Dear Candidate,

As part of our hiring process, we would like to invite you to participate in a technicality test. This test is designed to assess your skills and abilities relevant to the role you have applied for. You can check the attachment file link: [here](https://docs.google.com/document/d/1XbtAsSCrUx0A22lDpbJRxs3VDRICyzRG/edit?usp=sharing&ouid=102323437364740814749&rtpof=true&sd=true).

The deadline is 5 working days.

Kindly do what you can do on those technicality test at your best.

Thank you.

# Task 3 — Build an internal-only LLM chat / RAG (step‑by‑step)

## 0) What you’re building

A private “ChatGPT-for-your-company” that answers questions **only from internal data**. It retrieves the most relevant passages, **grounds** the model on those passages, and returns an answer **with citations**—or says “I don’t know” if evidence is insufficient.

## 1) Data prep (from internal DB/files)

* **Inventory sources:** DB tables, Confluence/SharePoint, PDFs, emails, tickets, code repos.
* **Access controls:** service account, fine‑grained RBAC/ABAC; never use a superuser key.
* **Export/ingest:**
  + DB: SQL extract views (mask PII at source when possible).
  + Files: use a text extraction pipeline (PDF, DOCX, HTML → plain text + metadata).
* **Normalize & clean:** remove boilerplate, headers/footers, menus, duplicated blocks.
* **Chunking:** split docs into **semantic chunks** (e.g., ~500–1,200 tokens) with **10–20% overlap**. Keep **metadata** (doc\_id, title, author, created\_at, acl\_tags, system\_version, url).
* **Embeddings:** choose a domain-suited model; store **vector + metadata**.
  + Vector stores: **pgvector** (Postgres), **Pinecone**, **Weaviate**, **Milvus**, **Azure AI Search**, **Vertex AI Matching Engine**, **Amazon OpenSearch** (kNN).
* **Indexing:** build **hybrid** indexes: BM25 (keyword) **+** vector.

## 2) Retrieval pipeline

1. **Query rewriting** (optional): expand acronyms, apply HyDE/synonyms from glossary.
2. **Hybrid search:** top‑k from keyword (BM25) and vector; combine via **RRF** or weighted sum.
3. **Rerank:** cross‑encoder (e.g., MiniLM) to rerank top‑50 into top‑5 passages.
4. **Filter by ACL/metadata:** enforce row/document‑level security (team, region, data\_class).
5. **Compose context:** select ~2–5 passages, deduplicate, keep total tokens under your model’s limit.

## 3) Generation pipeline

* **Prompt template (system):** “Answer only from the provided context. Cite sources. If not in context, say you don’t know.”
* **User message:** their question.
* **Context message:** concatenated passages (with doc\_id + snippet).
* **Output spec:** JSON schema (answer, citations[], confidence, used\_docs[]).

### Minimal Python skeleton (framework-agnostic)

python

SalinEdit

# Pseudocode: swap in your libs (LangChain, LlamaIndex, Haystack)

def retrieve(query, acl\_tags):

q2 = rewrite\_query(query) # optional

hits\_kw = bm25.search(q2, top\_k=50, filters=acl\_tags)

hits\_vec = vectordb.search(q2, top\_k=50, filters=acl\_tags)

merged = reciprocal\_rank\_fuse(hits\_kw, hits\_vec)[:50]

reranked = cross\_encoder\_rerank(q2, merged)[:5]

return reranked # [{text, doc\_id, score, meta}]

def generate\_answer(query, passages):

context = format\_passages(passages) # add doc ids + titles

prompt = render\_prompt(query=query, context=context)

out = llm.generate(prompt, response\_format=ANSWER\_SCHEMA) # enforce JSON

return out

def answer\_question(query, user\_acl):

ctx = retrieve(query, acl\_tags=user\_acl)

if not sufficient\_coverage(ctx): # score threshold / coverage check

return {"answer":"I don't know.", "citations":[], "confidence":0.2}

return generate\_answer(query, ctx)

## 4) Hallucination‑reduction strategies (practical)

* **Grounding & citations by design:** always pass only retrieved text in a separate context block; **require citations** (doc\_id + line).
* **Abstention:** if **relevance score < τ** or **coverage < threshold**, answer “I don’t know.”
* **Hybrid retrieval + reranking:** reduces missed evidence (recall) and off-topic context (precision).
* **Tight prompts:** “Do not guess. Quote exact lines when possible.”
* **Structured outputs:** JSON schema validation; reject if missing citations.
* **Context hygiene:** de‑duplicate, strip low-signal text, limit to top‑N passages.
* **Temporal filters:** prefer **latest** doc version; decay old docs.
* **Guardrails:** PII/secret scanning on outputs; allow only certain functions/knowledge.
* **Eval loop:** log “no context” and “low confidence” cases for curation/KB improvements.
* **Fine‑tuning (optional):** instruction‑tune on internal Q&A pairs + correct citations.

## 5) Evaluation (offline + online)

* **Retrieval metrics:** nDCG@k, Recall@k, MRR on labeled query→gold passage pairs.
* **Answer quality:**
  + **Groundedness/Faithfulness** (LLM-as-judge with rubric): does answer match context?
  + **Correctness** vs gold answers (Exact Match / F1 for extractive, rubric for abstractive).
  + **Citation accuracy:** does each citation support its sentence?
* **Safety:** red‑teaming prompts, data leakage checks, PII.
* **Latency & cost:** p50/p95 latency; $/1k queries.
* **Online:** A/B prompts, human feedback, ticket deflection %, CSAT.

## 6) Security & privacy

* **Network:** private VPC/VNet, no public endpoints; egress control.
* **Secrets:** KMS/Key Vault/Cloud KMS; per‑service IAM.
* **RBAC/ABAC:** propagate user identity; filter retrieval by ACL.
* **At-rest & in-flight encryption.**
* **Audit logs** for queries, retrieved docs, and returned citations.

## 7) Deploy & operate

* **Containerize** (Docker), deploy to **Kubernetes** or serverless.
* **Observability:** structured logs (prompt, context hash, citations), traces, dashboards.
* **Caching:** query & embedding cache (e.g., Redis); pre‑warm popular docs.
* **Index refresh:** CDC from DB; nightly re-embed changed docs; blue/green index swaps.

# Task 4 — Design an end‑to‑end data platform to the cloud

## 0) Target outcomes

* Reliable pipelines for **internal & external data**.
* Medallion‑style layers (**Bronze/Raw → Silver/Cleansed → Gold/Curated**).
* Warehouse/lakehouse for BI + **ML/LLM** workloads.
* Governed, secure, cost‑efficient, observable.

## 1) Reference architecture (map to any cloud)

**Ingestion**

* **Batch:** Files/DB extracts → Object storage (S3 / GCS / ADLS).
* **Stream:** Kafka/Kinesis/Pub/Sub/Event Hubs.

**Storage**

* **Data Lake:** Parquet/Delta/Iceberg in S3/GCS/ADLS (Bronze/Silver/Gold).
* **Warehouse/Lakehouse:** Redshift/BigQuery/Snowflake/Synapse/Databricks SQL.

**Processing**

* **Transform:** dbt (SQL ELT), Spark/Databricks, Beam/Dataflow, Glue/EMR.
* **Quality:** Great Expectations / Deequ; quarantine bad records.
* **Orchestration:** Airflow/Composer, Step Functions, Azure Data Factory, Dagster.

**Serving**

* **BI:** Looker/Power BI/QuickSight.
* **ML/LLM:** Vertex AI / SageMaker / Azure ML; feature store; RAG vector DB.
* **APIs:** FastAPI/Cloud Run/Lambda + API Gateway.

**Governance & Security**

* **Catalog/Lineage:** Data Catalog (GCP) / Purview (Azure) / Glue Data Catalog (AWS), OpenLineage.
* **Access:** IAM + Lake/Warehouse row/col policies; tokenization; DLP scanners.
* **Monitoring:** Cloud Logging/Monitoring + custom data SLAs (freshness, completeness).
* **IaC:** Terraform for everything; CI/CD with GitHub Actions/Azure DevOps/CodePipeline.

## 2) Cloud mappings (pick your stack)

| **Layer** | **GCP** | **Azure** | **AWS** |
| --- | --- | --- | --- |
| Object store | Cloud Storage | ADLS Gen2 | S3 |
| Stream | Pub/Sub | Event Hubs | Kinesis |
| Compute batch | Dataproc / Dataflow | Synapse / Databricks | EMR / Glue |
| Orchestration | Cloud Composer | Data Factory | Step Functions / MWAA |
| Warehouse | BigQuery | Synapse | Redshift / Athena |
| Lakehouse | BigLake/Iceberg/Delta via Dataproc | Delta Lake on ADLS | Delta/Iceberg on S3 |
| Catalog | Data Catalog | Purview | Glue Catalog |
| ML | Vertex AI | Azure ML | SageMaker |
| RAG vector | Matching Engine / AlloyDB PGVector | Azure AI Search / Cosmos DB + vectors | OpenSearch / Aurora PGVector / Pinecone |

## 3) Data lifecycle (step‑by‑step)

1. **Landing zone & foundations**
   * Org, projects/subscriptions, VPCs, subnets, Private Service Connect/Peering.
   * Centralized IAM & secrets management; audit logging enabled; budgets/quotas.
2. **Ingestion**
   * **Batch:** scheduled dbt seeds or extractor jobs (Airflow/ADF) → **Bronze** (as‑is).
   * **Stream:** CDC from OLTP via Debezium/DMS to Kafka/Kinesis → **Bronze** topics.
   * Apply **schema registry**, contracts, and PII tagging at ingress.
3. **Transformation**
   * **Silver:** standardize types, dedupe, conform dimensions, SCD2 for entities.
   * **Quality gates:** Great Expectations tests on each model (nulls, ranges, uniqueness).
   * **Gold:** business-ready marts (star schemas) for BI and ML features.
4. **Serving**
   * **BI:** publish approved datasets; row/column security for users.
   * **ML:** register features/models; enable RAG indexes built from curated docs/tables.
   * **APIs:** expose gold data via read‑only endpoints (rate-limited, cached).
5. **Governance**
   * Auto‑catalog new tables; classify PII; lineage from jobs to columns.
   * Policies: data retention, deletion, legal holds; data contracts with producers.
6. **Ops & FinOps**
   * SLAs/SLOs for pipelines; on‑call alerts (freshness, volume anomalies).
   * Cost dashboards per workspace/project; lifecycle policies (tiering, TTL).
   * Blue/green deploys for pipelines and warehouses; backfills via parameterized runs.

## 4) CI/CD & IaC (sketch)

* **Terraform** modules: network, storage, compute, secrets, catalog, warehouse, ML.
* **Data CI:** pre‑commit hooks, SQL lint, unit tests on dbt, data diffs (e.g., Elementary).
* **App CI:** build Docker images; scan vulnerabilities; deploy via ArgoCD/GitOps.

## 5) Evaluation & SLAs

* **Data quality KPIs:** freshness lag, % valid rows, duplication rate, schema drift events.
* **Pipeline KPIs:** success rate, p95 latency, backfill duration.
* **Consumption KPIs:** BI query latency, ML feature staleness, RAG answer groundedness.

## 6) Example dbt + Great Expectations pattern (concise)

sql

SalinEdit

-- models/silver/customers.sql

SELECT DISTINCT

id,

LOWER(email) AS email,

SAFE\_CAST(created\_at AS TIMESTAMP) AS created\_at,

country\_code

FROM {{ ref('bronze\_customers') }};

yaml

SalinEdit

# great\_expectations/expectations/customers.yml

expect\_column\_values\_to\_not\_be\_null:

column: email

expect\_column\_values\_to\_match\_regex:

column: email

regex: '^[^@]+@[^@]+\.[^@]+$'

## 7) RAG in the platform (where it fits)

* Build a **curated documents** zone in **Gold**, then index to your **vector DB**.
* Expose a **“/ask”** service (Vertex AI/SageMaker/Azure ML or custom) with the Task‑3 pipeline.
* Enforce **row‑level security** by propagating user identity to retrieval filters.

## Bonus — How to sketch this in Miro/draw.io quickly

* **Swimlanes:** Ingestion, Storage, Processing, Serving, Governance, Observability.
* **Shapes:**
  + Cylinders for storage (S3/GCS/ADLS, Warehouse).
  + Stacked docs for Bronze/Silver/Gold.
  + Gears for jobs (Airflow/ADF/Composer).
  + Magnifier for Catalog/Lineage.
  + Brain icon for LLM/RAG service.
* **Connectors:** solid arrows for data flows; dashed for control/metadata (catalog, lineage).
* **Callouts:** list SLAs, security policies, and cost controls next to each lane.

If you tell me your preferred cloud (GCP/Azure/AWS) and vector DB (e.g., pgvector vs Pinecone), I can tailor both the **RAG stack** and a **Terraform starter layout** to your environment.

**EXPLANATION**

### Vector Embeddings

A **vector embedding** is the process of converting data—like text, images, or audio—into a numerical representation called a vector. This vector is a list of floating-point numbers that captures the semantic meaning and characteristics of the original data. In simpler terms, it translates data into a format that a computer can more easily process and understand for tasks like finding relationships and similarities. For example, two words with similar meanings will have vectors that are mathematically "close" to each other in a high-dimensional space.

### Database Indexing

**Database indexing**, on the other hand, is the technique of organizing data to optimize the speed of data retrieval operations. It's similar to the index in the back of a book; instead of scanning the entire database, the index allows the system to quickly locate the relevant data. In the context of vector databases, this means organizing the vector embeddings in a way that makes similarity searches more efficient. Because vector embeddings are high-dimensional, they require specialized indexing techniques, such as Approximate Nearest Neighbor (ANN) algorithms, to enable fast searches without having to compare every single vector.

### The Relationship

In essence, **vector embeddings are the data being stored and indexed, while indexing is the method used to organize and search that data efficiently.** The embeddings are created first, and then the database uses various indexing techniques to store and manage them for rapid similarity searches. The indexing is a process that operates on the vector embeddings, but the embeddings themselves are not a part of the indexing process.

This video provides a straightforward explanation of vector databases, embeddings, and indexing. [Vector Databases simply explained! (Embeddings & Indexes)](https://www.youtube.com/watch?v=dN0lsF2cvm4" \t "_blank)

A **retrieval pipeline** is basically the “find the right stuff” part of a Retrieval-Augmented Generation (RAG) system.  
It’s the sequence of steps that takes the **user’s question** and finds the **most relevant internal documents or passages** before sending them to the LLM for answering.

Think of it like the “research assistant” phase before the “writer” (LLM) does its job.  
If retrieval is poor, the LLM either **hallucinates** or **answers incorrectly**.

## The steps you quoted

### 1. **Query rewriting** (optional but helpful)

Purpose: Make the user’s question easier for the search engine to understand.  
Techniques:

* **Expand acronyms:**  
  If the user writes “ROI of CRM in FY23”, rewrite to “return on investment of customer relationship management in fiscal year 2023”.
* **Synonym expansion:**  
  If your glossary says “car” ≈ “automobile”, rewrite to include synonyms for better matching.
* **HyDE (Hypothetical Document Embeddings):**  
  Generate a fake but plausible answer with the LLM, embed it, and use that embedding to search.  
  → This often helps when the question is very short or vague, because the hypothetical answer captures more context.
* **Spelling corrections / normalization:**  
  Fix typos, unify date formats, expand “Q1” → “first quarter”.

###### ChatGPT bilang:

A **retrieval pipeline** is basically the “find the right stuff” part of a Retrieval-Augmented Generation (RAG) system.  
It’s the sequence of steps that takes the **user’s question** and finds the **most relevant internal documents or passages** before sending them to the LLM for answering.

Think of it like the “research assistant” phase before the “writer” (LLM) does its job.  
If retrieval is poor, the LLM either **hallucinates** or **answers incorrectly**.

## The steps you quoted

### 1. **Query rewriting** (optional but helpful)

Purpose: Make the user’s question easier for the search engine to understand.  
Techniques:

* **Expand acronyms:**  
  If the user writes “ROI of CRM in FY23”, rewrite to “return on investment of customer relationship management in fiscal year 2023”.
* **Synonym expansion:**  
  If your glossary says “car” ≈ “automobile”, rewrite to include synonyms for better matching.
* **HyDE (Hypothetical Document Embeddings):**  
  Generate a fake but plausible answer with the LLM, embed it, and use that embedding to search.  
  → This often helps when the question is very short or vague, because the hypothetical answer captures more context.
* **Spelling corrections / normalization:**  
  Fix typos, unify date formats, expand “Q1” → “first quarter”.

### 2. **Hybrid search**

Purpose: Use two search methods together so you don’t miss relevant results.

#### Two main methods:

1. **Keyword search** (BM25 or similar)
   * Works like a classic search engine.
   * Scores based on matching words and their frequency/importance.
   * Great for exact phrase matches and catching terms not well-represented in embeddings.
2. **Vector search** (semantic search)
   * Uses embeddings to find texts with similar **meaning**, not just exact words.
   * Great for paraphrased or conceptually similar matches.

#### Combining them

* **RRF (Reciprocal Rank Fusion):**  
  A method that merges two ranked lists (keyword and vector) by giving each result a score based on its position in each list.  
  Formula: score = Σ (1 / (k + rank)) (k is usually ~60 to avoid over-biasing top results).
* **Weighted sum:**  
  Normalize both scores (e.g., to 0–1), then combine like:  
  final\_score = 0.4 \* keyword\_score + 0.6 \* vector\_score.

Why hybrid?

* Keyword search is **precise** (low recall for paraphrases).
* Vector search is **semantic** (better recall, but can bring in noise).
* Together, you get **both recall and precision**.