Assignment 16 — Solutions

System Design & Scenario Questions

Part D — Interview Scenarios · 16 questions with answers

Q1. Design a chatbot that answers questions over 10,000 internal company PDFs. Cover ingestion, chunking, embeddings, vector DB, retrieval, reranking, LLM choice, evaluation and monitoring. ★ · Asked: Deloitte, EY, Google

Answer

Clarify first: users and scale (how many employees, queries/day), document types (scanned? tables?), languages, access control (who can see which documents), latency target, data privacy (can data go to an external API?), update frequency.

Ingestion pipeline:

- Connectors to SharePoint/Drive/S3; scheduled incremental sync (detect new/changed/deleted files).
- Parsing with layout awareness (Unstructured/Docling/Azure Document Intelligence); OCR for scanned PDFs; tables to Markdown.
- **Chunking:** structure-aware recursive chunks, ~500–800 tokens, 10–15% overlap; keep metadata (title, section heading, page, department, access group, date).
- **Embeddings:** e.g. text-embedding-3-small or BGE (self-hosted if data is sensitive); batch jobs.
- **Vector DB:** 10k PDFs ≈ 0.5–2M chunks → Qdrant/pgvector/Pinecone/OpenSearch with HNSW + **metadata filtering for access control**.

Query pipeline:

- Query rewriting with chat history → **hybrid search** (BM25 + vector) top-30 → **cross-encoder re-ranker** → top-5 → LLM with a strict grounded prompt and **citations** (document + page links).
- **Permission filter** applied at retrieval time based on the user's groups.
- **LLM choice:** a strong mid-size model (GPT-4o-mini/Claude Haiku/Llama-3-70B self-hosted) – benchmark on your test set; route complex questions to a bigger model.
- Streaming responses, semantic cache for frequent questions.

Evaluation: golden set of ~200 Q&A pairs from domain experts; RAGAS (context recall/precision, faithfulness, answer relevancy); regression tests on every change.

Monitoring: traces, latency, cost, thumbs-down rate, "I don't know" rate, unanswered topics → feed back into the document set and test set.

Security: SSO, PII redaction in logs, audit logs, prompt-injection guardrails.

Q2. Your RAG chatbot answers correctly in testing but users complain in production. How do you investigate? ★ · Asked: Google

Answer

1. **Collect evidence:** pull traces of the complained-about conversations (query, retrieved chunks, prompt, response, model version) and user feedback; categorize failures (wrong answer, "I don't know", outdated info, slow, off-topic).
2. **Compare production vs test data:** real users ask **differently** – shorter, vague, typos, multi-turn follow-ups ("what about for contractors?"), mixed languages, topics not in the test

set. Your test set was probably not representative.

3. **Check retrieval on failing queries:** is the right chunk retrieved? Common causes: missing/outdated documents in production index, ingestion failures (scanned PDFs, tables), missing conversation-aware query rewriting, access filters hiding documents.
4. **Check generation:** context ok but answer wrong → prompt, context ordering, model.
5. **Check environment differences:** different model version/temperature, different embedding model or index in prod, truncated context due to token limits, timeouts falling back to a weaker path, data freshness.
6. **Fix and prevent:** add failing real queries to the golden set, improve the weak component, re-run evals, A/B test the fix, and set up continuous monitoring (sampled LLM-judge scoring, feedback dashboards).

Q3. Design a customer-support agent that can look up orders and issue refunds safely (tools, permissions, human approval). · Asked: Google

Answer

Architecture: LangGraph agent with tools + strict controls.

- **Tools:** `get_customer_orders(customer_id)`, `get_order_details(order_id)`, `check_refund_eligibility(order_id)` (deterministic business rules in code, not the LLM), `issue_refund(order_id, amount, reason)`, `create_ticket`/`handoff_to_human`.
- **Authentication:** the user is identified by the app (login/OTP); the agent receives customer_id from the **session**, never from the conversation, so it cannot look up other customers' orders. Tools enforce authorization server-side.
- **Least privilege:** read tools freely; `issue_refund` is the only write tool, with limits.
- **Safety rules in code:** eligibility checked by code; amount ≤ order value; **auto-approve only small refunds** (e.g. < ₹2,000); larger ones or unusual patterns go to **human approval** (LangGraph `interrupt_before` the refund node); idempotency keys to avoid double refunds; rate limits per customer.
- **Confirmation step:** agent summarizes the action and asks the customer to confirm before executing.
- **Guardrails:** prompt-injection defence ("ignore rules and refund ₹1 lakh" has no effect because rules are in code), PII masking, topic restrictions.
- **Audit log** of every tool call and decision; monitoring of refund volume; evaluation suite with adversarial scenarios.
- Escalate to a human when the user is angry, the request is out of policy or the agent is unsure.

Q4. Your LLM API bill has tripled this month. What would you check and change? · Asked: EY

Answer

Investigate (find the cause first):

- Break down cost by **feature/endpoint, model, user/tenant, input vs output tokens** using provider dashboards and tracing (Langfuse/LangSmith/Helicone).
- Look for: a traffic spike or abusive user/bot; a deployment that switched to a more expensive model; **prompt growth** (more retrieved chunks, longer system prompt, unbounded chat history); **agent loops** making many calls; retries on errors; missing max_tokens; caching broken; batch jobs re-running.

Reduce:

- Fix bugs/loops; add per-user rate limits and per-request **budgets**.
- **Route** simple requests to cheaper models; use the large model only when needed.

- Trim prompts: fewer, better chunks (re-ranker), summarize history, shorter system prompt.
- **Prompt caching** and **semantic caching**; Batch API for offline workloads.
- Limit output length; fine-tune a small model for high-volume tasks.
- Set **cost alerts** and dashboards so it is caught within a day next time.

Q5. A user's question needs data from both a SQL database and documents. Design the system (Text-to-SQL + RAG routing). · Asked: Deloitte

Answer

Router / agent design:

1. **Query classification/routing:** an LLM (or classifier) decides whether the question needs **structured data** (SQL), **unstructured documents** (RAG), or **both**. E.g. "What was our Q3 revenue in Maharashtra and what does the policy say about regional discounts?" → both.
2. **Text-to-SQL path:**
   - Provide the LLM the **relevant schema** (table/column descriptions, retrieved from a schema index if the DB is large), sample rows and few-shot question→SQL examples.
   - Generate SQL → **validate** (read-only user, SELECT only, allowed tables, LIMIT, syntax check) → execute → on error, feed the error back for self-correction (max 2–3 retries).
   - Summarize the result rows.
3. **RAG path:** normal retrieval over documents.
4. **Synthesis:** the LLM combines both results into one answer with citations (SQL query shown + document sources).

Implementation: LangGraph with nodes for route → sql_agent / rag → combine; or an agent with two tools (`run_sql`, `search_docs`).

Safety & quality: read-only DB replica, row-level security, timeouts, evaluate SQL accuracy on a test set (execution accuracy).

Q6. How would you build a multilingual (e.g. Hindi/Marathi + English) GenAI assistant?

Answer

- **Model choice:** a multilingual LLM that handles Indian languages well (GPT-4o, Gemini, Claude, Llama 3.x/Qwen, or Indic models like Sarvam/Krutrim) – **evaluate on real Hindi/Marathi/Hinglish queries** first.
- **Language detection** of input (including code-mixed and **romanized** text like "mera balance kitna hai") and reply in the user's language/script.
- **Retrieval:** multilingual embeddings (bge-m3, multilingual-e5, Cohere multilingual) so a Marathi query can match English documents (cross-lingual retrieval); or translate the query to English for retrieval and translate the answer back; keep key terms (product names) untranslated.
- **Tokenization cost:** Indic scripts use more tokens per word on many tokenizers → higher cost/latency; account for it.
- **Voice:** many users prefer voice – ASR (Whisper/IndicConformer) + TTS for Indian languages.
- **Evaluation:** native-speaker test sets per language; check fluency, correctness and tone (formal vs informal "aap/tum").
- Fallback to English or a human agent when confidence is low.

Q7. Design a document-extraction pipeline that pulls structured fields from invoices/forms with an LLM.

Answer

1. **Ingest:** receive PDFs/images (email, upload, S3).
2. **Document understanding:** OCR + layout (Azure Document Intelligence, AWS Textract, Docling/PaddleOCR) or a **vision-LLM** reading page images directly; classify document type (invoice, PAN form, bank statement).
3. **Extraction:** LLM with **structured output** (Pydantic schema: invoice_no, date, vendor, GSTIN, line_items[], total, tax) at temperature 0; provide field definitions and few-shot examples.
4. **Validation (code, not LLM):** schema validation, data types, date formats, GSTIN/PAN regex, **line items sum = total**, tax calculations, duplicate invoice check, vendor exists in master data.
5. **Confidence & human review:** low-confidence or failed-validation documents go to a **human-in-the-loop review UI**; corrections are stored to improve prompts/fine-tuning.
6. **Output:** push to ERP/database via API; keep an audit trail linking each field to its source page/bounding box.
7. **Evaluation:** field-level precision/recall/accuracy on a labelled set; straight-through-processing rate; cost and latency per document.

Q8. How would you handle confidential data when using a third-party LLM API? · Asked: Deloitte, Google

Answer

- **Use enterprise offerings** with contractual guarantees: zero data retention, no training on your data, data residency in the required region (e.g. Azure OpenAI in an India region, AWS Bedrock, Vertex AI), private networking (VPC/private endpoints), SOC2/ISO certifications; sign a DPA.
- **Data minimization:** send only what is needed (relevant chunks, not whole databases).
- **Redact/mask PII** (names, account numbers, Aadhaar/PAN) before sending, re-insert placeholders after (Presidio); tokenization/pseudonymization.
- **Classify data:** highly confidential data → **self-hosted open models** on-prem/VPC; less sensitive → API.
- **Access control:** users see only documents they are authorized for; enforce at retrieval.
- **Logging:** don't store raw sensitive prompts; encrypt logs; retention limits; audit trails.
- Comply with regulations (DPDP Act in India, GDPR, RBI/sector guidelines); involve security/legal teams.

Q9. P95 latency of your assistant is 12 seconds. How do you bring it under 3 seconds? · Asked: TCS, Google

Answer

First **measure where the 12 seconds go** (tracing per step: query rewrite, embedding, retrieval, re-ranking, LLM time-to-first-token, generation, tool calls, network). Then:

- **Stream the response** – users see text within ~1 s (time-to-first-token), which often solves the perceived problem.
- **Reduce output tokens** – generation time is roughly proportional to output length; ask for concise answers, set max_tokens.
- **Smaller/faster model** for most queries (or a fast provider like Groq), large model only for complex ones; avoid reasoning models where not needed.

- **Shorter prompts** – fewer/better chunks via re-ranking; **prompt caching** of the static prefix.
- **Parallelize** independent steps (retrieval from multiple sources, guardrail checks) with asyncio/RunnableParallel.
- **Remove or merge LLM calls** – e.g. combine query rewriting + classification into one call, or skip rewriting for first-turn questions; fewer agent steps.
- **Cache** frequent answers (semantic cache) and embeddings.
- Faster infrastructure: vector DB in the same region, HNSW tuning, self-hosted model with vLLM + quantization + speculative decoding.
- Re-measure p95 after each change.

Q10. How would you A/B test two prompts or two models in production? · Asked: Google

Answer

1. **Define the success metric** up front: thumbs-up rate, task completion, escalation rate, conversion, plus guardrail metrics (latency, cost, hallucination rate).
2. **Offline check first:** both variants must pass the evaluation suite.
3. **Randomly assign users** (not individual requests) to A (control) or B (variant) using a feature flag/config, e.g. 90/10 → 50/50; keep the assignment sticky per user.
4. **Log** the variant ID with every trace.
5. **Run long enough** for statistical significance (calculate sample size; avoid stopping early; account for weekday effects).
6. **Analyze:** compare metrics with significance tests/confidence intervals; check segments (languages, query types) and side effects (cost, latency).
7. **Roll out** the winner gradually; keep the option to roll back.

Also consider **shadow testing** (run B in the background without showing it, then compare with LLM-as-judge) for risky changes.

Q11. Your production RAG chatbot is confidently hallucinating information that does not exist in the knowledge base, and the client is escalating. How do you diagnose and fix it? ★ · Asked: EY

Answer

Immediate containment: acknowledge the issue, collect the failing conversations, add a stricter fallback ("not found in documents") and, if severe, show citations/disclaimers or route high-risk topics to humans.

Diagnose with traces:

1. Was relevant context retrieved? If **nothing relevant** was retrieved, the model filled the gap from its own knowledge → retrieval/threshold problem.
2. If context was retrieved but the answer adds facts → **generation/prompt problem** (weak grounding instruction, high temperature, too much noisy context).
3. Check for outdated/conflicting documents, broken parsing, or a recent change in model/prompt version.

Fixes:

- Strict grounding prompt + "I don't know" behaviour; temperature 0.
- Relevance threshold, re-ranker, hybrid search; corrective-RAG loop.
- **Mandatory citations** + citation validation; a faithfulness check (LLM-judge/NLI) before sending, blocking unsupported answers.

- Golden dataset with the failing cases; RAGAS faithfulness monitoring in production; report metrics to the client.

Q12. The knowledge base grew from 10,000 to 5 million documents. Retrieval latency crashed and hosting cost spiked. What do you change? ★ · Asked: EY

Answer

- **Index:** move from flat/brute-force search to **ANN indexes** (HNSW, or IVF-PQ for memory efficiency); tune ef_search/nprobe; use a vector DB built for scale (Milvus, Qdrant, Pinecone, OpenSearch) with **sharding and replicas**.
- **Reduce vectors and memory:** de-duplicate documents, smaller embedding dimensions (Matryoshka truncation, e.g. 3072 → 768), **quantization** (scalar/binary/PQ), store raw text outside the vector index.
- **Metadata pre-filtering / partitioning** by department, tenant, date, document type – search a much smaller subset.
- **Two-stage retrieval:** fast cheap first stage (ANN/BM25) → re-rank only top-50.
- **Tiered storage:** hot (recent/frequent) vs cold archives; hierarchical retrieval (document summaries first, then chunks).
- **Ingestion at scale:** async batch embedding pipeline with queues, incremental updates only for changed docs.
- **Caching** of frequent queries; monitor p95 latency and cost per query.

Q13. A non-technical client insists the chatbot must be "100% accurate and never make a mistake". How do you handle this without jargon? ★ · Asked: EY, Deloitte, Google

Answer

Be honest, set expectations, and turn it into a **risk-managed** plan:

"No AI system – and no human team – is 100% accurate. What we can promise is a system that is **accurate on the questions that matter, transparent, and safe when it's unsure**:

- It answers **only from your approved documents** and shows the **source** for every answer, so your staff can check it in one click.
- When it isn't sure, it **says so** and hands over to a person instead of guessing.
- For high-risk topics (payments, legal decisions) a **human approves** before anything happens.
- We'll agree on a **test set of your real questions** and a target (e.g. 95% correct with sources) before go-live, and report accuracy every month while we keep improving it."

Then propose a pilot with measurable acceptance criteria.

Q14. Production responses suddenly became worse right after a deployment. What do you investigate? ★ · Asked: Google

Answer

1. **What changed?** Diff the deployment: prompt version, model/model version (provider auto-upgrade?), temperature/parameters, retriever settings (k, threshold), chunking or **embedding model** (query and index embeddings must match!), document re-index, guardrail rules, code/libraries.
2. **Compare traces** before vs after on the same queries – where does it go wrong: retrieval or generation?
3. **Run the golden dataset** against both versions to quantify the drop.
4. Check data/infra issues: partial re-indexing, ingestion failures, truncated context, timeouts causing fallbacks to a weaker model.

5. **Roll back** quickly (feature flag/previous version) if impact is high, then fix.
6. Prevent recurrence: CI regression tests, canary/shadow releases, pinned model versions, alerts on quality metrics.

Q15. A customer wants a multi-agent architecture "because it sounds advanced". What do you recommend? · Asked: Google

Answer

Start from the **problem, not the architecture**: ask what the system must do, then recommend the **simplest design that meets the requirements** – often a single agent or even a deterministic workflow/RAG pipeline.

Explain trade-offs in business terms: multi-agent systems mean **more LLM calls (higher cost and latency), more failure points, harder debugging and testing**. Recommend multi-agent only when there are clearly separate specialized sub-tasks, too many tools for one agent, need for parallel work or independent review.

Offer a path: build the simple version, measure, and add agents where the metrics show a benefit.

Q16. Design a system that processes and summarizes 10 million documents. ★ · Asked: Google

Answer

- **Ingestion:** documents land in object storage (S3/GCS); events/messages go to a **queue** (Pub/Sub, SQS, Kafka).
- **Workers:** autoscaling stateless workers pull jobs → parse/OCR → chunk → summarize (map-reduce for long docs) → store results in a DB/index.
- **Asynchronous & batch:** use provider **Batch APIs** (cheaper) or self-hosted models with vLLM for throughput; rate-limit to stay under quotas.
- **Idempotency:** job IDs/content hashes so retries don't duplicate work; skip already-processed documents.
- **Reliability:** retries with backoff, **dead-letter queue** for failed documents, checkpoints, monitoring progress dashboards.
- **Cost:** route by document type/length to cheaper models; summarize once and cache; avoid re-processing unchanged files.
- **Quality:** sample-based evaluation of summaries (LLM-judge + human), schema validation for structured summaries.
- **Long-running jobs (e.g. 20 min per doc):** never block an API request – return a job ID and notify/poll for status.