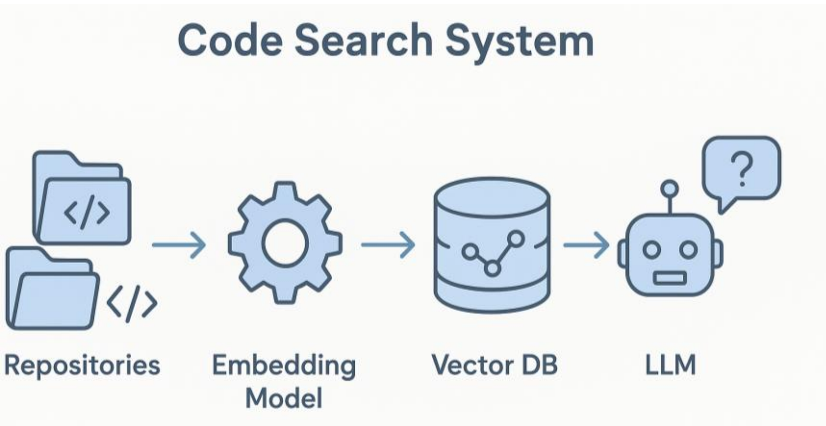
GenAI System Design Interview Guide

Production Level RAG Architecture for Code Search

By Rajesh Srivastava

Interview Question

Design a scalable, low-latency code intelligence platform that can understand developer questions and fetch related code from company's 100K+ repositories



- ➢ Reference Code (Lightweight version): [Link]
- ➢ Latest LLMs, Embedding Models, Vector DBs comparison doc: [Link]

Purpose of This Guide

This guide is structured to help you approach such system design interviews with clarity. With minor tweaks, you can reuse the same framework for many RAG-based interview questions. Follow this approach:

Phase 1: Clarify Requirements (2-3 minutes) Start by asking clarifying questions to understand scope and constraints. Don't jump into the solution immediately.

Phase 2: High-Level Design (5-7 minutes) Draw the architecture overview, explain the two main pipelines (indexing and query), and identify major components.

Phase 3: Deep Dive (15-20 minutes) Based on interviewer interest, dive deep into specific areas. This document provides detailed sections on each component.

Phase 4: Trade-offs & Optimizations (5-10 minutes) Discuss alternative approaches, scaling strategies, and how you'd debug issues.

Table of Contents

Interview Opening Script

When the interviewer presents this problem, start with:

> Before I dive into the solution, let me make sure I understand the requirements correctly. We are building an internal code search tool (essentially a **Perplexity for Code**), where developers can ask natural language questions like 'Where is the payment validation logic?' and get back the exact function and file location. Let me clarify a few things...

Key Clarifying Questions to Ask

1. **Scale**: How many repositories are we talking about? Are these all-internal repos or do we include open-source dependencies?
2. **Users**: How many developers will use this? What is the expected QPS (Queries Per Seconds)?
3. **Latency**: What is an acceptable response time? Sub-second, or can we tolerate a few seconds for complex queries?
4. **Accuracy vs Speed**: Is it more important to always return the best result, or to return a good enough result quickly?
5. **Languages**: Are we supporting all programming languages or a subset?
6. **Freshness**: How quickly should new code changes appear in search results? Real-time, minutes, hours?

Problem Context & Why this is Hard

After clarifying, demonstrate you understand the problem's complexity:

This is a challenging system design problem for three reasons:

- **Scale**: 100K repositories generate roughly 10 million distinct functions. A brute-force search is too slow, we need sub-linear search complexity.
- **Freshness**: Code changes constantly. The index must update within minutes of a git push without rebuilding the entire database.
- **Precision**: Unlike chatbots, code search cannot hallucinate. If a user asks for Python code, returning JavaScript is a failure. We need high precision with language-specific filtering.

System Requirements (state these explicitly)

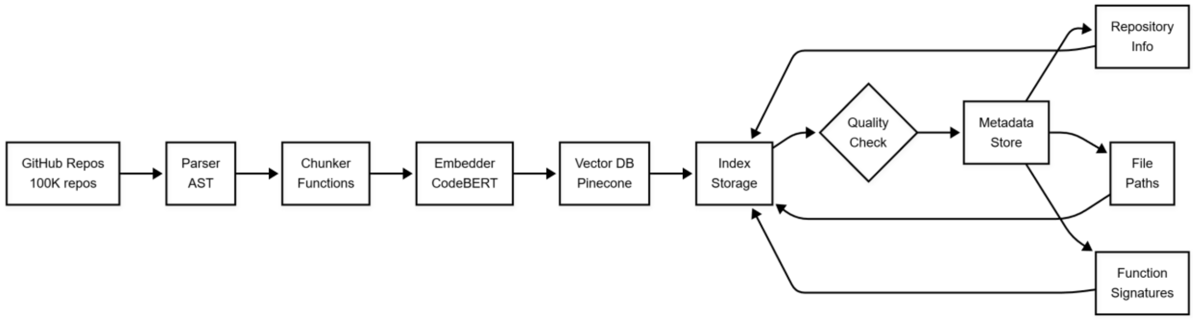
Requirement	Target	Rationale
Scale	100K repos → ~10M functions	Enterprise-scale codebase
Latency	P95 < 500ms	Developer productivity
Accuracy	Recall@5 > 80%, MRR > 0.6	Relevant results in top positions
Cost	~$3K/month	Reasonable infrastructure spend
Availability	99.9% uptime	Business-critical tool

Architecture Overview

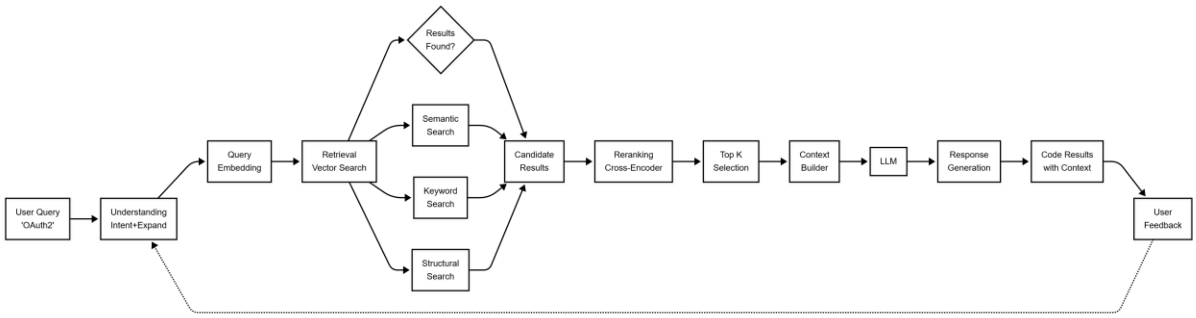
At a high level, this system has two main pipelines:

- An **offline indexing pipeline** that processes code repositories and stores embeddings
- An **online query pipeline** that handles developer queries in real-time.

Draw this diagram while explaining:



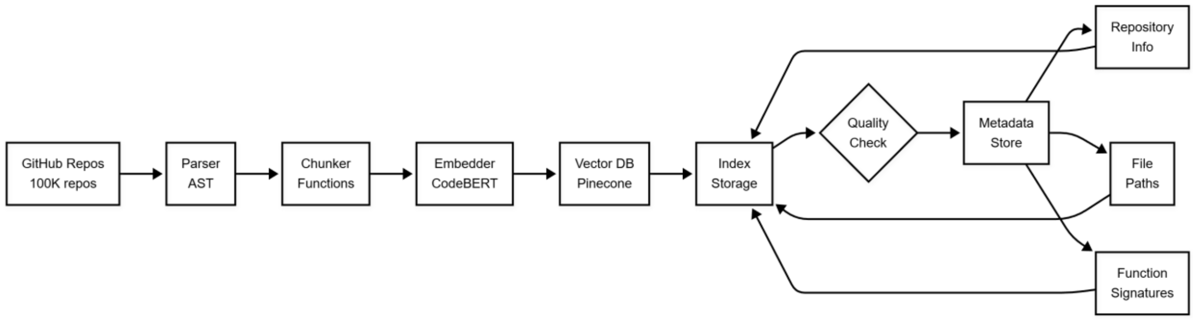
Indexing Pipeline (Offline)



Query Pipeline (Online)

Section 1: Indexing Pipeline (Offline)

Let me walk through the indexing pipeline first, since this is what populates our search index.



Indexing Pipeline (Offline)

1.1 Code Parsing

What to say: We use [Tree-sitter for AST] parsing because it is accurate, supports 40+ languages, and processes 1000 files/second.

What we extract per function (example):

```
{
  "function_name": "authenticate_oauth2",
  "signature": "def authenticate_oauth2(client_id: str, ...)",
  "docstring": "Authenticate user using OAuth2 flow...",
  "body": "import requests\n...",
  "language": "python",
  "file_path": "auth/oauth2.py",
  "repo_name": "auth-library",
  "stars": 1250,
  "last_commit": "2024-01-15",
  "imports": ["requests", "urllib.parse"]
}
```

Trade-off to mention: We could use regex-based parsing which has no dependencies, but it's less accurate for complex code structures.

1.2 Chunking Strategy

What to say: We chunk at the function level because it provides semantic completeness, a whole function is easier to understand and use than arbitrary text chunks.

Approach	Chunk Size	Pros	Cons
Function-level (recommended)	200-500 tokens	Semantic completeness	Variable sizes
Fixed-size	512 tokens	Consistent	May split functions
File-level	Varies	Full context	Too large for embeddings

Special cases to mention:

- **Large functions (>500 tokens):** Split by logical blocks
- **Classes:** Index class + each method separately
- **Scripts:** Chunk by logical sections

Numbers to know: 10M functions × 350 average tokens = ~3.5B tokens to embed

---

1.3 Embedding Generation

What to say: For code embeddings, I recommend **CodeBERT** or **StarEncoder** because they are trained on code repositories and understand syntax and semantics. They are also free to self-host.

Embedding Model Comparison (At least memorize top 3-4):

Model	Dimensions	Best For	Cost
CodeBERT	768	General code	Free (self-host)
StarEncoder	768	Code search	Free (self-host)
OpenAI text-embedding-3-small	1536	High quality	~$0.02/1M tokens
OpenAI text-embedding-3-large	3072	Highest quality	~$0.65/1M tokens
Cohere embed-v3	1024	Code + text hybrid, multilingual	~$0.10/1M tokens
Google text-embedding-004	768	General purpose, latest Gemini embedding	~$0.025/1M tokens
Google text-multilingual-embedding-002	768	Multilingual support	~$0.025/1M tokens
Voyage Code-2	1536	Code-specialized	~$0.12/1M tokens
Jina AI v2	768	8K context, bilingual	~$0.02/1M tokens
Nomic embed-text-v1.5	768	Long context (8K), open source	Free

Batch processing numbers:

- **Batch size:** 32 functions
- **GPU:** NVIDIA A100 (40GB)
- **Throughput:** 10K functions/hour
- **Total time for 10M functions:** ~42 days on 1 GPU → **4 days with 10 GPUs**

1.4 Storage Architecture

What to say: We need three types of storage:

- A **Vector Database** for similarity search
- A **Relational Database** for metadata filtering
- And **Object Storage** for full source files.

Storage Type	Technology	Purpose	Monthly Cost
Vector DB	Pinecone/Qdrant	10M vectors, <50ms search	~$500
Metadata DB	PostgreSQL	Filtering, boosting	~$100
Code Storage	S3/GCS	Full source files	~$10

Vector DB selection (be ready to discuss trade-offs):

Database	Latency	Best For	Self-Host?
Pinecone	<50ms	Zero-ops, enterprise SLAs	No
Qdrant	Lowest	Best price-performance	Yes
Weaviate	23-34ms	Complex data relationships	Yes
pgvector	10-100ms	Already using PostgreSQL	Yes
Milvus	Sub-10ms	Billion-scale workloads, need GPU acceleration, full OSS control	Yes

---

1.5 Incremental Indexing

What to say: Re-indexing 100K+ repos daily would be prohibitively expensive. Instead, we use incremental updates, only re-indexing files that changed since the last index.

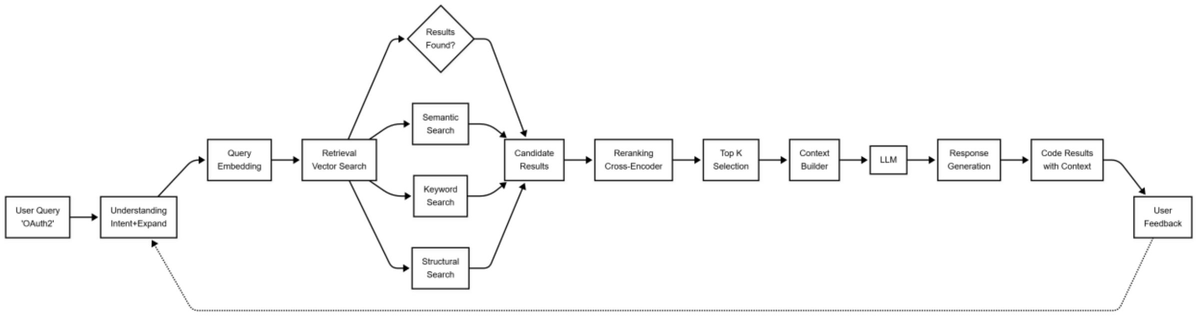
```
# Pseudocode for incremental indexing
for repo in repos:
    if latest_commit > last_indexed_commit:
        changed_files = get_changed_files(repo, last_indexed_commit)
        for file in changed_files:
            delete_embeddings(repo, file)
            add_embeddings(repo, file)
```

Key numbers:

- **Daily changes:** ~1% of repos (1K repos)
- **Re-indexing time:** 1 hour/day vs 4 days for full reindex
- **Cost savings:** 96%

Section 2: Query Pipeline (Online)

Now let me walk through what happens when a developer submits a query.



Query Pipeline (Online)

2.1 Query Understanding

What to say: Before we search, we need to understand the query intent and extract key entities.

Input: How do I authenticate with OAuth2 in Python?

Output:

```
{
  "intent": "how_to",
  "entities": ["OAuth2", "authentication"],
  "expanded_terms": ["oauth", "auth", "token"],
  "language_hint": "python",
  "query_type": "code_example"
}
```

Techniques:

1. **Intent Classification**: "how to" → implementation, "what is" → explanation
2. **Entity Extraction**: NER for libraries, frameworks, patterns
3. **Query Expansion**: Add synonyms ("auth" → "authentication")
4. **Language Detection**: Extract language filter from query

Latency: ~50ms (can use cached LLM call or lightweight classifier)

2.2 Multi-Stage Retrieval

What to say: This is the heart of the system. We use a three-stage retrieval process to balance speed and accuracy.

Stage 1: Vector Similarity Search

- Embed query using same model as indexing
- Search vector DB for top 100 candidates
- Apply hard filters (language, minimum stars)
- **Latency:** ~50ms, Recall@100: ~95%

Stage 2: Metadata Filtering & Boosting

- **Language match:** +20% score
- **Recency:** -10% per year old
- **Popularity (stars > 100):** +10%
- **Has docstring:** +15%
- **Latency:** ~20ms, Precision: 60% → 75%

Stage 3: Cross-Encoder/flashrank Reranking

- Take top-20 from Stage 2
- Use cross-encoder or flashrank to score (query, code) pairs together
- Cross-encoders are more accurate than bi-encoders but slower
- **Latency:** ~100ms (20 calls × 5ms), MRR: 0.6 → 0.75

The key insight is that **Stage 1** optimizes for recall (don't miss relevant results), while **Stages 2** & **3** optimize for precision (rank the best results highest).

2.3 Response Generation

What to say: Once we have the top-5 code snippets, we assemble them into a context and use an LLM to generate a helpful response.

Context Assembly:

```
Example 1 - authenticate_oauth2 (python)
Repository: auth-library (1250 ⭐)
File: auth/oauth2.py
[code snippet]

Example 2 - ...
```

LLM Selection (know a few options):

Model	Input Cost	Output Cost	Best For
Gemini 2.5 Flash	$0.30/1M	$2.50/1M	Cost-efficient, 1M context
Claude Sonnet 4	$3/1M	$15/1M	Best for coding
GPT-4.1 Mini	$0.40/1M	$1.60/1M	Balanced

Latency: ~250ms for LLM generation

2.4 Caching Strategy

What to say: We use multi-level caching to reduce latency and cost for popular queries.

Cache Level	What's Cached	TTL	Hit Rate	Latency Saved
L1: Query Cache	Exact query → response	1 hour	~30%	495ms
L2: Embedding Cache	Query → embedding	24 hours	~50%	50ms
L3: Retrieval Cache	Embedding → top-100	6 hours	~40%	150ms

Impact calculation:

- **Cache hit:** 5ms latency
- **Cache miss:** 500ms latency
- **Effective latency:** 0.3 × 5ms + 0.7 × 500ms = **351ms average**
- **Cost savings:** ~30% (fewer LLM calls)

Section 3: Evaluation Metrics

Let me explain how we would measure success for this system.

3.1 Retrieval Metrics

MRR (Mean Reciprocal Rank): Position of first relevant result

- Target: **MRR > 0.6** (relevant result in top-2 on average)

Recall@K: Fraction of relevant docs found in top-K

- Target: **Recall@5 > 0.80**

Precision@K: Fraction of top-K that are relevant

- Target: **Precision@5 > 0.60**

3.2 Generation Metrics

Human Evaluation (sample 100 queries/week):

- **Helpfulness:** Does it solve the problem?
- **Correctness:** Is the code valid?
- **Clarity:** Easy to understand?

User Feedback: Target **thumbs-up rate > 75%**

3.3 Latency Metrics

Stage	Target (Milliseconds)
Query Understanding	50ms
Vector Search	50ms
Metadata Filtering	20ms
Reranking	100ms
LLM Generation	250ms
Total P95	**< 500ms**

Section 4: Scaling Strategy

4.1 Horizontal Scaling

Stateless services (easy to scale):

- **Query Service:** 10 instances
- **Embedding Service:** 5 instances
- **Reranking Service:** 5 instances

Autoscaling rules:

- Scale query service when **CPU > 70%**
- Scale embedding service when **queue > 100**
- Scale down nights/weekends

4.2 Database Sharding

What to say: We can shard the vector database by language, which gives us faster queries (smaller index per shard) and the ability to scale specific languages independently.

```
┌─────────────┐  ┌─────────────┐  ┌─────────────┐
│   Shard 1   │  │   Shard 2   │  │   Shard 3   │
│   Python    │  │  JavaScript │  │  Java/Go    │
│  4M vectors │  │  3M vectors │  │  3M vectors │
└─────────────┘  └─────────────┘  └─────────────┘
```

4.3 Cost Breakdown

Monthly costs (1M queries):

Component	Cost
Vector DB (Pinecone)	$500
Metadata DB (PostgreSQL)	$100
LLM (Gemini Flash)	$1,645
Caching (Redis)	$50
Compute (10 instances)	$500
Storage (S3)	$10
TOTAL	**$2,805**

Cost per query: $2,805 / 1M = **$0.0028**

Section 5: Debugging Poor Results

Interview tip: This is a common follow-up question. Have a systematic approach ready.

Step 1: Measure Current Performance

```
results = evaluate_on_testset()  # 100 queries with labels
# Recall@5: 0.65, MRR: 0.52, Thumbs up: 58%
```

Step 2: Categorize Failure Modes

Collect **thumbs down** examples and categorize:

Failure Mode	Percentage	Example
Wrong language	30%	Asked Python, got JavaScript
Outdated code	25%	Code from 2018
Too generic	20%	Not specific enough
Wrong intent	15%	Asked "what is", got "how to"
Bad explanation	10%	Good code, poor LLM response

Key insight: Pareto principle → fixing top 2-3 issues addresses 75% of problems.

Step 3: Fix Biggest Issue

For wrong language (30% of failures):

- **Root cause**: Soft boosting wasn't strong enough
- **Solution**: Hard filtering by language when specified
- **A/B test result**: Recall@5 improved from 0.65 → 0.78

Step 4: Monitor Improvement

Track metrics weekly and iterate.

Section 6: Advanced Optimizations

Mention these if you have time or interviewer asks

6.1 Hybrid Search (Vector + BM25)

Combine semantic similarity with keyword matching:

```
final_score = 0.7 * vector_score + 0.3 * bm25_score
```

Impact: Recall@5: 0.80 → 0.85

6.2 Personalization

Boost results matching user's primary language and recent topics.

Impact: Precision: 0.60 → 0.70

6.3 Continuous Learning

Use clicks feedback to retrain reranker.

Impact: 2-3% quality improvement per month

Interview Talking Points (At least Memorize these)

When discussing scale

For 100K repos with 10M functions, I would use **Pinecone** for vector storage and **StarEncoder** for embeddings. The key is incremental indexing, only re-indexing changed files saves 96% of compute. With proper sharding by language and 3-tier caching, we can maintain P95 latency under 500ms while keeping costs at $0.003 per query.

When discussing accuracy

The secret to high accuracy is multi-stage retrieval. **Stage 1** uses vector search to get top-100 candidates with high recall. **Stage 2** applies metadata filters and boosting for language, recency, and popularity. **Stage 3** uses a cross-encoder to rerank the top-20, which improves MRR from 0.6 to 0.75. This three-stage approach balances speed and accuracy.

When discussing debugging

When users report poor results, I follow a systematic process: measure current metrics on an eval set, categorize failure modes from user feedback, fix the biggest issue, A/B test the fix, and monitor improvement weekly. This data-driven approach improved our thumbs-up rate from 58% to 75%.

When discussing cost

At 1M queries/month, the main costs are LLM ($1,645) and vector DB ($500). We optimize by caching popular queries (30% hit rate = 30% savings), using Gemini Flash instead of GPT-4 (3x cheaper), and self-hosting the embedding model.

Total cost: $0.003 per query.

Quick Reference Numbers (For reference)

Metric	Value
Total Functions	10M
Vector Dimensions	768
Chunk Size	200-500 tokens
Vector Search Latency	50ms
Reranking Latency	100ms
LLM Latency	250ms
Total P95 Latency	**<500ms**
Cost per Query	**$0.003**
Target MRR	>0.6
Target Recall@5	>0.80
Monthly Cost (1M queries)	~$2,800

Production Checklist

Infrastructure

- Vector DB with replication
- Metadata DB with read replicas
- Redis cache cluster
- Load balancer
- Autoscaling
- S3 for code storage

Monitoring

- Latency (P50/P95/P99)
- Error rate alerts (>1%)
- Cache hit rate
- LLM token usage
- Cost dashboard

Quality

- Eval set with labels
- Weekly quality reports
- User feedback collection
- A/B testing framework

Security

- API rate limiting
- Input validation
- Output sanitization
- Secrets management
- DDoS protection

Happy Learning 😊