AI Senior Engineer Interview Questions & Answers

3+ Years Experience - Comprehensive Study Guide

Table of Contents

1. Machine Learning Fundamentals

Q1: What is the difference between Supervised, Unsupervised, and Reinforcement Learning?

Answer:

Type	Description	Examples
Supervised Learning	Model learns from labeled data (input-output pairs)	Classification, Regression
Unsupervised Learning	Model finds patterns in unlabeled data	Clustering, Dimensionality Reduction
Reinforcement Learning	Agent learns through rewards/penalties from environment	Game AI, Robotics

Example:

- Supervised: Predicting house prices (regression) with features like size, location
- Unsupervised: Customer segmentation without predefined groups
- Reinforcement: Training a robot to walk through trial and error

Q2: Explain Bias-Variance Tradeoff

Answer:

Bias: Error from oversimplified assumptions (underfitting)

- High bias = model misses relevant relations
- Example: Linear model for non-linear data

Variance: Error from sensitivity to training data fluctuations (overfitting)

- High variance = model captures noise as patterns
- Example: Deep tree memorizing training data

Tradeoff:

```
Total Error = Bias² + Variance + Irreducible Error
```

Solutions:

- **High Bias:** Increase model complexity, add features, reduce regularization
- **High Variance:** More training data, regularization (L1/L2), dropout, cross-validation

---

Q3: What are Precision, Recall, F1-Score? When to use each?

Answer:

```
Precision = TP / (TP + FP)  → "Of all positive predictions, how many correct?"
Recall    = TP / (TP + FN)  → "Of all actual positives, how many found?"
F1-Score  = 2 × (Precision × Recall) / (Precision + Recall)
```

When to use:

Metric	Use When
Precision	False positives are costly (spam detection, fraud alerts)
Recall	False negatives are costly (cancer detection, security threats)
F1-Score	Need balance between precision and recall
AUC-ROC	Comparing models across thresholds

Q4: Explain Cross-Validation and its types

Answer:

Cross-validation evaluates model performance on unseen data by splitting data into folds.

Types:

1. **K-Fold CV:** Split into K parts, train on K-1, test on 1, rotate
2. **Stratified K-Fold:** Maintains class distribution in each fold
3. **Leave-One-Out (LOOCV):** K = number of samples (expensive)
4. **Time Series CV:** Respects temporal order (no future data leakage)

```
from sklearn.model_selection import cross_val_score, StratifiedKFold

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(model, X, y, cv=cv, scoring='f1')
print(f"Mean F1: {scores.mean():.3f} ± {scores.std():.3f}")
```

Q5: What is Regularization? Explain L1 vs L2

Answer:

Regularization prevents overfitting by adding penalty to loss function.

Type	Formula	Effect	Use Case
L1 (Lasso)	$\lambda\sum\|w\|$	Sparse weights (feature selection)	Many irrelevant features
L2 (Ridge)	$\lambda\sum w^2$	Small weights (no zeros)	All features relevant
Elastic Net	$\alpha\times L1 + (1-\alpha)\times L2$	Combination	Best of both

Example:

```
from sklearn.linear_model import Lasso, Ridge, ElasticNet

lasso = Lasso(alpha=0.1)  # L1 - some coefficients become 0
ridge = Ridge(alpha=0.1)  # L2 - coefficients shrink but non-zero
elastic = ElasticNet(alpha=0.1, l1_ratio=0.5)  # Mix
```

Q6: How do you handle imbalanced datasets?

Answer:

Techniques:

1. **Resampling:**
   - Oversampling minority (SMOTE, ADASYN)
   - Undersampling majority (Random, Tomek links)
2. **Algorithm-level:**
   - Class weights (`class_weight='balanced'`)

- Cost-sensitive learning

3. **Ensemble methods:**
   - BalancedRandomForest
   - EasyEnsemble
4. **Evaluation:**
   - Use F1, AUC-ROC instead of accuracy
   - Precision-Recall curves

```
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import RandomForestClassifier

smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)

# Or use class weights
model = RandomForestClassifier(class_weight='balanced')
```

---

2. Deep Learning & Neural Networks

Q7: Explain Backpropagation

Answer:

Backpropagation is the algorithm to compute gradients for updating neural network weights.

Steps:

1. **Forward Pass:** Compute predictions layer by layer
2. **Loss Calculation:** Compare predictions with targets
3. **Backward Pass:** Compute gradients using chain rule
4. **Weight Update:** Adjust weights using optimizer (SGD, Adam)

```
∂Loss/∂w = ∂Loss/∂output × ∂output/∂activation × ∂activation/∂w
```

Key concepts:

- Chain rule for gradient computation
- Computational graph tracks operations
- Gradients flow backwards through layers

Q8: What is Vanishing/Exploding Gradient Problem? How to solve?

Answer:

Vanishing Gradients: Gradients become very small in deep networks (sigmoid/tanh) **Exploding Gradients:** Gradients become very large (unstable training)

Solutions:

Problem	Solutions
Vanishing	ReLU activation, Batch Normalization, Residual connections, LSTM/GRU
Exploding	Gradient clipping, Weight initialization (Xavier/He), Batch Normalization

```
# Gradient Clipping
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

# Proper initialization
nn.init.kaiming_normal_(layer.weight, mode='fan_out', nonlinearity='relu')

# Batch Normalization
self.bn = nn.BatchNorm1d(hidden_size)
```

Q9: Compare CNN vs RNN vs Transformer architectures

Answer:

Architecture	Best For	Key Features	Limitations
CNN	Images, spatial data	Local patterns, translation invariance, parameter sharing	Fixed receptive field
RNN	Sequential data	Memory of past inputs	Vanishing gradients, slow (sequential)
Transformer	NLP, long sequences	Self-attention, parallel processing	$O(n^2)$ attention, memory intensive

When to use:

- **CNN:** Image classification, object detection, 1D signals
- **RNN/LSTM:** Time series, short sequences, streaming data
- **Transformer:** NLP tasks, long-range dependencies, when parallelization needed

---

Q10: Explain Attention Mechanism and Self-Attention

Answer:

Attention: Mechanism to focus on relevant parts of input when producing output.

Self-Attention (Transformer):

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

Where:

- **Q (Query):** What we're looking for
- **K (Key):** What we match against
- **V (Value):** What we retrieve
- **√d_k:** Scaling factor for stability

Multi-Head Attention: Multiple attention heads capture different relationships

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        self.num_heads = num_heads
        self.d_k = d_model // num_heads
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)

    def forward(self, Q, K, V):
        # Split into heads, compute attention, concatenate
        ...
```

Q11: What are different Optimizers? Compare SGD, Adam, AdamW

Answer:

Optimizer	Description	Pros	Cons
SGD	Basic gradient descent	Simple, good generalization	Slow, sensitive to LR
SGD+Momentum	Accelerates in consistent direction	Faster convergence	Still needs LR tuning
Adam	Adaptive learning rates + momentum	Fast, works well out-of-box	May not generalize well
AdamW	Adam with decoupled weight decay	Better generalization	Slightly more computation

Best Practices:

- **Computer Vision:** SGD with momentum often better
- **NLP/Transformers:** AdamW preferred
- **Quick experiments:** Adam for fast convergence

```
# AdamW with learning rate scheduler
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4, weight_decay=0.01)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs)
```

---

3. Natural Language Processing (NLP)

Q12: Explain Word Embeddings (Word2Vec, GloVe, FastText)

Answer:

Word embeddings represent words as dense vectors capturing semantic meaning.

Method	Training	Strengths
Word2Vec	Skip-gram or CBOW	Fast, captures analogies
GloVe	Global co-occurrence matrix	Captures global statistics
FastText	Character n-grams	Handles OOV words, morphology

Key Properties:

- Similar words have similar vectors
- Analogies: king - man + woman ≈ queen
- Typically 100-300 dimensions

```
from gensim.models import Word2Vec

model = Word2Vec(sentences, vector_size=100, window=5, min_count=1)
similar = model.wv.most_similar('python')
```

Q13: What is BERT? How does it work?

Answer:

BERT (Bidirectional Encoder Representations from Transformers)

Architecture:

- Transformer encoder (12 or 24 layers)
- Bidirectional context (sees both left and right)
- Pre-trained on large corpus, fine-tuned for tasks

Pre-training Tasks:

1. **Masked Language Modeling (MLM):** Predict masked tokens

2. **Next Sentence Prediction (NSP):** Predict if sentences are consecutive

Usage:

```
from transformers import BertTokenizer, BertModel

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

inputs = tokenizer("Hello world", return_tensors="pt")
outputs = model(**inputs)
# outputs.last_hidden_state: [batch, seq_len, 768]
# outputs.pooler_output: [batch, 768] for classification
```

Variants: RoBERTa, ALBERT, DistilBERT, DeBERTa

Q14: Explain Tokenization strategies (BPE, WordPiece, SentencePiece)

Answer:

Method	Description	Used By
BPE	Iteratively merge frequent byte pairs	GPT, RoBERTa
WordPiece	Similar to BPE, likelihood-based	BERT
SentencePiece	Language-agnostic, raw text	T5, mBART
Unigram	Probabilistic, keeps most likely subwords	XLNet

Example (BPE):

```
"unhappiness" → ["un", "happiness"] → ["un", "happ", "iness"]
```

Key Considerations:

- Vocabulary size tradeoff (smaller = more subwords per word)
- Unknown token handling
- Special tokens ([CLS], [SEP], [PAD])

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")
tokens = tokenizer.tokenize("unhappiness")  # ['un', 'happ', 'iness']
```

---

4. Large Language Models (LLMs)

Q15: Explain the GPT Architecture

Answer:

GPT (Generative Pre-trained Transformer)

Architecture:

- Decoder-only Transformer
- Unidirectional (causal) attention - can only see past tokens
- Autoregressive generation

Key Components:

```
Input → Token Embedding + Position Embedding →
N × (Masked Self-Attention → Feed Forward) →
Output Logits → Softmax → Next Token Prediction
```

Differences from BERT:

BERT	GPT
Encoder-only	Decoder-only
Bidirectional	Unidirectional
MLM + NSP	Next token prediction
Classification, QA	Text generation

Q16: What is Prompt Engineering? Best practices?

Answer:

Prompt Engineering is the art of crafting inputs to get desired outputs from LLMs.

Techniques:

1. **Zero-shot:** Direct question without examples
2. **Few-shot:** Include examples in prompt
3. **Chain-of-Thought:** Ask model to show reasoning
4. **Role-playing:** Assign persona to model

Best Practices:

```
# Zero-shot
prompt = "Classify the sentiment: 'This movie was great!' → "
```

```
# Few-shot
prompt = """
Classify sentiment:
Text: "I love this!" → Positive
Text: "Terrible experience" → Negative
Text: "This movie was great!" → """

# Chain-of-Thought
prompt = """
Solve step by step:
Q: If John has 5 apples and gives 2 to Mary, how many does he have?
A: Let's think step by step:
1. John starts with 5 apples
2. He gives 2 to Mary
3. 5 - 2 = 3
John has 3 apples.

Q: If a train travels 60 mph for 2.5 hours, how far does it go?
A: Let's think step by step:"""
```

Q17: Explain Fine-tuning vs RAG vs Agents

Answer:

Approach	Description	Use Case	Pros	Cons
Fine-tuning	Train model on domain data	Specific style/domain	Customized behavior	Expensive, outdated knowledge
RAG	Retrieve context + generate	Dynamic knowledge	Up-to-date, verifiable	Retrieval quality dependent
Agents	LLM + tools + reasoning	Complex tasks	Flexible, multi-step	Unpredictable, slower

When to use:

- **Fine-tuning:** Need specific tone, format, or domain expertise
- **RAG:** Need current information, cited sources, enterprise data
- **Agents:** Need to take actions, use tools, multi-step reasoning

---

Q18: What are Hallucinations in LLMs? How to mitigate?

Answer:

Hallucinations: LLMs generating plausible but incorrect/fabricated information.

Types:

1. **Factual errors:** Wrong facts confidently stated
2. **Fabrication:** Made-up references, quotes, data
3. **Inconsistency:** Contradicting previous statements

Mitigation Strategies:

Strategy	Description
RAG	Ground responses in retrieved documents
Temperature	Lower temperature (0.0-0.3) for factual tasks
Explicit instructions	"Only answer based on provided context"
Self-consistency	Generate multiple answers, check agreement
Fact verification	Post-process with fact-checking
Citations	Force model to cite sources

```
system_prompt = """
You are a helpful assistant. IMPORTANT RULES:
1. Only answer based on the provided context
2. If the answer is not in the context, say "I don't have information about that"
3. Always cite the source document
4. Never make up information
"""
```

---

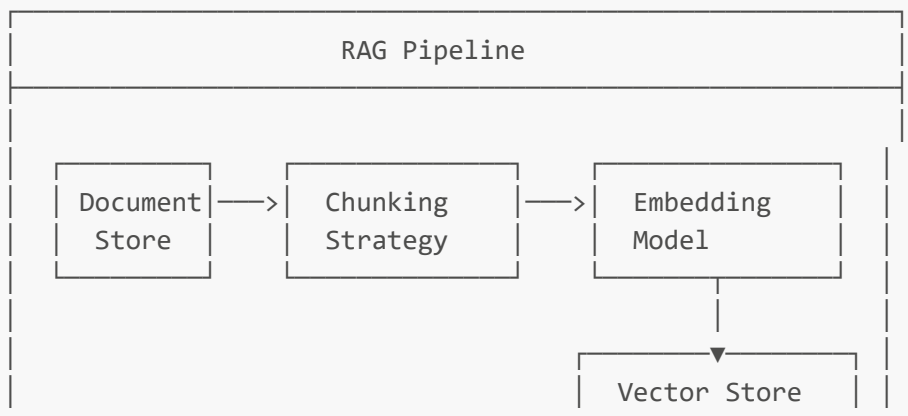
5. RAG (Retrieval Augmented Generation)

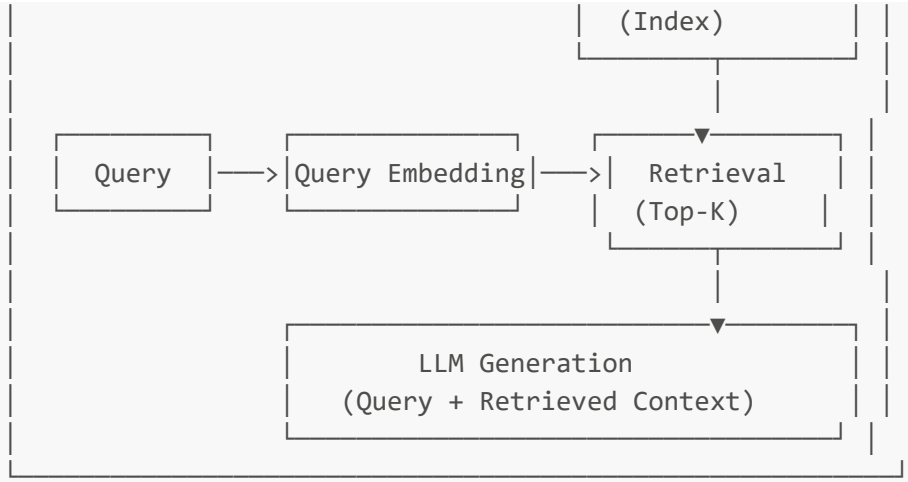
Q19: Explain RAG Architecture and Components

Answer:

RAG combines retrieval with generation for knowledge-grounded responses.

Components:





Key Decisions:

1. **Chunking:** Size (512-1024 tokens), overlap (10-20%)
2. **Embedding:** OpenAI ada-002, Cohere, sentence-transformers
3. **Vector Store:** Pinecone, Weaviate, Azure AI Search, FAISS
4. **Retrieval:** Semantic search, hybrid (keyword + vector)
5. **Generation:** GPT-4, Claude, Llama

Q20: What are different Chunking Strategies?

Answer:

Strategy	Description	Use Case
Fixed-size	Split by character/token count	Simple, consistent
Sentence	Split by sentence boundaries	Preserves meaning
Paragraph	Split by paragraphs	Structured docs
Semantic	Split by topic/meaning change	Best quality, complex
Recursive	Hierarchical splitting	Long documents

Best Practices:

- Chunk size: 512-1024 tokens typical
- Overlap: 10-20% to preserve context
- Metadata: Keep source, page, section info

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
    separators=["\n\n", "\n", ".", " "]
```

```
)
chunks = splitter.split_text(document)
```

Q21: How do you evaluate RAG systems?

Answer:

Metrics:

Metric	Measures	How
Retrieval Precision	Relevance of retrieved docs	% relevant in top-K
Retrieval Recall	Coverage of relevant docs	% of relevant docs retrieved
MRR	Ranking quality	1/rank of first relevant doc
NDCG	Graded relevance ranking	Normalized discounted cumulative gain
Answer Faithfulness	Grounded in context	LLM judge / human eval
Answer Relevance	Answers the question	LLM judge / human eval

Evaluation Frameworks:

- **RAGAS:** Automated RAG evaluation
- **LangSmith:** Tracing and evaluation
- **Human evaluation:** Gold standard

```
from ragas import evaluate
from ragas.metrics import faithfulness, answer_relevancy, context_precision

result = evaluate(
    dataset,
    metrics=[faithfulness, answer_relevancy, context_precision]
)
```

Q22: What is Hybrid Search? When to use it?

Answer:

Hybrid Search combines keyword (BM25) and semantic (vector) search.

```
# Hybrid scoring
final_score = α × vector_score + (1-α) × keyword_score
```

When to use:

Use Case	Best Approach
Exact matches needed (IDs, names)	Keyword
Semantic understanding	Vector
General enterprise search	Hybrid
Code search	Hybrid

Implementation (Azure AI Search):

```
results = search_client.search(
    search_text=query,  # Keyword search
    vector_queries=[VectorizedQuery(
        vector=query_embedding,
        k_nearest_neighbors=10,
        fields="content_vector"
    )],
    query_type="semantic",  # Add semantic ranking
    semantic_configuration_name="my-semantic-config"
)
```

6. Vector Databases & Embeddings

Q23: Compare Vector Databases (Pinecone, Weaviate, Azure AI Search, FAISS)

Answer:

Database	Type	Strengths	Best For
Pinecone	Managed cloud	Easy setup, scalable	Production, quick start
Weaviate	Open source	Hybrid search, GraphQL	Self-hosted, flexibility
Azure AI Search	Azure native	Enterprise, hybrid, security	Azure ecosystem
FAISS	Library	Fast, local, free	Prototyping, small scale
Chroma	Open source	Simple API, local	Development, small projects
Qdrant	Open source	Filtering, payloads	Complex filtering needs

Selection Criteria:

- Scale: Cloud for large scale
- Cost: Open source for budget
- Features: Hybrid search, filtering, metadata
- Integration: Match your cloud provider

Q24: Explain Similarity Metrics (Cosine, Euclidean, Dot Product)

Answer:

Metric	Formula	Range	Use Case
Cosine	A·B / (||A|| × ||B||)	[-1, 1]	Normalized vectors, text
Euclidean	√(Σ(A-B)²)	[0, ∞)	Spatial data
Dot Product	Σ(A×B)	(-∞, ∞)	When magnitude matters

Best Practices:

- **Cosine:** Most common for embeddings, direction matters
- **Euclidean:** When absolute distance matters
- **Dot Product:** OpenAI embeddings (already normalized)

```
import numpy as np

def cosine_similarity(a, b):
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))

def euclidean_distance(a, b):
    return np.linalg.norm(a - b)

def dot_product(a, b):
    return np.dot(a, b)
```

Q25: How do you handle embedding model updates in production?

Answer:

Challenge: New embedding model = all vectors incompatible

Strategies:

1. **Blue-Green Deployment:**
    - Create new index with new embeddings
    - Switch traffic when ready
    - Keep old index as fallback
2. **Gradual Migration:**
    - Re-embed documents in batches
    - Track migration progress
    - Use versioned index names

3. **Dual Index:**
    - Query both indexes during transition
    - Merge results
    - Phase out old index

```python
# Versioned index approach
INDEX_V1 = "documents-ada-002-v1"
INDEX_V2 = "documents-text-embedding-3-v2"

async def migrate_embeddings():
    documents = get_all_documents()
    for batch in chunks(documents, 100):
        new_embeddings = embedding_model_v2.embed(batch)
        index_v2.upsert(batch, new_embeddings)

# Feature flag for switching
if config.USE_NEW_EMBEDDINGS:
    results = search_v2(query)
else:
    results = search_v1(query)
```

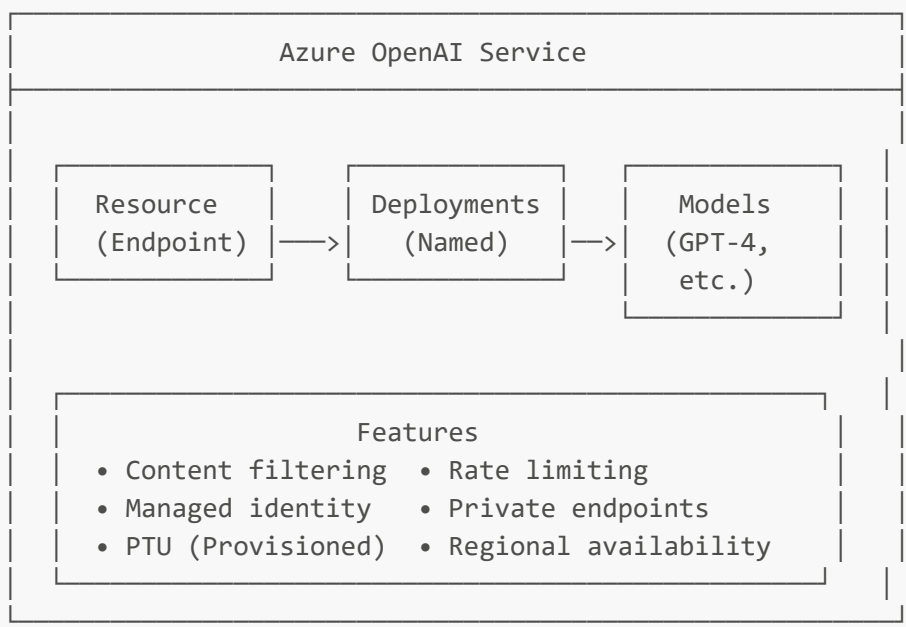
---

7. Azure AI Services

Q26: Explain Azure OpenAI Service architecture

Answer:

Components:



Usage:

```
from openai import AzureOpenAI

client = AzureOpenAI(
    azure_endpoint=os.getenv("AZURE_OPENAI_ENDPOINT"),
    api_key=os.getenv("AZURE_OPENAI_API_KEY"),
    api_version="2024-02-15-preview"
)

response = client.chat.completions.create(
    model="gpt-4-deployment",  # Deployment name, not model name
    messages=[{"role": "user", "content": "Hello!"}],
    temperature=0.7
)
```

Q27: Explain Azure AI Search for RAG

Answer:

Key Features for RAG:

1. **Vector Search:** Native vector indexing
2. **Hybrid Search:** Combine keyword + vector
3. **Semantic Ranking:** Re-rank with deep learning
4. **Security:** Row-level security, AAD integration

Index Schema for RAG:

```
from azure.search.documents.indexes.models import (
    SearchIndex, SearchField, VectorSearch,
    HnswAlgorithmConfiguration, SearchFieldDataType
)

index = SearchIndex(
    name="documents",
    fields=[
        SearchField(name="id", type=SearchFieldDataType.String, key=True),
        SearchField(name="content", type=SearchFieldDataType.String,
searchable=True),
        SearchField(name="content_vector",
type=SearchFieldDataType.Collection(SearchFieldDataType.Single),
                    vector_search_dimensions=1536,
vector_search_profile_name="myHnswProfile"),
        SearchField(name="owner_id", type=SearchFieldDataType.String,
filterable=True),
        SearchField(name="allowed_users",
type=SearchFieldDataType.Collection(SearchFieldDataType.String), filterable=True),
```

```
    ],
    vector_search=VectorSearch(
        algorithms=[HnswAlgorithmConfiguration(name="myHnsw")],
        profiles=[VectorSearchProfile(name="myHnswProfile",
algorithm_configuration_name="myHnsw")]
    )
)
```

Q28: How do you implement security trimming in Azure AI Search?

Answer:

Security Trimming: Filtering search results based on user permissions.

Implementation:

1. **Store permissions in index:**

```
{
    "id": "doc123",
    "content": "...",
    "owner_id": "user-abc",
    "allowed_users": ["user-abc", "user-xyz"],
    "allowed_groups": ["group-123", "group-456"]
}
```

2. **Build filter at query time:**

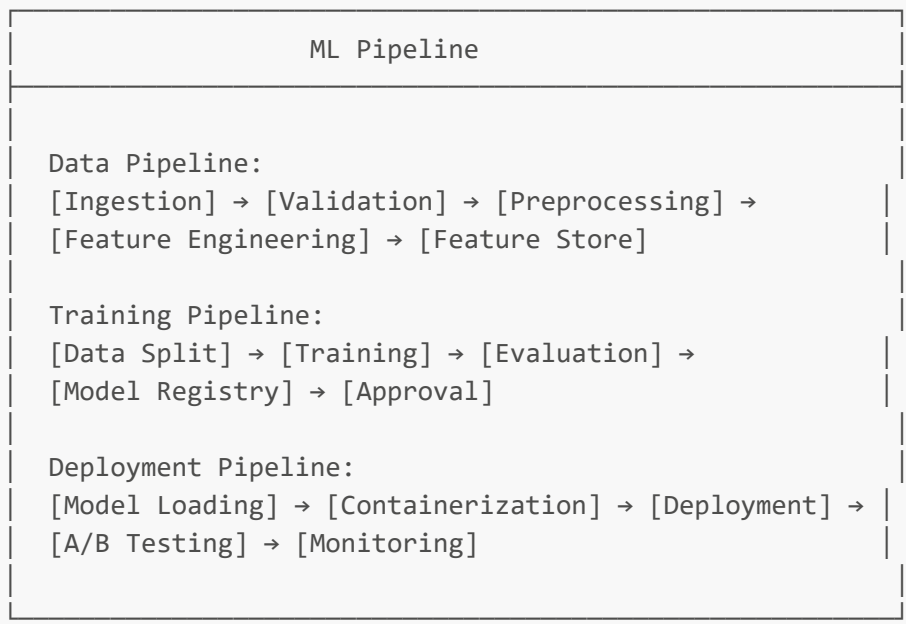
```
def build_security_filter(user_id: str, group_ids: List[str]) -> str:
    filter_parts = [f"owner_id eq '{user_id}'"]
    filter_parts.append(f"allowed_users/any(u: u eq '{user_id}')")

    for group_id in group_ids:
        filter_parts.append(f"allowed_groups/any(g: g eq '{group_id}')")

    return " or ".join(filter_parts)

# Apply filter
results = search_client.search(
    search_text=query,
    filter=build_security_filter(user_id, user_groups),
    vector_queries=[vector_query]
)
```

---

8. MLOps & Production AI

Q29: Explain ML Pipeline components

Answer:



Tools:

- **Orchestration:** Airflow, Kubeflow, Azure ML Pipelines
- **Experiment Tracking:** MLflow, Weights & Biases, Neptune
- **Model Registry:** MLflow, Azure ML, SageMaker
- **Feature Store:** Feast, Tecton, Azure ML Feature Store

Q30: How do you monitor LLM applications in production?

Answer:

Key Metrics:

Category	Metrics
Latency	P50, P95, P99 response time
Throughput	Requests/second, tokens/second
Errors	Error rate, timeout rate
Cost	Tokens used, cost per query
Quality	User feedback, hallucination rate

Monitoring Stack:

```python
# Tracing with LangSmith/Langfuse
from langsmith import traceable

@traceable(name="chat_query")
async def process_query(query: str, user_id: str):
    # Track retrieval
    with trace_span("retrieval"):
        chunks = retrieve(query)

    # Track generation
    with trace_span("generation"):
        response = generate(query, chunks)

    # Log metrics
    log_metric("tokens_used", response.usage.total_tokens)
    log_metric("retrieval_count", len(chunks))

    return response
```

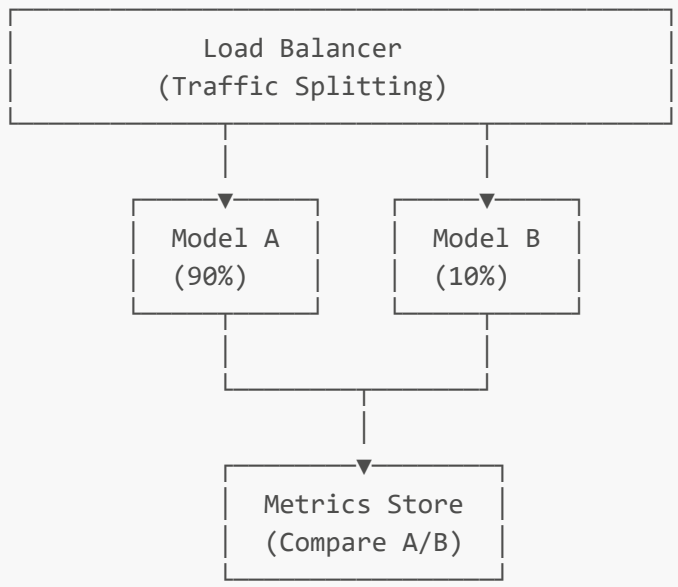
Alerting:

- High latency (>5s P95)
- Error rate spike
- Token usage anomaly
- Low user satisfaction

Q31: Explain A/B testing for ML models

Answer:

Setup:



Key Considerations:

1. **Statistical significance:** Minimum sample size
2. **Metrics:** Define primary/secondary metrics
3. **Duration:** Run long enough for confidence
4. **Segmentation:** User groups, regions

```
# Feature flag based routing
def get_model_version(user_id: str) -> str:
    if is_in_experiment(user_id, "new_model_test"):
        return "model_v2" if hash(user_id) % 100 < 10 else "model_v1"
    return "model_v1"

# Track metrics by variant
async def process_with_tracking(query, user_id):
    variant = get_model_version(user_id)
    response = await model_router[variant].generate(query)

    log_experiment_metric(
        experiment="new_model_test",
        variant=variant,
        user_id=user_id,
        metrics={"latency": response.latency, "satisfaction": ...}
    )
    return response
```

---

9. Python & Frameworks

Q32: Explain async/await in Python for AI applications

Answer:

Why async matters for AI:

- I/O bound operations (API calls, database)
- Handle multiple concurrent requests
- Don't block while waiting for LLM responses

```
import asyncio
import httpx

# Synchronous - slow
def sync_embeddings(texts):
    results = []
    for text in texts:
        response = requests.post(EMBEDDING_API, json={"input": text})
        results.append(response.json())
```

```
    return results  # Takes N * latency

# Asynchronous - fast
async def async_embeddings(texts):
    async with httpx.AsyncClient() as client:
        tasks = [
            client.post(EMBEDDING_API, json={"input": text})
            for text in texts
        ]
        responses = await asyncio.gather(*tasks)
        return [r.json() for r in responses]  # Takes ~1 * latency

# Usage in FastAPI
@app.post("/embed")
async def embed_documents(docs: List[str]):
    embeddings = await async_embeddings(docs)
    return {"embeddings": embeddings}
```

Q33: Explain FastAPI best practices for AI services

Answer:

```
from fastapi import FastAPI, Depends, HTTPException, BackgroundTasks
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from typing import List, Optional
import asyncio

app = FastAPI(title="AI Service", version="1.0.0")

# 1. CORS for frontend
app.add_middleware(CORSMiddleware, allow_origins=["*"], allow_methods=["*"])

# 2. Pydantic models for validation
class QueryRequest(BaseModel):
    query: str
    top_k: Optional[int] = 5

class QueryResponse(BaseModel):
    answer: str
    sources: List[dict]
    latency_ms: float

# 3. Dependency injection
async def get_current_user(token: str = Depends(oauth2_scheme)):
    return await verify_token(token)

# 4. Async endpoints
@app.post("/query", response_model=QueryResponse)
async def query(
```

```
    request: QueryRequest,
    user: dict = Depends(get_current_user),
    background_tasks: BackgroundTasks
):
    start = time.time()

    # Async operations
    chunks = await retriever.search(request.query, request.top_k)
    answer = await llm.generate(request.query, chunks)

    # Background logging
    background_tasks.add_task(log_query, user["id"], request.query)

    return QueryResponse(
        answer=answer,
        sources=chunks,
        latency_ms=(time.time() - start) * 1000
    )

# 5. Streaming responses
@app.post("/stream")
async def stream_response(request: QueryRequest):
    async def generate():
        async for chunk in llm.stream(request.query):
            yield f"data: {chunk}\n\n"

    return StreamingResponse(generate(), media_type="text/event-stream")

# 6. Health check
@app.get("/health")
async def health():
    return {"status": "healthy", "model_loaded": model is not None}
```

Q34: How do you handle errors and retries in LLM applications?

Answer:

```
import asyncio
from tenacity import retry, stop_after_attempt, wait_exponential,
retry_if_exception_type
import httpx

# Retry decorator
@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=1, max=10),
    retry=retry_if_exception_type((httpx.TimeoutException, httpx.HTTPStatusError))
)
async def call_llm_with_retry(prompt: str):
    async with httpx.AsyncClient(timeout=30.0) as client:
```

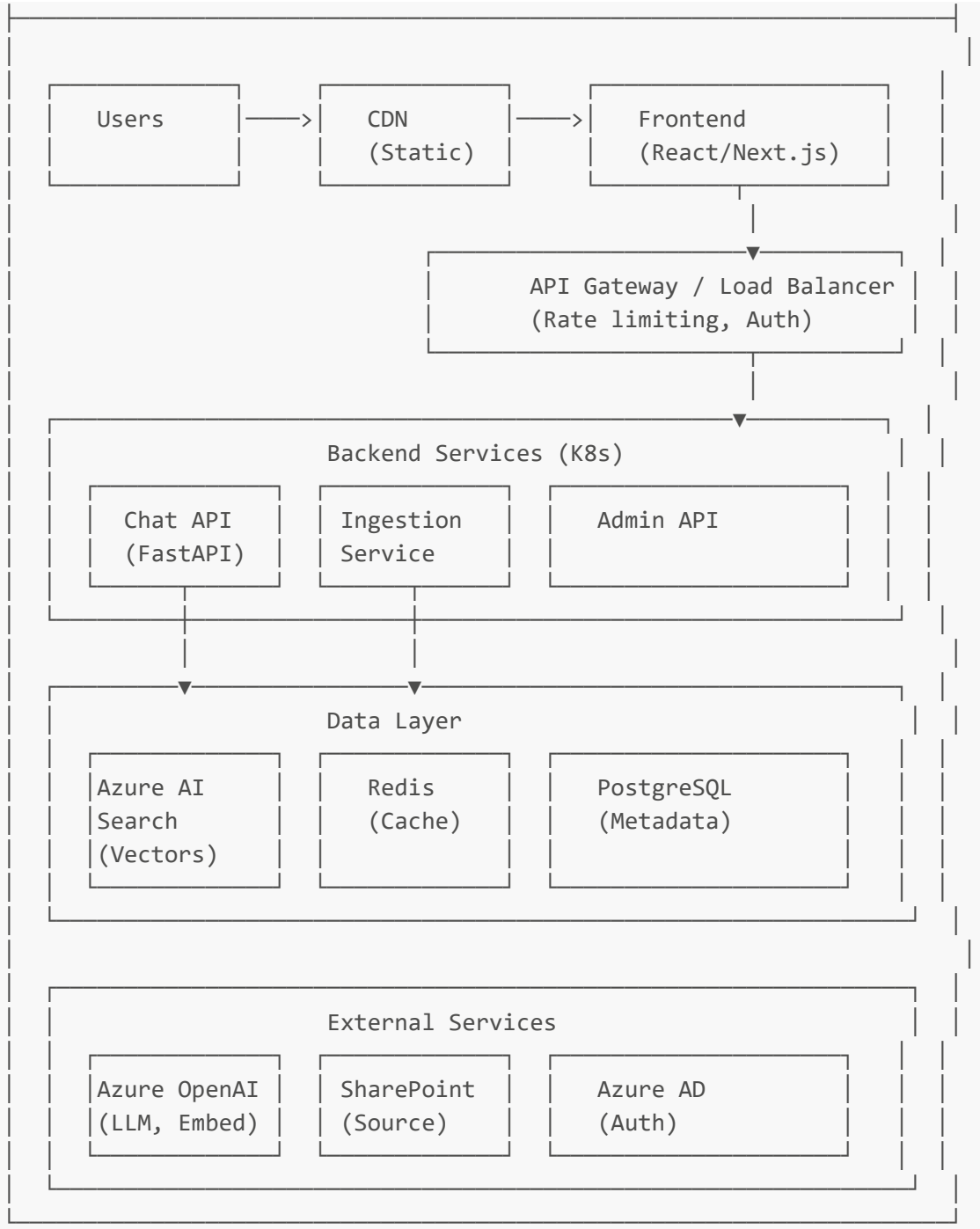
```
        response = await client.post(
            LLM_ENDPOINT,
            json={"prompt": prompt},
            headers={"Authorization": f"Bearer {API_KEY}"}
        )
        response.raise_for_status()
        return response.json()

# Circuit breaker pattern
class CircuitBreaker:
    def __init__(self, failure_threshold=5, reset_timeout=60):
        self.failures = 0
        self.threshold = failure_threshold
        self.reset_timeout = reset_timeout
        self.last_failure = None
        self.state = "closed"

    async def call(self, func, *args, **kwargs):
        if self.state == "open":
            if time.time() - self.last_failure > self.reset_timeout:
                self.state = "half-open"
            else:
                raise Exception("Circuit breaker is open")

        try:
            result = await func(*args, **kwargs)
            if self.state == "half-open":
                self.state = "closed"
                self.failures = 0
            return result
        except Exception as e:
            self.failures += 1
            self.last_failure = time.time()
            if self.failures >= self.threshold:
                self.state = "open"
            raise

# Usage
breaker = CircuitBreaker()
result = await breaker.call(call_llm_with_retry, "Hello")
```

10. System Design for AI

Q35: Design a scalable RAG chatbot system

Answer:

```
┌─────────────────────────────────────────────────────────────────┐
│                        Architecture                             │
```



Key Design Decisions:

1. **Caching:**
   - Cache embeddings for repeated queries
   - Cache LLM responses for identical queries
   - Cache user permissions
2. **Scaling:**
   - Horizontal scaling of API servers
   - Separate ingestion workers
   - Auto-scaling based on load

3. **Reliability:**
    - Retry with exponential backoff
    - Circuit breakers for external services
    - Fallback responses

Q36: How do you handle rate limiting and cost control for LLM APIs?

Answer:

```
import asyncio
from collections import defaultdict
import time

class RateLimiter:
    def __init__(self, requests_per_minute: int, tokens_per_minute: int):
        self.rpm = requests_per_minute
        self.tpm = tokens_per_minute
        self.requests = []
        self.tokens = []
        self.lock = asyncio.Lock()

    async def acquire(self, estimated_tokens: int = 1000):
        async with self.lock:
            now = time.time()
            minute_ago = now - 60

            # Clean old entries
            self.requests = [t for t in self.requests if t > minute_ago]
            self.tokens = [(t, n) for t, n in self.tokens if t > minute_ago]

            # Check limits
            current_requests = len(self.requests)
            current_tokens = sum(n for _, n in self.tokens)

            if current_requests >= self.rpm:
                wait_time = self.requests[0] - minute_ago
                await asyncio.sleep(wait_time)

            if current_tokens + estimated_tokens > self.tpm:
                # Wait or reject
                raise Exception("Token limit exceeded")

            self.requests.append(now)
            self.tokens.append((now, estimated_tokens))

    def record_actual_tokens(self, tokens: int):
        # Update with actual token usage
        pass

# Cost tracking
```

```python
class CostTracker:
    PRICING = {
        "gpt-4": {"input": 0.03, "output": 0.06},  # per 1K tokens
        "gpt-3.5-turbo": {"input": 0.001, "output": 0.002},
        "text-embedding-3-small": {"input": 0.00002}
    }

    def __init__(self):
        self.usage = defaultdict(lambda: {"input": 0, "output": 0, "cost": 0})

    def track(self, model: str, input_tokens: int, output_tokens: int = 0):
        pricing = self.PRICING.get(model, {"input": 0, "output": 0})
        cost = (input_tokens / 1000 * pricing["input"] +
                output_tokens / 1000 * pricing["output"])

        self.usage[model]["input"] += input_tokens
        self.usage[model]["output"] += output_tokens
        self.usage[model]["cost"] += cost

        return cost

    def get_daily_cost(self) -> float:
        return sum(u["cost"] for u in self.usage.values())

# Usage
limiter = RateLimiter(requests_per_minute=60, tokens_per_minute=90000)
cost_tracker = CostTracker()

async def call_llm(prompt: str):
    await limiter.acquire(estimated_tokens=2000)

    response = await openai_client.chat.completions.create(...)

    cost = cost_tracker.track(
        "gpt-4",
        response.usage.prompt_tokens,
        response.usage.completion_tokens
    )

    # Alert if daily cost exceeds threshold
    if cost_tracker.get_daily_cost() > DAILY_BUDGET:
        send_alert("Daily budget exceeded!")

    return response
```

---

11. Behavioral & Scenario Questions

Q37: Tell me about a challenging ML project you worked on

Answer Framework (STAR):

Situation: "At [Company], we needed to build a document Q&A system that could handle 500K documents with strict security requirements."

Task: "I was responsible for designing the RAG architecture and implementing the permission-aware search system."

Action:

- Evaluated vector databases (chose Azure AI Search for enterprise features)
- Designed chunking strategy for various document types
- Implemented security trimming using Azure AD groups
- Built async ingestion pipeline for performance
- Created evaluation framework to measure accuracy

Result:

- Reduced search latency from 5s to 200ms
- Achieved 95% user satisfaction in pilot
- System handles 1000 queries/hour with 99.9% uptime

Q38: How do you stay updated with AI/ML developments?

Answer:

Resources:

1. **Papers:** arXiv, Papers with Code, Semantic Scholar
2. **News:** The Batch (Andrew Ng), AI News
3. **Communities:** Twitter/X, Reddit r/MachineLearning, Discord
4. **Courses:** Coursera, fast.ai, Hugging Face courses
5. **Conferences:** NeurIPS, ICML, ACL, EMNLP proceedings
6. **Hands-on:** Kaggle competitions, personal projects

Recent examples:

- "I've been following the developments in mixture-of-experts models like Mixtral"
- "Recently implemented structured outputs from GPT-4 after reading the release notes"
- "Experimenting with new embedding models like text-embedding-3"

Q39: How do you explain AI concepts to non-technical stakeholders?

Answer:

Techniques:

1. **Analogies:**
    - RAG: "Like giving the AI a reference book to look up answers instead of relying on memory"
    - Embeddings: "Converting words into coordinates on a map where similar meanings are close together"

- Fine-tuning: "Like sending the AI to specialized training school"

2. **Business terms:**
   - Instead of "accuracy": "How often the system gets it right"
   - Instead of "latency": "Response time"
   - Instead of "tokens": "Words processed"

3. **Visuals:**
   - Flowcharts for pipelines
   - Before/after comparisons
   - ROI calculations

Example: "Our RAG system is like a very smart research assistant. When you ask a question, it first searches through all our company documents to find relevant information, then uses that information to write a clear answer. This means it can only tell you things that are actually in our documents, reducing the risk of made-up answers."

Q40: What would you do if an LLM is giving incorrect answers in production?

Answer:

Immediate Response:

1. Check error logs and recent changes
2. Identify pattern (all queries? specific topics? certain users?)
3. Enable verbose logging for debugging
4. Consider temporary fallback (human review, cached responses)

Investigation:

```
# Add debugging
async def debug_query(query: str):
    # 1. Check retrieval
    chunks = await retriever.search(query)
    logger.info(f"Retrieved {len(chunks)} chunks")
    for i, chunk in enumerate(chunks):
        logger.info(f"Chunk {i}: {chunk['content'][:100]}...")
        logger.info(f"Score: {chunk['score']}")

    # 2. Check prompt
    prompt = build_prompt(query, chunks)
    logger.info(f"Full prompt: {prompt}")

    # 3. Check LLM response
    response = await llm.generate(prompt, temperature=0)
    logger.info(f"Response: {response}")

    return response
```

Root Cause Categories:

1. **Retrieval issues:** Wrong chunks, poor embedding quality
2. **Prompt issues:** Unclear instructions, missing context
3. **Model issues:** Hallucination, outdated knowledge
4. **Data issues:** Stale content, permission problems

Long-term Fixes:

- Add evaluation suite
- Implement monitoring for quality metrics
- Create feedback loop for continuous improvement
- A/B test prompt changes

Quick Reference Card

Common Interview Topics Checklist

- ☐ ML fundamentals (bias-variance, regularization, metrics)
- ☐ Deep learning (backprop, attention, transformers)
- ☐ NLP (embeddings, BERT, tokenization)
- ☐ LLMs (GPT, prompting, hallucinations)
- ☐ RAG (architecture, chunking, evaluation)
- ☐ Vector databases (comparison, similarity metrics)
- ☐ Azure AI services (OpenAI, AI Search, security)
- ☐ MLOps (pipelines, monitoring, A/B testing)
- ☐ Python (async, FastAPI, error handling)
- ☐ System design (scalability, cost, reliability)

Key Numbers to Remember

Metric	Typical Value
BERT hidden size	768
GPT-4 context	128K tokens
Embedding dimensions	1536 (OpenAI ada-002), 3072 (text-embedding-3-large)
Chunk size	512-1024 tokens
Chunk overlap	10-20%
Top-K retrieval	3-10 chunks
Temperature for facts	0.0-0.3
Temperature for creativity	0.7-1.0

Good luck with your interview! 🚀

Remember: Be confident, give concrete examples, and don't be afraid to say "I don't know, but here's how I'd find out."