend compares with product embeddings
→ Returns similar products
→ Frontend displays results

Timeline:
• User types → Results: ~500-800ms
• Embedding generation: ~50-100ms
• Search: ~100-200ms
• Network + rendering: ~300-500ms

Trigger 3: Image Upload (Visual Search)

Flow:
User uploads image
→ Frontend sends image bytes
→ Backend calls Python service
→ Python generates image embedding
→ Backend finds similar product images
→ Returns visually similar products

Timeline:
• Image upload: ~500ms-1s (depends on size/network)
• Embedding generation: ~200-400ms
• Search: ~100-200ms
• Total: ~1-2 seconds (acceptable for visual search)

Trigger 4: Batch Processing

```
For existing products without embeddings:

Flow:
Admin triggers batch job
→ Fetch products in batches (100 at a time)
→ Generate embeddings for batch
→ Save all embeddings
→ Repeat for next batch

Timeline:
• 100 products: ~10-15 seconds
• 10,000 products: ~15-20 minutes
• Run once, or nightly for new products
```

Embedding Storage

Data Stored:

```
Product Table includes:
├── id (primary key)
├── name, description, price, etc.
├── text_embedding (vector type, 384 dimensions)
└── image_embedding (vector type, 512 dimensions)

Each embedding stored as array of floats:
text_embedding: [0.234, 0.876, 0.123, ... 384 numbers]
image_embedding: [0.912, 0.234, 0.567, ... 512 numbers]

Storage per product:
• Text: 384 floats × 4 bytes = 1.5 KB
• Image: 512 floats × 4 bytes = 2.0 KB
• Total: 3.5 KB per product

For 10,000 products: 35 MB (tiny!)
For 1,000,000 products: 3.5 GB (manageable!)
```

Similarity Search Process

How Similarity Is Calculated:

```
Cosine Similarity Formula:

similarity = (A · B) / (||A|| × ||B||)

Where:
A · B = dot product (sum of multiplying each pair)
||A|| = magnitude of vector A
||B|| = magnitude of vector B

Result: Number between 0 (completely different) and 1 (identical)

Example:
Product A: [0.8, 0.6]
Product B: [0.9, 0.5]

A · B = (0.8 × 0.9) + (0.6 × 0.5) = 0.72 + 0.30 = 1.02
||A|| = √(0.8² + 0.6²) = √1.00 = 1.0
||B|| = √(0.9² + 0.5²) = √1.06 = 1.03

similarity = 1.02 / (1.0 × 1.03) = 0.99 (very similar!)
```

Search Performance:

```
Without Index:
• Compares with every product
• 10,000 products: ~2-5 seconds
• 100,000 products: ~20-50 seconds
• NOT acceptable!

With pgvector Index (ivfflat):
• Uses approximate nearest neighbor (ANN)
• Groups similar vectors together
• Only searches relevant groups
• 10,000 products: ~50-100ms
• 100,000 products: ~100-300ms
• 1,000,000 products: ~200-500ms
• Acceptable! □

Trade-off:
• 100% accuracy vs 95-99% accuracy
```

```
• Worth it for speed improvement
```

Model Selection

Text Embeddings:

```
Model: all-MiniLM-L6-v2 (Sentence Transformers)

Pros:
 Fast (50ms per embedding on CPU)
 Small size (90 MB model)
 Good quality for e-commerce
 384 dimensions (manageable)
 Free and open source

Use for:
• Product descriptions
• Search queries
• Category matching
• Semantic similarity

Alternative if needed:
• all-mpnet-base-v2 (better quality, slower, 768 dims)
• multilingual models (if serving multiple countries)
```

Image Embeddings:

```
Model: CLIP (openai/clip-vit-base-patch32)

Pros:
 Understands both images AND text
 Can search images with text!
 Good visual similarity
 512 dimensions
 Free and open source

Use for:
• Product images
• Visual search
```

```
• Style matching
• "Shop the look"

Unique Feature:
Can compare text with images!
Query: "red shoes" (text embedding)
Find: Red shoe products (image embedding)
Cross-modal search! □□
```

Optimization Strategies

Caching:

```
Strategy 1: Cache Similar Products
• After computing similarities, cache results
• TTL: 24 hours (products don't change often)
• Key: product_id
• Value: [similar_product_ids]

Strategy 2: Cache Query Embeddings
• Common searches generate same embeddings
• Cache: "red shoes" → [0.23, 0.87, ...]
• TTL: 1 hour
• Saves Python service calls

Strategy 3: Cache Popular Product Embeddings
• Keep hot product embeddings in Redis
• Faster than PostgreSQL lookup
• Updates when product changes
```

Batch Processing:

```
Instead of:
• Generate 1 embedding → 50ms
• Generate 100 embeddings → 5 seconds (sequential)

Better:
• Generate 100 embeddings in batch → 2 seconds!
• 2.5x faster by batching
```

```
Use for:
• Initial product catalog population
• Nightly updates
• Bulk imports
```

Index Tuning:

```
pgvector index parameters:

lists: Number of clusters
• More lists = faster search, less accuracy
• Fewer lists = slower search, more accuracy
• Sweet spot: √(number of products)
• 10,000 products → lists = 100
• 100,000 products → lists = 316

Recommendation:
Start with lists = 100
Monitor search speed and accuracy
Adjust if needed
```

? Feed Algorithm (Complete Flow)

Feed Generation Process (Step-by-Step)

User Opens App → Request Feed

```
REQUEST:
GET /api/feed?limit=20

HEADERS:
Authorization: Bearer {user_token}
```

Backend Process:

Step 1: User Profile Loading (< 50ms)

```
Check Redis cache:
• User's following list (Key: user:{id}:following)
• User's interested categories (Key: user:{id}:categories)
• User's seen posts (Key: user:{id}:seen_posts)

If not cached:
• Query PostgreSQL
• Store in Redis (TTL: 1 hour for following, 6 hours for categories)
```

Step 2: Candidate Gathering (< 200ms)

```
Parallel Queries (executed simultaneously):

Query A: Posts from Following
• Get posts from last 7 days
• From sellers user follows
• Not seen by user
• In stock
→ Returns ~40-60 posts

Query B: Trending in Categories
• User's top 3 categories
• High engagement rate (>5%)
• Posted in last 48 hours
• Marked as trending (pre-computed)
→ Returns ~20-30 posts

Query C: Local Sellers
• Same city/region as user
• Active posts (last 7 days)
• Not seen by user
→ Returns ~10-20 posts

Query D: Sponsored
• Active campaigns
• Targeted to user's interests
• Budget remaining
→ Returns ~5-10 posts

Total Candidates: ~80-120 posts
```

Step 3: Filtering (< 50ms)

```
Remove:
• Already seen by user (Redis lookup)
• Out of stock products
• Blocked/reported sellers
• Duplicate products
• Posts from blocked users

Remaining: ~60-100 posts
```

Step 4: Scoring Each Post (< 300ms)

```
For each post, calculate score (0-100):

A. Social Score (35 points max):
├── Following seller? +15
├── Friends engaged? +2 per friend (max 10)
├── Seller verified? +5
└── Seller rating: +0 to 5 (based on stars)

B. Engagement Prediction (25 points):
├── Category match? +10
├── Price in range? +8
└── Similar to past likes? +7

C. Personalization (20 points):
├── Matches search keywords? +2 each (max 8)
├── Local seller? +7
└── Language/preferences? +5

D. Recency (15 points):
├── <1 hour: +15
├── <6 hours: +12
├── <24 hours: +8
├── <3 days: +4
└── Older: +1

E. Quality (5 points):
├── 3+ images? +2
```

```
├── Detailed description? +1
└── High engagement rate? +2

Example Post Score:
• Social: 15 (following) + 4 (2 friends) + 5 (verified) + 4.5 (rating) = 28.5
• Engagement: 10 (category) + 8 (price) + 7 (similar) = 25
• Personalization: 4 (keywords) + 7 (local) + 5 (prefs) = 16
• Recency: 12 (<6 hours) = 12
• Quality: 2 (images) + 1 (description) + 2 (engagement) = 5
→ Total: 86.5 points □□□
```

Step 5: Ranking & Diversity (< 100ms)

```
Primary Sort: By score (highest first)

Apply Diversity Rules:
1. No more than 3 consecutive posts from same seller
   If post N and N+1 and N+2 are from Seller A:
   → Move post N+2 down

2. Category distribution in top 10
   Count categories in positions 1-10
   If one category > 5 posts:
   → Demote some, promote others

3. Price variance
   Check price distribution in top 20
   If all high-priced or all low-priced:
   → Inject middle-price items

4. Content type mix
   Aim for: 70% product posts, 20% reviews, 10% tutorials
   Adjust positions to achieve balance

5. Freshness boost (random)
   Randomly boost 2-3 recent high-quality posts (score >70)
   Prevents feed from being too predictable

6. Sponsored insertion
   Insert 1 sponsored post at position 7-10
```

```
   Feels native, not intrusive
```

Step 6: Pagination & Response (< 50ms)

```
Take top 20 posts from ranked list

Generate cursor:
• cursor = ID of 20th post
• Next request uses: ?cursor=12345&limit=20
• Returns posts with ID < 12345

Mark as seen:
• Add post IDs to Redis set: user:{id}:seen_posts
• TTL: 30 days
• Prevents showing same posts again

Track impressions:
• Log to analytics: user viewed these posts
• Used for engagement metrics
• Improve future recommendations

Response format:
{
  "posts": [...20 posts with full data...],
  "nextCursor": "12345",
  "hasMore": true
}
```

Total Time: ~750ms (well under 1 second target!)

Feed Diversity Strategies

Why Diversity Matters:

```
Without diversity:
• All posts from one seller (boring!)
• All same category (no discovery)
• All same price range (limits audience)
• Predictable feed (user stops scrolling)
```

```
With diversity:
• Varied content (keeps interest)
• Category discovery (impulse buys)
• Price options (something for everyone)
• Surprising finds (engagement boost)
```

Diversity Techniques:

1. Temporal Diversity (Avoid Staleness)

```
Problem: Old high-scored posts dominate

Solution: Recency Decay
• Posts >3 days old: Reduce score by 30%
• Posts >7 days old: Reduce score by 60%
• Forces fresh content into feed

Balance:
• Still show quality old posts if very high score
• But prioritize newer content generally
```

2. Seller Diversity (Avoid Spam Feeling)

```
Problem: Popular seller floods feed

Solution: Consecutive Limit
• Track last 3 posts shown
• If post N, N+1, N+2 from same seller:
• Demote post N+2 by 20 positions
• Gives other sellers chance

Exception:
• User explicitly follows only one seller
• Then okay to show more from that seller
```

3. Category Diversity (Enable Discovery)

```
Problem: User pigeonholed into one category

Solution: Category Quotas in Top 20
```

• Max 8 posts from dominant category
• Min 2 posts from each of user's top 3 categories
• 2-4 posts from exploratory categories

Example:
User loves "Electronics" (dominant)
Also views "Fashion", "Home"
Feed includes:
• 7 Electronics posts
• 4 Fashion posts
• 3 Home posts
• 3 Sports posts (exploratory)
• 3 Beauty posts (exploratory)

4. Price Diversity (Cater to Moods)

Problem: All luxury or all budget items

Solution: Price Distribution
Target in top 20 posts:
• 30% Budget (<$50)
• 40% Mid-range ($50-200)
• 30% Premium (>$200)

Rationale:
• Some users browsing casually (show budget)
• Some ready to buy (show mid-range)
• Some aspirational shopping (show premium)
• Different moods, different needs

5. Content Type Diversity

Mix post types:
• 70% Direct product posts (main content)
• 15% Review/testimonial posts (social proof)
• 10% Tutorial/how-to posts (educational value)
• 5% Behind-the-scenes (brand building)

Prevents feed from being pure sales pitch
Provides value beyond "buy now"

Trending Algorithm

What Makes Content "Trending"?

Criteria:

```
1. High Engagement Rate
   Likes + Comments + Shares ÷ Views > 10%

2. Velocity (Speed of engagement)
   Gaining engagement faster than average

3. Recency
   Posted within last 48 hours

4. Social Spread
   Engaged by users from different networks
   (not just one seller's followers)
```

Trending Score Formula:

```
Trending Score =
    (0.40 × Engagement Rate) +
    (0.30 × Velocity Score) +
    (0.20 × Reach Score) +
    (0.10 × Recency Boost)

Where:
Engagement Rate = (Likes + Comments × 3 + Shares × 5) / Views
Velocity Score = Current hourly rate ÷ Average hourly rate
Reach Score = Unique user networks engaged / Total networks
Recency Boost = 100 if <6 hours, 80 if <24 hours, 60 if <48 hours

Example:
Post:
• 1000 views, 80 likes, 20 comments, 10 shares
• Posted 4 hours ago
• Gaining 15 engagements/hour (average: 5/hour)
• Engaged by users from 8 different networks
```

```
Engagement Rate = (80 + 20×3 + 10×5) / 1000 = 190/1000 = 0.19 = 19%
Velocity = 15/5 = 3.0 (3x faster than average)
Reach = 8/20 active networks = 0.40 = 40%
Recency = 100 (<6 hours)

Trending Score = (0.40×19) + (0.30×3×100/3) + (0.20×40) + (0.10×100)
               = 7.6 + 30 + 8 + 10
               = 55.6 → Mark as trending!
```

Trending Update Frequency:

```
Background job runs every 5 minutes:
1. Query posts from last 48 hours
2. Calculate trending scores
3. Mark posts with score >50 as trending
4. Store in Redis (key: trending_posts)
5. Feed service reads from this cache

Why every 5 minutes?
• Balance between freshness and server load
• Trending changes relatively slowly (minutes, not seconds)
• Users won't notice 5-minute delay
```

? Search System (Complete Flow)

Text Search Process

User Types Query → Get Results

```
REQUEST:
GET /api/search?q=comfortable+running+shoes&limit=20

BACKEND PROCESS:
```

Step 1: Query Processing (< 50ms)

```
Input: "comfortable running shoes"

Clean & Normalize:
• Lowercase: "comfortable running shoes"
• Remove special chars
• Trim whitespace
• Tokenize: ["comfortable", "running", "shoes"]

Check for special patterns:
• Price filter: "under $100"
• Color: "red", "blue"
• Brand: "nike", "adidas"
• Size: "size 10", "large"

Extract filters if present:
query = "comfortable running shoes"
filters = {price_max: 100, category: "footwear"}
```

Step 2: Generate Query Embedding (< 100ms)

```
Call Python Embedding Service:
POST http://embedding-service:5000/embed-text
Body: {"text": "comfortable running shoes"}

Python processes:
• Sentence Transformers model loaded in memory
• Generates embedding: [0.45, 0.78, 0.23, ... 384 numbers]

Returns: {"embedding": [0.45, 0.78, ...], "dimensions": 384}

Spring Boot receives embedding
```

Step 3: Similarity Search (< 200ms)

```
PostgreSQL query with pgvector:

Find products where:
• text_embedding similar to query embedding (cosine similarity)
• Apply filters (price, category, etc.)
• In stock
```

```
• Not blocked

Using pgvector index (ivfflat):
• Fast approximate nearest neighbor search
• Returns top 100 candidates with similarity scores
```

Step 4: Ranking & Boosting (< 100ms)

```
Base Ranking: Similarity score (0-1)

Apply Boosts:
• In stock: +10% score
• Popular (high sales): +8% score
• New arrival (<30 days): +5% score
• High rated (4.5+ stars): +5% score
• Exact keyword match in name: +15% score

Apply User Personalization:
• User's favorite brands: +10% score
• User's typical price range: +8% score
• User previously viewed: +5% score

Penalties:
• Low stock warning: -5% score
• Low rating (<3.5 stars): -10% score
• No image: -8% score

Final Sort: By boosted score
```

Step 5: Response (< 50ms)

```
Take top 20 results

Format response:
{
  "query": "comfortable running shoes",
  "results": [
    {
      "id": 123,
      "name": "Nike Air Max Running Shoes",
```

```
      "price": 89.99,
      "similarity": 0.92,
      "rating": 4.7,
      ...
    },
    ... 19 more
  ],
  "total": 847,
  "took": "287ms"
}
```

Total Time: ~500ms (good search experience!)

Visual Search Process

User Uploads Image → Get Visually Similar Products

```
REQUEST:
POST /api/search/visual
Content-Type: multipart/form-data
Body: image file

BACKEND PROCESS:
```

Step 1: Image Reception & Validation (< 100ms)

```
Receive uploaded file

Validate:
• File size < 10MB
• Format: JPG, PNG, WEBP
• Not corrupted

Preprocess:
• Resize if needed (max 1024x1024)
• Convert to RGB if grayscale
• Optimize file size

Temporarily store:
```

```
• Option A: Memory (for immediate processing)
• Option B: Temp file system (for large images)
• Option C: S3 (for tracking/analytics)
```

Step 2: Generate Image Embedding (< 300ms)

```
Convert image to bytes

Call Python Embedding Service:
POST http://embedding-service:5000/embed-image-bytes
Body: {"image_base64": "iVBORw0KGgo..."}

Python processes:
• CLIP model loaded in memory
• Processes image pixels (not filename!)
• Understands visual features: color, shape, style
• Generates embedding: [0.91, 0.23, 0.67, ... 512 numbers]

Returns: {"embedding": [0.91, 0.23, ...], "dimensions": 512}

Spring Boot receives embedding
```

Step 3: Visual Similarity Search (< 200ms)

```
PostgreSQL query with pgvector:

Find products where:
• image_embedding similar to upload embedding
• In stock
• Has image (obviously)

Using pgvector index on image_embedding:
• Fast ANN search
• Returns top 50 candidates with visual similarity scores
```

Step 4: Ranking & Filtering (< 100ms)

```
Base Ranking: Visual similarity (0-1)

Filter Options (if user specifies):
• Category filter: "Show only shoes"
```

```
• Price range: "$50-150"
• Brand preference

Apply Boosts:
• Popular in category: +10%
• High quality images: +5%
• Multiple product images: +5%

Sort by final score
```

Step 5: Response (< 50ms)

```
Return top 20 visually similar products

Response format:
{
  "results": [
    {
      "id": 456,
      "name": "Similar Red Sneakers",
      "image_url": "...",
      "similarity": 0.88,
      "price": 79.99
    },
    ... 19 more
  ],
  "message": "Found 20 visually similar products"
}
```

Total Time: ~750ms-1s (acceptable for visual search!)

Search Enhancements

Query Understanding (NLP Techniques):

```
Intent Detection:

"gift for dad" → Gift intent
→ Boost: Gift-appropriate products
```

```
→ Filter: Hide inappropriate items

"cheap laptop" → Budget intent
→ Sort: Price low to high
→ Filter: <$500

"best running shoes" → Quality intent
→ Sort: Rating high to low
→ Boost: Reviews, testimonials

"red nike shoes size 10" → Specific intent
→ Exact filters applied
→ Precise matching
```

Spell Correction:

```
User types: "iphone 15 pro mac"
System detects: "mac" likely typo
Suggestion: "Did you mean: iphone 15 pro max?"

Auto-correct for common typos:
• "iphone" → "iPhone"
• "macbook" → "MacBook"
• "airpods" → "AirPods"
```

Search Suggestions (As User Types):

```
User types: "run"
Suggestions appear:
• "running shoes" (popular)
• "running shorts" (trending)
• "running watch" (category)

Based on:
• Popular searches (last 7 days)
• User's past searches
• Trending products
• Category completions
```

Related Searches:

```
After search results, suggest:
"People also searched for:"
• "nike running shoes" (brand specific)
• "trail running shoes" (category variant)
• "running shoes for women" (gender variant)

Generated from:
• Search session data (what others searched next)
• Similar query embeddings
• Category relationships
```

?? Technology Stack

Backend (Spring Boot)

Core Framework:

- Spring Boot 3.2+
- Java 17 or 21
- Spring Web (REST APIs)
- Spring Data JPA (Database)
- Spring Security (Authentication)
- Spring WebSocket (Real-time features)

Libraries & Dependencies:

- Lombok (reduce boilerplate)
- MapStruct (object mapping)
- Hibernate (ORM)
- Jackson (JSON processing)
- Resilience4j (circuit breaker, retry)
- Micrometer (metrics)

Database

Primary Database:

- PostgreSQL 16+
- pgvector extension (for vector similarity)

Caching:

- Redis 7+
- Used for: sessions, feed cache, seen posts, hot data

Embedding Service (Python)

Framework:

- Flask or FastAPI
- Python 3.10+

ML Libraries:

- Sentence Transformers (text embeddings)
- Transformers (Hugging Face)
- CLIP (image embeddings)
- PyTorch or TensorFlow
- NumPy, Pandas

Models:

- Text: all-MiniLM-L6-v2 (384 dimensions)
- Image: openai/clip-vit-base-patch32 (512 dimensions)

Infrastructure

Containerization:

- Docker
- Docker Compose (local development)

Orchestration (Production):

- Kubernetes (if scaling large)
- OR Docker Swarm (simpler alternative)
- OR Cloud services (AWS ECS, Google Cloud Run)

Cloud Services (Optional):

- AWS: S3 (images), RDS (database), ElastiCache (Redis)
- Google Cloud: Cloud Storage, Cloud SQL, Memorystore
- Azure: Blob Storage, Azure Database, Azure Cache

Monitoring & Observability

Metrics:

- Prometheus (metrics collection)
- Grafana (visualization)
- Spring Boot Actuator (health endpoints)

Logging:

- Logback or Log4j2
- ELK Stack (Elasticsearch, Logstash, Kibana) optional
- OR Cloud logging (CloudWatch, Stackdriver)

Tracing:

- Spring Cloud Sleuth + Zipkin (optional)
- Jaeger (distributed tracing)

? Implementation Phases

Phase 1: MVP - Basic Recommendations (Weeks 1-3)

Goal: Launch with working feed and basic recommendations

Features to Implement:

Social Feed (Core)

- User can create posts about products
- Follow/unfollow sellers
- Like, comment, share posts
- Basic feed: Posts from following + trending
- Simple scoring (social + recency only)
- Cursor-based pagination

Basic Product Similarity

- "Similar Products" using category + price range
- No embeddings yet (too complex for MVP)
- Collaborative filtering: "Bought together"

Simple Search

- Keyword-based search
- Filter by category, price
- Sort by relevance, price, rating
- No semantic search yet

Engagement Tracking

- Track views, likes, comments, shares
- Store user activity (views, searches)
- Basic analytics

Group Buy (Core)

- Create group purchase
- Join existing group
- Group countdown timer
- Basic "find groups" (category match)

Installment Display

- Show monthly payment option
- Simple affordability check
- Payment plan calculator

No Embeddings in Phase 1!

- Focus on core functionality
- Prove product-market fit
- Gather user data for Phase 2

Success Metrics:

- Feed engagement rate > 3%
- User retention (day 7) > 30%
- Group buy completion rate > 50%

Phase 2: Enhanced Recommendations (Weeks 4-6)

Goal: Add semantic search and better personalization

Features to Implement:

Text Embeddings

- Set up Python embedding service
- Generate text embeddings for all products
- Update products when created/edited

Semantic Search

- Search using text embeddings
- Understand query meaning
- Better search relevance

Improved Product Similarity

- Use text embeddings
- Semantic similarity matching
- Better "Similar Products" section

User Interest Profiling

- Track user's category preferences
- Calculate interest scores
- Use in feed personalization

Enhanced Feed Algorithm

- Add personalization score
- Category matching
- Price range matching
- Engagement prediction

Group Buy Intelligence

- Match users to relevant groups using embeddings
- Smart group invitations (social graph)
- Urgency notifications
- "Complete the group" campaigns

Installment Recommendations

- Budget analysis per user
- Premium product suggestions
- Upgrade recommendations
- Bundle suggestions

Success Metrics:

- Search click-through rate > 8%
- "Similar products" click rate > 12%

- Feed engagement rate > 5%
- Group buy completion rate > 65%

Phase 3: Visual Search & Advanced Features (Weeks 7-9)

Goal: Add image-based features and optimize

Features to Implement:

Image Embeddings

- Generate image embeddings for products
- Store in database
- Index for fast search

Visual Search

- Upload image to find products
- Image similarity matching
- Visual style recommendations

Shop the Look

- Identify items in images
- Match to catalog
- Complete outfit suggestions

Combined Search

- Text + Image combined queries
- "Find red shoes like this image"
- Cross-modal search (CLIP)

Feed Optimization

- Diversity rules
- Quality boosting
- Trending algorithm refinement

Caching Strategy

- Redis caching for hot data
- Pre-computed similarities
- Query caching

A/B Testing Framework

- Test different scoring weights
- Test feed compositions
- Measure impact on conversions

Success Metrics:

- Visual search adoption > 15% of searches
- Feed engagement rate > 7%
- Conversion rate > 3%
- Average session time > 12 minutes

Phase 4: Optimization & ML (Weeks 10+)

Goal: Fine-tune and scale

Features to Implement:

Performance Optimization

- Query optimization
- Index tuning
- Embedding search speed improvements
- Caching refinements

Advanced Personalization

- ML-based engagement prediction
- Click-through rate modeling
- Purchase probability scoring

Recommendation Quality

- Feedback loops (did user buy?)
- Continuously improve scoring weights
- Seasonal adjustments

Scaling

- Load testing
- Database replication
- Caching layers
- CDN for images

Analytics Dashboard

- Recommendation performance metrics
- A/B test results
- User behavior insights
- Business intelligence

Success Metrics:

- System response time < 500ms (p95)
- Handle 10,000+ concurrent users
- Recommendation click rate > 15%
- Overall conversion rate > 5%

? Performance & Scaling

Performance Targets

API Response Times (95th percentile):

```
Feed API: < 500ms
Search API: < 600ms
Visual Search: < 1.5s
Product Detail: < 200ms
Group Buy List: < 300ms
```

Database Query Times:

```
Simple queries: < 50ms
Vector similarity (with index): < 200ms
Complex joins: < 300ms
```

Caching Hit Rates:

```
User following list: > 90%
Trending posts: > 95%
Product embeddings: > 80%
Search query results: > 60%
```

Optimization Strategies

1. Database Optimization

```
Indexing Strategy:
• Primary keys (automatic)
• Foreign keys (user_id, product_id, etc.)
• Frequently filtered columns (category, price, created_at)
• Vector columns (pgvector ivfflat index)
• Composite indexes for common queries

Query Optimization:
• Use EXPLAIN ANALYZE to identify slow queries
• Avoid N+1 queries (use JOIN or batch fetch)
• Limit result sets appropriately
• Use covering indexes where possible

Connection Pooling:
• HikariCP (default in Spring Boot)
• Pool size: 10-20 connections per instance
• Connection timeout: 30 seconds
• Idle timeout: 10 minutes
```

2. Caching Strategy

```
Redis Cache Layers:

Layer 1: Hot Data (TTL: 5-15 minutes)
• Trending posts
• Active group buys
• Real-time counters

Layer 2: Warm Data (TTL: 1-6 hours)
• User following lists
• User interest profiles
• Popular products

Layer 3: Cold Data (TTL: 24 hours)
• Similar product lists (pre-computed)
• Static content
```

• Configuration data

Layer 4: Session Data (TTL: 7-30 days)
• User seen posts
• Shopping cart
• Search history

Cache Invalidation:
• Update cache when data changes
• Lazy invalidation (TTL expiry)
• Active invalidation (delete on update)

3. Embedding Service Optimization

Model Loading:
• Load models at startup (not per request)
• Keep in memory (RAM)
• Use GPU if available (5-10x faster)

Batch Processing:
• Process multiple embeddings together
• Amortize model overhead
• 2-3x throughput improvement

Request Queuing:
• Queue embedding requests
• Batch process every 100ms
• Balance latency vs throughput

Caching:
• Cache common query embeddings
• Cache product embeddings in Redis
• Reduce Python service calls

4. Feed Generation Optimization

Pre-computation:
• Background job: Calculate trending posts (every 5 min)
• Background job: Update user interests (daily)
• Background job: Pre-compute popular similarities

```
Parallel Processing:
• Fetch candidates in parallel (following + trending + local)
• Score posts in parallel (if list is large)
• Use CompletableFuture or reactive programming

Result Caching:
• Cache full feed for 1-2 minutes (same user, same request)
• Cache candidate lists for 5 minutes
• Invalidate on user action (post, like, follow)
```

5. Search Optimization

```
Index Strategy:
• pgvector ivfflat index on embeddings
• Full-text search index (if needed)
• Composite indexes on filters

Query Rewriting:
• Combine filters in single query
• Use covering indexes
• Avoid sequential scans

Result Caching:
• Cache popular search queries
• TTL: 10-30 minutes
• Personalized results: 2-5 minutes
```

Scaling Strategy

Horizontal Scaling (Add More Servers):

```
Application Layer (Spring Boot):
• Stateless design (session in Redis, not memory)
• Load balancer distributes requests
• Scale out to 5, 10, 20+ instances
• Auto-scaling based on CPU/memory

Embedding Service (Python):
• Run multiple instances
```

• Load balancer in front
• Each instance has model loaded
• Scale based on request rate

Database (PostgreSQL):
• Read replicas for heavy read load
• Write to master, read from replicas
• Connection pooling per instance
• Separate analytics queries to replica

Vertical Scaling (Bigger Servers):

When to Scale Up:
• Database: More RAM for cache, faster CPU for queries
• Redis: More RAM for bigger cache
• Embedding service: GPU for faster inference

Sweet Spot:
• Application: 4-8 CPU, 8-16 GB RAM
• Database: 8-16 CPU, 32-64 GB RAM
• Redis: 4 CPU, 16-32 GB RAM
• Embedding: 4-8 CPU, 16-32 GB RAM, GPU optional

Caching Layers:

Level 1: Application Cache (Spring @Cacheable)
• Short-lived data (seconds to minutes)
• User session data
• Request-scoped cache

Level 2: Redis Cache
• Medium-lived data (minutes to hours)
• Cross-instance sharing
• High performance

Level 3: CDN (Images, Static Assets)
• Long-lived data (hours to days)
• Globally distributed
• Reduce origin load

```
Level 4: Database Query Cache
• PostgreSQL query result cache
• Automatic by database
• Benefit from repeated queries
```

Database Partitioning (Future):

```
When Needed:
• >10 million products
• >100 million posts
• Query performance degrades

Partitioning Strategy:
• Range partition by date (posts)
• Hash partition by user_id (user data)
• List partition by category (products)

Sharding (Advanced):
• Separate database per region
• Shard by user_id hash
• Cross-shard queries expensive (avoid)
```

? Metrics & Monitoring

Key Performance Indicators (KPIs)

User Engagement:

```
Feed Metrics:
• Feed Engagement Rate: (Likes + Comments + Shares) / Views
  Target: >5%
• Scroll Depth: Average posts viewed per session
  Target: >30 posts
• Session Duration: Time spent in app
  Target: >10 minutes
• Return Rate: % users returning within 7 days
  Target: >40%
```

Conversion Metrics:

```
Purchase Funnel:
• Feed View → Product Page: 8-12%
• Product Page → Add to Cart: 15-20%
• Cart → Purchase: 30-40%
• Overall Conversion: 3-5%

Group Buy:
• Group View → Join: 25-35%
• Group Completion Rate: 70-80%
• Average Group Fill Time: < 48 hours

Installment:
• Installment Shown → Selected: 40-50%
• Premium Product with Installment: 20-25% conversion boost
```

Recommendation Quality:

```
Click-Through Rates:
• "Similar Products": 12-18%
• Search Results: 8-12%
• Feed Recommendations: 5-8%
• Group Buy Suggestions: 20-30%

Relevance Metrics:
• Precision@10: >60% (of top 10 results are relevant)
• Mean Reciprocal Rank: >0.7 (relevant result in top 3)
• User Satisfaction: Survey score >4/5
```

System Performance:

```
Response Times (p95):
• Feed Load: < 500ms
• Search: < 600ms
• Product Page: < 200ms
• Visual Search: < 1.5s

Availability:
• Uptime: >99.9% (less than 45 min downtime/month)
```

```
• Error Rate: <0.1%
• Successful API calls: >99.5%

Resource Utilization:
• CPU: 60-80% (headroom for spikes)
• Memory: 70-85%
• Database connections: 50-70% of pool
```

Monitoring Setup

Application Monitoring:

```
Spring Boot Actuator:
• /actuator/health: System health
• /actuator/metrics: Performance metrics
• /actuator/prometheus: Prometheus export

Custom Metrics:
• Feed generation time
• Embedding generation time
• Similarity search time
• Cache hit rates
• Recommendation click-through rates
```

Database Monitoring:

```
PostgreSQL Metrics:
• Query execution time
• Index usage
• Table sizes
• Connection pool status
• Slow query log

pgvector Specific:
• Vector search time
• Index efficiency
• Embedding storage size
```

Cache Monitoring:

```
Redis Metrics:
• Hit rate
• Memory usage
• Eviction rate
• Command execution time
• Key expiration rate
```

Business Metrics Dashboard:

```
Real-Time:
• Active users now
• Feeds served (per minute)
• Searches executed
• Group buys active
• Purchases completed

Daily:
• New users
• Daily active users (DAU)
• Group buy completion rate
• Average order value
• Conversion rate

Weekly/Monthly:
• User retention (cohort analysis)
• Revenue trends
• Top categories
• Best performing recommendations
• A/B test results
```

? Success Criteria

Phase 1 (MVP) Success:

- □ Feed loads in < 1 second
- □ Users scroll through 20+ posts per session
- □ Group buy completion rate > 50%
- □ Basic recommendations work (not perfect)

- □ System handles 1,000 concurrent users

Phase 2 (Enhanced) Success:

- □ Search relevance improved (user feedback positive)
- □ "Similar products" click rate > 12%
- □ Feed engagement rate > 5%
- □ Group buy completion rate > 65%
- □ Installment adoption > 30% for expensive items

Phase 3 (Advanced) Success:

- □ Visual search works well (user satisfaction > 4/5)
- □ Feed engagement rate > 7%
- □ Overall conversion rate > 3%
- □ System handles 10,000+ concurrent users
- □ Recommendation quality high (precision > 60%)

Long-Term Success:

- □ Platform is primary discovery channel (not search)
- □ Users return 3+ times per week
- □ Average session > 15 minutes
- □ Conversion rate > 5%
- □ User satisfaction > 4.5/5
- □ System scales to 1M+ users

? Final Notes

Critical Principles

1. Start Simple, Scale Smart:

- Phase 1: No embeddings, basic scoring
- Phase 2: Add text embeddings
- Phase 3: Add image embeddings
- Don't over-engineer from day 1

2. User Data is Gold:

- Every interaction teaches the system
- Track everything (respecting privacy)
- Use data to improve recommendations
- Feedback loops are essential

3. Social Proof Wins:

- "7 people in group" > "Save $150"
- "Your friend bought this" > "Trending"
- Community drives engagement
- Leverage network effects

4. Affordability = Access:

- Group buying makes expensive affordable
- Installments remove price barriers
- Together = 2-3x conversion boost
- Key differentiator for Nexgate

5. Feed > Search:

- Most users browse, not search
- Discovery drives impulse buys
- Feed is primary experience
- Search is support feature

6. Diversity Matters:

- Avoid echo chambers
- Mix categories, prices, sellers
- Enable serendipitous discovery
- Balance personalization with exploration

7. Performance is Feature:

- Fast feed = more scrolling
- Slow search = user leaves
- Every 100ms matters
- Optimize relentlessly

When to Use What

Embeddings:

- Product similarity
- Semantic search

- Visual search
- Style matching

Scoring System:

- Main feed
- Personalization
- Trending
- Urgency ranking

Social Graph:

- Who to follow
- Group invitations
- Friend activity
- Network effects

Collaborative Filtering:

- "Bought together"
- "Also viewed"
- Purchase patterns
- Cross-sell

All Together:

- Best results come from hybrid approach
- Combine multiple signals
- Weight based on context
- Continuous improvement

? Conclusion

This architecture document provides a complete blueprint for Nexgate's recommendation system. It combines:

- **Social commerce** (TikTok Shop style feed)
- **Group buying** (collective purchasing power)
- **Installment payments** (affordability)
- **Smart recommendations** (embeddings + scoring)
- **Community engagement** (social graph)

The system is designed to:

1. **Start simple** (MVP with basic features)
2. **Scale smart** (add complexity as needed)
3. **Learn continuously** (from user behavior)
4. **Prioritize performance** (fast = engaging)
5. **Drive conversions** (discovery → purchase)

Remember: The best recommendation system is one that:

- Serves users first (not algorithms)
- Balances personalization with discovery
- Performs fast and reliably
- Improves over time with data
- Drives business goals

Good luck building Nexgate! □□

Document Version: 1.0
Last Updated: November 2025
Next Review: After Phase 1 completion

Understanding Recommendation Systems - From Zero to Hero ?

? What You'll Learn

This guide explains recommendation systems from first principles, with real-world examples, formulas, and the math behind them. **No code, just concepts!**

? Chapter 1: What Are Recommendation Systems?

The Simple Definition

A recommendation system is a tool that **predicts what you might like** based on:

- What you've done before
- What others like you have done
- Properties of the items themselves

Real-World Analogy

Imagine a smart bookstore clerk:

- Remembers every book you bought
- Knows what other customers bought
- Understands book genres and themes
- Suggests books you'll probably enjoy

That's essentially what a recommendation system does!

?? Chapter 2: The Three Main Types

Type 1: Content-Based Filtering

Concept: Recommend items similar to what you liked before.

How it works:

1. Analyze features of items you liked
2. Find other items with similar features
3. Recommend those items

Example:

```
You liked:
- "Harry Potter" (Fantasy, Magic, Young Adult, Adventure)
- "Lord of the Rings" (Fantasy, Magic, Epic, Adventure)

System recommends:
- "The Hobbit" (Fantasy, Magic, Adventure) □ Very similar!
- "Chronicles of Narnia" (Fantasy, Magic, Young Adult) □ Good match!
```

The Math Behind It:

Each item is represented as a **feature vector**:

```
Harry Potter = [Fantasy: 1, Magic: 1, Young Adult: 1, Adventure: 1, Romance: 0]
Lord of the Rings = [Fantasy: 1, Magic: 1, Young Adult: 0, Adventure: 1, Romance: 0]
The Hobbit = [Fantasy: 1, Magic: 1, Young Adult: 0, Adventure: 1, Romance: 0]
```

Similarity Calculation (Cosine Similarity):

```
Similarity = (A · B) / (||A|| × ||B||)

Where:
A · B = Dot product (multiply matching features)
||A|| = Magnitude of vector A
||B|| = Magnitude of vector B

Result: Number between 0 (totally different) and 1 (identical)
```

Pros:

- ✅ Doesn't need other users' data
- ✅ Can recommend new items immediately
- ✅ Easy to explain why something was recommended

Cons:

- ❌ Limited to features you can describe
- ❌ Can't discover new interests
- ❌ Gets stuck in a "filter bubble"

Type 2: Collaborative Filtering

Concept: "People like you also liked..."

How it works:

1. Find users similar to you
2. See what they liked
3. Recommend those items to you

Example:

```
You (Alice):
- Liked: iPhone, MacBook, AirPods
- Rating: 5 stars, 5 stars, 4 stars

Similar User (Bob):
- Liked: iPhone, MacBook, AirPods, Apple Watch
- Rating: 5 stars, 5 stars, 5 stars, 5 stars

Recommendation for Alice:
→ Apple Watch (because Bob, who has similar taste, loves it!)
```

Two Approaches:

A. User-Based Collaborative Filtering

Formula for User Similarity (Pearson Correlation):

```
similarity(user_a, user_b) =
  Σ(rating_a - avg_a)(rating_b - avg_b)
```

```
  / √[Σ(rating_a - avg_a)²] × √[Σ(rating_b - avg_b)²]

Result: Number between -1 (opposite taste) and 1 (identical taste)
```

Example Calculation:

```
Alice's ratings: [5, 4, 3, ?, 2]
Bob's ratings:   [5, 5, 3, 4, 2]
Carol's ratings: [1, 2, 3, 4, 5]

Similarity(Alice, Bob) = 0.95 (very similar!)
Similarity(Alice, Carol) = -0.8 (opposite taste!)

Predict Alice's rating for item 4:
→ Use Bob's rating (4) because Bob is most similar
```

B. Item-Based Collaborative Filtering

Instead of finding similar users, find similar items!

Example:

```
People who bought iPhone also bought:
- iPhone Case (90% of buyers)
- Screen Protector (85% of buyers)
- AirPods (60% of buyers)
- Apple Watch (40% of buyers)

You bought iPhone → Recommend iPhone Case (highest correlation!)
```

Formula for Item Similarity:

```
similarity(item_i, item_j) =
  Number of users who liked both items
  / √(Users who liked item_i × Users who liked item_j)

This is called "Jaccard Similarity"
```

Pros:

- □ Discovers new interests
- □ Doesn't need item features

- Works well with lots of user data

Cons:

- Cold start problem (new users/items)
- Sparsity (most users rate few items)
- Popularity bias (recommends popular items)

Type 3: Hybrid Systems

Concept: Combine multiple approaches for better results!

Common Combinations:

A. Weighted Hybrid

```
Final Score =
  (0.5 × Content-Based Score) +
  (0.5 × Collaborative Score)

Example:
Product X:
- Content similarity to your likes: 0.8
- People like you also bought it: 0.6
- Final score: (0.5 × 0.8) + (0.5 × 0.6) = 0.7
```

B. Switching Hybrid

```
IF user is new (no history):
    → Use Content-Based (based on item features)
ELSE IF user has lots of history:
    → Use Collaborative (based on similar users)
```

C. Cascade Hybrid

```
Step 1: Content-Based filters 1000 → 100 items
Step 2: Collaborative ranks those 100 → Top 10
Step 3: Show top 10 to user
```

? Chapter 3: The Math Explained Simply

Similarity Measures

These are ways to measure "how alike" two things are.

1. **Cosine Similarity** (Most Common)

Imagine two arrows in space:

```
Arrow A points → (3, 4)
Arrow B points → (4, 3)

Angle between them = small → Similar!
Angle = 90° → Completely different
```

Formula:

```
cosine_similarity = cos(θ) = (A · B) / (|A| × |B|)

Where:
A · B = (3×4) + (4×3) = 12 + 12 = 24
|A| = √(3² + 4²) = √25 = 5
|B| = √(4² + 3²) = √25 = 5

Result = 24 / (5 × 5) = 24/25 = 0.96 (very similar!)
```

Range: 0 (perpendicular) to 1 (identical direction)

---

2. Euclidean Distance

Think of it as "crow flies" distance:

```
Point A = (1, 2)
Point B = (4, 6)

Distance = √[(4-1)² + (6-2)²]
         = √[9 + 16]
```

```
          = √25
          = 5

Closer distance = More similar
```

Problem: Doesn't work well with different scales!

```
Price: $10 vs $15 (difference = 5)
Rating: 3 vs 4 stars (difference = 1)

The price difference dominates unfairly!
```

Solution: Normalize first (scale everything 0-1)

---

3. Pearson Correlation

Measures if two things move together:

```
Alice rates: [5, 4, 3, 2, 1]
Bob rates:   [5, 4, 3, 2, 1]
→ Perfect correlation = 1.0 (they always agree!)

Alice rates: [5, 4, 3, 2, 1]
Carol rates: [1, 2, 3, 4, 5]
→ Perfect negative correlation = -1.0 (opposite taste!)
```

Formula:

```
r = Σ[(x - x̄)(y - ȳ)] / √[Σ(x - x̄)² × Σ(y - ȳ)²]

Where:
x̄ = average of x
ȳ = average of y
```

Range: -1 (opposite) to +1 (identical)

Matrix Factorization (Advanced!)

The Idea: Break down the user-item matrix into hidden patterns.

Real-World Example:

```
Movie ratings matrix:
           Action  Comedy  Drama
Alice        5       2       4
Bob          5       1       3
Carol        1       5       2

Hidden factors might be:
Factor 1: "Likes serious content"
Factor 2: "Likes funny content"

Alice = [High Factor 1, Low Factor 2] → Likes Action/Drama
Carol = [Low Factor 1, High Factor 2] → Likes Comedy
```

This is what Netflix does!

They discovered hidden factors like:

- "Likes quirky independent films"
- "Prefers big-budget blockbusters"
- "Enjoys thought-provoking documentaries"

Formula (Simplified):

```
Rating = User_Vector · Item_Vector

Alice's vector = [0.9, 0.2] (serious, not funny)
Action movie vector = [0.8, 0.1] (serious, not funny)

Predicted rating = (0.9 × 0.8) + (0.2 × 0.1)
                 = 0.72 + 0.02
                 = 0.74 (normalized)
                 ≈ 4.5 stars
```

? Chapter 4: Real-World Examples Explained

Example 1: Netflix

What they use: Hybrid system with heavy collaborative filtering + content-based

How it works:

```
Step 1: Collaborative Filtering
- Find users who rated movies similarly to you
- Weight: 60%

Step 2: Content-Based
- Analyze genres, actors, directors you like
- Weight: 25%

Step 3: Trending/Popular
- What's hot right now
- Weight: 15%

Final Score = (0.6 × Collaborative) + (0.25 × Content) + (0.15 × Trending)
```

Why it works:

- Cold start: New users get recommendations based on genres they select
- Warm users: Get personalized recommendations from similar users
- Diversity: Trending ensures you see new popular content

Example 2: Amazon

What they use: Primarily item-based collaborative filtering

The Famous Algorithm: "Customers who bought X also bought Y"

How it's calculated:

```
iPhone → Case: 85% co-purchase rate
iPhone → Screen Protector: 78% co-purchase rate
iPhone → Charger: 65% co-purchase rate
iPhone → Laptop: 5% co-purchase rate

Formula:
Co-purchase rate =
```

```
  (Times X and Y bought together) / (Times X was bought)

Example:
iPhone bought: 1000 times
iPhone + Case bought together: 850 times
Co-purchase rate = 850/1000 = 85%
```

Why it works:

- Very accurate for complementary products
- Doesn't need user profiles
- Works immediately for new users
- Based on actual purchase behavior (not just browsing)

Example 3: Spotify

What they use: Hybrid with collaborative + audio analysis + social

Three Recommendation Types:

A. Collaborative Filtering

```
Your playlists: [Pop, Rock, Indie]
Similar user's playlists: [Pop, Rock, Indie, Alternative]
→ Recommend Alternative music
```

B. Audio Analysis (Content-Based)

```
Song features analyzed:
- Tempo: 120 BPM
- Key: C Major
- Energy: High
- Valence (happiness): Medium
- Acousticness: Low

Find songs with similar audio features!
```

C. Social

```
Your friends listen to:
- Artist X: 80% of friends
```

```
- Artist Y: 60% of friends
→ Recommend Artist X
```

Weekly Discover Playlist:

```
= 30% Collaborative (users like you)
+ 30% Audio similarity (songs like yours)
+ 20% New releases in your genres
+ 20% Social (what friends listen to)
```

Example 4: TikTok (The King!)

What they use: Engagement prediction model (ML-based)

How it works:

```
For each video, predict:
- Will user watch to the end? (Completion rate)
- Will user like it?
- Will user comment?
- Will user share?
- Will user follow creator?

Score =
  (10 × Completion prediction) +
  (5 × Like prediction) +
  (8 × Comment prediction) +
  (12 × Share prediction) +
  (15 × Follow prediction)

Show videos with highest predicted score!
```

Features considered:

```
Video features:
- Category/hashtags
- Music used
- Duration
- Captions
```

```
User features:
- Past liked categories
- Watch time patterns
- Engagement history
- Language preference

Interaction features:
- Time of day
- Device type
- Network speed
```

Why it's so addictive:

- Optimizes for ENGAGEMENT, not just relevance
- Learns quickly (every swipe teaches the algorithm)
- Heavy personalization (your feed is unique)

? Chapter 5: Common Formulas Reference

1. Weighted Score (Most Common in Practice!)

```
Final Score = Σ(Weight_i × Score_i)

Example (E-commerce):
Product Score =
  (0.35 × Social_Score) +
  (0.25 × Engagement_Score) +
  (0.20 × Personalization_Score) +
  (0.15 × Recency_Score) +
  (0.05 × Quality_Score)

Each component score is 0-100, normalized
```

---

2. Recency Decay

```
Recency Score = Base_Score × e^(-λ × time)

Where:
λ (lambda) = decay rate (how fast score decreases)
time = hours/days since creation
e = 2.71828 (natural logarithm base)

Example:
Base score = 100
λ = 0.1 (slow decay)
After 24 hours: 100 × e^(-0.1 × 24) = 100 × 0.091 = 9.1

Interpretation: Old content gets much lower score
```

Simpler Alternative (Step Function):

```
IF age < 1 hour: Score = 100
ELSE IF age < 6 hours: Score = 80
ELSE IF age < 24 hours: Score = 50
ELSE IF age < 7 days: Score = 20
ELSE: Score = 5
```

---

3. Engagement Rate

```
Engagement Rate =
  (Likes + Comments + Shares) / Views

Example:
Video: 10,000 views, 500 likes, 50 comments, 30 shares
Engagement = (500 + 50 + 30) / 10,000 = 0.058 = 5.8%

Good engagement: > 5%
Viral content: > 15%
```

---

4. Click-Through Rate (CTR)

```
CTR = Clicks / Impressions

Example:
Product shown 1000 times
Clicked 50 times
CTR = 50/1000 = 0.05 = 5%

Use CTR to rank items:
Higher CTR = Better recommendation
```

---

5. Conversion Rate

```
Conversion Rate = Purchases / Clicks

Example:
Product clicked 100 times
Purchased 10 times
Conversion = 10/100 = 10%

Ultimate metric: Did recommendation lead to action?
```

? Chapter 6: Choosing the Right System

Decision Framework

Use Content-Based When:

- □ Items have rich descriptions
- □ Few users (cold start)
- □ Need to explain recommendations
- □ Items change frequently

Examples: News articles, blog posts, jobs

Use Collaborative Filtering When:

- □ Lots of user interaction data
- □ Items don't have clear features
- □ Want to discover unexpected items
- □ Users have diverse tastes

Examples: Movies, music, products

Use Hybrid When:

- □ You have both item features AND user data
- □ Want best of both worlds
- □ Can handle complexity
- □ Need to solve cold start

Examples: E-commerce (like Amazon), streaming (like Netflix)

Use Social/Graph-Based When:

- □ Platform has social connections
- □ Social proof matters
- □ Viral/trending important
- □ Community-driven

Examples: Social commerce, TikTok, Instagram Shopping

? Chapter 7: Learning Resources

Books (No Code!)

1. "Recommendation Systems: The Textbook" by Charu Aggarwal

- Comprehensive coverage
- Mathematical explanations
- Theory + Practice
- □□ Best for deep understanding

2. "Practical Recommender Systems" by Kim Falk

- Real-world examples
- Less math, more intuition
- Case studies
- □□ Best for beginners

3. "Programming Collective Intelligence" by Toby Segaran

- Intuitive explanations
- Simple examples
- Practical algorithms
- □□ Best for implementation ideas

Online Courses

1. Coursera: "Recommender Systems" by University of Minnesota

- Free to audit
- Video lectures
- Covers all types
- □□ Best structured course

2. YouTube: "StatQuest with Josh Starmer"

- Amazing explanations
- Visual animations
- Covers collaborative filtering, PCA, SVD
- □□ Best for visual learners

3. Google's Machine Learning Crash Course

- Section on recommendations
- Interactive examples
- Free and well-designed
- □□ Best for ML context

Papers (Foundational)

1. "Amazon.com Recommendations: Item-to-Item Collaborative Filtering"

- How Amazon does it
- Industry standard
- Very readable
- □□ Must-read!

2. "The Netflix Prize" papers

- Competition that advanced the field
- Matrix factorization explained
- Real-world constraints
- □□ Historical importance

3. "BPR: Bayesian Personalized Ranking"

- Modern ranking approach
- Implicit feedback (views, not ratings)
- Used by many companies
- □□ Advanced but important

Websites

1. Towards Data Science (Medium)

- Blog posts explaining concepts
- Real-world case studies
- Beginner to advanced
- □□ Free with email

2. Papers With Code

- Research papers + implementations
- See state-of-the-art methods
- Compare approaches
- □□ Great for staying current

3. Google Research Blog

- How Google does recommendations
- YouTube algorithm explanations
- Cutting-edge research
- □□ Straight from the source

? Chapter 8: Working Example (No Code!)

Scenario: Recommend Products for Alice

Alice's History:

```
Bought: iPhone ($999), AirPods ($199), MacBook ($1299)
Viewed: iPad, Apple Watch, iPhone Case
Searched: "wireless earbuds", "laptop accessories"
```

```
Budget range: $150-1500
```

Available Products:

```
1. Apple Watch ($399)
2. iPad ($329)
3. Samsung Phone ($899)
4. Laptop Stand ($49)
5. Wireless Keyboard ($129)
6. iPhone Case ($29)
7. AirPods Pro ($249)
```

Method 1: Content-Based Scoring

Step 1: Define Item Features

```
Apple Watch:
- Brand: Apple (1)
- Category: Electronics (1)
- Price Range: Mid ($399 in her range □)
- Compatibility: iPhone (1)

Samsung Phone:
- Brand: Samsung (0 - she buys Apple)
- Category: Electronics (1)
- Price Range: High ($899 □)
- Compatibility: Android (0)
```

Step 2: Calculate Similarity

```
Apple Watch vs Alice's preferences:
Brand match: 100% (all Apple)
Category match: 100% (all electronics)
Price match: 80% (slightly lower than average)
Compatibility: 100% (has iPhone)

Similarity Score = (100 + 100 + 80 + 100) / 4 = 95%

Samsung Phone:
```

```
Brand match: 0%
Category match: 100%
Price match: 90%
Compatibility: 0%

Similarity Score = (0 + 100 + 90 + 0) / 4 = 47.5%
```

Ranking:

1. Apple Watch (95%)
2. AirPods Pro (92%)
3. iPad (88%)
4. Samsung Phone (47.5%)

Method 2: Collaborative Filtering

Step 1: Find Similar Users

```
Alice bought: [iPhone, AirPods, MacBook]

Bob bought: [iPhone, AirPods, MacBook, Apple Watch]
Similarity: 3/3 common items = 100% overlap!

Carol bought: [iPhone, Samsung Phone, Android Tablet]
Similarity: 1/3 common items = 33% overlap

Dan bought: [Dell Laptop, Android Phone]
Similarity: 0/3 common items = 0% overlap
```

Step 2: Recommend What Similar Users Bought

```
Bob (100% similar) also bought:
→ Apple Watch  Strong recommendation!

Carol (33% similar) also bought:
→ Samsung Phone  Weak recommendation

Dan (0% similar):
→ Ignore his purchases
```

Ranking:

1. Apple Watch (Bob recommends, 100% similarity)
2. iPad (viewed but not bought - weaker signal)

Method 3: Hybrid Approach (Best!)

Combine Both Methods:

```
Apple Watch:
- Content similarity: 95%
- Collaborative: 100% (Bob bought it)
- Final: (0.5 × 95) + (0.5 × 100) = 97.5 

iPad:
- Content similarity: 88%
- Collaborative: 50% (Alice viewed, no strong signal)
- Final: (0.5 × 88) + (0.5 × 50) = 69

Samsung Phone:
- Content similarity: 47.5%
- Collaborative: 33% (Carol bought, low similarity)
- Final: (0.5 × 47.5) + (0.5 × 33) = 40.25
```

Final Ranking:

1. **Apple Watch (97.5)** ← Recommend this!
2. AirPods Pro (92)
3. iPad (69)
4. Wireless Keyboard (55)
5. Samsung Phone (40.25)

Adding More Factors

Recency Boost:

```
Apple Watch: Released 2 months ago → +5 points
iPad: Released 6 months ago → +3 points
Samsung Phone: Released 2 years ago → +0 points
```

```
Updated scores:
1. Apple Watch (102.5)
2. AirPods Pro (92)
3. iPad (72)
```

Social Proof:

```
Apple Watch: 4.8 stars, 10,000 reviews → +8 points
iPad: 4.7 stars, 8,000 reviews → +7 points
Samsung Phone: 4.5 stars, 5,000 reviews → +5 points

Final scores:
1. Apple Watch (110.5) □□□
2. AirPods Pro (92)
3. iPad (79)
```

? Key Takeaways

The Golden Rules

1. Simple Often Wins

- Don't need complex ML for good recommendations
- Weighted scoring can be 80% as effective
- Start simple, add complexity only if needed

2. Context Matters

- Social platform → Use social signals heavily
- E-commerce → Use purchase history + collaborative
- Content platform → Use engagement metrics

3. Multiple Signals Are Better

- Combine content + collaborative + social + popularity
- No single method is perfect
- Hybrid approaches work best in practice

4. Measure What Matters

- Track engagement, conversion, retention

- A/B test different approaches
- Optimize for business goals, not just accuracy

5. Cold Start Is Hard

- New users: Use popular items + content-based
- New items: Use content-based + social proof
- Have fallback strategies

? Summary Cheat Sheet

```
┌─────────────────────────────────────────┐
│         Recommendation Method Picker    │
└─────────────────────────────────────────┘


Have item features? → Content-Based
Have user behavior data? → Collaborative
Have both? → Hybrid □

Social platform? → Add social signals
Need explainability? → Content-Based
Want serendipity? → Collaborative

Cold start problem? → Content-Based first,
                      then Collaborative

Popular approach: Weighted Hybrid
= (Weight × Content) + (Weight × Collab) +
  (Weight × Social) + (Weight × Recency)
```

You now understand recommendation systems from first principles! □□

Next steps:

1. Re-read sections that were unclear
2. Draw diagrams to visualize concepts
3. Work through more examples on paper
4. Apply to your Nexgate platform design

Remember: The best recommendation system is one that works for YOUR specific use case and users! □□