Generative AI

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Explain the self-attention mechanism in transformers and how it differs from traditional RNN attention.

Self-Attention Mechanism

Self-attention allows each token in a sequence to attend to all other tokens simultaneously, computing attention weights based on query, key, and value projections. Unlike RNN attention which processes sequentially, self-attention is parallelizable.

Key Differences:

- **Parallelization:** Self-attention processes all positions simultaneously, while RNNs must process sequentially
- **Long-range dependencies:** Direct connections between any two positions regardless of distance
- **Computational complexity:** $O(n^2d)$ for self-attention vs $O(nd^2)$ for RNNs, where n is sequence length and d is dimension
- **No temporal bias:** Requires positional encodings, unlike RNNs which have inherent sequential processing

Formula:

Attention(Q, K, V) = softmax(QK^T / √d_k)V

Where queries, keys, and values are linear projections of the input, and d_k is the dimension of keys used for scaling to prevent vanishing gradients in softmax.

2. What is the difference between autoregressive and non-autoregressive generation in language models?

Autoregressive Generation

Autoregressive models (like GPT) generate tokens sequentially, where each token depends on all previously generated tokens. The model predicts P(x_t | x_1, ..., x_{t-1}).

```
for i in range(max_length):
    logits = model(tokens[:i])
    next_token = sample(logits)
    tokens.append(next_token)
    if next_token == EOS:
        break
```

Non-Autoregressive Generation

Non-autoregressive models generate all tokens in parallel or with limited iterations, predicting P(x_1, ..., x_n) simultaneously.

Trade-offs:

- **Speed:** Non-autoregressive is much faster (parallel generation)
- **Quality:** Autoregressive typically produces higher quality due to conditional dependencies
- **Flexibility:** Autoregressive allows dynamic length; non-autoregressive requires length prediction
- **Use cases:** Autoregressive for creative text; non-autoregressive for fixed-length tasks like translation

3. Describe the architecture and training process of diffusion models like Stable Diffusion.

Diffusion Model Architecture

Diffusion models consist of two processes: a forward diffusion process that gradually adds Gaussian noise to data, and a reverse denoising process learned by a neural network (typically a U-Net with attention layers).

Forward Process:

$q(x_t | x_{t-1}) = N(\sqrt{1-\beta_t}\, x_{t-1}, \beta_t I)$

Noise is added over T timesteps until the image becomes pure Gaussian noise.

Training Process:

- **Step 1:** Sample a training image x_0 and random timestep t
- **Step 2:** Add noise to get x_t using the closed-form: $x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\, \varepsilon$
- **Step 3:** Train a neural network ε_θ to predict the noise ε
- **Loss:** $L = ||\varepsilon - \varepsilon_\theta(x_t, t)||^2$

Stable Diffusion Specifics:

- Operates in **latent space** using a VAE encoder/decoder for efficiency
- Uses **cross-attention** to condition on text embeddings from CLIP
- Employs classifier-free guidance for better text alignment

4. What are LoRA (Low-Rank Adaptation) and QLoRA, and when would you use them for fine-tuning LLMs?

LoRA (Low-Rank Adaptation)

LoRA fine-tunes large language models by adding trainable low-rank decomposition matrices to frozen pretrained weights, drastically reducing trainable parameters.

For a weight matrix $W \in R^{d\times k}$, LoRA adds: $\Delta W = BA$, where $B \in R^{d\times r}$ and $A \in R^{r\times k}$, with $r << \min(d,k)$

```
class LoRALayer(nn.Module):
    def __init__(self, in_dim, out_dim, rank=4):
        self.A = nn.Parameter(torch.randn(rank, in_dim))
        self.B = nn.Parameter(torch.zeros(out_dim, rank))

    def forward(self, x, base_output):
        return base_output + (x @ self.A.T @ self.B.T)
```

QLoRA (Quantized LoRA)

Extends LoRA by quantizing the base model to 4-bit precision while keeping LoRA adapters in higher precision, enabling fine-tuning of 65B+ models on consumer GPUs.

When to Use:

- **Limited GPU memory:** LoRA reduces memory by 3-10x
- **Multiple task adaptation:** Store multiple small LoRA adapters instead of full models
- **Fast experimentation:** Training is 2-3x faster
- **QLoRA specifically:** When working with very large models (30B+) on limited hardware

5. Explain the concept of temperature and top-k/top-p sampling in text generation. How do they affect output quality?

Temperature Sampling

Temperature controls randomness by scaling logits before softmax: $p_i = \exp(z_i/T) / \Sigma \exp(z_j/T)$

- **T < 1:** More deterministic, peaks sharper (conservative)
- **T = 1:** Standard softmax distribution

- **T > 1:** More random, flatter distribution (creative)

```
logits = model(input_ids)
scaled_logits = logits / temperature
probs = softmax(scaled_logits)
next_token = sample(probs)
```

Top-k Sampling

Restricts sampling to the k most probable tokens, setting others to zero probability.

Top-p (Nucleus) Sampling

Top-p samples from the smallest set of tokens whose cumulative probability exceeds p, dynamically adjusting vocabulary size.

```
sorted_probs, indices = torch.sort(probs, descending=True)
cumsum = torch.cumsum(sorted_probs, dim=-1)
mask = cumsum <= p
filtered_probs = probs * mask
```

Impact on Quality:

- **Temperature:** Low values for factual tasks, high for creative writing
- **Top-k:** Prevents unlikely tokens but can cut off valid options
- **Top-p:** More adaptive, typically produces more coherent text (p=0.9 common)
- **Combined:** Often use temperature + top-p together for best results

6. What is the difference between RLHF (Reinforcement Learning from Human Feedback) and DPO (Direct Preference Optimization)?

RLHF (Reinforcement Learning from Human Feedback)

RLHF is a three-stage process used to align LLMs with human preferences:

- **Stage 1:** Supervised fine-tuning on high-quality demonstrations
- **Stage 2:** Train a reward model on human preference comparisons
- **Stage 3:** Use PPO (Proximal Policy Optimization) to optimize the LLM policy against the reward model

Objective: maximize $E[r(x,y)] - \beta \cdot KL(\pi_\theta \,||\, \pi_{ref})$, where r is the reward model and KL prevents drift from reference policy.

DPO (Direct Preference Optimization)

DPO eliminates the reward model and RL training by directly optimizing the policy using a closed-form preference loss:

$L = -E[\log \sigma(\beta \log \pi_\theta(y_w|x)/\pi_{ref}(y_w|x) - \beta \log \pi_\theta(y_l|x)/\pi_{ref}(y_l|x))]$

Where y_w is the preferred completion and y_l is the rejected one.

Key Differences:

- **Complexity:** DPO is simpler, no separate reward model or RL training
- **Stability:** DPO is more stable, avoids RL hyperparameter tuning
- **Efficiency:** DPO requires less compute and memory
- **Performance:** Both achieve similar alignment quality
- **Use case:** DPO preferred for most applications due to simplicity; RLHF when you need explicit reward modeling

7. Explain how Flash Attention optimizes the attention mechanism. What are the memory and speed improvements?

Flash Attention Optimization

Flash Attention is an IO-aware exact attention algorithm that reduces memory reads/writes between GPU HBM (high bandwidth memory) and SRAM by reordering attention computation.

Standard Attention Problems:

- Materializes the full N×N attention matrix in HBM
- Memory complexity: $O(N^2)$ for sequence length N
- Multiple HBM reads/writes create bottlenecks

Flash Attention Strategy:

- **Tiling:** Breaks Q, K, V into blocks that fit in SRAM
- **Kernel fusion:** Computes softmax and matrix multiplication in a single kernel
- **Recomputation:** Uses selective recomputation in backward pass instead of storing attention matrix
- **Online softmax:** Computes softmax incrementally without materializing full matrix

Improvements:

- **Memory:** Reduces from $O(N^2)$ to $O(N)$, enabling 4-8x longer sequences
- **Speed:** 2-4x faster on A100 GPUs due to reduced HBM access
- **Exact:** Produces identical results to standard attention (not an approximation)
- **Training:** Enables training on sequences up to 64K tokens

Flash Attention 2 further improves by better parallelization and work partitioning across thread blocks.

8. What are the key differences between encoder-only (BERT), decoder-only (GPT), and encoder-decoder (T5) transformer architectures?

Encoder-Only (BERT)

Architecture: Bidirectional self-attention across all tokens simultaneously.

- **Training:** Masked Language Modeling (MLM) - predict masked tokens
- **Use cases:** Classification, NER, question answering, embeddings
- **Strengths:** Full context understanding, excellent for understanding tasks
- **Limitations:** Cannot generate text autoregressively

Decoder-Only (GPT)

Architecture: Causal (unidirectional) self-attention with masking to prevent attending to future tokens.

- **Training:** Next token prediction (autoregressive)
- **Use cases:** Text generation, completion, few-shot learning
- **Strengths:** Natural generative capabilities, scales well with size
- **Limitations:** Only sees left context during generation

Encoder-Decoder (T5, BART)

Architecture: Encoder with bidirectional attention + decoder with causal attention and cross-attention to encoder.

- **Training:** Sequence-to-sequence objectives (span corruption, denoising)
- **Use cases:** Translation, summarization, any seq2seq task
- **Strengths:** Best for tasks requiring both understanding and generation
- **Limitations:** More parameters, slower inference than decoder-only

Modern Trend:

Decoder-only models (GPT, LLaMA) dominate due to simpler architecture and better scaling properties, even for tasks traditionally suited to encoders.

9. Describe the challenges of inference optimization for large language models and common solutions like quantization and KV cache.

Inference Challenges

- **Memory bandwidth:** Models like LLaMA-70B require 140GB+ in FP16
- **Latency:** Autoregressive generation is sequential, not parallelizable

- **Cost:** GPU hours are expensive for production serving
- **Throughput:** Batch size limited by memory

Quantization Solutions

Quantization reduces precision from FP16/FP32 to INT8/INT4, reducing memory and increasing throughput.

- **Post-training quantization (PTQ):** GPTQ, AWQ - no retraining needed
- **Quantization-aware training (QAT):** Better accuracy, requires retraining
- **Benefits:** 2-4x memory reduction, 2-3x speedup on modern GPUs

```
from transformers import AutoModelForCausalLM
model = AutoModelForCausalLM.from_pretrained(
    "model_id",
    load_in_8bit=True,
    device_map="auto"
)
```

KV Cache Optimization

KV cache stores computed key and value tensors from previous tokens to avoid recomputation.

- **Memory:** Grows linearly with sequence length (batch × seq_len × layers × hidden_dim × 2)
- **Optimization:** Multi-Query Attention (MQA) or Grouped-Query Attention (GQA) shares KV heads
- **PagedAttention:** vLLM uses paged memory management for efficient KV cache

Other Techniques:

- **Speculative decoding:** Use small model to draft, large model to verify
- **Continuous batching:** Dynamic batching as sequences complete
- **Flash Attention:** IO-efficient attention computation

10. What is prompt engineering and what are advanced techniques like chain-of-thought, few-shot learning, and ReAct prompting?

Prompt Engineering

Prompt engineering is the practice of designing inputs to guide LLM behavior and improve output quality without model fine-tuning.

Few-Shot Learning

Provide examples in the prompt to demonstrate the desired task format and style.

```
Translate to French:
Hello -> Bonjour
Goodbye -> Au revoir
Thank you -> Merci
How are you? ->
```

Chain-of-Thought (CoT)

CoT prompting encourages step-by-step reasoning by including intermediate reasoning steps in examples or explicitly requesting them.

Example: "Let's think step by step: First, identify... Then, calculate... Finally, conclude..."

- **Zero-shot CoT:** Simply add "Let's think step by step" before the question
- **Benefits:** Dramatically improves performance on reasoning tasks (math, logic)

ReAct (Reasoning + Acting)

ReAct interleaves reasoning traces with actions (tool use, API calls, searches).

Format: Thought -> Action -> Observation -> Thought -> ...

- Enables LLMs to use external tools and knowledge
- Combines chain-of-thought with interactive environment access

Other Advanced Techniques:

- **Self-consistency:** Sample multiple reasoning paths and take majority vote
- **Tree of Thoughts:** Explore multiple reasoning branches
- **Role prompting:** "You are an expert..." to set context
- **Instruction tuning:** Models trained specifically to follow instructions

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement an LRU (Least Recently Used) Cache with O(1) time complexity for both get and put operations.

LRU Cache Implementation

An **LRU Cache** requires a combination of a **hash map** (for O(1) lookups) and a **doubly linked list** (for O(1) insertion/deletion). The hash map stores key-node pairs, while the doubly linked list maintains access order.

```
class LRUCache:
    def __init__(self, capacity):
        self.cache = {}
        self.capacity = capacity
        self.head = Node(0, 0)
        self.tail = Node(0, 0)
        self.head.next = self.tail
        self.tail.prev = self.head
```

- **Get operation:** Move accessed node to front (most recent)
- **Put operation:** Add to front, evict from tail if capacity exceeded
- **Time Complexity:** O(1) for both operations
- **Space Complexity:** O(capacity)

2. How do you find all pairs in an array that sum to a target value? What's the optimal approach?

Two Sum / Pair Sum Problem

The optimal approach uses a **hash set** to achieve O(n) time complexity with a single pass through the array.

```
def find_pairs(arr, target):
    seen = set()
    pairs = []
    for num in arr:
        complement = target - num
        if complement in seen:
            pairs.append((complement, num))
        seen.add(num)
    return pairs
```

- **Time Complexity:** O(n) - single pass
- **Space Complexity:** O(n) - hash set storage
- **Alternative:** Two-pointer approach requires O(n log n) sorting first
- **Key insight:** Store complements for constant-time lookups

3. Explain the sliding window technique and implement maximum sum of a subarray of size k.

Sliding Window Technique

The **sliding window** technique maintains a window of elements and efficiently updates it by adding new elements and removing old ones, avoiding redundant calculations.

```
def max_sum_subarray(arr, k):
    window_sum = sum(arr[:k])
    max_sum = window_sum
    for i in range(k, len(arr)):
        window_sum += arr[i] - arr[i-k]
        max_sum = max(max_sum, window_sum)
    return max_sum
```

- **Time Complexity:** O(n) - single pass after initial window
- **Space Complexity:** O(1) - constant space
- **Use cases:** Substring problems, array subarrays, streaming data
- **Key benefit:** Reduces O(n*k) brute force to O(n)

4. What is the difference between a stack and a queue? Implement a queue using two stacks.

Stack vs Queue

A **stack** follows LIFO (Last In First Out), while a **queue** follows FIFO (First In First Out). Implementing a queue with two stacks demonstrates amortized O(1) operations.

```
class QueueWithStacks:
    def __init__(self):
        self.stack_in = []
        self.stack_out = []
    def enqueue(self, x):
        self.stack_in.append(x)
    def dequeue(self):
        if not self.stack_out:
            while self.stack_in:
                self.stack_out.append(self.stack_in.pop())
        return self.stack_out.pop()
```

- **Enqueue:** O(1) - push to stack_in
- **Dequeue:** Amortized O(1) - transfer when stack_out empty
- **Key insight:** Reversing twice restores FIFO order

5. Explain time complexity of dictionary/hash map operations. When would they degrade to O(n)?

Hash Map Time Complexity

Hash maps provide average O(1) operations but can degrade under specific conditions.

- **Average case:** O(1) for insert, delete, lookup
- **Worst case:** O(n) when hash collisions occur
- **Collision scenarios:** Poor hash function, high load factor, hash flooding attacks
- **Load factor:** Ratio of elements to buckets; rehashing occurs at ~0.75
- **Collision resolution:** Chaining (linked lists) or open addressing (probing)
- **Python dict:** Uses open addressing with random probing
- **Mitigation:** Good hash functions, dynamic resizing, balanced load factors

Real-world consideration: In production systems, assume O(1) but design for worst-case scenarios in security-critical applications.

6. Implement a function to detect if a linked list has a cycle. What's the optimal space complexity?

Cycle Detection in Linked List

The **Floyd's Cycle Detection Algorithm** (tortoise and hare) uses two pointers moving at different speeds to detect cycles with O(1) space.

```
def has_cycle(head):
    if not head:
        return False
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
```

```
        return True
    return False
```

- **Time Complexity:** O(n) - at most 2n iterations
- **Space Complexity:** O(1) - only two pointers
- **Alternative:** Hash set approach uses O(n) space
- **Extension:** Find cycle start by resetting one pointer

7. What is a Trie data structure? Implement insertion and search operations.

Trie (Prefix Tree)

A **Trie** is a tree-like data structure optimized for string operations, particularly prefix matching. Each node represents a character, and paths form words.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()
    def insert(self, word):
        node = self.root
        for char in word:
            node = node.children.setdefault(char, TrieNode())
        node.is_end = True
```

- **Time Complexity:** O(m) where m is word length
- **Space Complexity:** O(n*m) for n words
- **Use cases:** Autocomplete, spell checkers, IP routing
- **Advantage:** Faster prefix searches than hash maps

8. Explain the difference between BFS and DFS. When would you choose one over the other?

BFS vs DFS

Breadth-First Search (BFS) explores level by level using a queue, while **Depth-First Search (DFS)** explores as deep as possible using a stack or recursion.

- **BFS Time/Space:** O(V+E) time, O(V) space for queue
- **DFS Time/Space:** O(V+E) time, O(h) space for recursion stack (h=height)
- **Choose BFS when:** Finding shortest path, level-order traversal, nearest neighbor
- **Choose DFS when:** Detecting cycles, topological sort, path existence, using less memory
- **BFS guarantees:** Shortest path in unweighted graphs
- **DFS advantages:** Lower space complexity, easier to implement recursively
- **Graph representation:** Both work with adjacency lists or matrices

9. Implement a min heap and explain its time complexity for insertion and extraction.

Min Heap Implementation

A **min heap** is a complete binary tree where each parent is smaller than its children, typically implemented using an array.

```
class MinHeap:
    def __init__(self):
        self.heap = []
    def insert(self, val):
        self.heap.append(val)
        self._bubble_up(len(self.heap)-1)
    def extract_min(self):
        if not self.heap: return None
        self._swap(0, len(self.heap)-1)
        min_val = self.heap.pop()
        self._bubble_down(0)
    return min_val
```

- **Insert:** O(log n) - bubble up to maintain heap property
- **Extract min:** O(log n) - bubble down after removing root
- **Get min:** O(1) - root element
- **Heapify:** O(n) - build heap from array
- **Use cases:** Priority queues, Dijkstra's algorithm, top K elements

10. What is dynamic programming? Explain with the Fibonacci sequence and optimization from recursion to memoization to tabulation.

Dynamic Programming

Dynamic Programming (DP) solves problems by breaking them into overlapping subproblems and storing results to avoid redundant calculations.

Naive Recursion: O(2^n) time, O(n) space

```
def fib_recursive(n):
    if n <= 1: return n
    return fib_recursive(n-1) + fib_recursive(n-2)
```

Memoization (Top-Down): O(n) time, O(n) space

```
def fib_memo(n, cache={}):
    if n in cache: return cache[n]
    if n <= 1: return n
    cache[n] = fib_memo(n-1) + fib_memo(n-2)
    return cache[n]
```

Tabulation (Bottom-Up): O(n) time, O(1) space optimized

```
def fib_tab(n):
    if n <= 1: return n
    prev, curr = 0, 1
    for _ in range(2, n+1):
        prev, curr = curr, prev + curr
    return curr
```

- **Key principles:** Optimal substructure, overlapping subproblems
- **Common problems:** Knapsack, longest common subsequence, coin change

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly. What are the key components and how would you handle high traffic?

Key Components

- **URL Generation Service:** Creates short codes using base62 encoding or hash functions
- **Database:** NoSQL (Cassandra/DynamoDB) for high write throughput
- **Cache Layer:** Redis/Memcached for frequently accessed URLs
- **Load Balancer:** Distribute traffic across application servers
- **CDN:** Serve redirects from edge locations

Architecture Considerations

- **Stateless Design:** Application servers don't store session data, enabling horizontal scaling
- **Database Sharding:** Partition by hash of short code for distributed storage
- **Read-Heavy Optimization:** Cache-aside pattern with 80/20 rule (cache hot URLs)
- **Write Strategy:** Pre-generate short codes in batches, assign from pool

CAP Theorem Trade-off

Favor **Availability and Partition Tolerance (AP)**. Eventual consistency is acceptable - if a URL takes milliseconds to propagate, users can retry. Use multi-master replication across regions.

Sample Code Generation

```
import hashlib
import base64

def generate_short_code(url, counter):
    hash_input = f"{url}{counter}".encode()
    hash_digest = hashlib.md5(hash_input).digest()
    short_code = base64.b64encode(hash_digest)[:7].decode()
    return short_code.replace('/', '_').replace('+', '-')
```

Scalability Metrics

- Handle 10K+ requests/second with horizontal scaling
- Sub-10ms redirect latency with proper caching
- 99.99% availability with multi-region deployment

2. How would you design a real-time chat application supporting millions of concurrent users? Discuss the architecture, protocols, and data consistency challenges.

Architecture Overview

- **WebSocket Gateway:** Maintains persistent connections with clients
- **Message Queue:** Kafka/RabbitMQ for reliable message delivery
- **Presence Service:** Tracks online/offline status using Redis
- **Message Storage:** Cassandra for chat history (time-series data)
- **Push Notification Service:** For offline message delivery

Protocol Selection

WebSocket for real-time bidirectional communication. Fallback to Server-Sent Events (SSE) or long polling for older clients. Use **Protocol Buffers** for efficient message serialization.

Scaling Strategy

- **Connection Servers:** Stateful WebSocket servers, use consistent hashing to route users
- **Message Routing:** Publish-subscribe pattern via message broker
- **Horizontal Scaling:** Add more WebSocket servers behind load balancer with sticky sessions
- **Database Partitioning:** Shard by conversation_id or user_id

Data Consistency

Use **eventual consistency** for message delivery. Implement message acknowledgments and sequence numbers to handle out-of-order delivery.

```
class MessageHandler:
    def send_message(self, user_id, msg):
        msg['timestamp'] = time.time()
        msg['seq_num'] = self.get_next_seq()
        self.kafka.publish('chat-messages', msg)
        self.store_message(msg)
        return msg['seq_num']
```

CAP Considerations

Prioritize **Availability and Partition Tolerance**. Messages may arrive slightly out of order but system remains responsive. Use vector clocks for conflict resolution.

3. Design a distributed rate limiter that can handle 100K requests per second. How would you ensure accuracy and prevent race conditions?

Algorithm Selection

- **Token Bucket:** Best for smooth rate limiting with burst handling
- **Sliding Window Log:** Most accurate but memory intensive
- **Sliding Window Counter:** Balance between accuracy and efficiency

Distributed Architecture

- **Redis Cluster:** Centralized rate limit counters with atomic operations
- **Local Cache + Sync:** Each server maintains local counters, periodically syncs
- **Consistent Hashing:** Route requests for same user/API key to same rate limiter

Redis Implementation (Token Bucket)

```
-- Lua script for atomic rate limiting
local key = KEYS[1]
local capacity = tonumber(ARGV[1])
local rate = tonumber(ARGV[2])
local now = tonumber(ARGV[3])
local tokens = redis.call('hget', key, 'tokens')
if not tokens then tokens = capacity end
local last = redis.call('hget', key, 'last') or now
tokens = math.min(capacity, tokens + (now-last)*rate)
if tokens >= 1 then return 1 else return 0 end
```

Race Condition Prevention

- **Atomic Operations:** Use Redis Lua scripts for read-modify-write atomicity
- **Optimistic Locking:** Version numbers with compare-and-swap
- **Distributed Locks:** Redlock algorithm for critical sections

Accuracy vs Performance Trade-offs

Strict Mode: Every request hits Redis (accurate, slower). **Relaxed Mode:** Local counters with periodic sync (faster, ~5% error margin acceptable for most use cases).

Scalability

- Redis Cluster with 10+ nodes handles 100K+ ops/sec

- Partition by user_id or API key for horizontal scaling
- Monitor with metrics: rejection_rate, latency_p99

4. Design a news feed system like Facebook or Twitter. How would you handle feed generation, ranking, and real-time updates at scale?

Feed Generation Approaches

- **Fanout-on-Write (Push):** Pre-compute feeds when content is posted, store in user's feed cache
- **Fanout-on-Read (Pull):** Compute feed on-demand by querying followed users' posts
- **Hybrid:** Push for most users, pull for celebrities with millions of followers

Architecture Components

- **Post Service:** Handles content creation and storage
- **Feed Generation Service:** Computes personalized feeds
- **Ranking Service:** ML-based scoring for post relevance
- **Feed Cache:** Redis sorted sets storing top N posts per user
- **Graph Database:** Neo4j/Neptune for social graph queries

Fanout Strategy

```
def fanout_post(user_id, post):
    followers = get_followers(user_id)
    if len(followers) > 10000:
        return  # Skip fanout for celebrities
    for follower_id in followers:
        feed_key = f"feed:{follower_id}"
        redis.zadd(feed_key, {post.id: post.score})
        redis.zremrangebyrank(feed_key, 0, -1001)
```

Ranking Algorithm

Combine multiple signals: **recency, engagement (likes/comments), relationship strength, content type**. Use machine learning model trained on user interactions.

Score = w1*recency + w2*engagement + w3*affinity + w4*content_quality

Real-time Updates

- **WebSocket Connections:** Push new posts to active users
- **Long Polling:** Fallback for clients without WebSocket support
- **Incremental Loading:** Pagination with cursor-based approach

Scalability Considerations

- Partition feed cache by user_id across Redis cluster
- Asynchronous fanout using Kafka message queue
- CDN for media content (images, videos)
- Database sharding by user_id for social graph data

5. Design a distributed task scheduler that can execute millions of jobs with precise timing. How would you handle failures and ensure exactly-once execution?

Core Components

- **Job Queue:** Kafka/RabbitMQ with priority queues
- **Scheduler Service:** Determines when jobs should execute
- **Worker Pool:** Distributed workers pull and execute jobs
- **Coordination Service:** ZooKeeper/etcd for leader election and distributed locking
- **State Store:** PostgreSQL for job metadata and execution history

Timing Precision

Use **delay queues** with multiple time buckets (immediate, 1min, 5min, 1hour, 1day). Background service moves jobs between buckets as execution time approaches.

```
class JobScheduler:
    def schedule_job(self, job, delay_seconds):
        execute_at = time.time() + delay_seconds
        bucket = self.get_time_bucket(execute_at)
        self.redis.zadd(f"jobs:{bucket}",
                        {job.id: execute_at})
        self.db.save_job(job, execute_at)
```

Exactly-Once Execution

- **Idempotency Keys:** Each job has unique ID, workers check before execution
- **Distributed Locks:** Acquire lock before processing job
- **Two-Phase Commit:** Mark job as 'processing', execute, mark as 'completed'
- **Visibility Timeout:** Job becomes visible again if worker fails

Failure Handling

- **Retry Logic:** Exponential backoff with max retry count
- **Dead Letter Queue:** Failed jobs after max retries
- **Worker Health Checks:** Heartbeat mechanism to detect dead workers
- **Job Timeout:** Kill long-running jobs, return to queue

Scalability

Partition jobs by hash of job_id across multiple queue partitions. Workers subscribe to specific partitions. Use **consistent hashing** for worker assignment to minimize rebalancing.

6. Design a content delivery network (CDN) from scratch. What are the key challenges in cache invalidation, routing, and global distribution?

Architecture Components

- **Edge Servers:** Geographically distributed cache servers (PoPs)
- **Origin Servers:** Source of truth for content
- **DNS/Routing Layer:** Directs users to nearest edge server
- **Cache Management:** Handles invalidation and eviction policies
- **Load Balancers:** Distribute traffic within each PoP

Routing Strategy

Use **GeoDNS** to return IP of closest edge server based on user location. Implement **Anycast** routing where multiple servers share same IP, BGP routes to nearest one.

```
def route_request(user_ip, content_url):
    user_location = geoip_lookup(user_ip)
    edge_servers = get_healthy_edges(user_location)
    selected = consistent_hash(content_url, edge_servers)
    return selected.ip_address
```

Cache Invalidation Strategies

- **TTL-based:** Content expires after time period
- **Purge API:** Origin triggers invalidation of specific URLs
- **Tag-based:** Group related content, purge by tag
- **Versioned URLs:** Change URL when content updates (cache-busting)

Cache Hierarchy

L1 Cache: In-memory (Redis), hot content. **L2 Cache:** SSD storage, warm content. **L3 Cache:** Regional cache tier before origin.

Consistency Challenges

- **Eventual Consistency:** Invalidation propagates gradually across PoPs
- **Stale Content Risk:** Use versioning or short TTLs for critical content
- **Cache Stampede:** Prevent thundering herd with request coalescing

Performance Optimization

- HTTP/2 and HTTP/3 (QUIC) for multiplexing
- Brotli/Gzip compression
- TCP connection pooling and keepalive
- Prefetching and predictive caching

7. Design a distributed file storage system like Google Drive or Dropbox. How would you handle file synchronization, versioning, and conflict resolution?

System Architecture

- **Metadata Service:** Stores file hierarchy, permissions, versions
- **Block Storage:** Chunked file storage (S3, GCS, or custom)
- **Sync Service:** Handles client synchronization
- **Notification Service:** Alerts clients of remote changes
- **Client Application:** Desktop/mobile apps with local cache

File Chunking Strategy

Split files into 4MB blocks using **content-defined chunking** (Rabin fingerprinting). This enables deduplication and efficient delta syncing.

```
def chunk_file(file_path):
    chunks = []
    with open(file_path, 'rb') as f:
        while True:
            chunk = f.read(4 * 1024 * 1024)
            if not chunk: break
            chunk_hash = sha256(chunk).hexdigest()
            chunks.append({'hash': chunk_hash, 'data': chunk})
    return chunks
```

Synchronization Protocol

- **Watch for Changes:** Client monitors local filesystem events
- **Upload Changes:** Send only modified chunks to server
- **Poll for Updates:** Long-polling or WebSocket for remote changes
- **Download Deltas:** Fetch only new/changed chunks

Versioning

Maintain **version history** with metadata linking to chunk sets. Each file version points to list of chunk hashes. Implement **copy-on-write** for efficient storage.

Conflict Resolution

- **Last-Write-Wins:** Simple but may lose data
- **Version Vectors:** Track causality, detect true conflicts
- **Operational Transform:** For real-time collaborative editing
- **User Intervention:** Create conflict copies for manual merge

Scalability Considerations

- Shard metadata by user_id or folder_id
- Use object storage (S3) for unlimited block storage
- CDN for downloading popular files
- Database: PostgreSQL for metadata, Cassandra for activity logs

8. Design a real-time collaborative document editing system like Google Docs. What algorithms and architecture would you use to handle concurrent edits?

Core Algorithms

- **Operational Transformation (OT):** Transforms operations to maintain consistency
- **Conflict-free Replicated Data Types (CRDTs):** Mathematically guarantee convergence
- **Differential Synchronization:** Periodically sync and merge differences

Architecture Components

- **WebSocket Gateway:** Real-time bidirectional communication
- **Document Service:** Manages document state and operations
- **Operation Queue:** Orders and applies concurrent operations
- **Storage Layer:** Persists document snapshots and operation logs
- **Presence Service:** Tracks active collaborators and cursor positions

Operational Transformation Example

```
def transform_operations(op1, op2):
    # op1: insert 'X' at position 5
    # op2: delete at position 3
    if op2.type == 'delete' and op2.pos < op1.pos:
        op1.pos -= 1  # Adjust insert position
    elif op1.type == 'insert' and op1.pos <= op2.pos:
        op2.pos += 1  # Adjust delete position
    return op1, op2
```

CRDT Approach

Use **YJS or Automerge** libraries. Each character has unique ID (site_id, counter). Insertions and deletions are commutative, associative, and idempotent.

Consistency Model

- **Strong Eventual Consistency:** All clients converge to same state
- **Causal Ordering:** Operations applied in causal order using vector clocks
- **Optimistic Updates:** Apply local changes immediately, transform remote ops

Scalability Challenges

- **Document Partitioning:** Split large documents into sections
- **Operation Compaction:** Periodically create snapshots, discard old ops
- **Horizontal Scaling:** Shard documents across servers
- **Caching:** Redis for active document state

Presence and Cursors

Broadcast cursor positions and selections via separate channel. Use **throttling** (100ms intervals) to reduce network overhead.

9. Design a distributed search engine like Elasticsearch. How would you handle indexing, sharding, and query optimization for billions of documents?

Architecture Overview

- **Indexing Service:** Processes and indexes documents
- **Shard Manager:** Distributes index across nodes
- **Query Coordinator:** Routes queries to relevant shards
- **Inverted Index:** Core data structure mapping terms to documents
- **Replication Layer:** Ensures data durability and availability

Inverted Index Structure

```
inverted_index = {
    'distributed': [(doc1, [3, 15]), (doc5, [8])],
    'systems': [(doc1, [4]), (doc3, [2, 10])],
    'design': [(doc1, [1]), (doc2, [0]), (doc5, [1])]
}
# Format: term -> [(doc_id, [positions])]
```

Sharding Strategy

- **Document-based Sharding:** Hash document ID, distribute evenly
- **Term-based Sharding:** Partition by term range (less common)
- **Shard Size:** Keep shards 20-50GB for optimal performance

- **Routing:** Use routing key to direct related docs to same shard

Indexing Pipeline

- **Document Parsing:** Extract text from various formats
- **Analysis:** Tokenization, stemming, stop-word removal
- **Indexing:** Build inverted index with term frequencies
- **Segment Creation:** Write immutable index segments
- **Merge Process:** Periodically merge small segments

Query Optimization

- **Query Rewriting:** Expand synonyms, correct spelling
- **Early Termination:** Stop when enough results found
- **Caching:** Cache frequent queries and filter results
- **Skip Lists:** Fast intersection of posting lists

Ranking Algorithm

TF-IDF with BM25: Score = IDF * (TF * (k+1)) / (TF + k * (1 - b + b * doc_length/avg_length))

Scalability

- Replicate each shard 2-3 times for read scaling
- Use SSD storage for low-latency random reads
- Implement query-time boosting and filtering
- Monitor shard health and rebalance automatically

10. Design a video streaming platform like Netflix or YouTube. How would you handle video encoding, adaptive bitrate streaming, and content delivery at global scale?

System Components

- **Upload Service:** Receives video files from content creators
- **Transcoding Pipeline:** Converts videos to multiple formats/resolutions
- **Storage Layer:** Object storage (S3) for video files
- **CDN:** Distributes content globally
- **Streaming Service:** Handles playback requests
- **Recommendation Engine:** Suggests content to users

Video Encoding Strategy

Use **H.264/AVC** for compatibility, **H.265/HEVC** for efficiency, **VP9** or **AV1** for modern browsers. Encode multiple bitrates: 240p, 360p, 480p, 720p, 1080p, 4K.

```
def transcode_video(input_file):
    profiles = [
        {'resolution': '1920x1080', 'bitrate': '5000k'},
        {'resolution': '1280x720', 'bitrate': '2500k'},
        {'resolution': '854x480', 'bitrate': '1000k'},
        {'resolution': '640x360', 'bitrate': '500k'}
    ]
    for profile in profiles:
        output = f"output_{profile['resolution']}.mp4"
        encode(input_file, output, profile)
```

Adaptive Bitrate Streaming

Use **HLS (HTTP Live Streaming)** or **DASH (Dynamic Adaptive Streaming)**. Video split into 2-10 second segments. Manifest file lists available bitrates. Client switches quality based on bandwidth.

Content Delivery

- **Multi-tier CDN:** Edge locations + regional caches + origin
- **Prefetching:** Pre-load popular content to edge servers
- **Geo-routing:** Direct users to nearest CDN PoP
- **Chunked Transfer:** Stream segments progressively

Storage Optimization

- **Deduplication:** Store identical segments once
- **Tiered Storage:** Hot content on SSD, cold on HDD/Glacier
- **Compression:** Use efficient codecs to reduce storage costs

Scalability Considerations

- Distributed transcoding using worker pools (Kubernetes)
- Message queue (Kafka) for job coordination
- Database sharding by user_id for metadata
- Real-time analytics for monitoring playback quality
- Implement DRM for content protection

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a Python function to flatten a nested list of arbitrary depth without using external libraries.

Solution

Here's an elegant recursive approach to flatten nested lists:

```
def flatten(nested_list):
    result = []
    for item in nested_list:
        if isinstance(item, list):
            result.extend(flatten(item))
        else:
            result.append(item)
    return result
```

Key Points:

- Uses recursion to handle arbitrary nesting depth
- isinstance() checks if item is a list
- extend() merges flattened sublists efficiently
- Time complexity: O(n) where n is total number of elements

2. Implement a function to check if a string is a palindrome, considering only alphanumeric characters and ignoring case.

Optimal Solution

```
def is_palindrome(s):
    cleaned = ''.join(c.lower() for c in s if c.isalnum())
    return cleaned == cleaned[::-1]
```

Alternative Two-Pointer Approach (O(1) space):

```
def is_palindrome(s):
    left, right = 0, len(s) - 1
    while left < right:
        while left < right and not s[left].isalnum(): left += 1
        while left < right and not s[right].isalnum(): right -= 1
        if s[left].lower() != s[right].lower(): return False
        left, right = left + 1, right - 1
    return True
```

3. How would you debug a memory leak in a Python application? What tools and techniques would you use?

Memory Leak Debugging Strategy

Tools:

- **tracemalloc**: Built-in module for tracking memory allocations
- **memory_profiler**: Line-by-line memory usage profiling
- **objgraph**: Visualize object references and find circular references
- **gc module**: Inspect garbage collector and detect uncollected objects

Example using tracemalloc:

```
import tracemalloc
```

```
tracemalloc.start()
# Your code here
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')
for stat in top_stats[:10]:
    print(stat)
```

Common causes: Circular references, global variables holding large objects, unclosed file handles, cache without size limits

4. Explain exception handling best practices. Write code demonstrating proper use of try-except-else-finally blocks.

Exception Handling Best Practices

```
def process_file(filename):
    file = None
    try:
        file = open(filename, 'r')
        data = file.read()
    except FileNotFoundError:
        print(f"File {filename} not found")
        return None
    except PermissionError:
        print(f"Permission denied for {filename}")
        return None
    else:
        return data.strip()
    finally:
        if file:
            file.close()
```

Key Principles:

- Catch specific exceptions, not bare except
- Use **else** for code that runs only if no exception occurred
- Use **finally** for cleanup that must always execute
- Consider context managers (with statement) for resource management

5. What is monkey patching? Provide an example and discuss when it's appropriate to use.

Monkey Patching Explained

Definition: Dynamically modifying a class or module at runtime to change or extend behavior.

Example:

```
import datetime
original_now = datetime.datetime.now

def mock_now():
    return datetime.datetime(2024, 1, 1, 12, 0, 0)

datetime.datetime.now = mock_now
print(datetime.datetime.now())
```

Appropriate Use Cases:

- Testing: Mock external dependencies or time-dependent code
- Hot-fixing third-party libraries temporarily
- Adding functionality to frozen classes

Risks:

- Makes code harder to understand and maintain
- Can cause conflicts with other patches
- Better alternatives: dependency injection, subclassing, decorators

6. Write a function to reverse a string in-place. Discuss why this is challenging in Python.

String Reversal Challenge

The Problem: Strings in Python are **immutable**, so true in-place reversal is impossible.

Standard Approach:

```
def reverse_string(s):
    return s[::-1]
```

Simulating In-Place with List:

```
def reverse_in_place(s):
    chars = list(s)
    left, right = 0, len(chars) - 1
    while left < right:
        chars[left], chars[right] = chars[right], chars[left]
        left, right = left + 1, right - 1
    return ''.join(chars)
```

Why Python Strings Are Immutable:

- Enables string interning for memory efficiency
- Allows strings to be hashable (usable as dict keys)
- Thread-safe by design

7. Demonstrate how to use Python's logging module for debugging in production environments.

Production-Grade Logging

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[logging.FileHandler('app.log'), logging.StreamHandler()]
)

logger = logging.getLogger(__name__)
logger.info('Application started')
logger.error('Error occurred', exc_info=True)
```

Best Practices:

- Use appropriate log levels: DEBUG, INFO, WARNING, ERROR, CRITICAL
- Include contextual information (user ID, request ID, timestamps)
- Use **exc_info=True** to capture stack traces
- Implement log rotation to manage file sizes
- Avoid logging sensitive data (passwords, tokens)
- Use structured logging (JSON) for easier parsing

8. Write a decorator that measures and logs the execution time of a function. Handle both sync and async functions.

Execution Time Decorator

```
import time
import functools

def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        end = time.perf_counter()
        print(f"{func.__name__} took {end - start:.4f}s")
        return result
    return wrapper
```

For Async Functions:

```
import asyncio

def async_timer(func):
    @functools.wraps(func)
    async def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = await func(*args, **kwargs)
        end = time.perf_counter()
        print(f"{func.__name__} took {end - start:.4f}s")
        return result
    return wrapper
```

9. Explain how to profile CPU and memory usage in Python. Provide code examples using cProfile and memory_profiler.

Performance Profiling

CPU Profiling with cProfile:

```
import cProfile
import pstats

def my_function():
    return sum(i**2 for i in range(10000))

profiler = cProfile.Profile()
profiler.enable()
my_function()
profiler.disable()
stats = pstats.Stats(profiler)
stats.sort_stats('cumulative')
stats.print_stats(10)
```

Memory Profiling:

```
from memory_profiler import profile

@profile
def memory_intensive():
    large_list = [i for i in range(1000000)]
    return sum(large_list)
```

Key Metrics:

- **ncalls**: Number of calls
- **tottime**: Total time excluding subcalls
- **cumtime**: Total time including subcalls

10. Write a function to detect and handle circular imports in Python. What strategies prevent circular import issues?

Circular Import Handling

Detection Example:

```
import sys

def check_circular_imports():
    for name, module in sys.modules.items():
        if hasattr(module, '__file__') and module.__file__:
            print(f"{name}: {module.__file__}")
```

Prevention Strategies:

- **Restructure code**: Move shared dependencies to a separate module

- **Import at function level**: Delay imports until needed
- **Use import-time guards**: Conditional imports
- **Dependency injection**: Pass dependencies as parameters

Example Fix:

```
# Instead of module-level import
def process_data():
    from .utils import helper
    return helper.transform(data)
```

Best Practice: Design modules with clear, unidirectional dependencies

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you had to implement a generative AI solution under tight deadlines. How did you manage it?

Situation: Our e-commerce client needed a product description generator for 50,000 SKUs before the holiday season launch in 6 weeks.

Task: I was responsible for designing and deploying an LLM-based solution that could generate accurate, SEO-optimized descriptions while maintaining brand voice consistency.

Action: I chose to fine-tune GPT-3.5 instead of building from scratch, created a prompt template system with few-shot examples, implemented batch processing with rate limiting, and set up a human-in-the-loop review workflow for quality assurance. I also cached common attribute combinations to reduce API costs.

Result: We delivered 2 weeks early, generated 48,000 descriptions with 92% approval rate on first pass, reduced costs by 40% through caching, and the client reported a 23% increase in conversion rates.

2. Describe a situation where a generative AI model you deployed produced unexpected or biased outputs. How did you handle it?

Situation: After deploying a resume screening assistant using an LLM, HR reported that the system was consistently ranking candidates with certain university names higher, regardless of actual qualifications.

Task: I needed to identify the bias source, mitigate it, and restore trust with stakeholders while maintaining the system's utility.

Action: I conducted a thorough audit of training prompts and discovered implicit bias in our few-shot examples. I redesigned the prompt engineering approach to focus on skills and experience metrics, implemented blind screening by removing institution names during initial evaluation, added bias detection tests to our CI/CD pipeline, and created a diverse evaluation dataset with demographic parity metrics.

Result: Bias metrics improved by 78%, the system achieved demographic parity within 5% across protected groups, and HR adoption increased from 45% to 89%. We also open-sourced our bias testing framework.

3. Can you share an example of when you had to optimize the cost of running generative AI models in production?

Situation: Our customer support chatbot using GPT-4 was costing $15,000/month with 500K queries, making the solution financially unsustainable.

Task: Reduce costs by at least 60% while maintaining response quality and user satisfaction scores above 4.2/5.

Action: I implemented a tiered model approach where GPT-3.5-turbo handled 70% of simple queries, cached frequent question-answer pairs in Redis, reduced token usage through prompt compression techniques, implemented semantic similarity search to retrieve cached responses, and reserved GPT-4 only for complex queries identified by a lightweight classifier. I also negotiated batch processing discounts with OpenAI.

Result: Monthly costs dropped to $4,800 (68% reduction), average response time improved by 35% due to caching, user satisfaction remained at 4.3/5, and we processed 40% more queries with the same budget.

**4. Tell me about a time when you had to explain the limitations of generative AI to non-

technical stakeholders who had unrealistic expectations.

Situation: The executive team expected our generative AI system to completely replace the content writing team and produce publication-ready articles with zero human oversight.

Task: I needed to set realistic expectations about AI capabilities while demonstrating value and maintaining project support.

Action: I organized a live demonstration showing both successes and failure cases, including hallucinations and factual errors. I created a capability matrix comparing AI vs. human performance across dimensions like creativity, fact-checking, and brand consistency. I proposed a hybrid model where AI handled first drafts and research, while humans focused on editing, fact-verification, and strategic direction. I also provided ROI projections for the hybrid approach.

Result: Leadership approved the hybrid model, we retained the writing team in elevated roles as AI editors, content production increased 3x, and quality scores improved by 15% due to writers focusing on refinement rather than initial drafting.

5. Describe a challenging technical problem you faced while fine-tuning or customizing a large language model for your organization's specific use case.

Situation: We needed to fine-tune an LLM for medical documentation, but the model kept generating clinically accurate text that violated our institution's specific documentation standards and compliance requirements.

Task: Achieve both clinical accuracy and 100% compliance with institutional guidelines while working with limited labeled data (only 2,000 examples).

Action: I implemented a multi-stage approach: first, I used retrieval-augmented generation (RAG) to inject institutional guidelines into context; second, I created synthetic training data by having domain experts annotate AI-generated outputs; third, I fine-tuned Llama-2-70B with LoRA for parameter efficiency; and fourth, I implemented a rule-based post-processing layer for hard compliance requirements like required sections and formatting.

Result: Compliance rate increased from 67% to 98%, clinical accuracy remained at 94%, documentation time reduced by 45%, and the model was successfully deployed across 3 departments serving 200+ physicians.

6. Tell me about a time when you had to balance innovation with responsible AI practices in a generative AI project.

Situation: We were developing an AI-powered code generation tool for our development team, and there were concerns about generating code with security vulnerabilities or licensing issues from training data.

Task: Deploy an innovative productivity tool while ensuring code security, license compliance, and maintaining developer trust.

Action: I established a responsible AI framework that included: static analysis scanning of all generated code before insertion, a whitelist of approved open-source licenses, prompt engineering to explicitly request secure coding patterns, integration with our existing security scanning pipeline, and comprehensive logging for audit trails. I also created developer guidelines and conducted training sessions on AI-assisted development best practices.

Result: The tool was adopted by 85% of developers within 3 months, we detected and prevented 34 potential security issues in the first quarter, zero licensing violations occurred, and developer productivity increased by 28% as measured by story points completed.

7. Share an experience where you had to troubleshoot and debug unexpected behavior in a generative AI system that was difficult to reproduce.

Situation: Our production text summarization service occasionally produced completely irrelevant summaries for about 2% of documents, but the issue was intermittent and couldn't be reproduced in testing environments.

Task: Identify root cause, fix the issue, and prevent similar problems while minimizing production downtime.

Action: I implemented comprehensive logging to capture full request context including input text, temperature settings, and generated outputs. I discovered that edge cases with unusual character encodings and extremely long paragraphs were causing context window truncation in unpredictable ways. I added input validation and preprocessing, implemented graceful degradation with chunk-based processing for long documents, added monitoring alerts for output quality metrics, and created a regression test suite with the problematic edge cases.

Result: Error rate dropped from 2% to 0.1%, we identified 12 additional edge cases proactively, average processing reliability improved to 99.9%, and the monitoring system now catches anomalies within 5 minutes.

8. Describe a situation where you had to collaborate with cross-functional teams (data scientists, product managers, legal) on a generative AI initiative.

Situation: We were launching an AI-powered customer email response system that required coordination between engineering, data science, customer service, legal, and product teams, each with different priorities and concerns.

Task: Lead the technical implementation while ensuring all stakeholder requirements were met and maintaining project timeline for a 4-month delivery.

Action: I established a weekly sync with all stakeholders and created role-specific documentation. I worked with data scientists to define model evaluation metrics, collaborated with legal to implement content filtering and data retention policies, partnered with product to define the user experience and escalation workflows, and conducted workshops with customer service to gather domain expertise for prompt engineering. I also created a shared dashboard showing progress against each team's success criteria.

Result: Launched on schedule with 100% stakeholder approval, the system handled 60% of tier-1 support emails autonomously, customer satisfaction remained at 4.4/5, legal compliance was maintained at 100%, and the cross-functional collaboration model became a template for future AI projects.

9. Tell me about a time when you had to make a difficult architectural decision for a generative AI system, such as build vs. buy or choosing between different model providers.

Situation: We needed to decide between using OpenAI's API, deploying open-source models on our infrastructure, or building a custom model for our document analysis platform handling sensitive financial data.

Task: Make an architecture decision that balanced cost, performance, data privacy, and long-term maintainability while meeting our 3-month MVP deadline.

Action: I conducted a comprehensive analysis creating a decision matrix evaluating total cost of ownership, data residency requirements, performance benchmarks, and team capabilities. I built proof-of-concepts for each approach: OpenAI API with data encryption, self-hosted Llama-2 on AWS, and a smaller custom model. I presented findings showing OpenAI violated data residency requirements, custom models would take 8 months, but self-hosted open-source models met all criteria with acceptable performance.

Result: We deployed Llama-2-70B on dedicated AWS instances with 99.9% uptime, maintained full data control, achieved 89% of GPT-4 performance on our specific use case, total monthly cost was $3,200 vs. projected $8,000 with APIs, and we retained flexibility to fine-tune and customize.

10. Describe an instance where you had to rapidly learn and implement a new generative AI technology or framework to solve a business problem.

Situation: Our company needed to implement Retrieval-Augmented Generation (RAG) for an internal knowledge base system within 5 weeks, but I had no prior hands-on experience with vector databases or embedding models.

Task: Learn RAG architecture, select appropriate technologies, and deliver a working prototype that could search across 100,000 internal documents with high accuracy.

Action: I dedicated the first week to intensive learning through research papers, documentation, and building small experiments. I evaluated vector databases (Pinecone, Weaviate, Chroma) and chose Weaviate for its hybrid search capabilities. I implemented a pipeline using sentence-transformers for embeddings, LangChain for orchestration, and GPT-3.5 for generation. I iterated rapidly with user feedback from 10 beta testers and optimized chunk sizes and retrieval parameters based on evaluation metrics.

Result: Delivered the prototype in 4.5 weeks, achieved 87% accuracy on test queries (exceeding the 80% target), reduced information retrieval time from 15 minutes to 30 seconds, and the system was adopted by 400+ employees within 2 months. I also created internal documentation and training materials for the team.