

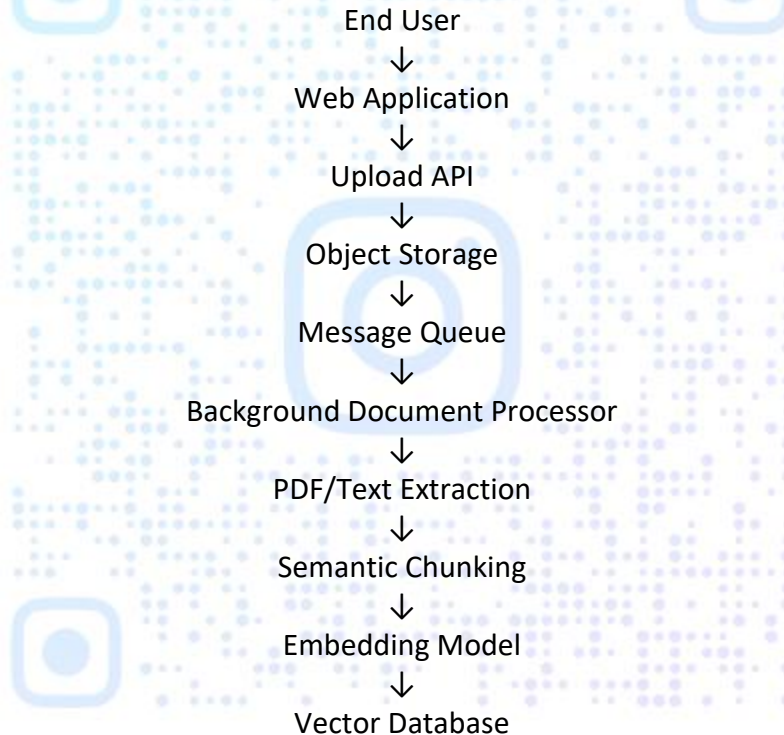
RAG Pipeline Interview Preparation

1. Can you explain the architecture of your RAG pipeline and the application?

The application is an enterprise web-based RAG application used by internal engineering/technical users to search and ask questions over enterprise documents.

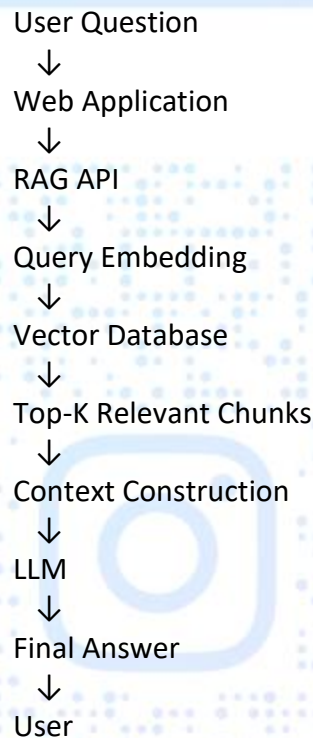
The architecture has two major flows:

Document Ingestion Flow



@RAWHUL213

Query Flow



2. Who is the end user?

The primary users are internal enterprise engineering and technical teams who need to search technical and business documentation without manually going through large numbers of documents.

3. How does the user upload the document?

The user logs into the web application and uploads a PDF through the UI. The upload API validates the document and stores the actual PDF in secure object storage. After the document is successfully stored, the application publishes a message to a queue to trigger asynchronous document processing.

4. Is document upload and processing synchronous or asynchronous?

The document ingestion pipeline is asynchronous. The upload API does not wait for PDF extraction, chunking and embedding to finish. It stores the document and publishes a processing job to a message queue. A background worker consumes the message and processes the document.

<https://www.instagram.com/rawhul213>

This gives us better scalability, avoids HTTP timeouts and provides retry and failure-handling capabilities.

5. Why did you use asynchronous processing for document ingestion?

Document processing can take significant time, especially when the document contains many pages and therefore many chunks.

If we processed everything synchronously, the HTTP request could remain open for a long time and potentially timeout.

Asynchronous processing allows:

- Better scalability
- Background processing
- Retry mechanisms
- Failure isolation
- Better user experience

The UI can show document status such as:

UPLOADED → PROCESSING → COMPLETED

↓
FAILED

6. What is in the message queue?

The actual PDF is not placed in the queue.

The queue contains a lightweight event/job message containing metadata and a reference to the document in object storage.

Example:

```
{
  "event_id": "evt-12345",
  "event_type": "DOCUMENT_UPLOADED",
  "event_version": "1.0",
  "timestamp": "2026-08-17T10:30:00Z",
  "document": {
    "document_id": "doc-12345",
    "file_name": "hr_policy.pdf",
    "content_type": "application/pdf",
    "storage_uri": "documents/hr/hr_policy.pdf",
    "checksum": "sha256:abc123",
    "version": 1
  },
  "source": {
    "uploaded_by": "user-456",
    "department": "HR",
    "tenant_id": "vw-001"
  }
}
```

<https://www.instagram.com/rawhul213>

```
,
"processing": {
  "pipeline": "rag-ingestion",
  "requested_operations": [
    "text_extraction",
    "semantic_chunking",
    "embedding",
    "indexing"
  ]
},
"correlation": {
  "correlation_id": "corr-789",
  "trace_id": "trace-456"
}
}
```

7. What is the contract between document storage/upload and document processing?

It is a versioned event/message contract.

The contract defines fields such as:

- event_id
- event_type
- event_version
- timestamp
- document_id
- storage_uri
- file_name
- content_type
- checksum
- document version
- tenant information
- uploader information
- processing requirements
- correlation/trace IDs

The contract is versioned so changes in the upload service don't unexpectedly break the document-processing service.

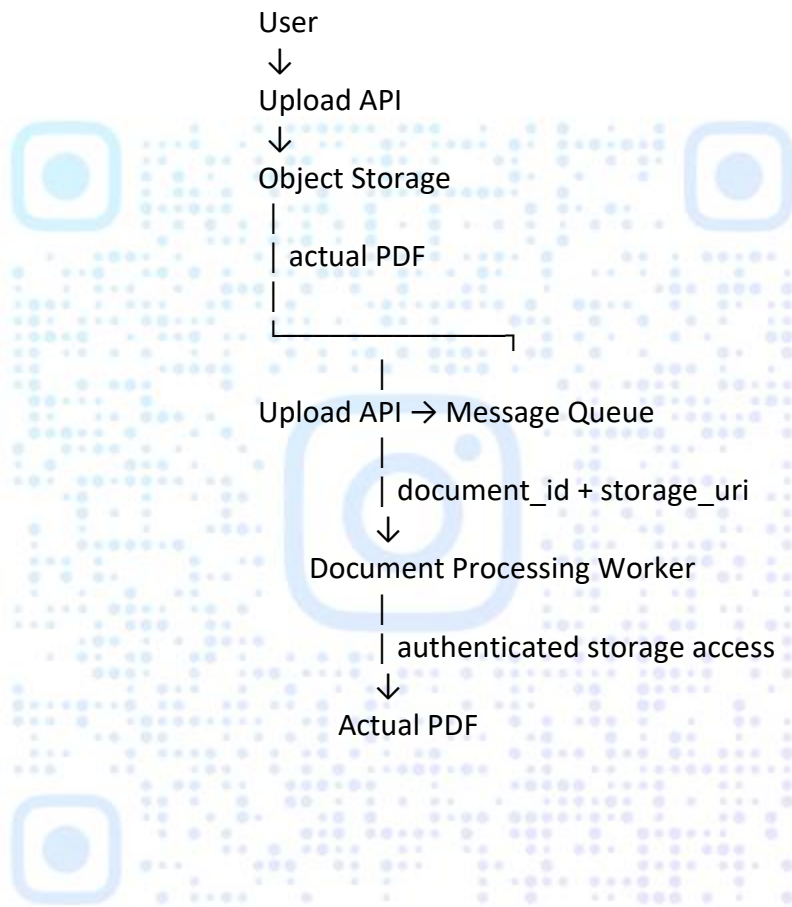
8. The queue contains only metadata. How does the document-processing service get the actual PDF?

The actual PDF is stored in object storage.

The queue only contains the document reference.

<https://www.instagram.com/rawhul213>

The background worker consumes the message, reads the `document_id` or `storage_uri`, authenticates with the storage service using its service identity, and downloads the actual PDF.



@RAWHUL213

Queue = instruction/reference

Object Storage = actual document

Worker = retrieves and processes document

9. Why don't you put the PDF directly into the message queue?

Queues are designed primarily for lightweight messages and event delivery.

Putting large binary documents into the queue would increase:

- Payload size
- Cost
- Latency
- Failure risk
- Queue resource consumption

Object storage is designed for large files, while the queue carries the processing instruction and document reference.

10. How does the document-processing service authenticate to object storage?

The processing service uses a managed identity or service account with permission to read the required storage container/object.

We don't put storage credentials or permanent public URLs into the queue message.

Access is controlled through authentication and authorization.

11. What chunking strategy did you finalize on?

We did not rely on a fixed chunk size and fixed overlap as the primary chunking strategy.

Instead, we identify natural semantic boundaries and keep related information together.

The goal is to create chunks based on meaning rather than simply splitting every fixed number of characters or tokens.

12. Did you have a fixed chunk size and overlap?

No, not as the primary strategy.

Because we used semantic chunking, chunks were determined based on the semantic structure and context of the document.

However, if a semantic section became excessively large, a secondary splitting mechanism could be applied to keep it within practical model/context limits.

13. What happens after semantic chunking?

Each semantic chunk is passed to the embedding model.

The embedding model converts the text chunk into a numerical vector.

The vector, original chunk text and metadata are stored in the vector database.

Document



Semantic Chunking



Chunk 1 → Embedding → Vector DB

Chunk 2 → Embedding → Vector DB

Chunk 3 → Embedding → Vector DB

14. What is embedding dimension?

Embedding dimension is the number of numerical values in the vector representation generated by an embedding model.

For example, if an embedding model generates a 384-dimensional vector:

[0.12, -0.04, 0.81, ...]

there are 384 numerical values.

The dimension represents the length of the vector, not the number of words or tokens.

15. What does the number 384 actually signify?

384 represents the number of coordinates in the mathematical vector space learned by that embedding model.

It does not mean 384 words, 384 features or 384 individual human-understandable meanings.

The model learns to use all dimensions together to represent semantic characteristics of the text.

Similar text should result in vectors that are close together in that vector space.

16. Give an example of embedding dimension.

For example, all-MiniLM-L6-v2 generates 384-dimensional embeddings.

"Engine temperature is too high"



Embedding Model



384-dimensional vector

Another semantically similar sentence such as:

"Vehicle is overheating"

should produce a vector that is relatively close in vector space.

17. Does embedding dimension determine which LLM we use?

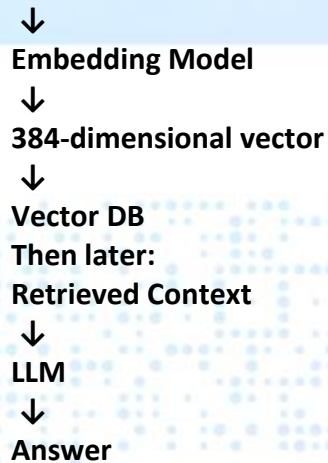
No. The embedding model and the generation LLM are separate components.

The embedding dimension is determined by the embedding model.

The LLM used for generation does not need to have the same dimension.

For example:

Document



The embedding dimension primarily affects the embedding and vector-search layer.

18. Should the embedding dimension be higher or lower?

Not necessarily higher.

Higher dimensionality can provide more representational capacity, but it also increases:

- Storage requirements
- Memory requirements
- Index size
- Computation
- Retrieval cost

Lower dimensions can be faster and cheaper but may potentially lose some semantic information.

Therefore, I would not select an embedding model simply because it has the largest dimension.

I would evaluate:

- Retrieval quality
- Recall@K
- Precision@K
- MRR
- Latency
- Cost
- Domain performance

The goal is maximum retrieval quality for an acceptable cost and latency, not maximum dimensions.

19. Which embedding model would you choose?

I would select the embedding model based on retrieval evaluation rather than dimension alone. For example, all-MiniLM-L6-v2 produces 384-dimensional embeddings and is lightweight and fast.

<https://www.instagram.com/rawhul213>

For an enterprise Azure environment, I would also evaluate Azure/OpenAI embedding models based on:

- Retrieval quality
- Domain performance
- Language support
- Latency
- Cost
- Infrastructure compatibility

I would benchmark the candidate models against actual enterprise documents and select the model that provides the best quality/cost/latency trade-off.

20. Do you choose the dimension first and then the embedding model?

No. "I choose the embedding model based on retrieval quality, domain performance, latency and cost. The embedding dimension follows from the selected model."

Query and Retrieval Pipeline

@RAWHUL213

21. Assume embeddings are already stored. What happens when a user asks a question?

The user question goes through the RAG query pipeline.



The question is converted into an embedding using the same embedding model used for the document chunks.

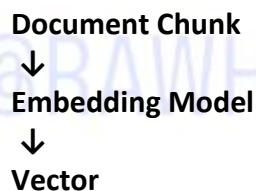
The vector database performs similarity search and returns the most relevant chunks.

Those chunks are then provided as context to the LLM.

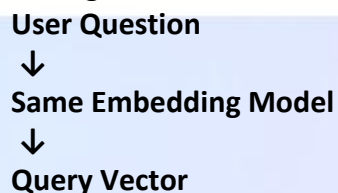
22. Why must the same embedding model be used for query and documents?

Because the query vector and document vectors must exist in the same embedding space.

During ingestion:



During retrieval:



Then the vectors can be compared using similarity metrics such as cosine similarity.

23. How does vector retrieval work?

Suppose the query is converted into a vector.

The vector database compares that query vector against stored document vectors and calculates similarity.

For example:

Query
↓
Query Vector
↓
Vector Search
Chunk 1 → similarity 0.91
Chunk 2 → similarity 0.88
Chunk 3 → similarity 0.86
Chunk 4 → similarity 0.81

The highest-scoring chunks are returned as Top-K results.

24. What is Top-K?

Top-K represents the number of most relevant chunks returned by the retrieval system. For example, if $K = 5$, the vector database returns the five highest-ranked relevant chunks. Top-K is a retrieval parameter and can be tuned based on:

- Context-window size
- Retrieval quality
- Latency
- Document complexity
- LLM requirements

25. Did you use reranking?

In our finalized implementation, we used Top-K retrieval directly and did not have a separate reranking stage.

The flow was:

Question
↓
Query Embedding
↓
Vector Search
↓
Top-K Chunks
↓
Context Construction
↓
LLM

Do not introduce reranking if it was not actually part of the implementation.

26. What is the difference between Top-K and reranking?

They are different concepts.

Top-K means how many candidates are retrieved.

Reranking means using an additional model or scoring mechanism to reorder those retrieved candidates.

Vector Search

<https://www.instagram.com/rawhul213>



Top-K candidates



Optional Reranker



Final relevant chunks

Top-K does not automatically mean reranking.

27. Why didn't you use reranking?

We evaluated the complexity versus the retrieval improvement.

For our use case, Top-K vector retrieval provided sufficient relevance, so we kept the pipeline simpler and avoided additional latency.

If retrieval quality became a bottleneck, a reranker could be introduced later.

28. How do you make sure retrieved chunks belong to the correct document/user?

Every chunk contains metadata such as:

- Document ID
- Document version
- Tenant
- Department
- Page number
- Access information

During retrieval, we can apply metadata filters so that users only retrieve documents they are authorized to access.

@RAWHUL213

29. Is the query API synchronous or asynchronous?

The original RAG query API was synchronous.

The user sends a question and waits for the answer.



The document ingestion pipeline was asynchronous, while the normal query path was synchronous.

30. What was the HTTP API timeout?

The configured HTTP API timeout was 60 seconds.

The normal end-to-end RAG response was typically below 30 seconds.

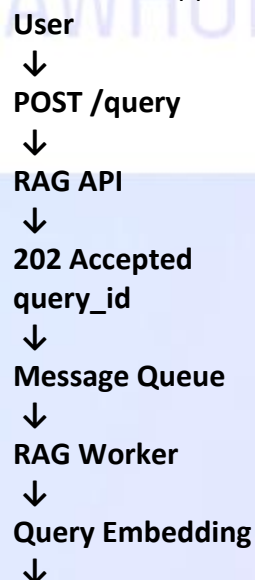
The 60-second value was a safety boundary rather than the normal response time.

31. What if there are many documents and retrieval takes more than 30 seconds?

I would redesign the query path as an asynchronous job-based architecture instead of simply increasing the HTTP timeout.

The API would immediately return 202 Accepted with a query_id.

The actual retrieval and LLM generation would happen in a background worker.



Vector Search



Top-K



LLM



Result Store

This prevents a long-running HTTP request from holding server resources.

32. What does the asynchronous query API return?

The API immediately returns:

202 Accepted

For example:

```
{  
  "query_id": "q-12345",  
  "status": "PROCESSING"  
}
```

The query_id is used to correlate the original request with the background processing and final result.

33. How does the background worker know what question to process?

The API publishes a query job to a message queue.

For example:

```
{  
  "query_id": "q-12345",  
  "question": "What is the HR leave policy?",  
  "user_id": "user-456",  
  "tenant_id": "vw-001"  
}
```

The worker consumes this message and starts the RAG pipeline.

Server-Sent Events

34. The API returned 202. How do you send the final answer back to the UI?

Because the original HTTP request has already completed, the background worker cannot return the answer through that original request.

For a conversational application, I would use Server-Sent Events (SSE).

The UI establishes an SSE connection using the query_id.

When processing completes, the backend pushes the result to the UI through that connection.

35. Explain the complete SSE flow.



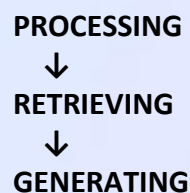
The UI opens: GET /query/q123/events

Accept: text/event-stream

The server keeps the SSE connection open.

The worker processes the request and updates the query status.

For example:



<https://www.instagram.com/rawhul213>



COMPLETED

The backend can push events to the UI.

36. What does an SSE event look like?

For example:

event: status

data: {"query_id":"q123","status":"RETRIEVING"}

Then:

event: status

data: {"query_id":"q123","status":"GENERATING"}

Finally:

event: completed

```
data: {  
  "query_id":"q123",  
  "status":"COMPLETED",  
  "answer":"According to the HR policy..."  
}
```

The browser receives these events and updates the UI.

37. How does the UI update itself?

The UI creates an SSE connection and listens for events.

Conceptually:

```
const events = new EventSource("/query/q123/events");  
events.addEventListener("completed", (event) => {  
  const result = JSON.parse(event.data);  
  displayAnswer(result.answer);  
  events.close();  
});
```

So the UI does not need to continuously ask the server whether the answer is ready.

The server pushes the update to the UI.

38. Does SSE repeatedly check the backend?

No. That is an important distinction.

SSE creates a persistent HTTP connection.

The UI doesn't repeatedly poll:

"Is it ready?"

"Is it ready?"

"Is it ready?"

Instead:

UI ————— persistent SSE connection —————▶ Server

Result ready



<https://www.instagram.com/rawhul213>

UI ←————— SSE event —————→

The server pushes the event when the status changes or processing completes.

39. What happens if the SSE connection breaks?

The UI can reconnect using the same query_id.

The result is also stored durably in a result store.

Therefore, even if the SSE connection is temporarily lost, the completed result is not lost.

When the client reconnects, the backend can return the latest status/result.

40. Why use SSE instead of WebSockets?

SSE is a good fit when communication is primarily server-to-client.

Our use case is:

Client → submit query

Server → process query

Server → push status/result

SSE is simpler for this one-way streaming pattern.

If we required continuous bidirectional communication between client and server, WebSockets would be more appropriate.

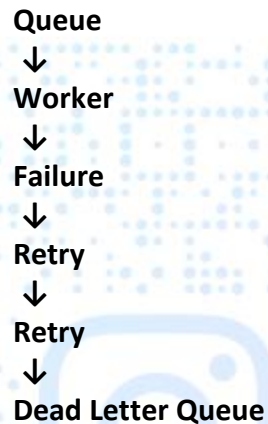
Important Production Points

41. What happens if the background job fails?

The worker can retry the message.

If it repeatedly fails, the message can be moved to a Dead Letter Queue.

The query status can be updated to FAILED, and the UI can receive a failure event.



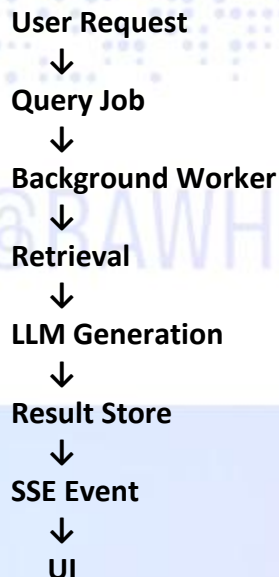
42. How do you prevent duplicate processing?

We make the processing idempotent using identifiers such as document_id + version for ingestion and query_id for query processing.

If the same message is delivered twice, the service checks whether that operation has already been completed before creating duplicate embeddings or duplicate results.

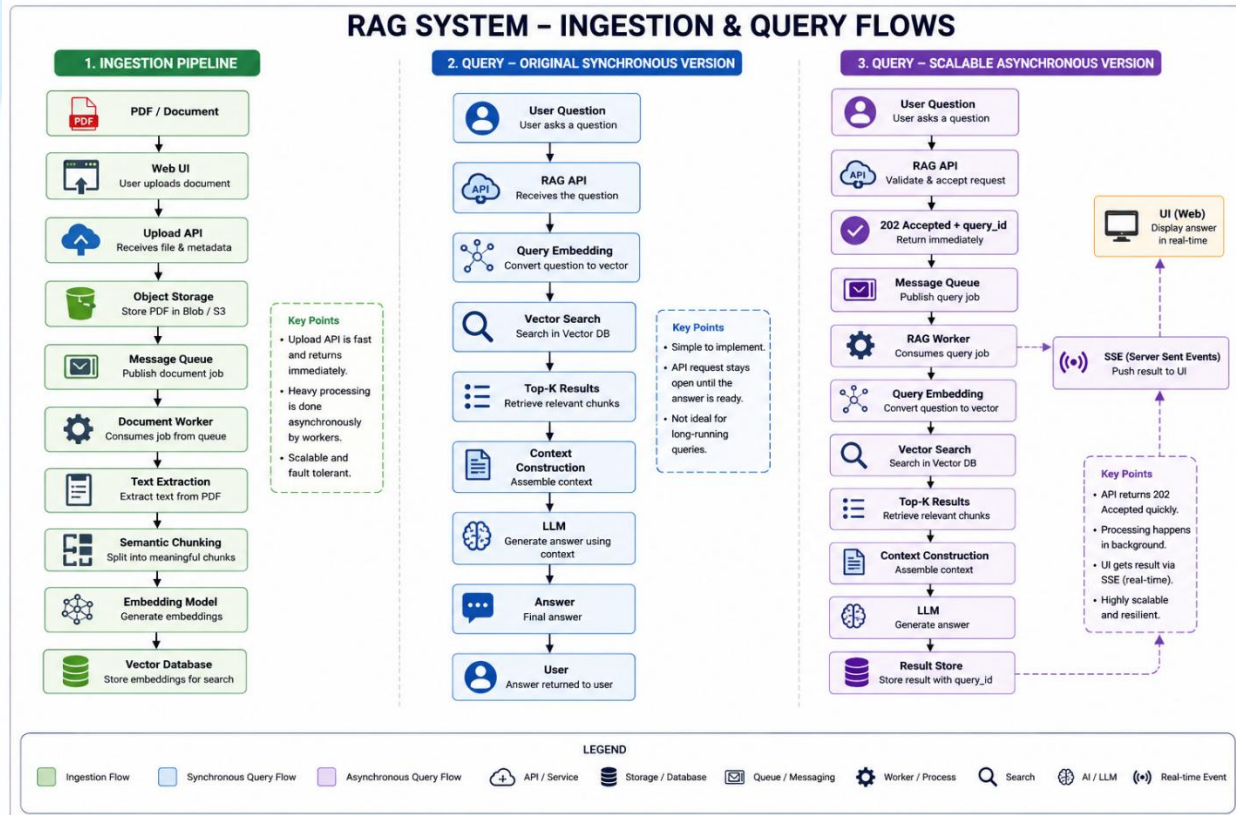
43. What is the role of the query_id?

The query_id is the correlation key for the entire asynchronous query lifecycle.



It also helps with logging, tracing and debugging.

Final Architecture to Remember



The 5 points to remember in the interview

1. Document ingestion is asynchronous.
2. Actual PDFs live in object storage; the queue contains metadata/reference.
3. Semantic chunking was used instead of fixed chunk size/overlap.
4. Query retrieval used query embedding → vector search → Top-K → LLM; no separate reranker in the finalized implementation.
5. If synchronous query latency becomes too high, redesign using 202 Accepted + query_id + background worker + result store + SSE.