SNAP Principal Engineer Interview
Large Scale Ads System Design

Complete Preparation Guide - December 1, 2024

December 2, 2025

Contents

1 Interview Overview

1.1 What to Expect

Interview Type: Principal Engineer Technical Interview (60 minutes)
Interviewer: Principal Engineer Beijie Xu
Format:

- 20 minutes: Behavioral/Leadership (”Tell me about a time when...”)
- 35 minutes: Large Scale Ads Product System Design
- 5 minutes: Q&A (your questions)

Focus Areas:

- Leading large scale initiatives
- Evaluating technical tradeoffs
- Creating strategic roadmaps
- System design exercise (ads platform)
- Deep technical details and latency requirements

1.2 Success Criteria

1. **Leadership**: Demonstrate experience leading large scale projects with cross-functional impact
2. **Technical Depth**: Go deep into ads system architecture, tradeoffs, and performance optimization
3. **Strategic Thinking**: Show ability to create roadmaps and evaluate long-term technical decisions
4. **Collaboration**: Think through solutions collaboratively, ask clarifying questions
5. **Avoid Brute Force**: Think through tradeoffs before jumping to solutions

Pro Tips

Key Insight: This is a Principal Engineer role - focus on:

- Breadth of technical leadership
- Cross-team coordination
- Long-term strategic impact
- Mentoring and technical influence

Part I
Behavioral Interview Prep (20 minutes)

2 STAR Method Framework

2.1 Structure for Every Story

Situation (2-3 sentences):

- Set context: company, team size, business problem
- Quantify scale: users, requests/sec, data volume
- State the problem or opportunity

Task (1-2 sentences):

- Your specific role and responsibility
- What you were asked to accomplish
- Why it mattered to the business

Action (Most important - 60% of time):

- Technical decisions you made (with alternatives considered)
- How you led the team (delegation, unblocking, mentoring)
- Tradeoffs you evaluated
- Cross-functional coordination
- Risk mitigation strategies

Result (2-3 sentences):

- Quantifiable impact: latency improvement, cost reduction, revenue increase
- Team/organizational impact: processes created, standards established
- Long-term strategic value

2.2 Story Bank - Top 5 Stories to Prepare

Critical Talking Points

Prepare 5 stories covering different leadership dimensions:

1. **Large Scale System Design**: Led architecture for high-throughput system
2. **Cross-Team Initiative**: Coordinated multiple teams to achieve shared goal
3. **Technical Tradeoff**: Made difficult technical decision under constraints
4. **Mentoring/Influence**: Elevated team technical capabilities
5. **Strategic Roadmap**: Created multi-quarter technical strategy

3 Common Leadership Questions

3.1 Tell me about leading a large scale initiative

Example Stories

Example Structure:
Situation: "At Roblox, we needed to migrate 500M+ records from a monolithic search index to a multi-tenant architecture serving 10K+ organizations. The existing system had 2s p99 latency and frequent OOM crashes."
Task: "As tech lead, I was responsible for designing the new architecture, coordinating 3 teams (backend, infra, data), and executing a zero-downtime migration over 6 months."
Action:

- **Technical Design:** Evaluated 3 approaches (index-per-tenant, shared index, hybrid). Selected hybrid model with tiered backends (Enterprise dedicated clusters, Premium dedicated indices, Shared multi-tenant index) based on cost vs performance tradeoffs.
- **Tradeoff Analysis:** Index-per-tenant would cost $500K/year in infrastructure but guarantee isolation. Shared index would cost $50K but risk noisy neighbor. Hybrid gave 90% cost savings while isolating top 5% of customers.
- **Risk Mitigation:** Built dual-write system to write to both old and new indices for 30 days. Added feature flag for instant rollback. Created shadow traffic testing to compare results before cutover.
- **Team Coordination:** Established weekly sync with backend, infra, and SRE teams. Created migration playbook with rollback procedures. Trained on-call team on new architecture.
- **Execution:** Migrated in waves - internal orgs first, then 10% customers, then full rollout. Monitored latency, error rates, and cost metrics at each stage.

Result:

- Reduced p99 latency from 2s to 300ms (85% improvement)
- Zero downtime during migration
- Infrastructure cost reduced by $400K/year
- Architecture enabled new features: cross-workspace search, tenant-specific relevance tuning
- Created migration playbook used by 4 other teams

3.2 Tell me about evaluating technical tradeoffs

Example Stories

Example: Real-time vs Batch Processing for Ad Attribution
Situation: Need to attribute ad clicks to conversions for campaign ROI calculation. 100M clicks/day, 5M conversions/day.
Task: Decide between real-time stream processing vs batch processing.
Tradeoff Analysis:
Option 1: Real-time (Kafka + Flink)

- **Pros:** Fresh data (¡1 min), enables real-time bidding optimization
- **Cons:** Complex infrastructure, higher cost ($100K/year), difficult debugging
- **Latency:** ¡1 minute end-to-end
- **Cost:** High operational complexity

Option 2: Micro-batch (5 min intervals)

- **Pros:** Simpler code, easier debugging, 50% lower cost
- **Cons:** 5 min data delay
- **Latency:** 5-10 minutes
- **Cost:** Medium complexity

Option 3: Hourly batch

- **Pros:** Simplest, lowest cost ($20K/year), reliable
- **Cons:** 1-2 hour delay, no real-time optimization
- **Latency:** 1-2 hours
- **Cost:** Low complexity

Decision: Selected micro-batch (5 min) because:

- Advertisers check dashboards every 15-30 minutes (not second-by-second)
- 5 min delay acceptable for campaign optimization decisions
- 50% cost reduction vs real-time
- Simpler debugging = faster iteration on attribution models
- Could upgrade to real-time later if business justified it

Result: Delivered attribution system in 3 months instead of 6, saved $50K/year, achieved 99.9% accuracy.

3.3 Tell me about creating a strategic roadmap

Example Stories

Example: Multi-Quarter Ads Platform Modernization
Situation: Legacy ads platform serving 10K advertisers with monolithic architecture, single database, no A/B testing capability. Scaling limited to 20K req/sec.
Task: Create 12-month roadmap to modernize platform to support 10x growth (100K advertisers, 200K req/sec).
Strategic Roadmap (4 Phases):
Q1: Foundation (Observability & Data)

- Add distributed tracing (OpenTelemetry)
- Build data warehouse for ads analytics
- Implement feature flags for gradual rollouts
- **Why First:** Can't improve what you can't measure. Need visibility before making changes.

Q2: Decouple Services

- Extract Ad Serving into separate service
- Migrate to Redis for ad metadata caching
- Implement API gateway for rate limiting
- **Why Second:** Most latency-sensitive path. Quick wins on performance.

Q3: Scale Database

- Shard database by advertiser_id
- Implement read replicas for reporting queries
- Move campaign analytics to data warehouse
- **Why Third:** Database is bottleneck. Need sharding before hitting 50K req/sec.

Q4: ML & Optimization

- Build ML-based bid optimization
- Implement real-time budget pacing
- Add A/B testing framework for ad creatives
- **Why Last:** Needs stable infrastructure first. Revenue optimization after scale problems solved.

Principles:

- Each phase delivers standalone value
- No "big bang" rewrites - incremental migration
- Each phase derisk next phase
- Balance technical debt cleanup with new features

Result: Executed 90% of roadmap on time. Scaled to 150K req/sec (7.5x improvement). Enabled 3 new revenue features. Reduced incident rate by 60%.

Part II
Ads System Design Deep Dive (35 minutes)

4 Leveraging Search Expertise for Ads Systems

Critical Talking Points

Key Insight: Ads Serving = Search Problem
Ad serving is fundamentally a retrieval and ranking problem. Your search expertise directly applies!
Mental Framework:

- **Query** = User context (demographics, interests, behavior)
- **Documents** = Ad campaigns (millions of candidates)
- **Retrieval** = Find top-K relevant ads from candidate pool
- **Ranking** = Score ads by relevance × bid × quality
- **Result** = Winning ad served to user

4.1 Search Concepts Applied to Ads

4.1.1 1. Multi-Stage Retrieval (Your Expertise!)

Search Problem: Find relevant documents from billions of items in ¡10ms.
Search Solution: Multi-stage retrieval

```
Stage 1: Broad retrieval (10M docs -> 1000 candidates) - 5ms
Stage 2: Reranking (1000 -> 100) - 10ms
Stage 3: Final ranking (100 -> 10) - 20ms
```

Ads Application: Find relevant ads from millions of campaigns

```
# Stage 1: Candidate Retrieval (Inverted Index)
# "Male, 18-24, US, interested in sports"
def retrieve_candidates(user_profile):
    # Use inverted index (like search!)
    candidates = []

    # Geo filter (must match)
    candidates = index.filter(country=user_profile.country)

    # Demographic filter
    candidates = candidates.filter(
        age_range__contains=user_profile.age
    )

    # Interest-based retrieval (BM25 scoring)
    for interest in user_profile.interests:
        scored_ads = index.search(
            field="targeting_keywords",
            query=interest,
            limit=1000
        )
        candidates.extend(scored_ads)

    return candidates[:1000]  # Top 1000 candidates

# Stage 2: Fast Ranking (Logistic Regression)
def fast_rank(candidates, user):
    scored = []
    for ad in candidates:
        # Simple features (like search ranking!)
        score = (
            ad.bid *
            ad.historical_ctr *
            user_ad_affinity(user, ad.category) *
            ad.quality_score
        )
        scored.append((ad, score))

    scored.sort(key=lambda x: x[1], reverse=True)
    return scored[:20]  # Top 20 for next stage

# Stage 3: Accurate Ranking (Deep Learning)
def accurate_rank(top_candidates, user):
    # Like search reranking with BERT!
    for ad in top_candidates:
        features = extract_deep_features(user, ad)
        pCTR = deep_model.predict(features)
        final_score = ad.bid * pCTR * ad.quality_score

    return sorted_by_score[:3]  # Top 3 for auction
```

How to Present This:

- ”In my search experience, we used multi-stage retrieval to handle billions of documents. The same principle applies to ads - Stage 1 uses inverted indices for broad filtering, Stage 2 uses fast scoring, Stage 3 uses ML for accuracy.”
- ”This is exactly like Elasticsearch's query-then-fetch or vector search with two-stage ranking.”

4.1.2 2. Hybrid Search = Hybrid Ad Targeting

Your Search Expertise: Hybrid search combining BM25 + vector embeddings

```
# Search: Combine keyword match + semantic similarity
def hybrid_search(query, documents):
    # BM25 for keyword matching
    bm25_results = bm25_index.search(query, top_k=100)

    # Vector search for semantic similarity
    query_embedding = encode(query)
    vector_results = vector_index.search(
        query_embedding,
        top_k=100
    )

    # Combine scores
    combined = reciprocal_rank_fusion(
        bm25_results,
        vector_results,
        weights=[0.7, 0.3]
    )

    return combined[:10]
```

Ads Application: Hybrid ad targeting

```
# Ads: Combine rule-based + semantic targeting
def hybrid_ad_targeting(user):
    # Rule-based targeting (like BM25)
    # Explicit matches: age, gender, location
    rule_based_ads = get_ads_matching_demographics(user)

    # Semantic targeting (like vector search)
    # Infer interests from behavior
    user_embedding = encode_user_behavior(
        user.recent_story_views,
        user.recent_lens_uses,
        user.recent_searches
    )

    # Find ads with similar semantic meaning
    semantic_ads = vector_index.search(
        user_embedding,
        top_k=100
    )

    # Combine (rule-based for precision, semantic for recall)
    combined = merge_results(
        rule_based_ads,  # High precision, narrow
        semantic_ads,    # High recall, broad
        weights=[0.6, 0.4]
    )

    return combined[:20]
```

How to Present This:

- ”At Roblox, we built hybrid search combining BM25 and vector embeddings. For ads, I'd use the same approach - rule-based targeting for precision (exact demographic matches) and semantic embeddings for recall (discover similar interests).”
- ”This addresses the cold-start problem - new campaigns without historical CTR data can leverage semantic similarity to user interests.”

4.1.3 3. Learning to Rank (LTR)

Your Search Expertise: Learning to Rank for search results
Ads Application: Predicting ad CTR and ranking

```
# Search LTR: Predict relevance score
class SearchRanker:
    def rank(self, query, documents):
        features = []
        for doc in documents:
            features.append([
                bm25_score(query, doc),
                query_doc_cosine_sim(query, doc),
                doc.pagerank,
                doc.freshness,
                doc.length
            ])

        # Predict relevance
        scores = model.predict(features)
        return sort_by_scores(documents, scores)

# Ads LTR: Predict CTR (click-through rate)
class AdRanker:
    def rank(self, user, ads):
        features = []
        for ad in ads:
            features.append([
                user_ad_affinity(user, ad),
                ad.historical_ctr,
                ad.recency,
                user_engagement_with_category(user, ad.category),
                time_of_day_factor(),
                # Add more features...
            ])

        # Predict CTR
        pCTR = model.predict(features)

        # Rank by eCPM (expected revenue)
        eCPM = ad.bid * pCTR * 1000
        return sort_by_eCPM(ads)
```

How to Present This:

- "In search, we used LTR models trained on click logs. For ads, it's the same concept - train on impression/click data to predict pCTR, then rank by bid × pCTR."
- "The features are similar: user-item affinity, historical engagement, temporal factors, contextual signals."

4.1.4 4. Vector Embeddings for Semantic Matching

Your Search Expertise: Dense retrieval with BERT/sentence transformers
Ads Application: User and ad embeddings for semantic targeting

```
# Search: Encode queries and documents
query_embedding = bert_model.encode(query)
doc_embeddings = bert_model.encode(documents)
similarities = cosine_similarity(query_embedding, doc_embeddings)

# Ads: Encode users and ads
class SemanticAdMatcher:
    def encode_user(self, user):
        """
        Create user embedding from behavior.
        """
        behaviors = [
            user.recent_story_views,  # Stories watched
            user.recent_searches,     # Search queries
            user.recent_lens_uses     # Lenses tried
        ]

        # Concatenate behavior text
        behavior_text = " ".join([
            story.title for story in user.recent_story_views
        ] + [
            search.query for search in user.recent_searches
        ])

        # Encode with sentence transformer
        user_embedding = model.encode(behavior_text)
        return user_embedding

    def encode_ad(self, ad):
        """
        Create ad embedding from campaign metadata.
        """
        ad_text = f"{ad.title} {ad.description} {ad.targeting_keywords}
            "
        ad_embedding = model.encode(ad_text)
        return ad_embedding

    def find_similar_ads(self, user):
        """
        Dense retrieval: find ads semantically similar to user
            interests.
        """
        user_embedding = self.encode_user(user)

        # Vector search in embedding space
        similar_ads = vector_index.search(
            user_embedding,
            top_k=100,
            metric="cosine"
        )

        return similar_ads
```

How to Present This:

- ”We used sentence transformers for semantic search at Roblox. For Snapchat ads, I'd encode user behavior (Stories watched, Lenses used) into embeddings and match against ad embeddings for semantic targeting.”
- ”This goes beyond keyword matching - a user interested in ’fitness’ content would also match ads about ’yoga’, ’running’, ’healthy eating’ through semantic similarity.”

4.1.5 5. Multi-Tenancy and Isolation

Your Search Expertise: Multi-tenant search with workspace isolation
Ads Application: Campaign isolation and budget management

```
# Search: Workspace isolation
def search(user, query):
    # MUST filter by workspace - prevent data leaks
    workspaces = get_user_workspaces(user.id)

    results = elasticsearch.search({
        "query": {
            "bool": {
                "must": [{"match": {"content": query}}],
                "filter": [
                    {"terms": {"workspace_id": workspaces}}
                ]
            }
        }
    })

    return results

# Ads: Advertiser isolation and budget enforcement
def serve_ad(user):
    # Get eligible campaigns (budget remaining)
    campaigns = get_campaigns_with_budget()

    # Each advertiser's campaigns are isolated
    # Budget tracked per advertiser (like workspace quota)

    eligible_ads = []
    for campaign in campaigns:
        # Check budget (like checking workspace quota)
        if has_budget_remaining(campaign.advertiser_id):
            eligible_ads.append(campaign)

    # Rank and serve
    ad = rank_and_select(eligible_ads, user)
    return ad
```

How to Present This:

- ”At Roblox, we built multi-tenant search with strict workspace isolation. Ads systems have similar requirements - each advertiser's budget must be tracked separately, campaigns must respect spend limits, just like workspace quotas.”
- ”The sharding strategy is similar - shard by advertiser_id (like workspace_id) to isolate data and avoid cross-advertiser queries.”

4.2 Key Talking Points: Search → Ads Translation

Critical Talking Points

When interviewer asks about ads system design, frame it as:

1. **"Ads serving is a retrieval problem"**
   - "In my search work, we retrieved top-K documents from billions. Ads is the same - retrieve top candidates from millions of campaigns using inverted indices, then rank by bid $\times$ relevance."
2. **"Multi-stage ranking from search applies directly"**
   - "We used 3-stage retrieval in search: broad retrieval (inverted index), fast ranking (simple features), accurate ranking (deep model). Same strategy for ads to meet 100ms latency SLA."
3. **"Hybrid search = Hybrid targeting"**
   - "I built hybrid search combining BM25 and vector embeddings. For ads, I'd combine rule-based targeting (demographics) with semantic targeting (user behavior embeddings) for better precision-recall balance."
4. **"Learning to Rank → CTR prediction"**
   - "LTR models in search predict relevance from click logs. For ads, train on impression/click data to predict pCTR, then rank by eCPM = bid $\times$ pCTR."
5. **"Cold start problem solutions"**
   - "In search, new documents use content-based signals before getting engagement data. For ads, new campaigns can use semantic similarity to find users with related interests before accumulating CTR history."
6. **"Multi-tenancy patterns are identical"**
   - "Multi-tenant search with workspace isolation is exactly like advertiser isolation with budget constraints. Same sharding strategies, same quota enforcement patterns."

4.3 Example: How to Open the Discussion

Example Stories

Interviewer: "Design an ad serving system for Snapchat."
Your Response:
"Great question! Before diving in, I want to clarify - in my experience building search systems at Roblox, I've found that ad serving is fundamentally a retrieval and ranking problem. The core challenge is:
Given a user context (the 'query'), find the most relevant ads (the 'documents') from millions of campaigns, rank them by relevance × bid, and serve the winner - all in ¡100ms.
I'd approach this using the same multi-stage retrieval architecture we used for search:
Stage 1: Candidate Retrieval (20ms)

- Use inverted indices to filter by demographics, geo, interests
- Retrieve top 1000 candidates (like search's initial retrieval)

Stage 2: Fast Ranking (30ms)

- Simple scoring: bid × historical_ctr × user_affinity
- Narrow to top 20 candidates

Stage 3: Accurate Ranking (30ms)

- Deep learning model to predict pCTR (like search reranking with BERT)
- Final score: bid × pCTR

Stage 4: Auction (10ms)

- Second-price auction among top 3

Now, for Snapchat specifically, there are unique considerations around video ads, mobile-first delivery, and privacy-first targeting. Let me dive into those..."
Why this works:

- Shows you understand the problem fundamentally (retrieval/ranking)
- Leverages your search expertise explicitly
- Demonstrates structured thinking (multi-stage architecture)
- Sets you up to go deep into each component
- Shows confidence translating expertise to new domain

5 Large Scale Ads System Architecture

5.1 Problem Statement (Typical Interview Question)

Interview Component

Design an ad serving system for Snapchat.
Requirements:

- Support 400M+ daily active users (Snapchat scale)
- Serve 3B+ ad impressions/day (35K-100K req/sec peak)
- **Vertical video ads** - full-screen, mobile-first format
- Multiple ad formats: Snap Ads, Story Ads, AR Lens Ads, Collection Ads
- Real-time bidding with multiple advertisers competing
- Sub-100ms latency for ad serving (p99)
- Video CDN integration for fast delivery
- Campaign budget tracking and pacing
- Fraud detection and brand safety
- **Privacy-first** (Gen Z audience, strict regulations)

Snapchat-Specific Challenges:

- **Video-First:** 99% of ads are vertical video (not display)
- **Ephemeral Content:** Stories disappear after 24h (different from Facebook/Instagram)
- **Mobile-Only:** iOS and Android native apps (no web)
- **High Engagement, Short Sessions:** Users open 30+ times/day, but 5-10 min sessions
- **Young Demographic:** 75% under age 34 - privacy, safety critical
- **AR Integration:** Sponsored lenses/filters - unique ad format

Key Metrics:

- **CTR (Click-Through Rate):** clicks / impressions (target: 1-3%)
- **CPM (Cost Per Mille):** cost per 1000 impressions ($1-10)
- **CPC (Cost Per Click):** cost per click ($0.50-2.00)
- **Fill Rate:** % of ad requests successfully filled (90-95%)
- **eCPM (effective CPM):** CPC $\times$ CTR $\times$ 1000 ($5-30)
- **Latency:** p50, p99, p999 for ad serving (target: p99 ¡100ms)
- **Video Completion Rate:** % watching full 6-10 second ad (target: 70-80%)

5.2 High-Level Architecture

Strategy & Approach

Core Components:

```
User App --> API Gateway --> Ad Serving Layer --> Ad Selection Engine
                                      |                    |
                                                     +-----+-----+
                                      |              |           |
                                   Cache       Targeting   Ranking
                                  (Redis)       Engine      Engine
                                      |              |           |
                                      |         +----+----+      |
                                      |         |         |      |
                                      +-----> User   Campaign    |
                                             Profile   Data       |
                                               DB        DB       |
                                                                  |
                          Budget Service <------------------------+
                                |
                          (Tracks spend in real-time)
```

Key Design Decisions:

1. **Separate Ad Serving from Ad Selection:** Ad serving is latency-critical (100ms SLA). Ad selection can be pre-computed or cached.
2. **Cache-First Architecture:** 90% of requests served from Redis cache. Database only for cache misses.
3. **Asynchronous Budget Tracking:** Don't block ad serving on budget checks. Use eventual consistency.
4. **Tiered Targeting:** Broad targeting (country, age) in first pass. Granular targeting (interests, behavior) in second pass.

5.3 Ad Serving Flow (Critical Path)

Critical Talking Points

Optimize for the 100ms latency requirement:
Step 1: Request Processing (5ms)

- Validate request (user_id, context, ad_slot)
- Extract user features (from cookie/token)
- Enrich with geo, device, app context

Step 2: Targeting & Retrieval (20ms)

- Query Redis for eligible ad campaigns (filter by geo, age, device)
- Retrieve top 100 candidate ads (from pre-filtered sets)
- **Optimization:** Pre-compute user segments (e.g., "US_Male_18-24") and cache eligible ads per segment

Step 3: Ranking & Selection (30ms)

- Score each candidate ad: $score = bid \times pCTR \times quality$
- Apply business rules: frequency capping, budget pacing, brand safety
- Select top 3 ads for auction

Step 4: Auction & Pricing (10ms)

- Run second-price auction: winner pays 2nd highest bid + $0.01
- Calculate eCPM: $eCPM = CPC \times pCTR \times 1000$
- Return winning ad with tracking pixels

Step 5: Response & Logging (35ms)

- Return ad creative (CDN URL, title, CTA)
- Log impression event (async to Kafka)
- Update budget tracker (async)

Total: 5 + 20 + 30 + 10 + 35 = 100ms

5.4 Detailed Component Design

5.4.1 1. User Targeting Engine

Challenge: Match ads to users in ¡20ms from pool of 1M active campaigns.
Solution: Multi-Level Indexing

```
# Level 1: Broad Segments (pre-computed hourly)
user_segment = compute_segment(user_profile)
# Example: "US_Male_18-24_Sports"

# Level 2: Redis Sorted Sets per Segment
```

```python
# Key: segment:{user_segment}
# Score: bid * pCTR (pre-computed)
# Members: campaign_ids

candidate_campaigns = redis.zrevrange(
    f"segment:{user_segment}",
    start=0,
    end=100  # Top 100 candidates
)

# Level 3: Fine-Grained Filtering (in-memory)
eligible_campaigns = []
for campaign_id in candidate_campaigns:
    campaign = get_campaign_from_cache(campaign_id)

    # Check detailed targeting criteria
    if matches_interests(user, campaign.interests):
        if has_budget_remaining(campaign):
            if passes_frequency_cap(user, campaign):
                eligible_campaigns.append(campaign)

    if len(eligible_campaigns) >= 20:
        break  # Early exit

return eligible_campaigns
```

Key Optimizations:

- Pre-compute user segments (reduce targeting dimensions from 100 to 5)
- Use Redis sorted sets for fast top-K retrieval
- Cache campaign metadata in application memory (100MB)
- Early exit once 20 eligible candidates found

5.4.2 2. Ranking Engine (ML-Based)

Challenge: Predict ad performance (pCTR) in ¡10ms.
Solution: Two-Stage Ranking

```python
class AdRanker:
    def __init__(self):
        # Stage 1: Simple logistic regression (in-memory)
        self.fast_model = LogisticRegressionModel()

        # Stage 2: Deep learning model (for top-K)
        self.accurate_model = DNNModel()

    def rank(self, user, candidates):
        # Stage 1: Fast scoring (all candidates)
        scored = []
        for ad in candidates:
            features = extract_features(user, ad)
            pCTR = self.fast_model.predict(features)
            score = ad.bid * pCTR * ad.quality_score
            scored.append((ad, score, pCTR))

        # Sort by score
        scored.sort(key=lambda x: x[1], reverse=True)

        # Stage 2: Accurate scoring (top 10 only)
        top_10 = scored[:10]
        refined = []
        for ad, score, _ in top_10:
            features = extract_detailed_features(user, ad)
            pCTR = self.accurate_model.predict(features)
            final_score = ad.bid * pCTR * ad.quality_score
            refined.append((ad, final_score, pCTR))

        refined.sort(key=lambda x: x[1], reverse=True)
        return refined

    def extract_features(self, user, ad):
        # Fast features (10-20 features)
        return [
            user.age_bucket,
            user.gender,
            ad.category,
            ad.historical_ctr,
            hour_of_day(),
            day_of_week()
        ]

    def extract_detailed_features(self, user, ad):
        # Comprehensive features (100-200 features)
        return [
            *self.extract_features(user, ad),
            user.recent_clicks,  # Last 10 clicked categories
            user.recent_views,   # Last 50 viewed content
            ad.creative_features,  # Image embeddings
            user_ad_affinity_score(user, ad),
            time_since_last_ad_of_category(user, ad.category)
        ]
```

Model Training Pipeline:

- Train on 30 days of click logs (1B impressions, 20M clicks)
- Features: user demographics, ad attributes, context, historical CTR
- Target: binary (clicked or not)
- Update models daily with fresh data
- A/B test new models (10% traffic) before full rollout

5.4.3 3. Budget Management Service

Challenge: Prevent campaign overspend while serving ads at 50K req/sec.
Problem: If we check budget on every request:

- Database bottleneck (50K writes/sec)
- Race conditions (concurrent requests)
- Latency impact (adds 20-50ms)

Solution: Asynchronous Budget Tracking with Pacing

```python
class BudgetManager:
    def __init__(self):
        # In-memory budget cache (updated every 10s)
        self.budget_cache = {}  # {campaign_id: remaining_budget}

        # Redis for distributed coordination
        self.redis = RedisClient()

        # Kafka for async spend updates
        self.kafka_producer = KafkaProducer()

    def can_serve(self, campaign_id, cost):
        """
        Fast check: is budget likely available?
        Don't block on exact budget check.
        """
        cached_budget = self.budget_cache.get(campaign_id)

        if cached_budget is None:
            # Cache miss - fetch from Redis
            cached_budget = self.redis.get(f"budget:{campaign_id}")
            self.budget_cache[campaign_id] = cached_budget

        # Fuzzy check with 10% buffer
        if cached_budget < cost * 0.9:
            return False

        # Pacing: don't spend entire budget too quickly
        pacing_rate = self.get_pacing_rate(campaign_id)
        if random.random() > pacing_rate:
            return False  # Throttle to spread spend over day

        return True

    def record_impression(self, campaign_id, cost):
        """
        Async: send spend event to Kafka.
        Don't wait for confirmation.
        """
        event = {
            "campaign_id": campaign_id,
            "cost": cost,
            "timestamp": time.time()
        }
        self.kafka_producer.send("ad_spend", event)

        # Optimistic update to local cache
        self.budget_cache[campaign_id] -= cost

    def get_pacing_rate(self, campaign_id):
        """
        Pacing: spread budget evenly over campaign duration.

        Example: $1000 budget for 10-day campaign
        - Should spend $100/day
        - If it's day 3 and spent $200, we're on pace
        - If spent $400, we're ahead - slow down (return 0.5)
        - If spent $100, we're behind - speed up (return 1.0)
        """
        campaign = self.get_campaign(campaign_id)

        days_elapsed = (now() - campaign.start_date).days
        days_remaining = (campaign.end_date - now()).days

        expected_spend = (campaign.budget * days_elapsed
                         / campaign.duration_days)
        actual_spend = campaign.total_spend

        if actual_spend > expected_spend * 1.2:
            return 0.5  # Slow down (50% of requests)
        elif actual_spend < expected_spend * 0.8:
            return 1.0  # Speed up (100% of requests)
        else:
            return 0.9  # On pace (90% of requests)

# Separate service: Budget Updater (consumes Kafka)
class BudgetUpdater:
    def run(self):
        for message in kafka_consumer:
            campaign_id = message["campaign_id"]
            cost = message["cost"]

            # Update database (batched every 10 seconds)
            self.pending_updates[campaign_id] += cost

            if time.time() - self.last_flush > 10:
                self.flush_updates()

    def flush_updates(self):
        """
        Batch update database every 10 seconds.
        """
        with db.transaction():
            for campaign_id, cost in self.pending_updates.items():
                db.execute("""
                    UPDATE campaigns
                    SET total_spend = total_spend + %s
                    WHERE id = %s
                """, (cost, campaign_id))

                # Update Redis cache
                redis.decrby(f"budget:{campaign_id}", cost)

        self.pending_updates.clear()
        self.last_flush = time.time()
```

Tradeoffs:

- **Pro:** Zero latency impact on ad serving (async updates)
- **Pro:** High throughput (50K req/sec without database bottleneck)
- **Con:** Eventual consistency - may overspend by 1-2% before detection
- **Con:** Complex state management (cache, Redis, database)

Mitigation for Overspend:

- Set soft limit at 95% of budget (hard stop at 100%)
- Alert advertisers at 80%, 90%, 95% spend
- Refund overspend (typically ¡1% of budget)

5.5 Scaling Considerations

5.5.1 Database Sharding Strategy

Challenge: Single database can't handle 50K req/sec + analytics queries.

Solution: Separate OLTP and OLAP

```
Shard Key: advertiser_id

Shard 1 (advertisers 0-999):
  - Campaigns for advertisers 0-999
  - Real-time metrics (impressions, clicks, spend)
  - Read replicas for reporting queries

Shard 2 (advertisers 1000-1999):
  - Campaigns for advertisers 1000-1999
  - ...

Data Warehouse (Redshift/BigQuery):
  - ETL pipeline (every 5 minutes)
  - Historical data (90 days)
  - Complex analytics queries
  - Campaign performance reports
```

Why Shard by advertiser_id:

- Advertisers query only their own campaigns (no cross-shard queries)
- Budget tracking per advertiser (no distributed transactions)
- Easy to add shards as advertiser count grows

5.5.2 Caching Strategy

Multi-Layer Cache:

```python
class AdCache:
    def __init__(self):
        # L1: Application memory (100MB)
        self.l1_cache = {}  # Hot campaigns (top 1000)

        # L2: Redis (10GB)
        self.redis = RedisClient()

        # L3: Database (source of truth)
        self.db = DatabaseClient()

    def get_campaign(self, campaign_id):
        # L1 cache check (1ms)
        if campaign_id in self.l1_cache:
            return self.l1_cache[campaign_id]

        # L2 cache check (5ms)
        campaign = self.redis.get(f"campaign:{campaign_id}")
        if campaign:
            self.l1_cache[campaign_id] = campaign  # Populate L1
            return campaign

        # L3 database query (50ms)
        campaign = self.db.query(
            "SELECT * FROM campaigns WHERE id = %s",
            (campaign_id,)
        )

        # Populate L2 and L1
        self.redis.setex(f"campaign:{campaign_id}", 3600, campaign)
        self.l1_cache[campaign_id] = campaign

        return campaign

    def invalidate(self, campaign_id):
        """
        When campaign updated, invalidate all cache layers.
        """
        del self.l1_cache[campaign_id]
        self.redis.delete(f"campaign:{campaign_id}")
```

Cache Hit Rates:

- L1 (memory): 80% hit rate, 1ms latency
- L2 (Redis): 18% hit rate, 5ms latency
- L3 (database): 2% hit rate, 50ms latency
- **Overall p99 latency: 10ms**

5.5.3 Handling Traffic Spikes

Challenge: Super Bowl ads cause 10x traffic spike (500K req/sec).

Solutions:

1. Auto-Scaling

- Kubernetes HPA (Horizontal Pod Autoscaler)
- Scale out ad serving pods based on CPU/latency
- Pre-warm instances 1 hour before known events

2. Rate Limiting

- Limit per advertiser: 1000 req/sec max
- Shed load gracefully: return cached house ads if overloaded
- Priority tiers: premium advertisers get guaranteed QPS

3. Graceful Degradation

- Skip ML ranking if latency ¿80ms (use simple bid-based ranking)
- Skip personalization if latency ¿60ms (use geo-based targeting only)
- Always return an ad (even if suboptimal) - never blank space

5.6 Critical Tradeoffs Discussion

Key Considerations

Be ready to discuss these tradeoffs in depth:

1. Consistency vs Latency (Budget Tracking)

- Strong consistency: every request checks database (50ms latency, 1K req/sec max)
- Eventual consistency: async updates (5ms latency, 50K req/sec, 1-2% overspend risk)
- **Decision:** Eventual consistency with 95% soft limit and overspend refunds

2. Accuracy vs Speed (Ranking)

- Deep learning: +10% CTR improvement, 50ms latency
- Logistic regression: baseline CTR, 5ms latency
- **Decision:** Two-stage ranking (fast model for all, deep model for top-10)

3. Personalization vs Privacy

- Full personalization: use browsing history, clicks, location (privacy concerns)
- Contextual only: use current page context (less effective targeting)
- **Decision:** Hybrid with user consent + differential privacy for aggregated signals

4. Real-Time vs Batch (Analytics)

- Real-time: advertisers see metrics in 1 minute (complex, expensive)
- Batch: advertisers see metrics in 15 minutes (simpler, cheaper)
- **Decision:** Hybrid - impressions real-time (Kafka), conversions batch (hourly)

5. Cost vs Performance (Infrastructure)

- Dedicated Redis per advertiser: perfect isolation, $1M/year
- Shared Redis with namespacing: 99% isolation, $100K/year
- **Decision:** Shared Redis with strict monitoring and alerts

6 Advanced Topics

6.1 Real-Time Bidding (RTB)

Challenge: Integrate with external ad exchanges - must respond in 100ms.

```
External Ad Exchange (Google AdX, Facebook Audience Network)
           |
           | Bid Request (50ms timeout)
           v
    Our Bidding Service
           |
    +------+------+
    |             |
Targeting    Ranking
 Engine      Engine
    |             |
    +------+------+
           |
      Bid Response
   (bid price, ad creative)
```

Optimization: Pre-Compute Bids

- For each campaign, pre-compute bid ranges per user segment
- Store in Redis: `bid:campaign:{id}:segment:{segment}` = $2.50
- On bid request, lookup pre-computed bid (5ms) instead of running full ML model (30ms)

6.2 Fraud Detection

Common Fraud Patterns:

- **Click Farms:** Bot networks generating fake clicks
- **Impression Fraud:** Hidden iframes loading ads (no real views)
- **Attribution Fraud:** Fake install attributions to get CPI payouts

Detection Techniques:

```python
class FraudDetector:
    def is_suspicious(self, impression):
        score = 0

        # 1. Device fingerprinting
        if self.is_emulator(impression.device):
            score += 50

        # 2. Behavioral signals
        if impression.time_to_click < 0.1:  # Too fast (bot)
            score += 30

        if self.click_count(impression.ip) > 100:  # Too many clicks
            score += 40

        # 3. Geo signals
        if self.is_vpn(impression.ip):
            score += 20

        # 4. Historical patterns
        if impression.user_id in self.known_fraudsters:
            score += 100

        return score > 70  # Threshold for blocking

    def get_click_velocity(self, user_id):
        """
        Detect unnatural click patterns.
        Normal user: 1-3 clicks/hour
        Bot: 100+ clicks/hour
        """
        recent_clicks = redis.zcount(
            f"clicks:{user_id}",
            min=time.time() - 3600,  # Last hour
            max=time.time()
        )
        return recent_clicks
```

6.3 A/B Testing Framework

Use Cases:

- Test new ranking algorithms
- Test different ad formats
- Test pricing strategies (second-price vs first-price auction)

```
class ABTestFramework:
    def get_variant(self, user_id, experiment_name):
        """
        Consistent assignment: same user always gets same variant.
        """
        hash_val = hashlib.md5(
            f"{user_id}:{experiment_name}".encode()
        ).hexdigest()

        bucket = int(hash_val[:8], 16) % 100

        experiment = self.get_experiment(experiment_name)

        if bucket < experiment.control_pct:
            return "control"
        elif bucket < experiment.control_pct + experiment.test_pct:
            return "test"
        else:
            return "excluded"  # Not in experiment

    def serve_ad(self, user_id, context):
        variant = self.get_variant(user_id, "ranking_v2")

        if variant == "control":
            ranker = self.baseline_ranker
```

```python
        elif variant == "test":
            ranker = self.new_ranker
        else:
            ranker = self.baseline_ranker

        ad = ranker.rank(context)

        # Log for experiment analysis
        self.log_experiment_impression(
            user_id, "ranking_v2", variant, ad.id
        )

        return ad
```

6.4 Snapchat-Specific System Design Considerations

6.4.1 Video Ad Serving Architecture

Challenge: Serve vertical video ads at scale - 3B+ impressions/day, 100MB+ video files.
Architecture:

```python
# Video Ad Delivery Flow
class VideoAdServer:
    def serve_video_ad(self, user, context):
        # 1. Select ad (same as display ads - 50ms)
        ad = self.select_ad(user, context)

        # 2. Return video metadata (NOT the video file)
        # Client downloads video from CDN
        return {
            "ad_id": ad.id,
            "video_url": self.get_cdn_url(ad.video_id),
            "thumbnail_url": self.get_cdn_url(ad.thumbnail_id),
            "duration": ad.duration,  # e.g., 6 seconds
            "cta": ad.call_to_action,
            "tracking_pixels": {
                "impression": f"https://track.snap.com/imp/{ad.id}",
                "quartile_25": f"https://track.snap.com/q25/{ad.id}",
                "quartile_50": f"https://track.snap.com/q50/{ad.id}",
                "quartile_75": f"https://track.snap.com/q75/{ad.id}",
                "complete": f"https://track.snap.com/comp/{ad.id}",
                "click": f"https://track.snap.com/click/{ad.id}"
            }
        }

    def get_cdn_url(self, video_id):
        """
        Return geographically-optimized CDN URL.
        """
        user_region = self.get_user_region()

        # Multi-CDN strategy (Fastly, Akamai, CloudFront)
        cdn = self.select_cdn(user_region)

        return f"https://{cdn}.snapcdn.com/video/{video_id}.mp4"

    def select_cdn(self, region):
        """
        Route to fastest CDN per region.
        """
        if region == "US":
            return "us-west.cdn"
        elif region == "EU":
            return "eu-central.cdn"
        elif region == "APAC":
            return "apac-east.cdn"
```

Key Optimizations:

- **Pre-cache hot videos:** Top 1000 ads cached on edge servers globally
- **Adaptive bitrate:** Serve 480p/720p/1080p based on network speed
- **Progressive download:** Start playback before full video downloaded
- **Video preloading:** Prefetch next ad while user watches current content

6.4.2 AR Lens Ads - Unique Challenge

Problem: Sponsored AR lenses (face filters) require real-time rendering on device.
Architecture:

```
Lens Ad Flow:
1. User opens Snapchat camera
2. Client downloads lens manifest (JSON, 10KB) from CDN
3. Client downloads 3D assets (meshes, textures, 5MB) from CDN
4. Client renders AR effect using local GPU
5. Log impression event when lens applied
6. Log engagement event when photo/video shared
```

Challenges:

- Assets must be ¡5MB (mobile bandwidth constraints)
- Rendering must be 60fps (smooth AR experience)
- Works offline after initial download
- Difficult attribution (lens used days after download)

6.4.3 Privacy-First Ad Targeting

Challenge: Target ads without invasive tracking (Gen Z values privacy).
Snapchat's Approach:

```
class PrivacyFirstTargeting:
    def get_targetable_attributes(self, user):
        """
        Limited targeting - privacy-preserving.
        """
        return {
            # Basic demographics (user-provided)
            "age_range": user.age_range,  # e.g., "18-24"
            "gender": user.gender,
            "location": user.city,  # City-level (not precise GPS)

            # Interest-based (inferred from content, not browsing)
            "interests": self.get_snap_interests(user),
            # e.g., ["sports", "music", "fashion"]

            # NO cross-site tracking
            # NO third-party data brokers
            # NO pixel tracking on external websites
        }

    def get_snap_interests(self, user):
        """
        Infer interests from Snapchat activity ONLY.
        """
        interests = []

        # Stories user watches
        for story in user.recent_story_views:
            interests.append(story.category)

        # Lenses user tries
        for lens in user.recent_lens_uses:
            interests.append(lens.category)

        # Discover content user reads
        for article in user.recent_discover_views:
            interests.append(article.category)

        return list(set(interests))  # Deduplicate
```

Tradeoff:

- **Pro:** User trust, regulatory compliance, brand safety
- **Con:** Lower ad relevance (CPM $2-5 vs Facebook $7-10)
- **Decision:** Snapchat prioritizes user experience over ad revenue per user

6.4.4 Mobile-First Infrastructure

Challenge: 100% mobile traffic (iOS/Android), no web fallback.
Considerations:

- **Battery Efficiency:** Minimize CPU/GPU usage for ad rendering
- **Bandwidth Optimization:** Adaptive quality, video compression
- **Offline Support:** Cache ads for offline viewing
- **Push Notifications:** Re-engagement campaigns via push
- **App Version Fragmentation:** Support 10+ versions simultaneously

```
class MobileAdOptimizer:
    def optimize_for_device(self, ad, device_info):
        """
        Adapt ad delivery based on device capabilities.
        """
        if device_info.network == "wifi":
            # High quality, pre-cache multiple ads
            return self.get_hd_video(ad), self.prefetch_next_ads(3)

        elif device_info.network == "4G":
            # Standard quality, prefetch 1 ad
            return self.get_sd_video(ad), self.prefetch_next_ads(1)

        elif device_info.network == "3G":
            # Low quality, no prefetch
            return self.get_low_quality_video(ad), None

        if device_info.battery_level < 20:
            # Low battery mode: reduce quality, skip video preload
            return self.get_low_quality_video(ad), None

        if device_info.storage_available < 500MB:
            # Low storage: don't cache ads
            return ad, None
```

6.5 Multi-Region Deployment

Challenge: Serve ads globally with ¡100ms latency.

Architecture:

```
Region: US-East
  - Ad Serving Cluster
  - Redis Cache (hot campaigns)
  - User Profile DB (read replica)

Region: EU-West
  - Ad Serving Cluster
  - Redis Cache (hot campaigns)
  - User Profile DB (read replica)

Region: AP-Southeast
  - Ad Serving Cluster
  - Redis Cache (hot campaigns)
  - User Profile DB (read replica)

Central Region (US-West):
  - Campaign DB (master)
  - Budget Tracking Service
  - Analytics Data Warehouse
```

Data Consistency:

- Campaign metadata: replicated globally (10 min lag acceptable)
- User profiles: replicated globally (5 min lag acceptable)
- Budget tracking: centralized (eventual consistency, 1 min lag)
- Real-time metrics: local aggregation, global rollup every 15 min

7 2-Hour Quick Reference Cheat Sheet

7.1 Opening Statement: Leverage Your Search Expertise

Critical Talking Points

CRITICAL: Open with this framing to establish credibility
"I want to start by noting that in my experience building large-scale search systems at Roblox, I've found that **ad serving is fundamentally a retrieval and ranking problem**.
The core challenge: given a user context (the 'query'), find the most relevant ads (the 'documents') from millions of campaigns, rank by relevance × bid - all in ¡100ms.
My search background directly applies to Snapchat's ads platform:

- **Multi-stage retrieval:** inverted index → fast ranking → ML reranking (same as search)
- **Hybrid targeting:** BM25 + vector embeddings for semantic matching
- **Learning to Rank:** CTR prediction models (same features as search relevance)
- **Multi-tenancy:** advertiser isolation = workspace isolation patterns
- **Snapchat-specific:** Video CDN delivery, privacy-first targeting, AR lens ads

Let me walk through the architecture, starting with the latency budget..."
Pro tip: Pause after this to see if interviewer wants to go breadth-first or depth-first. Adapt based on their response.
Why this works: Immediately demonstrates you understand the problem fundamentally and have directly relevant expertise. Turns potential weakness (no ads background) into strength (deep retrieval/ranking expertise).

7.2 Top 3 Behavioral Talking Points

Critical Talking Points

1. **Large Scale Migration Story**
   - Context: 500M records, 3 teams, 6 months
   - My role: Tech lead, architecture, coordination
   - Technical decisions: Hybrid multi-tenancy (cost vs performance)
   - Risk mitigation: Dual-write, feature flags, shadow traffic
   - Result: 85% latency improvement, zero downtime, $400K savings
2. **Technical Tradeoff Example**
   - Problem: Real-time vs batch for ad attribution
   - Options: Real-time (complex, $100K), micro-batch (5 min, $50K), hourly batch ($20K)
   - Analysis: User behavior (check every 15-30 min) didn't justify real-time
   - Decision: Micro-batch (5 min) - sweet spot of freshness vs complexity
   - Result: 50% cost savings, delivered 2x faster
3. **Strategic Roadmap Creation**
   - Goal: Scale ads platform 10x (20K to 200K req/sec)
   - 4-phase roadmap: Observability → Decouple Services → Scale DB → ML Optimization
   - Principle: Each phase delivers standalone value and de-risks next phase
   - Result: Executed 90%, scaled 7.5x, reduced incidents 60%

7.3 Top 7 Snapchat-Specific Talking Points

Critical Talking Points

1. **Video Ad Serving at Snapchat Scale**
   - **Scale:** 400M DAU, 3B+ video impressions/day
   - **Format:** 99% vertical video (9:16), 6-10 second ads
   - **Delivery:** Metadata in ¡50ms, video from CDN (progressive download)
   - **Optimization:** Pre-cache top 1000 ads, adaptive bitrate, video preloading
   - **Tracking:** Quartile-based metrics (25%, 50%, 75%, completion)
2. **Ad Serving Latency Budget (100ms total) - Memorize This!**
   - Request processing: 5ms (validate, extract features)
   - Targeting & retrieval: 20ms (Redis, pre-filtered candidates)
   - Ranking: 30ms (Stage 1: fast model → Stage 2: deep model top-10)
   - Auction: 10ms (second-price auction)
   - Response & logging: 35ms (async Kafka)
   - **Key insight:** Multi-stage ranking same as search (fast → accurate)
3. **Privacy-First Targeting (Snapchat Differentiator)**
   - **Challenge:** Gen Z values privacy - no invasive tracking
   - **Snapchat Approach:** First-party data only (Stories, Lenses, Discover)
   - NO cross-site tracking, NO third-party data brokers
   - City-level location (not GPS), interest inference from in-app behavior
   - **Tradeoff:** Lower CPM ($2-5 vs Facebook $7-10), higher user trust
   - **Principal-level insight:** Product decision prioritizes UX over revenue/user
4. **AR Lens Ads - Unique Format**
   - Sponsored lenses/filters downloaded to device (¡5MB assets)
   - Client-side GPU rendering at 60fps for smooth AR
   - Works offline after download (unlike traditional ads)
   - Attribution challenge: lens used days after impression
   - High engagement: 10-20x interaction rate vs standard video ads
5. **Budget Tracking Tradeoff**
   - Challenge: 100K req/sec peak, can't check database every request
   - Solution: Eventual consistency with async updates
   - Cache budget in Redis, update every 10 seconds via Kafka batching
   - Pacing algorithm to spread budget over campaign duration
   - Accept 1-2% overspend risk, refund advertisers

7.4 Additional System Design Points

Critical Talking Points

1. **Scaling Strategy**
   - Database: Shard by advertiser_id (no cross-shard queries)
   - Separate OLTP (real-time serving) from OLAP (analytics)
   - Data warehouse for historical queries (ETL every 5 min)
   - Auto-scaling for traffic spikes (10x during major events)
   - Graceful degradation: skip ML ranking if latency ¿80ms
2. **Fraud Detection**
   - Device fingerprinting (detect emulators)
   - Behavioral signals (time to click, click velocity)
   - Geo signals (VPN detection)
   - Block score ¿70 threshold
   - Cost: prevent 5-10% fraud waste

7.5 Key Metrics to Know

Metric	**Formula**	**Typical Value**
CTR	clicks / impressions	1-3%
CPM	cost per 1000 impressions	$1-10
CPC	cost per click	$0.50-2.00
eCPM	CPC * CTR * 1000	$5-30
Fill Rate	filled requests / total requests	90-95%
Latency (p99)	99th percentile response time	¡100ms

7.6 Questions to Ask Interviewer

1. What's the biggest scaling challenge for Snapchat's video ad platform today?
2. How does Snapchat balance user experience (Gen Z expectations) with ad monetization?
3. What's unique about AR Lens ads from an infrastructure perspective?
4. How does Snapchat's privacy-first approach affect ad targeting effectiveness?
5. What's the team structure for ads infrastructure? How do iOS/Android/Backend teams coordinate?
6. How does Snapchat measure ad quality/relevance beyond CTR (e.g., completion rate, swipe-away rate)?
7. What's the strategy for competing with TikTok and Instagram Reels in the video ads space?

7.7 Red Flags to Avoid

Key Considerations

Don't say:

- "We'll just use microservices" (without justifying why)
- "NoSQL is always better than SQL" (depends on use case)
- "We'll make it real-time" (expensive, may not be needed)
- "This is how Google does it" (different scale, different constraints)

Instead say:

- "Let me think through the tradeoffs..."
- "What are the latency requirements?" (clarify before designing)
- "We could do X, but the downside is Y. Given the constraints, I'd choose Z"
- "I'd start with a simple architecture and add complexity only when needed"

7.8 Day-of-Interview Checklist

Pro Tips

30 Minutes Before Interview:

1. Review Opening Statement (search → ads connection)
2. Review Top 7 Snapchat-Specific Talking Points
3. Review Top 3 Behavioral Stories
4. Glance at Key Metrics table
5. Remember: Ask clarifying questions, think through tradeoffs, quantify impact

First 60 Seconds:

- When asked about system design, immediately establish the search-retrieval framing
- "Ad serving is fundamentally a retrieval and ranking problem..."
- This positions you as an expert from the start

7.9 Principal Engineer Differentiation

Critical Talking Points

How to stand out as Principal-level:

1. Think Beyond Code

- Mention: team coordination, technical roadmap, organization-wide impact
- ”I created a migration playbook that 4 other teams used”
- ”I established architectural review process for cross-team consistency”

2. Quantify Impact

- Not: ”improved performance”
- Instead: ”reduced p99 latency from 2s to 300ms, saving $400K/year”

3. Show Strategic Thinking

- Explain why you chose this approach over alternatives
- Discuss long-term maintainability, not just quick wins
- ”This design allows us to add feature X in Q3 without major refactor”

4. Acknowledge Tradeoffs

- ”We chose eventual consistency to optimize for latency, accepting 1-2% overspend risk”
- ”This adds operational complexity, but the cost savings justify it”

5. Cross-Functional Coordination

- ”I worked with product, data science, and infra teams to align on requirements”
- ”I created a shared RFC process for architectural decisions”