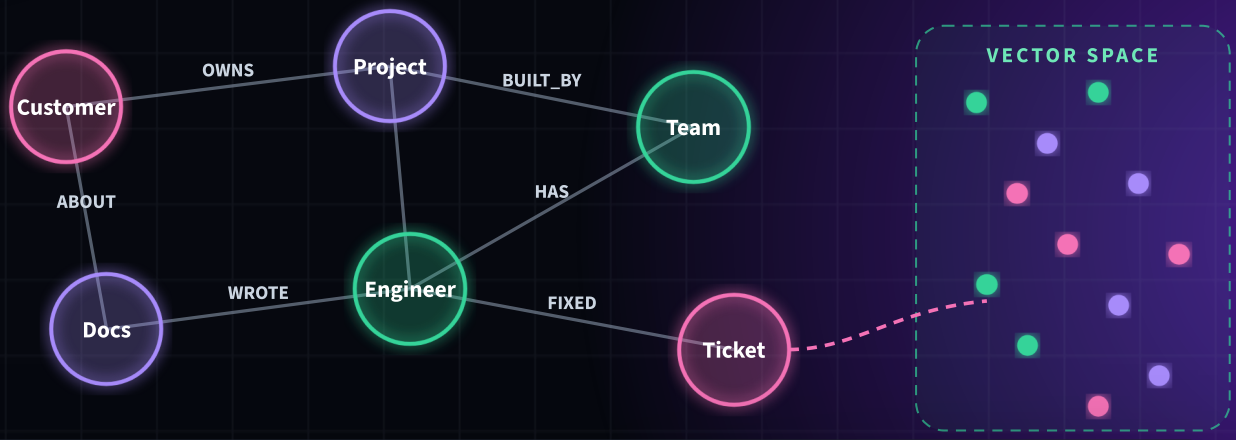
JOB-READY SERIES · JOB-SWITCH EDITION

RAG · GRAPHRAG · AGENTS

AI Engineering

RAG, GraphRAG & Enterprise Agents on PostgreSQL

Build what every company wants now: RAG with pgvector, knowledge graphs with Apache AGE, GraphRAG, and permission-aware enterprise agents with MCP. Real Python code, labs, scenarios, projects and 120+ interview questions.



psql: knowledge_graph

```
SELECT * FROM cypher('kg', $$
  MATCH (p:Person)-[:BUILT]→(:Project)←[:ABOUT]-(d:Doc)
  RETURN p.name, d.title $$) AS (who agtype, doc agtype);
-- + ORDER BY embedding ⟺ $query  (pgvector)  ⇒ grounded answer
```

LLMs · Embeddings · PostgreSQL · pgvector · Hybrid Search · RAG · Re-ranking · RAG Evals · Knowledge Graphs · Apache AGE · Cypher · GraphRAG · AI Agents · LangGraph · MCP · Guardrails

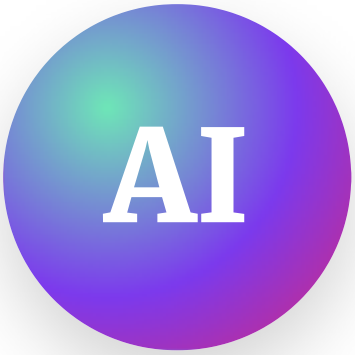
WRITTEN BY

Shreyash Tripathi

20 Chapters · Python + Postgres Labs · Interview Questions

Edition 2026

JOB-READY SERIES · JOB-SWITCH EDITION



AI Engineering

RAG, GraphRAG and Enterprise Agents on PostgreSQL

Retrieve | Connect the graph | Reason with agents | Evaluate | Ship to production

WRITTEN BY

Shreyash Tripathi

Job-Switch Edition, 2026

© Shreyash Tripathi. All rights reserved.

Preface

Every company now wants three things from AI: answers grounded in its own documents, a knowledge graph that connects its people, projects, customers and decisions, and agents that can act on that knowledge safely. The engineers who can build these systems are some of the most sought-after people in the industry right now.

This book teaches the whole stack on one strong, scalable foundation: PostgreSQL. You will store embeddings with pgvector, combine them with full-text search, build knowledge graphs inside the same database with Apache AGE, run GraphRAG over them, and put permission-aware agents on top, connected to company tools through MCP. Frameworks change every few months, so we learn the concepts in plain Python first and then show where LangChain, LlamaIndex and LangGraph fit.

Like every book in this series, it is a textbook: numbered chapters, simple language, a check with answers after every topic, labs, Interview Corners, At Work stories, real scenarios, and three portfolio projects. Build them, put them on GitHub, and you will have exactly what AI engineering interviews ask for.

Shreyash Tripathi

Features of Every Chapter

🎯 **INTERVIEW CORNER**

Questions interviewers really ask, with model answers.

💼 **AT WORK**

What really happens on the job and on call.

🧪 **HANDS-ON LAB**

Do it yourself with Docker, Python and PostgreSQL.

⚠️ **COMMON MISTAKE**

Traps that break production and fail interviews.

💡 **DO YOU KNOW?**

Real facts from the industry.

😄 **FUNNY SIDE**

Because hallucinations need jokes too.

✅ **CHECK YOUR PROGRESS**

Short questions with answers after every topic, plus exercises, scenario questions, a mini project and a full answer key at the end of every chapter.

Contents

PART I: FOUNDATIONS

1 The AI Engineer Job and Roadmap 6
2 LLMs for Developers 24
3 Embeddings and Vector Search 44
4 PostgreSQL for AI Engineers 63

PART II: RETRIEVAL-AUGMENTED GENERATION

5 RAG Architecture End to End 84
6 Ingestion, Parsing and Chunking 103
7 pgvector in Depth and Hybrid Search 123
8 Advanced RAG Techniques 142
9 Evaluating and Debugging RAG 160

PART III: KNOWLEDGE GRAPHS AND GRAPHRAG

10 Knowledge Graphs Fundamentals 180
11 Apache AGE: Graphs inside PostgreSQL 198
12 Building Knowledge Graphs with LLMs 221
13 GraphRAG 242
14 Hybrid Retrieval: Vectors and Graphs in One Postgres 260

PART IV: AGENTS AND ENTERPRISE AI

15 AI Agents and Tool Calling 280
16 Frameworks and Protocols: LangChain, LlamaIndex, LangGraph and MCP 299
17 Multi-Agent Systems and Enterprise Agents 315
18 Production AI Systems 333

PART V: GET THE JOB

19 AI Portfolio Projects and Real Scenarios 352
20 AI Engineering Interview Question Bank 370

APPENDICES

A SQL, pgvector and Cypher Cheat Sheet 390
B Code Templates 397
C Glossary 413
D Job Switch Toolkit 417

PART I

Foundations

The AI engineer job, LLMs, embeddings and PostgreSQL for AI.

CHAPTER 1

The AI Engineer Job and Roadmap

"Training a model is like building a car. Most companies just want someone who can drive it to the office without crashing."

After studying this chapter, you will be able to

- Explain how an AI engineer differs from an ML engineer and a data scientist.
- Describe why companies are building RAG systems, knowledge graphs and enterprise agents right now.
- Draw the full AI engineering stack, from the LLM to production monitoring.
- Follow a 90-day plan to switch into an AI engineer role.
- Plan three portfolio projects and write resume lines that get shortlisted.
- Prepare for the four common interview rounds for AI engineer jobs.

Think of a big Indian wedding. The farmer grows the rice, the miller cleans it, and the caterer turns it into biryani for 800 guests. The guests only care that the biryani is hot, tasty and on time. In the AI world, research labs are the farmers: they train huge language models. ML engineers are the millers who prepare and tune models. The **AI engineer** is the caterer. You take a ready model, mix it with company data, add the right spices (prompts, retrieval, tools), and serve something useful to real users, every day, without food poisoning.

This chapter covers the job itself, what employers want, and a 90-day switch plan. We also set up the lab that every later chapter uses.

1.1 What Does an AI Engineer Actually Do?

An AI engineer builds products on top of large language models (LLMs). You usually do not train the model; it comes from a provider such as OpenAI or Anthropic (Claude), or is an open-source model such as Llama or Qwen. Your job is everything around it: feeding it the right data, prompting it, checking answers, connecting it to company systems, and keeping it fast, cheap and safe.

A typical week: design a retrieval pipeline over 2,000 HR policy pages, store vectors in pgvector, discover that chunk boundaries cut a leave-policy table in half, fix it, prove the fix with a 60-question evaluation set, then add logging and cost tracking and ship to 200 internal users.

1.1.1 AI Engineer vs ML Engineer vs Data Scientist

These titles overlap, but the centre of gravity differs.

Aspect	Data Scientist	ML Engineer	AI Engineer
Main question	What does the data tell us?	How do we train and serve this model reliably?	How do we build a useful product with an existing LLM?
Daily work	Analysis, statistics, dashboards, experiments	Feature pipelines, training, model serving, MLOps	Prompts, RAG, agents, tool integrations, evaluation, APIs
Typical tools	SQL, pandas, notebooks, BI tools	PyTorch, Spark, Kubernetes, feature stores	LLM APIs, embeddings, pgvector, graph DBs, FastAPI
Maths needed	High (statistics)	High (ML theory)	Moderate (vectors, similarity, basic probability)
Output	Insights and reports	Trained models and pipelines	Working AI features inside products
Best entry path	Analytics, statistics	Data engineering, research	Backend or full-stack development

The good news for software developers: the AI engineer role sits closest to backend engineering. If you can build an API, write SQL and debug a production issue, you already have about half the skills.

🔑 **KEY POINT**

An AI engineer is first a software engineer. The model is one component in your system, like a database or a payment gateway. It just happens to be a component that sometimes makes things up.

🎯 **INTERVIEW CORNER**

Q: How is your role different from an ML engineer? **A:** An ML engineer mostly trains, tunes and serves models. I mostly build applications on top of pre-trained LLMs: retrieval, prompting, tool calling, evaluation and production reliability. I fine-tune only when prompting and retrieval are not enough.

Q: Do you need to know how transformers work? **A:** I need a working understanding: tokens, attention at a high level, context windows and next-token prediction. That helps me reason about cost, latency and failure modes. I do not need to derive backpropagation to ship a RAG system.

✅ **CHECK YOUR PROGRESS 1.1**

1. **Name two tasks an AI engineer does that a data scientist usually does not.**

   Answer: Building retrieval pipelines that feed company data to an LLM, and wiring the LLM to tools and APIs (agents) with production logging and evaluation.

2. **Why is backend development a good starting point for AI engineering?**

   Answer: Most AI engineering work is API design, data pipelines, databases, error handling and deployment. These are backend skills; the LLM is an added component.

1.2 Why Companies Are Building RAG, Knowledge Graphs and Agents Now

LLMs are trained on public data up to a cutoff date. They do not know your leave policy or yesterday's tickets. Three patterns bring company facts to the model.

1.2.1 RAG: Retrieval-Augmented Generation

RAG means: before asking the LLM, search your own documents for relevant pieces and paste them into the prompt. The LLM then answers using those pieces. It is like an open-book exam. The student (LLM) is smart, but the book (your data) has the facts.



Figure 1.1: The basic RAG flow

Why companies like RAG: no training, fresh data (just re-index) and answers that cite sources. Uses include knowledge assistants, support bots and compliance search.

1.2.2 Knowledge Graphs and GraphRAG

Plain vector search finds text that "sounds similar". It struggles with questions that need relationships: "Which suppliers of Plant B also supply our competitor's vendor?" A **knowledge graph** stores entities (people, products, suppliers) as nodes and relationships as edges. **GraphRAG** combines graph traversal with vector search so the LLM gets connected facts, not just similar paragraphs. In this book we use **Apache AGE**, a PostgreSQL extension that lets you run Cypher graph queries inside the same database as your vectors.

1.2.3 Enterprise Agents

An **agent** is an LLM that can decide to call tools (search, SQL, an API, a calculator) in a loop until a task is done. Instead of just answering "what is the refund status?", an agent can look up the order, check the refund API and draft the reply. Enterprises want agents for ticket triage, report generation, data lookup and workflow automation. They also fear agents, because an agent with write access can do real damage. That is why guardrails and evaluation are a big part of the job.

> 💡 **DO YOU KNOW?**
>
> The term "Retrieval-Augmented Generation" comes from a 2020 research paper by Patrick Lewis and colleagues at Facebook AI Research (now Meta AI). The idea became mainstream only after ChatGPT launched in late 2022, when every company suddenly wanted "ChatGPT on our documents".

🎯 **INTERVIEW CORNER**

Q: Why not just fine-tune the model on company documents? **A:** Fine-tuning is good for teaching style or format, not for injecting facts that change often. It is costly to repeat, hard to cite, and the model can still mix up facts. RAG keeps facts outside the model, so updates are just re-indexing and answers can show sources.

Q: When would you add a knowledge graph to a RAG system? **A:** When questions need relationships across documents, like "who reports to whom" or "which contracts mention suppliers of this part". If evaluation shows vector search missing multi-hop facts, a graph helps.

Q: What makes enterprise agents risky? **A:** They act, not just talk. A wrong tool call can send an email or update a record. You need limited permissions, human approval for risky actions, step limits and full logging.

✅ **CHECK YOUR PROGRESS 1.2**

1. **Explain RAG using an exam analogy.**
   Answer: RAG is an open-book exam. The LLM is the student who can write well; the retrieved chunks are the book pages that contain the actual facts.
2. **Give one question that plain vector search may fail but a graph can answer.**
   Answer: "Which projects are led by people who report to Priya?" It needs two hops: reports-to, then leads.

1.3 The Full Stack Map

Here is the stack we will build across this book. Each layer gets one or more chapters.

```
+-------------------------------------------------------------+
|  USERS: web app, Slack bot, internal portal                 |
+-------------------------------------------------------------+
|  API LAYER: FastAPI, auth, rate limits, streaming           |
+-------------------------------------------------------------+
|  ORCHESTRATION: prompts, agents, tool calls, guardrails     |
+------------------+----------------------+-------------------+
|  LLM             |  RETRIEVAL           |  TOOLS            |
|  call_llm()      |  vector + keyword    |  SQL, APIs,       |
|  OpenAI / Claude |  + graph traversal   |  search, email    |
|  / open models   |  + reranking         |                   |
+------------------+----------+-----------+-------------------+
|  DATA: PostgreSQL + pgvector (vectors) + Apache AGE (graph) |
|  embed() for chunks, metadata tables, documents             |
+-------------------------------------------------------------+
|  EVALUATION: test sets, LLM-as-judge, retrieval metrics     |
+-------------------------------------------------------------+
|  PRODUCTION: logging, tracing, cost, caching, CI/CD, Docker |
+-------------------------------------------------------------+
```

Figure 1.2: The AI engineering stack used in this book

1.3.1 Layer by Layer

- **LLM:** hidden behind a small `call_llm()` helper so you can switch between OpenAI, Anthropic Claude or an open-source model without rewriting your app.
- **Embeddings:** `embed(text)` turns text into a vector. Similar meanings give nearby vectors.
- **pgvector:** stores vectors in a `vector(N)` column and finds nearest neighbours with operators like `⟺` (cosine distance).
- **Apache AGE:** graph tables and Cypher queries in the same PostgreSQL.
- **Retrieval, agents, evaluation, production:** picking chunks, tool loops, measuring correctness, and keeping it fast, cheap and safe.

> 🔑 **KEY POINT**
>
> Why PostgreSQL for everything? Vectors, graph and normal tables in one database means easy joins, one backup plan and fewer moving parts. Dedicated vector databases are fine too; the concepts transfer.

> 😄 **FUNNY SIDE**
>
> Every AI startup pitch in 2023: "We are building a revolutionary AI platform." Every AI startup codebase in 2023: a prompt, a for-loop and a try/except that retries when the JSON is broken. This book will teach you the prompt, the for-loop and, most importantly, why the JSON broke.

1.3.2 Hands-On: Your Lab Database

Every lab uses one Docker container with PostgreSQL, pgvector and Apache AGE. We start from the official Apache AGE image and compile pgvector into it. Package names depend on the PostgreSQL version in the image, so check the docs for your version if the build fails.

```
# Dockerfile
FROM apache/age:latest
USER root
RUN apt-get update \
 && apt-get install -y --no-install-recommends \
      build-essential git postgresql-server-dev-$PG_MAJOR \
 && git clone --branch v0.8.0 https://github.com/pgvector/pgvector.git /tmp/pgvector \
 && cd /tmp/pgvector && make && make install \
 && rm -rf /tmp/pgvector /var/lib/apt/lists/*
USER postgres
```

```
# docker-compose.yml
services:
  db:
    build: .
    container_name: aidb
    environment:
      POSTGRES_USER: ai
      POSTGRES_PASSWORD: ai
      POSTGRES_DB: aidb
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data
volumes:
  pgdata:
```

```
$ docker compose up -d --build
[+] Running 2/2
 ✓ Volume "lab_pgdata"  Created
 ✓ Container aidb       Started

$ docker exec -it aidb psql -U ai -d aidb
```

```
CREATE EXTENSION IF NOT EXISTS vector;
CREATE EXTENSION IF NOT EXISTS age;
LOAD 'age';
SET search_path = ag_catalog, "$user", public;

SELECT extname, extversion FROM pg_extension;
 extname | extversion
---------+------------
 plpgsql | 1.0
 vector  | 0.8.0
 age     | 1.5.0
(3 rows)
```

Your version numbers may differ. That is fine as long as both `vector` and `age` appear.

💉 **HANDS-ON LAB 1.1**

Goal: prove that vectors and graphs live in one database.

1. Start the container with `docker compose up -d --build` and open `psql` as shown above.

2. Create a tiny vector table and query it:

```
CREATE TABLE public.items (id serial PRIMARY KEY, name text, embedding vector(3));
INSERT INTO public.items (name, embedding) VALUES
  ('apple',  '[1, 0.9, 0]'),
  ('banana', '[0.9, 1, 0.1]'),
  ('laptop', '[0, 0.1, 1]');
SELECT name, embedding ⟺ '[1, 1, 0]' AS cosine_distance
FROM public.items ORDER BY embedding ⟺ '[1, 1, 0]' LIMIT 2;
  name  |   cosine_distance
--------+--------------------
 apple  | 0.0013...
 banana | 0.0041...
(2 rows)
```

3. Create a graph and a relationship:

```
SELECT create_graph('g');
SELECT * FROM cypher('g', $$
  CREATE (:Person {name: 'Asha'})-[:WORKS_ON]→(:Project {name: 'HR Bot'})
$$) AS (v agtype);
SELECT * FROM cypher('g', $$
  MATCH (p:Person)-[:WORKS_ON]→(x:Project) RETURN p.name, x.name
$$) AS (person agtype, project agtype);
 person | project
--------+----------
 "Asha" | "HR Bot"
(1 row)
```

4. Take a screenshot. This is the first commit of your portfolio repo.

1.3.3 Troubleshooting

Symptom	Cause	Fix
`ERROR: extension "vector" is not available`	pgvector not installed in the image	Rebuild with the Dockerfile above: `docker compose build --no-cache`
`ERROR: function cypher(unknown, unknown) does not exist`	AGE not loaded or search_path not set in this session	Run `LOAD 'age';` and `SET search_path = ag_catalog, "$user", public;` again (they are per session)
`ERROR: type "vector" does not exist` after setting search_path	The vector extension lives in `public` and `public` was dropped from search_path	Keep `public` in search_path, or write `public.vector`

Symptom	Cause	Fix
`bind: address already in use` on port 5432	A local PostgreSQL is already running	Change the mapping to `"5433:5432"` and connect on 5433
`ERROR: expected 3 dimensions, not 4`	Inserted vector length does not match the column	Match the column size to your embedding model output

✅ CHECK YOUR PROGRESS 1.3

1. **Which pgvector operator gives cosine distance?**
   Answer: `<=>`. The L2 distance operator is `<->` and negative inner product is `<#>`.
2. **Which two commands must you run in each new session before using Cypher in AGE?**
   Answer: `LOAD 'age';` and `SET search_path = ag_catalog, "$user", public;`
3. **Why hide the LLM provider behind a `call_llm()` helper?**
   Answer: So you can switch providers or models for cost, quality or compliance reasons by changing one function, not the whole codebase.

1.4 Skills Employers Ask For

We read job posts for titles like "AI Engineer", "GenAI Developer" and "LLM Application Engineer". The same skills show up again and again. Group them into three tiers.

Tier	Skills	How often in job posts
Must have	Python, REST APIs (FastAPI/Flask), SQL, LLM APIs, prompt design, RAG, embeddings, a vector store, Git, Docker	Almost every post
Strong plus	Evaluation methods, agents and tool calling, a framework (LangChain, LlamaIndex or similar), cloud (AWS/Azure/GCP), CI/CD, observability	Most posts
Differentiators	Knowledge graphs, hybrid search and reranking, fine-tuning basics, guardrails and security, cost optimisation, open-source model hosting	Senior or specialised posts

No PhD or deep learning theory in "must have". Managers want people who ship.

⚠️ COMMON MISTAKE

Listing ten frameworks with no project behind them. Interviewers pick one and ask "what broke?" Two frameworks used deeply beat ten used for a tutorial.

🎯 **INTERVIEW CORNER**

Q: Which vector database have you used and why? **A:** PostgreSQL with pgvector. Our data was already in Postgres, so we could filter by metadata with normal SQL, join with user permissions and keep one backup strategy. At our scale of a few million vectors, an HNSW index gave us low latency.

Q: How do you keep up with a field that changes every month? **A:** I focus on stable concepts (retrieval, evaluation, tokens, cost) and treat frameworks as replaceable. I keep a small eval set so that when a new model comes out I can test it on my own data in an hour instead of trusting hype.

✅ **CHECK YOUR PROGRESS 1.4**

1. **List four must-have skills for an AI engineer job.**

   Answer: Python, SQL, LLM APIs with prompt design, and RAG with embeddings and a vector store. (Git, Docker and REST APIs are also accepted.)

2. **Why is evaluation a strong plus in hiring?**

   Answer: It shows you can prove that a change made answers better, instead of judging by a few manual tries. Companies need that to trust AI features in production.

1.5 The 90-Day Switch Plan

This plan assumes you work full time and can give about 10 to 12 hours a week: one hour on weekdays and three to four hours on weekends. Adjust if you have more time.

Weeks	Focus	Output by the end
1 to 2	Python refresh, FastAPI basics, SQL, Docker. Chapter 1 lab.	Lab container running; a small FastAPI app
3 to 4	LLM fundamentals: tokens, prompting, structured output, streaming, cost (Chapter 2).	`call_llm()` helper with tests; a CLI that summarises files
5 to 6	Embeddings, chunking, pgvector, HNSW indexes.	Document ingestion script; semantic search API
7 to 8	Full RAG: retrieval, prompts with citations, hybrid search, reranking. Start evaluation.	Portfolio Project 1 deployed with an eval report
9 to 10	Knowledge graphs with Apache AGE, entity extraction, GraphRAG.	Portfolio Project 2
11 to 12	Agents, tool calling, guardrails, production (logging, caching, cost).	Portfolio Project 3; resume updated
13	Mock interviews, system design practice, applications.	20 to 30 targeted applications sent



Figure 1.3: The 90-day path

1.5.1 Rules That Make the Plan Work

- **Build in public.** Push to GitHub every week and write a short post every two weeks ("What broke when I chunked PDFs").
- **Use your current job.** Even a small AI feature at work (a search bot, a ticket classifier) becomes your strongest resume line because it has real users.
- **Keep costs tiny.** Use small, cheap models during development. Local open-source models cost nothing per call.

> 💼 **AT WORK**
>
> Ravi, a Java developer at a logistics firm, saw his support team searching a 400-page SOP manual daily. In week 7 he built a pgvector RAG bot over it and demoed it. After two weeks of hardening, lookups dropped from minutes to under one. Those real usage numbers carried his interviews more than any certificate.

> ✅ **CHECK YOUR PROGRESS 1.5**
>
> 1. **Why does the plan start evaluation in weeks 7 to 8 and not at the very end?**
>
>   **Answer:** Evaluation guides every later change. Without it you cannot tell whether graph retrieval or agents actually improve answers.
>
> 2. **What is the strongest kind of resume project and why?**
>
>   **Answer:** One used by real people, ideally at your current job, because it comes with real metrics and real production problems you solved.

1.6 Three Portfolio Projects

Three projects, each showing a different layer of the stack, are enough. Quality over quantity. Each must have: a README with an architecture diagram, a one-command setup (`docker compose up`), an evaluation section with numbers, and a short demo GIF or video.

1.6.1 Project 1: Document Q&A with Citations (RAG)

Pick public documents such as RBI circulars or government scheme PDFs. Build ingestion into pgvector, a FastAPI `/ask` endpoint returning answers with citations, and a 50-question eval set. Report accuracy before and after one improvement.

1.6.2 Project 2: GraphRAG over a Domain

Use a dataset with relationships (company filings, a movie database). Extract entities and relations with an LLM, load them into Apache AGE, and combine Cypher with vector search. Show 10 questions where GraphRAG beats plain RAG.

1.6.3 Project 3: An Enterprise Agent with Guardrails

Build an agent that answers "analyst" questions using tools: a read-only SQL tool over a sample sales database, a document search tool, and a calculator. Add a step limit, a list of allowed tables, input validation and a full trace log. Show traces of a success, a failure and a blocked unsafe request.

> ⚠️ **COMMON MISTAKE**
>
> Building "ChatGPT clone #4,000" with a nice UI and no evaluation. Hiring managers have seen hundreds. A plain UI with an honest eval table ("accuracy went from 64% to 81% after hybrid search") stands out far more.

> ✅ **CHECK YOUR PROGRESS 1.6**
>
> 1. **What four things must every portfolio project include?**
>
>   **Answer:** A README with an architecture diagram, one-command setup, an evaluation section with numbers, and a short demo.
>
> 2. **Why should Project 3 show a blocked unsafe request?**
>
>   **Answer:** It proves you thought about safety and guardrails, which is a top concern for companies adopting agents.

1.7 Resume Tips for AI Engineer Roles

Recruiters spend well under a minute on a first scan. Your resume must show three things fast: you build with LLMs, you measure results, and you ship to production.

1.7.1 The Formula

Action verb + what you built + technology + measurable result.

Weak	Strong
Worked on a chatbot using LangChain.	Built a RAG assistant over 3,000 policy pages using pgvector and FastAPI; raised answer accuracy from 62% to 84% on a 120-question eval set.
Familiar with LLMs and prompt engineering.	Designed JSON-schema prompts for invoice extraction; cut manual data entry by about 70% for a 6-person finance team.

Put "Projects" near the top if your current job has no AI work yet. Keep your backend wins ("APIs serving 1M requests/day"); AI apps are APIs too. Never claim numbers you cannot explain.

🎯 **INTERVIEW CORNER**

Q: Your resume says accuracy improved from 62% to 84%. How did you measure that? **A:** I built 120 question and expected-answer pairs with domain experts. For each answer, an LLM judge with a strict rubric marked correct or not, and I manually checked a random 20% to confirm the judge agreed with humans. The improvement came mostly from better chunking and adding keyword search.

Q: What was the hardest bug in that project? **A:** Tables in PDFs were split across chunks, so numeric answers were wrong. I switched to a parser that kept tables whole and stored them as separate chunks with the section title attached.

✅ **CHECK YOUR PROGRESS 1.7**

1. **Rewrite "Used embeddings for search" as a strong resume line.**
   Answer: Example: "Built semantic search over 50,000 support tickets with embeddings in pgvector; improved top-5 relevant-result rate from 48% to 79%."
2. **Why is it risky to add numbers you cannot explain?**
   Answer: Interviewers probe them. Failing to explain how you measured a number damages trust in the whole resume.

1.8 The Interview Rounds

Most AI engineer interview loops have four types of rounds, sometimes merged. Knowing them lets you prepare in the right order.

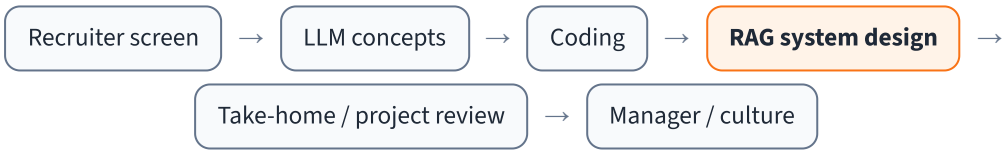


Figure 1.4: A typical AI engineer interview loop

1.8.1 Round 1: LLM Concepts

Rapid questions on tokens, temperature, context windows, hallucination, embeddings and fine-tuning vs RAG. Chapter 2 covers these.

1.8.2 Round 2: System Design for RAG

A typical prompt: "Design a support assistant over 1 million help articles for 10,000 daily users." They want to hear about ingestion, chunking, embedding, index choice, hybrid retrieval, reranking, prompt with citations, evaluation, caching, cost per query, access control and monitoring. Draw boxes, state numbers, and talk trade-offs.

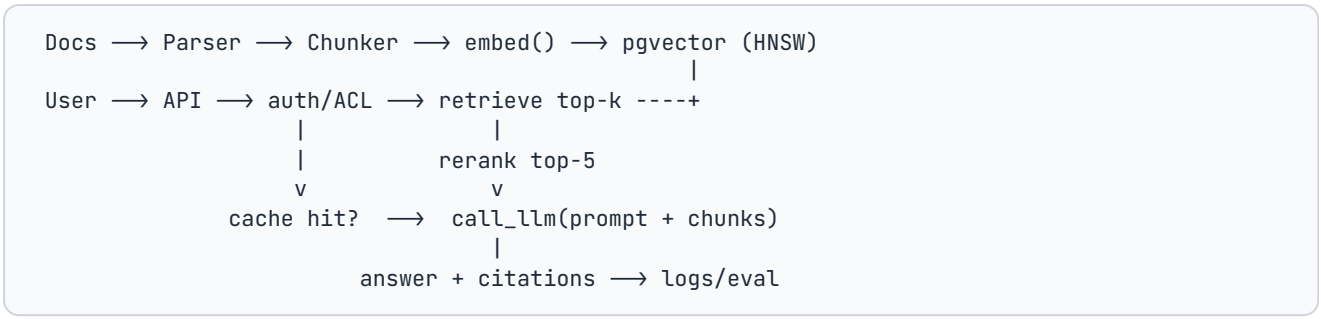


Figure 1.5: A whiteboard-level RAG design

1.8.3 Round 3: Coding

Usually practical: write a chunker with overlap, cosine similarity in plain Python, JSON validation with retries, or a streaming FastAPI endpoint. Keep basic DSA warm too.

```
import math

def cosine_similarity(a: list[float], b: list[float]) → float:
    if len(a) ≠ len(b):
        raise ValueError("vectors must have the same length")
    dot = sum(x * y for x, y in zip(a, b))
    norm_a = math.sqrt(sum(x * x for x in a))
    norm_b = math.sqrt(sum(y * y for y in b))
    if norm_a == 0 or norm_b == 0:
        return 0.0
    return dot / (norm_a * norm_b)

print(cosine_similarity([1, 1, 0], [1, 0.9, 0]))
```

```
$ python cos.py
0.9986178293325095
```

Note: pgvector's `⟺` returns cosine *distance*, which is `1 - cosine_similarity`. That is why "apple" had a distance near 0.0013 in Lab 1.1.

1.8.4 Round 4: Take-Home or Project Deep Dive

You build a small RAG app in 2 to 5 days or present a portfolio project. Graders want clean structure, an eval script, error handling, honest limitations and a "What I would do next" section.

> 🎯 **INTERVIEW CORNER**
>
> **Q:** In the RAG design, how do you make sure users only see documents they are allowed to see? **A:** Store access-control metadata (team, role, document owner) with each chunk and filter in the same SQL query as the vector search, for example `WHERE team_id = ANY(%s)` with the user's teams. Never rely on the LLM to hide restricted content.
>
> **Q:** How would you reduce cost per query? **A:** Cache answers for repeated questions, send fewer but better chunks after reranking, use a smaller model for simple queries and route only hard ones to a large model, and set a max output token limit.

CHECK YOUR PROGRESS 1.8

1. **Name five components you should mention in a RAG system design round.**
   Answer: Ingestion and chunking, embeddings and vector index, retrieval with reranking, prompting with citations, and evaluation and monitoring. (Caching, access control and cost are also valid.)
2. **pgvector returns 0.2 for `a` ⟺ `b`. What is the cosine similarity?**
   Answer: 1 - 0.2 = 0.8.

What Have We Learnt?

- An AI engineer builds products on top of pre-trained LLMs; the role is closest to backend engineering.
- Data scientists find insights, ML engineers train and serve models, AI engineers ship LLM-powered features.
- RAG adds company knowledge to an LLM at question time, like an open-book exam.
- Knowledge graphs and GraphRAG help with relationship and multi-hop questions.
- Agents let an LLM call tools in a loop; they need guardrails because they take actions.
- Our stack: call_llm(), embed(), PostgreSQL with pgvector and Apache AGE, retrieval, agents, evaluation and production practices.
- Must-have skills: Python, SQL, APIs, LLM APIs, prompting, RAG, embeddings, a vector store, Git and Docker.
- A 90-day part-time plan with weekly GitHub output is enough to make a credible switch.
- Three focused, evaluated portfolio projects beat many shallow demos.
- Interviews test LLM concepts, RAG system design, practical coding and a take-home or project deep dive.

Key terms: AI engineer, ML engineer, data scientist, LLM, RAG, embedding, vector, pgvector, cosine distance, HNSW, knowledge graph, GraphRAG, Apache AGE, Cypher, agent, tool calling, guardrails, evaluation, portfolio, system design.

RESUME LINE

Built a containerised AI lab on PostgreSQL with pgvector and Apache AGE, combining vector search and Cypher graph queries in a single database for 3 portfolio projects.

Delivered an internal RAG assistant over a 400-page SOP manual used by 25 support agents; cut average lookup time from about 4 minutes to under 1 minute.

Exercises

A. Multiple Choice Questions

1. Which role mainly builds products on top of pre-trained LLMs?
   a. Data scientist
   b. AI engineer
   c. Database administrator
   d. Research scientist
2. RAG is best described as:
   a. Training a model from scratch
   b. Compressing a model
   c. Retrieving relevant data and adding it to the prompt
   d. Translating prompts into SQL
3. In pgvector, the cosine distance operator is:
   a. `⟷`
   b. `<#>`
   c. `⟺`
   d. `=`
4. Which statement must run before Cypher queries work in a new AGE session (along with setting search_path)?
   a. `LOAD 'age';`
   b. `START GRAPH;`
   c. `USE age;`
   d. `IMPORT age;`
5. A question like "Which suppliers of Plant B also supply Vendor X?" is best served by:
   a. Higher temperature
   b. A knowledge graph
   c. A larger font
   d. Removing retrieval
6. Which is the strongest resume line?
   a. Passionate about AI
   b. Familiar with LangChain
   c. Built RAG over 3,000 pages; accuracy 62% to 84% on 120-question eval
   d. Completed 12 AI courses

B. Short Answer Questions

1. Explain the difference between an ML engineer and an AI engineer in three sentences.
2. Why do companies prefer RAG over fine-tuning for frequently changing facts?
3. Give two reasons to keep vectors and the graph in PostgreSQL rather than separate systems.
4. List the four common interview round types for AI engineer roles.

C. Scenario Questions

1. Your demo RAG bot works on your laptop. In the shared dev environment every query fails with `ERROR: function cypher(unknown, unknown) does not exist`. What do you do?
2. A manager reports that the HR bot answered a salary question using a document only the finance team should see. What do you do?

3. After a product launch, the bot's monthly LLM bill is three times the estimate. What do you check first?

D. Mini Project

Create a GitHub repo named `ai-engineer-lab`. Add the Dockerfile and docker-compose.yml from this chapter, a `setup.sql` that creates both extensions, a sample vector table and a sample graph, and a Python script `check_lab.py` that connects with psycopg 3 and prints the installed extension versions. Write a README with setup steps and a screenshot.

Answer Key: Chapter 1

A. MCQ

1. (b) AI engineers build applications on top of existing LLMs.
2. (c) RAG retrieves data and adds it to the prompt at question time.
3. (c) `⟺` is cosine distance; `⟷` is L2; `<#>` is negative inner product.
4. (a) `LOAD 'age';` loads the extension library into the session.
5. (b) Multi-hop relationship questions suit a graph.
6. (c) It has a concrete build, technology and measured result.

B. Short Answers

1. An ML engineer builds training pipelines and serves models, often creating or tuning them. An AI engineer mostly uses pre-trained LLMs and builds retrieval, prompts, tools and evaluation around them. The AI engineer's work looks more like backend product engineering.
2. Updating RAG means re-indexing documents, which is fast and cheap. Fine-tuning must be repeated for every change, cannot easily cite sources, and does not reliably store exact facts.
3. You can join vectors, graph data and normal tables (for example permissions) in one query, and you have a single system to back up, secure and monitor.
4. LLM concepts, RAG system design, coding, and take-home or project deep dive.

C. Scenario Questions

1. Step 1: run `SELECT extname FROM pg_extension;` on dev; if `age` is missing, `CREATE EXTENSION age;`. Step 2: AGE needs `LOAD 'age';` and the search_path per session; your laptop likely ran them by hand. Step 3: run both on every new pooled connection. Step 4: add a startup health check with a trivial Cypher query.
2. Step 1: treat it as a data leak; disable the affected collection. Step 2: the retrieval query likely has no access filter. Step 3: store team or role metadata per chunk and filter in SQL by the user's groups. Step 4: add a test that finance-only chunks never reach HR users. Step 5: inform security and review logs.
3. Check per-request token logs: input (too many chunks, long system prompt, full history each turn), output (no max limit) and model choice. Look for retry loops and long agent runs. Fix with reranking to fewer chunks, trimmed history, a smaller model for easy queries, caching and output limits.

D. Mini Project: Model Solution

Files: `Dockerfile`, `docker-compose.yml`, `setup.sql` (CREATE EXTENSION for vector and age, LOAD 'age', SET search_path, create items table and graph `g`), `check_lab.py`, `README.md`. The core of `check_lab.py`:

```
import psycopg

with psycopg.connect("postgresql://ai:ai@localhost:5432/aidb") as conn:
    rows = conn.execute(
        "SELECT extname, extversion FROM pg_extension ORDER BY extname"
    ).fetchall()
    for name, version in rows:
        print(f"{name:10} {version}")
```

```
$ python check_lab.py
age        1.5.0
plpgsql    1.0
vector     0.8.0
```

CHAPTER 2

LLMs for Developers

"An LLM is the most confident person at the chai stall. Always has an answer. Sometimes it is even correct."

After studying this chapter, you will be able to

- Explain tokens, context windows and how an LLM generates text one token at a time.
- Control output with temperature, top-p and maximum output tokens.
- Write system prompts, few-shot prompts and structured JSON output prompts that hold up in production.
- Explain tool calling and why LLMs hallucinate.
- Build a provider-neutral `call_llm()` helper in Python with streaming and cost tracking.
- Decide between prompting, RAG and fine-tuning for a given problem.

Think of autocomplete on your phone keyboard. You type "Happy" and it suggests "birthday". An LLM is that same idea, trained on a huge share of the internet and books, and made big enough that the "next word" guesses become essays, code and legal summaries. It has no database of facts inside it. It has patterns. When the pattern matches reality, it looks like genius. When it does not, it confidently invents a railway timetable for a train that never ran.

You do not need to build an LLM. You need to drive one well: know its cost, its failures, and how to make its output predictable.

2.1 Tokens and Tokenization

LLMs do not read characters or words. They read **tokens**: chunks of text from a fixed vocabulary, usually tens of thousands to a few hundred thousand entries. Common words are often one token. Rare words split into pieces. Spaces and punctuation are often part of tokens too.

```
"Tokenization is unbelievable"
  → ["Token", "ization", " is", " un", "believ", "able"]
     (illustrative split; real splits depend on the tokenizer)
```

Figure 2.1: A sentence broken into tokens

Most tokenizers use **Byte Pair Encoding (BPE)** or a cousin: start from bytes and repeatedly merge the most frequent pairs. Each model family has its own tokenizer, so token counts differ across models.

2.1.1 Counting Tokens in Python

OpenAI publishes the open-source `tiktoken` library. Other providers offer token-counting endpoints or their own tokenizers; check the docs for your provider.

```
$ pip install tiktoken

# count_tokens.py
import tiktoken

enc = tiktoken.get_encoding("cl100k_base")
ids = enc.encode("Hello world")
print(ids, len(ids))
print([enc.decode([i]) for i in ids])
```

```
$ python count_tokens.py
[9906, 1917] 2
['Hello', ' world']
```

2.1.2 Rules of Thumb

- For English, one token is about 4 characters, or about three-quarters of a word. So 1,000 words is roughly 1,300 to 1,400 tokens.
- Hindi, Tamil and other non-Latin scripts often need more tokens per word on many tokenizers, so they cost more for the same meaning. Measure with your model's tokenizer.
- Code, JSON and long numbers are token-heavy.

> 💡 **DO YOU KNOW?**
>
> Many LLMs struggle to count the letters in a word like "strawberry". The model never sees individual letters, only tokens such as "str" and "awberry". It is like asking someone how many times the letter "a" appears in a word they only ever heard spoken.

> 🎯 **INTERVIEW CORNER**
>
> **Q:** What is a token and why does it matter to you as an engineer? **A:** A token is the unit of text the model reads and writes, roughly 4 English characters. It matters because pricing, context window limits and latency are all measured in tokens.
>
> **Q:** Why can the same prompt cost different amounts on two providers? **A:** Different tokenizers split text differently, so token counts differ, and per-token prices differ too.

✅ **CHECK YOUR PROGRESS 2.1**

1. **Roughly how many tokens are in a 750-word English document?**
   Answer: About 1,000 tokens, using the "one token is about three-quarters of a word" rule.
2. **Why can a Hindi chatbot be more expensive than an English one?**
   Answer: Many tokenizers split non-Latin scripts into more tokens per word, so the same meaning costs more tokens.

2.2 Context Windows

The **context window** is the maximum number of tokens the model can handle in one request: your input (system prompt, history, retrieved documents, question) plus the output it generates. Think of it as the size of the exam answer sheet plus question paper together. If you fill it with questions, there is no space left to write the answer.

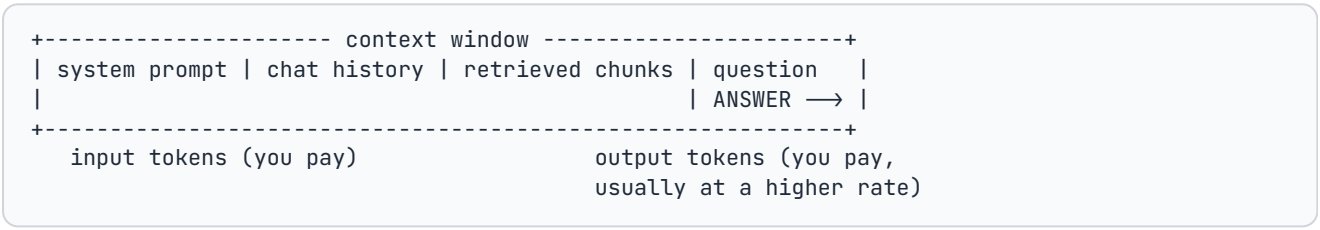


Figure 2.2: What fills a context window

Windows range from tens of thousands to around a million tokens. Bigger is not free:

- **Cost:** every input token is billed, every time. A 100,000-token prompt sent 1,000 times a day is 100 million tokens a day.
- **Latency:** longer input takes longer to process before the first output token appears.
- **Quality:** research such as the "Lost in the Middle" paper (Liu et al., 2023) found models use information at the start and end of a long context better than information in the middle. Newer models have improved, but you should still test.

🔑 **KEY POINT**

A large context window is not a replacement for good retrieval. Sending the five most relevant chunks usually beats sending fifty loosely related ones: cheaper, faster and often more accurate.

⚠️ **COMMON MISTAKE**

Sending the full chat history every turn. By turn 40 the bill climbs and requests fail with context-length errors. Keep the last few turns plus a short summary.

CHECK YOUR PROGRESS 2.2

1. **Name the parts of a request that use up the context window.**
   Answer: System prompt, conversation history, retrieved documents, the user's question and the generated output.
2. **Give two reasons not to fill a huge context window just because you can.**
   Answer: Higher cost per request and slower responses. Accuracy can also drop when key facts sit in the middle of long input.

2.3 How Generation Works

2.3.1 Next-Token Prediction

An LLM is trained to do one thing: given the tokens so far, predict a probability for every possible next token. Generation is a loop:



Figure 2.3: The generation loop

For the input "The capital of India is", the model might give " New" 0.92, " Delhi" 0.04, " a" 0.01, and tiny scores to everything else. It picks one, appends it, and runs again with "The capital of India is New". The loop stops when the model emits a special end token or hits the maximum output token limit you set.

Two consequences: output is produced token by token, so long answers are slow (hence streaming); and there is no "fact check" step, only likely-sounding text, which is the root of hallucination.

2.3.2 Temperature

Temperature reshapes the probabilities before picking. Low temperature sharpens them (the top token almost always wins). High temperature flattens them (unlikely tokens get a real chance).

Temperature	Behaviour	Use for
0 to 0.2	Nearly deterministic, focused	Extraction, classification, RAG answers, code
0.3 to 0.7	Balanced	General chat, summaries, emails
0.8 and above	Creative, more random	Brainstorming, taglines, story ideas

Even at temperature 0, outputs are not guaranteed identical across calls on many hosted APIs, because of how computation is batched on servers. Design your system to tolerate small differences.

2.3.3 Top-p (Nucleus Sampling)

Top-p keeps only the smallest set of tokens whose probabilities add up to p, then samples from that set. With top_p = 0.9 the model ignores the long tail of strange tokens that together hold the last

10%. Providers usually advise changing temperature or top-p, not both. Allowed ranges differ between providers, so check the docs for your version.

😄 **FUNNY SIDE**

Temperature 0: "Dear Sir, please find attached the report." Temperature 1.5: "Dear Sir, behold the report, born of moonlight and Excel, attached with the courage of a thousand interns." Know your use case before turning the knob.

🎯 **INTERVIEW CORNER**

Q: What temperature would you use for a RAG answer bot and why? **A:** Low, around 0 to 0.2. I want the answer to stick closely to the retrieved text, not get creative.

Q: Explain top-p in one line. **A:** Sample only from the smallest set of top tokens whose combined probability reaches p, cutting off the unlikely tail.

Q: If temperature is 0, why might two runs differ? **A:** Hosted inference can have small numerical differences from batching and parallel hardware, and ties can break differently. So temperature 0 reduces randomness but does not guarantee identical output.

✅ **CHECK YOUR PROGRESS 2.3**

1. **What two things can stop a generation loop?**
   Answer: The model producing an end-of-sequence token, or reaching the maximum output token limit (stop sequences, if configured, also end it).
2. **Which is better for invoice field extraction: temperature 0.1 or 1.0?**
   Answer: 0.1. Extraction needs consistent, focused output.

2.4 Prompting Techniques

A prompt is a specification. Vague specs give vague software. Good prompts state the role, the task, the input, the rules and the output format.

2.4.1 System Prompts

Most chat APIs separate a **system** message (standing instructions) from **user** messages (the request). Put stable rules in the system prompt.

```
SYSTEM:
You are a support assistant for ShopKart, an Indian e-commerce site.
Rules:
- Answer only from the CONTEXT section. If the answer is not there,
  say "I don't have that information" and suggest contacting support.
- Keep answers under 120 words.
- Never ask for card numbers, OTPs or passwords.
- Reply in the user's language (English or Hindi).

USER:
CONTEXT:
[1] Refunds for prepaid orders reach the original payment method
    in 5-7 business days after pickup.
QUESTION: My refund is not showing, it has been 3 days.
```

2.4.2 Few-Shot Prompting

Few-shot means giving a few input-output examples before the real input. It is the fastest way to teach format and edge cases.

```
Classify the ticket into one of: BILLING, DELIVERY, ACCOUNT, OTHER.

Ticket: "Charged twice for one order"            → BILLING
Ticket: "Delivery boy says address not found"    → DELIVERY
Ticket: "Can't log in after changing phone"      → ACCOUNT

Ticket: "Order shows delivered but I got nothing" →
```

Pick examples at tricky boundaries; three to five good ones beat twenty random ones.

2.4.3 Chain-of-Thought Style Reasoning

Asking the model to reason step by step before answering often improves maths, logic and multi-rule tasks. Some "reasoning" models do this internally and bill for those tokens. Reasoning adds cost and latency, and raw reasoning should not be shown to users: ask for steps in one field and the answer in another, and display only the answer.

2.4.4 Structured Output with JSON Schema

Software needs predictable fields. Describe the exact JSON shape, then **validate** it in code. Many providers offer native structured output or JSON modes; parameter names differ by provider and version, so check the docs. Validate either way.

```python
# extract.py
import json
from pydantic import BaseModel, ValidationError
from llm import call_llm

class Invoice(BaseModel):
    vendor: str
    invoice_number: str
    total_inr: float
    gstin: str | None = None

SCHEMA = json.dumps(Invoice.model_json_schema())

PROMPT = f"""Extract the invoice fields from the text.
Return ONLY a JSON object matching this JSON Schema, no prose:
{SCHEMA}
If a field is missing, use null.

TEXT:
{{text}}"""

def extract_invoice(text: str, retries: int = 2) -> Invoice:
    prompt = PROMPT.replace("{text}", text)
    for attempt in range(retries + 1):
        raw = call_llm(prompt, temperature=0)
        try:
            return Invoice.model_validate_json(raw)
        except ValidationError as err:
            prompt += f"\n\nYour last output was invalid: {err}. Return valid JSON only."
    raise RuntimeError("could not get valid JSON")
```

Note the double braces `{{text}}` in the f-string: they leave a literal `{text}` placeholder. We fill it with `.replace()`, not `.format()`, because the inserted schema is full of braces and `.format()` would raise a `KeyError` on them.

> ⚠️ **COMMON MISTAKE**
>
> Parsing LLM JSON with `json.loads()` and trusting it. Models sometimes wrap JSON in markdown code fences, add a friendly sentence, or skip a field. Strip fences if needed, validate with a schema, and retry with the error message.

> 🎯 **INTERVIEW CORNER**
>
> **Q:** How do you get reliable JSON from an LLM? **A:** Give the exact schema, use temperature 0, use the provider's structured output mode if available, validate with Pydantic, and retry with the validation error. Log failures so the prompt can be improved.
>
> **Q:** When does few-shot help more than a longer instruction? **A:** When the format or boundary between categories is easier to show than describe, such as classification labels or a specific writing style.

CHECK YOUR PROGRESS 2.4

1. **What belongs in a system prompt?**
   Answer: Stable instructions: role, rules, safety limits, tone and output format.
2. **Why should you not show raw chain-of-thought to users?**
   Answer: It can be long, confusing or wrong in places even when the final answer is right, and it may expose internal instructions. Show the final answer only.
3. **What should happen when the JSON fails validation?**
   Answer: Retry with the validation error added to the prompt, up to a small limit, then fail loudly and log it.

2.5 Tool Calling and Hallucinations

2.5.1 The Idea of Tool Calling

An LLM cannot check your database or today's exchange rate. With **tool calling** (also called function calling) you describe tools to the model: name, purpose and a JSON Schema for arguments. The model can then reply with "please call `get_order_status` with `{"order_id": "A123"}`" instead of plain text. **Your code** runs the tool and sends the result back. The model never runs anything itself.

```
User: "Where is order A123?"
  |
  v
LLM (+ tool descriptions) ──> tool call: get_order_status(A123)
                                          |
Your code runs the function <------+
  |  result: {"status": "out for delivery"}
  v
LLM (+ result) ──> "Your order A123 is out for delivery today."
```

Figure 2.4: The tool calling round trip

Both OpenAI and Anthropic APIs accept a list of tools with JSON Schema parameters, and open-source models served by many runtimes support similar formats. The exact request and response shapes differ, so we build agents properly later in the book.

2.5.2 Why LLMs Hallucinate

A **hallucination** is fluent, confident output that is false or unsupported. Causes:

- **Prediction, not lookup:** the model generates likely text. A plausible fake case citation is "likely" text.
- **Missing knowledge:** private data and events after the training cutoff are simply unknown.
- **Pressure to answer:** training often rewards helpful-sounding answers over "I don't know".
- **Bad context:** wrong or conflicting retrieved chunks lead to wrong answers grounded in bad text.

Reduce it with retrieval grounding, permission to say "I don't know", verified chunk-ID citations, low temperature, tools for facts and maths, and evaluation sets.

> 💼 **AT WORK**
>
> A fintech support bot told a customer that a loan prepayment had "no charges". The real policy had a 2% fee. Investigation showed retrieval had returned an old FAQ. The fix had three parts: delete stale documents and add an `effective_date` filter, require the answer to cite a chunk ID that is then verified in code, and add 30 policy questions with exact expected answers to the eval set, run on every deploy.

> ✅ **CHECK YOUR PROGRESS 2.5**
>
> 1. **In tool calling, who executes the function?**
>
>   **Answer:** Your application code. The model only proposes the tool name and arguments.
>
> 2. **Give three ways to reduce hallucinations.**
>
>   **Answer:** Ground with retrieval, require and verify citations, and allow the model to say "I don't know". Low temperature and evaluation also help.

2.6 Choosing a Model

Factor	Closed (hosted API)	Open-weight (self-hosted)
Examples	OpenAI GPT models, Anthropic Claude, Google Gemini	Llama, Mistral, Qwen, Gemma families
Quality	Usually top of leaderboards	Strong and improving; best ones are close for many tasks
Setup	API key, minutes	GPUs, serving stack (for example vLLM or Ollama)
Cost model	Pay per token	Pay for hardware, whether used or idle
Data control	Data leaves your network (check provider terms)	Stays inside your infrastructure
Customisation	Limited fine-tuning options	Full control, fine-tune freely (check the licence)

Larger models reason better but cost more and respond slower. Use **routing**: easy tasks (classification, short extraction) to a small fast model, hard ones to a large model. Decide with your own eval set, not leaderboards.

> 🔑 **KEY POINT**
>
> Model names and prices change every few months. Never hardcode a model name deep in your code. Read it from configuration so switching is a one-line change after you re-run your evals.

🎯 **INTERVIEW CORNER**

Q: A bank says customer data cannot leave its data centre. What model strategy do you suggest? **A:** Self-host an open-weight model inside their network, or use a provider offering a private or in-region deployment that meets their compliance rules. Then benchmark on their tasks; maybe a mid-size open model is enough.

Q: How do you choose between a small and a large model? **A:** Run both on the same eval set and compare accuracy, latency and cost per request. If the small model meets the quality bar, use it; route only failing categories to the large one.

✅ **CHECK YOUR PROGRESS 2.6**

1. **Give two reasons a company might choose an open-weight model.**

   Answer: Data must stay inside its network, and it wants full control to fine-tune or to run at high steady volume on its own hardware.

2. **What is model routing?**

   Answer: Sending each request to a model chosen by task difficulty, so easy tasks use a cheap model and hard ones use a strong model.

2.7 Calling an LLM from Python

We now build the `call_llm()` helper used for the rest of the book. It hides provider differences behind one function. Install the SDKs you need: `pip install openai anthropic`. Many local runtimes, such as Ollama, expose an OpenAI-compatible endpoint, so the OpenAI client can talk to them by changing `base_url`.

```
# llm.py
import os
from dataclasses import dataclass

PROVIDER = os.getenv("LLM_PROVIDER", "openai")   # openai | anthropic | local
MODEL = os.environ["LLM_MODEL"]                   # set from your provider's model list

@dataclass
class LLMResult:
    text: str
    input_tokens: int
    output_tokens: int

def call_llm(prompt: str, system: str = "You are a helpful assistant.",
             temperature: float = 0.2, max_tokens: int = 800) → str:
    return call_llm_full(prompt, system, temperature, max_tokens).text

def call_llm_full(prompt: str, system: str, temperature: float,
                  max_tokens: int) → LLMResult:
    if PROVIDER == "anthropic":
        import anthropic
        client = anthropic.Anthropic()          # reads ANTHROPIC_API_KEY
        resp = client.messages.create(
            model=MODEL, system=system, max_tokens=max_tokens,
            temperature=temperature,
            messages=[{"role": "user", "content": prompt}],
        )
        text = "".join(b.text for b in resp.content if b.type == "text")
        return LLMResult(text, resp.usage.input_tokens, resp.usage.output_tokens)

    from openai import OpenAI
    if PROVIDER == "local":                     # e.g. Ollama
        client = OpenAI(base_url="http://localhost:11434/v1", api_key="ollama")
    else:
        client = OpenAI()                       # reads OPENAI_API_KEY
    resp = client.chat.completions.create(
        model=MODEL, temperature=temperature, max_tokens=max_tokens,
        messages=[{"role": "system", "content": system},
                  {"role": "user", "content": prompt}],
    )
    u = resp.usage
    return LLMResult(resp.choices[0].message.content or "",
                     u.prompt_tokens if u else 0, u.completion_tokens if u else 0)
```

```
$ export LLM_PROVIDER=anthropic LLM_MODEL=<model-id-from-docs>
$ python -c "from llm import call_llm; print(call_llm('Say namaste in 3 languages.'))"
Namaste (Hindi), Vanakkam (Tamil), Namaskara (Kannada).
```

Notes: newer OpenAI models may prefer `max_completion_tokens` instead of `max_tokens`, and some reasoning models restrict temperature. Check the docs for the model you use. In production, create the client once and reuse it, and add timeouts and retries.

2.7.1 Streaming

Streaming sends tokens as they are generated. Total time is the same, but users see the first words almost at once, so chat feels fast.

```
# stream.py
from llm import PROVIDER, MODEL

def stream_llm(prompt: str, system: str = "You are a helpful assistant."):
    if PROVIDER == "anthropic":
        import anthropic
        client = anthropic.Anthropic()
        with client.messages.stream(
            model=MODEL, system=system, max_tokens=800,
            messages=[{"role": "user", "content": prompt}],
        ) as stream:
            for text in stream.text_stream:
                yield text
        return
    from openai import OpenAI
    client = OpenAI()
    stream = client.chat.completions.create(
        model=MODEL, stream=True,
        messages=[{"role": "system", "content": system},
                  {"role": "user", "content": prompt}],
    )
    for chunk in stream:
        if chunk.choices and chunk.choices[0].delta.content:
            yield chunk.choices[0].delta.content

for piece in stream_llm("Explain pgvector in two lines."):
    print(piece, end="", flush=True)
```

In FastAPI, return a `StreamingResponse` wrapping this generator. Remember: when streaming, you cannot validate JSON until the end, so stream only free text to users.

2.7.2 Cost Estimation

Cost per request = input tokens x input price + output tokens x output price. Prices are quoted per million tokens and change often, so keep them in config.

```
# cost.py  (prices are EXAMPLES; check your provider's pricing page)
PRICES_PER_M = {"small-model": (0.15, 0.60), "large-model": (3.00, 15.00)}

def cost_usd(model: str, in_tok: int, out_tok: int) -> float:
    p_in, p_out = PRICES_PER_M[model]
    return in_tok / 1e6 * p_in + out_tok / 1e6 * p_out

per_req = cost_usd("large-model", 3_000, 400)
print(f"per request: ${per_req:.4f}")
print(f"per month at 20k req/day: ${per_req * 20_000 * 30:,.0f}")
```

```
$ python cost.py
per request: $0.0150
per month at 20k req/day: $9,000
```

Same traffic on the small model: 3,000 x 0.15/1M + 400 x 0.60/1M = $0.00069 per request, about $414 a month. That gap is why routing and evaluation matter.

💉 HANDS-ON LAB 2.1

Goal: log every LLM call into PostgreSQL so you can see cost per feature.

1. Start the Chapter 1 container and create a table:

```
CREATE TABLE public.llm_calls (
  id bigserial PRIMARY KEY,
  feature text NOT NULL,
  model text NOT NULL,
  input_tokens int NOT NULL,
  output_tokens int NOT NULL,
  cost_usd numeric(12,6) NOT NULL,
  latency_ms int NOT NULL,
  created_at timestamptz DEFAULT now()
);
```

2. Wrap `call_llm_full()`:

```
import time, psycopg
from llm import call_llm_full, MODEL
from cost import cost_usd

DSN = "postgresql://ai:ai@localhost:5432/aidb"

def tracked_llm(feature: str, prompt: str, price_key: str) → str:
    t0 = time.perf_counter()
    r = call_llm_full(prompt, "You are concise.", 0.2, 400)
    ms = int((time.perf_counter() - t0) * 1000)
    with psycopg.connect(DSN) as conn:
        conn.execute(
            "INSERT INTO public.llm_calls (feature, model, input_tokens,"
            " output_tokens, cost_usd, latency_ms) VALUES (%s,%s,%s,%s,%s,%s)",
            (feature, MODEL, r.input_tokens, r.output_tokens,
             cost_usd(price_key, r.input_tokens, r.output_tokens), ms))
    return r.text
```

3. Call it 10 times for two features ("summary", "classify"), then run:

```
SELECT feature, count(*), sum(cost_usd) AS usd, avg(latency_ms)::int AS avg_ms
FROM public.llm_calls GROUP BY feature;
 feature  | count |   usd    | avg_ms
----------+-------+----------+--------
 classify |    10 | 0.000420 |    610
 summary  |    10 | 0.003150 |   1840
(2 rows)
```

Your numbers will differ. Add this table to your portfolio; cost dashboards impress interviewers.

2.7.3 Troubleshooting

Symptom	Cause	Fix
HTTP 401, "invalid API key" or authentication error	Key missing, wrong env var name, or key revoked	Check `OPENAI_API_KEY` / `ANTHROPIC_API_KEY` is exported in the same shell; regenerate if needed
HTTP 429, rate limit error	Too many requests or tokens per minute	Retry with exponential backoff and jitter; queue bulk jobs; request a higher limit
Error mentioning maximum context length	Input plus max output exceeds the window	Trim history, send fewer chunks, lower max output tokens
Answer cut off mid-sentence	Hit the max output token limit	Check the finish/stop reason in the response; raise the limit or ask for shorter output
`Connection refused` on localhost:11434	Local runtime not started	Start Ollama and pull the model first

🎯 **INTERVIEW CORNER**

Q: How would you handle 429 errors in a batch job of 50,000 documents? **A:** Limit concurrency, retry with exponential backoff and jitter, make the job resumable by storing progress, and use a provider batch API if available for lower cost.

Q: Why stream responses? **A:** It lowers time to first token, so users see progress quickly. It does not reduce total cost or total time.

✅ **CHECK YOUR PROGRESS 2.7**

1. **Why use an environment variable for the model name?**
   Answer: So you can switch models without code changes, after testing on your eval set.
2. **A request has 2,000 input and 500 output tokens. Prices are $1 and $4 per million. What is the cost?**
   Answer: 2,000 x 1/1M + 500 x 4/1M = $0.002 + $0.002 = $0.004.

2.8 Fine-Tuning vs RAG vs Prompting

This is one of the most asked interview questions. The rule: start with prompting, add RAG for knowledge, fine-tune for behaviour.

Need	Prompting	RAG	Fine-tuning
Use private or fresh facts	Only if they fit in the prompt	Best choice	Poor; facts go stale, no citations
Fixed output format or tone	Often enough	Does not help	Good when prompts fail

Need	Prompting	RAG	Fine-tuning
Show sources	No	Yes	No
Setup effort	Hours	Days to weeks	Weeks, plus labelled data
Update cost	Edit prompt	Re-index changed docs	Retrain
Cut cost at high volume	No	No	Yes, a small tuned model can replace a large one



Figure 2.5: The usual decision path

They combine too: a fine-tuned small model for format, fed by RAG for facts.

CHECK YOUR PROGRESS 2.8

1. **A legal team wants answers from 10,000 contracts that change weekly, with clause references. Which approach?**

   Answer: RAG, because facts change often and answers need citations.

2. **When is fine-tuning a good choice?**

   Answer: When you need consistent style or format that prompting cannot achieve, or want a small cheap model to match a big model on a narrow, high-volume task.

What Have We Learnt?

- LLMs read and write tokens; one English token is about 4 characters, and pricing and limits are in tokens.
- The context window holds input and output together; bigger windows cost more and can hurt accuracy.
- Generation is next-token prediction in a loop until an end token or the output limit.
- Temperature and top-p control randomness; use low temperature for factual and structured tasks.
- System prompts hold rules, few-shot examples teach format, and step-by-step reasoning helps on logic tasks.
- Structured output needs a schema, validation and retries, even with native JSON modes.
- Tool calling lets the model request actions; your code runs them.
- Hallucinations come from prediction without lookup; reduce them with grounding, citations and evaluation.
- A provider-neutral `call_llm()` with streaming and token logging keeps you flexible and cost-aware.
- Prompt first, RAG for knowledge, fine-tune for behaviour or cost.

Key terms: token, tokenizer, BPE, context window, next-token prediction, temperature, top-p, max output tokens, system prompt, few-shot, chain-of-thought, structured output, JSON Schema, Pydantic, tool calling, hallucination, open-weight model, model routing, streaming, time to first token, rate limit, fine-tuning.

🔑 **RESUME LINE**

Built a provider-neutral LLM client with streaming, retries and per-feature cost logging in PostgreSQL; routing simple tasks to a small model cut monthly LLM spend by about 60%.

Designed schema-validated JSON extraction for invoices with Pydantic and retry-on-error, reaching 97% valid outputs on 1,200 test documents.

Exercises

A. Multiple Choice Questions

1. About how many English characters make one token?
   a. 1 b. 4
   c. 20 d. 100
2. Which setting best suits extracting fields from invoices?
   a. Temperature 1.5 b. Temperature 0 to 0.2
   c. Top-p 0.1 and temperature 2 d. No output limit
3. In tool calling, the tool function is executed by:
   a. The LLM provider's GPU b. The model itself
   c. Your application code d. The tokenizer
4. An answer stops mid-sentence. The most likely cause is:
   a. High temperature b. Max output token limit reached
   c. Wrong API key d. Too few few-shot examples
5. Which approach best handles weekly-changing facts with citations?
   a. Fine-tuning b. Higher temperature
   c. RAG d. Bigger tokenizer
6. Streaming mainly improves:
   a. Total cost b. Time to first token
   c. Accuracy d. Context window size

B. Short Answer Questions

1. Explain next-token prediction and how it leads to hallucinations.
2. What is the difference between temperature and top-p?
3. List four steps to get reliable JSON from an LLM.
4. Give two advantages and two disadvantages of self-hosting an open-weight model.

C. Scenario Questions

1. After a marketing campaign, your chatbot returns many HTTP 429 errors and users see blank replies. What do you do?
2. Finance reports the LLM bill doubled this month though traffic grew only 10%. How do you investigate?
3. A downstream service crashes because the LLM sometimes returns text like "Sure! Here is the JSON:" before the object. Fix it.

D. Mini Project

Build `llm-toolkit`: the `call_llm()` helper supporting two providers (one may be a local model), a streaming FastAPI endpoint `/chat`, a Pydantic-validated `/extract` endpoint, and token/cost logging to PostgreSQL. Include a README table comparing two models on 20 test prompts for accuracy, latency and cost.

Answer Key: Chapter 2

A. MCQ

1. (b) About 4 characters per token in English.
2. (b) Low temperature gives focused, consistent output.
3. (c) The model proposes; your code executes.
4. (b) The output limit truncates the answer; check the stop reason.
5. (c) RAG re-indexes changes and can cite sources.
6. (b) Users see words sooner; total time and cost stay similar.

B. Short Answers

1. The model scores every possible next token, picks one, appends it and repeats. It chooses plausible text, not verified facts, so when it lacks knowledge it can still produce fluent but false text.
2. Temperature reshapes the whole probability distribution (sharper or flatter). Top-p cuts the candidate list to the smallest set reaching probability p, then samples from it.
3. Give an exact schema in the prompt, use low temperature (and native structured output if available), validate with Pydantic or a JSON Schema validator, and retry with the validation error.
4. Advantages: data stays in-house; full control and fine-tuning freedom. Disadvantages: GPU and operations cost even when idle; you own scaling, upgrades and uptime.

C. Scenario Questions

1. Step 1: confirm 429s in logs and check the provider's rate-limit headers or dashboard. Step 2: add retries with exponential backoff and jitter, and show a "busy, retrying" message instead of a blank reply. Step 3: cap concurrency with a queue. Step 4: cut tokens per request (shorter history, fewer chunks), cache frequent answers, and route easy queries to a smaller model with separate limits. Step 5: request a higher rate limit.
2. Step 1: query the `llm_calls` table grouped by feature, model and day. Step 2: compare average input and output tokens per request with last month. Step 3: common causes: a prompt change adding long context, chat history no longer trimmed, a default model switched to a larger one, retry loops, or an agent taking more steps. Step 4: fix the cause and add an alert on daily cost per feature.
3. Step 1: use the provider's structured output or JSON mode if available. Step 2: tighten the prompt ("Return only JSON"). Step 3: in code, strip code fences and extract the first JSON object, then validate with Pydantic. Step 4: retry with the error on failure, and return a clear error to the downstream service instead of crashing it. Step 5: add a test with bad outputs.

D. Mini Project: Model Solution

Files: `llm.py` (call_llm and call_llm_full from 2.7), `stream.py`, `cost.py`, `db.py` (tracked_llm from Lab 2.1), `main.py` (FastAPI with `/chat` returning `StreamingResponse(stream_llm(q), media_type="text/plain")` and `/extract` using `extract_invoice`), `eval/prompts.jsonl` (20

prompts with expected answers), `eval/compare.py` (runs both models, writes a markdown table), `docker-compose.yml` and `README.md` with the comparison table and setup steps.