

FIRST EDITION

AI for Business: A Hands-On Guide



Karthik Srinivasan

Contents

How to Use This Book	2
0.1 For Hobbyists and Casual Readers	2
0.2 For Students in a Course	2
0.3 For Practitioners Solving Specific Problems	2
0.4 Feeding Notebooks to AI Coding Tools	3
0.5 A Word of Caution: Stay Grounded	3
0.6 Recommended Textbooks and Resources	3
Chapter 1: AI at a Glance	6
1.1 The Lifecycle of Intelligence	6
1.2 Tokenization: How Models See Text	7
1.3 Hallucinations and the Swiss-Cheese Capability	7
1.4 Multimodality: Beyond Text	7
1.5 The Divided Ecosystem	8
1.6 From Chatbots to Agents	8
1.7 The Path Ahead	8
1.8 Key Takeaways	8
1.9 Exercises	8
1.10 References	9
Chapter 2: LLM Foundations	11
2.1 Tokenization: How Text Becomes Numbers	12
2.2 Word Embeddings: Teaching Machines the Meaning of Words	15
2.3 TF-IDF: Finding the Words That Actually Matter	18
2.4 The Attention Mechanism: How AI Learns to Focus	20
2.5 Simple Transformers: Building a Sentiment Classifier from Scratch	27
2.6 Attention in Image Analysis: Seeing What the Model Sees	32
Chapter 3: Prompt Engineering with Google AI Studio	36
3.1 Zero-Shot, Few-Shot, and the System Prompt	36
3.2 The PICFAT Framework	37
3.3 Prompting Reasoning Models	38
3.4 Sandwich Prompting and Markdown Structure	39
3.5 Controlling Behavior: Temperature and Friends	39
3.6 Grounding with Google Search	40
3.7 Hands-On Exercise in Google AI Studio	40
3.8 Key Takeaways	41
3.9 Exercises	41
3.10 References and Further Reading	41
Chapter 4: Setting Up API Keys	43
4.1 Storing keys in Colab	43
4.2 Storing keys locally	44
Chapter 5: Calling LLMs: Providers and Fallback	46
5.1 The llm_cascade Package	47
5.2 Calling the OpenAI API	49
5.3 Calling the Google Gemini API	51
5.4 Calling the Groq Cloud API	54
Chapter 6: Local LLMs	60

6.1	Why Run LLMs Locally?	60
6.2	LM Studio	60
6.3	Ollama	61
6.4	Other Tools in the Ecosystem	61
6.5	Model Selection and Quantization	61
6.6	When to Use Local vs. Cloud	61
6.7	Key Takeaways	62
6.8	Exercises	62
6.9	References	62
6.10	Running Local Models with LM Studio	63
Chapter 7: Vibe Coding		66
7.1	What Vibe Coding Actually Means	66
7.2	The Major Vibe Coding Tools	66
7.3	The Vibe Coding Workflow	67
7.4	The 90/10 Problem	67
7.5	Implications for Business Professionals	67
7.6	Key Takeaways	68
7.7	Exercises	68
7.8	References	68
Chapter 8: RAG from First Principles		70
8.1	RAG from First Principles — No Framework, No Magic	71
Chapter 9: RAG with Vector Databases		79
9.1	RAG Example (North America): ChromaDB + LLM Cascade	80
9.2	RAG over Many Documents with ChromaDB	88
9.3	RAG with a Featherweight LLM	95
9.4	LanceDB: A Local Vector Database	98
9.5	Pinecone: A Managed Vector Database	101
Chapter 10: RAG with Structured Data		106
10.1	RAG with a Local DBMS: Text-to-SQL using SQLite	107
10.2	Graph RAG: Retrieval-Augmented Generation with Graph Databases	116
Chapter 11: RAG with Images and Video		125
11.1	RAG with Images: Image Retrieval and Multimodal Q&A	126
11.2	RAG with Video: Visual Question-Answering over Video Frames	136
Chapter 12: LlamaIndex		152
12.1	LlamaIndex RAG Demo: Document Q&A with a Framework	153
12.2	LlamaIndex RAG: A Simpler Single-Company Demo	172
Chapter 13: LangChain		180
13.1	LangChain RAG Demo: Document Q&A with a Framework	181
Chapter 14: LangGraph		202
14.1	LangGraph RAG Demo: Graph-Based Retrieval Pipeline	203
Chapter 15: Fine-Tuning with QLoRA		229
15.1	QLoRA Fine-Tuning Demo: Teaching a Small LLM New Knowledge	230
Chapter 16: Agentic Systems		241
16.1	Multi-Agent System: SQL vs. RAG for Business Database Queries	242
16.2	Multi-Agent System: SQL vs. RAG Comparison (Single-Input Version)	250

16.3 Single-Agent System: One Agent Handles Everything	254
Chapter 17: Autonomous Agents	262
17.1 Autonomous Agent: Business Idea Validator	263
17.2 Autonomous Agent: Business Idea Validator (Multi-Vendor Edition)	269
Chapter 18: Tool-Using Agents and MCP	276
18.1 KU Parking Assistant: A Simple Tool-Using Agent	277
18.2 KU Parking Assistant via Cloud Run MCP Endpoint	291
Chapter 19: Deployment and No-Code	302
19.1 MCP Servers on Google Cloud Run	303
19.2 No-Code AI with Dify	304
Appendix	306
20.1 Learning Path and Chapter Dependencies	306
20.2 Glossary	308
20.3 Common Errors and Fixes	311
20.4 What to Build Next	314

How to Use This Book

This book is designed to be practical, modular, and immediately useful — whether you are reading it cover to cover, referencing a specific chapter to solve a problem, or feeding its notebooks to an AI coding agent to build something new. Different readers will use it differently, and that is by design. This short chapter explains the recommended approaches for different audiences and offers guidance on how to get the most value from the hands-on materials.

0.1 For Hobbyists and Casual Readers

If you are exploring AI out of curiosity or professional interest, the simplest approach is to read the chapters sequentially from beginning to end. Each chapter builds on the ones before it, introducing concepts in the order that makes them easiest to absorb. Start with the big picture in Chapter 1 (AI at a Glance), work through the foundations of how LLMs process language, set up your own local models and coding environment, and then move into the application chapters — RAG, frameworks, fine-tuning, and agents. You do not need to run every notebook. Reading the chapter introductions and skimming the code will give you a solid conceptual understanding. But when a topic catches your interest, clicking the Colab link and running the notebook yourself will deepen your understanding far beyond what reading alone can achieve.

0.2 For Students in a Course

If you are using this book as part of a university course, the recommended approach is to treat it as a **practical companion** to conceptual textbooks, not as a replacement for them. The chapter introductions provide context and motivation, but they are intentionally concise — they cover the “what” and “why” at a level sufficient to understand the code, not at the depth required to truly master the underlying theory.

For deeper conceptual grounding, pair this book with one or more of the textbooks and resources listed in the References section below. Read the theory in a textbook first, then come to this book to see that theory put into practice. For example, read about attention mechanisms and transformer architectures in a textbook, then run the attention notebook in Chapter 2 to see attention weights computed and visualized on real data. Read about retrieval-augmented generation in a research paper, then build a RAG pipeline from scratch in Chapter 8. The combination of conceptual depth from a textbook and hands-on practice from this book produces a far richer understanding than either alone.

0.3 For Practitioners Solving Specific Problems

If you already have a working knowledge of AI and are looking for starter code or reference implementations for a specific task, use this book as a **reference manual**. The table of contents is organized by application pattern, so you can jump directly to the chapter that matches your problem:

- Need to build a document Q&A system? Start with Chapter 8 (RAG from First Principles) or Chapter 12 (LlamaIndex).
- Need to query a SQL database with natural language? Go to Chapter 10 (RAG with Structured Data).
- Need to build an agent-based workflow? Chapters 16–18 cover agentic, autonomous, and tool-using architectures.
- Need to fine-tune a model on company-specific data? Chapter 15 (QLoRA) has a complete working example.
- Need to deploy an AI tool as a cloud service? Chapter 19 covers MCP servers on Google Cloud Run and no-code workflows with Dify.

Each notebook is self-contained — you can run it independently without working through the preceding chapters. The `llm_cascade` package handles API key management and provider fallback, so you only need to set one API key to get any notebook running. See Chapter 4 for step-by-step instructions on acquiring a free API key and storing it in Colab or a local `.env` file.

0.4 Feeding Notebooks to AI Coding Tools

One of the most powerful ways to use this book is as **seed material for vibe coding**. Take any notebook, feed it to Claude Code, Cursor, GitHub Copilot, or Antigravity, and ask the AI to adapt it to your specific business problem. For instance, you might take the KU Parking Assistant notebook and say: “Adapt this to find the nearest coffee shops to any office in our company’s building directory.” The AI coding agent understands the pattern (tool-calling agent with distance calculation) and can restructure the code for your use case in minutes.

This approach works because the notebooks are well-structured with clear function boundaries, consistent naming conventions, and explanatory markdown between code cells. They serve as effective templates that an AI agent can read, understand, and modify — far more effectively than starting from a blank file.

0.5 A Word of Caution: Stay Grounded

While the tools covered in this book — from `llm_cascade` to vibe coding — make AI development remarkably accessible, it is critically important to **understand what you are doing**. Do not blindly trust AI-generated outputs. Do not run code expecting it to produce correct results magically. Large language models are statistical systems that can hallucinate, make logical errors, and produce plausible-sounding but incorrect answers.

Every chapter in this book emphasizes grounding: RAG grounds answers in retrieved evidence, fine-tuning grounds the model in domain-specific knowledge, and tool-using agents ground reasoning in deterministic computations. But the final line of defense is always you — the human who understands the problem domain, can evaluate whether an answer makes sense, and takes responsibility for the decisions made with AI assistance.

As you work through the hands-on exercises, develop the habit of asking: *Does this output make sense given what I know about the problem?* This critical evaluation skill is what separates effective AI practitioners from those who merely run code and hope for the best.

0.6 Recommended Textbooks and Resources

The following resources provide deeper conceptual coverage of the topics in this book. They are listed in order of relevance to the material covered here.

Textbooks:

- Chen, W., & Chen, L. (2025). *Generative AI for Business: Frameworks, Techniques, and Governance*. GenAI Flows Publishing. ISBN 979-8-9997161-0-1. <https://genai4all.org/> — Comprehensive coverage of generative AI frameworks and governance considerations for business applications.
- Alammari, J., & Grootendorst, M. (2024). *Hands-On Large Language Models: Language Understanding and Generation*. O’Reilly Media. <https://www.amazon.com/Hands-Large-Language-Models-Understanding/dp/1098150961> — Excellent visual explanations of transformer architectures, embeddings, and language generation.
- Bouchard, L. F., & Peters, L. (2024). *Building LLMs for Production: Enhancing LLM Abilities and Reliability with Prompting, Fine-Tuning, and RAG*. Towards AI, Inc. <https://a.co/d/5a0kIUx> — Practical guide to production RAG systems, prompt engineering, and fine-tuning.
- Raschka, S. (2024). *Build a Large Language Model (From Scratch)*. Manning Publications. <https://www.manning.com/books/build-a-large-language-model-from-scratch> — For readers who want to understand every layer of the transformer by implementing one.

Video Lectures:

- Karpathy, A. (2025). *Deep Dive into LLMs like ChatGPT* [Video, 3.5 hours]. YouTube. <https://www.youtube.com/watch?v=7xTGNLPyMI> — The single best overview of how LLMs are built, from pre-training to RLHF to System 2 reasoning. Highly recommended before starting this book.
- Karpathy, A. (2024). *Let’s Build GPT: From Scratch, in Code, Spelled Out* [Video]. YouTube. <https://www.youtube.com/watch?v=kCc8FmEb1nY> — A hands-on coding walkthrough of building a GPT-style model from the ground up.

Online Courses and Workshops:

- Google & Kaggle. (2025). *5-Day Gen AI Intensive Course*. <https://www.kaggle.com/learn-guide/5-day-genai> — A structured five-day workshop covering embeddings, RAG, agents, and MLOps with hands-on Kaggle notebooks.
- LangChain Academy. (2026). *LangChain and LangGraph Courses*. <https://academy.langchain.com/collections> — Official courses from the LangChain team covering chains, agents, and graph-based orchestration.
- Gábors Data Analysis. (2026). *AI Course Materials*. <https://gabors-data-analysis.com/ai-course/> — Data analysis and AI course materials with a focus on practical applications.
- DeepLearning.AI. (2026). *Short Courses on LLMs and RAG*. <https://www.deeplearning.ai/short-courses/> — A growing library of focused courses by Andrew Ng and collaborators on specific AI topics including RAG, fine-tuning, and agents.

Each of these resources complements this book from a different angle. The textbooks provide theoretical depth. The video lectures build intuition. The online courses offer structured learning paths. This book provides the hands-on practice environment where all of those concepts come to life.

Chapter 1

AI at a Glance

Before diving into hands-on coding and practical implementations, it is essential to build a robust mental model of how generative AI systems are constructed, why they succeed, why they fail, and where the technology is heading. Generative AI is not magic. It is a sophisticated, multi-stage engineering pipeline that transforms raw internet data into highly capable reasoning agents. This opening chapter surveys the full landscape — from the atomic units of AI perception to the frontier of agentic systems — so that every concept in the chapters that follow sits on a solid foundation.

1.1 The Lifecycle of Intelligence

Creating a powerful AI assistant is not a single monolithic task. It is a progressively refined process that begins with absorbing vast quantities of internet text and gradually shapes that raw data into a model that is helpful, aligned with human values, and capable of complex reasoning. The lifecycle unfolds across three distinct stages.

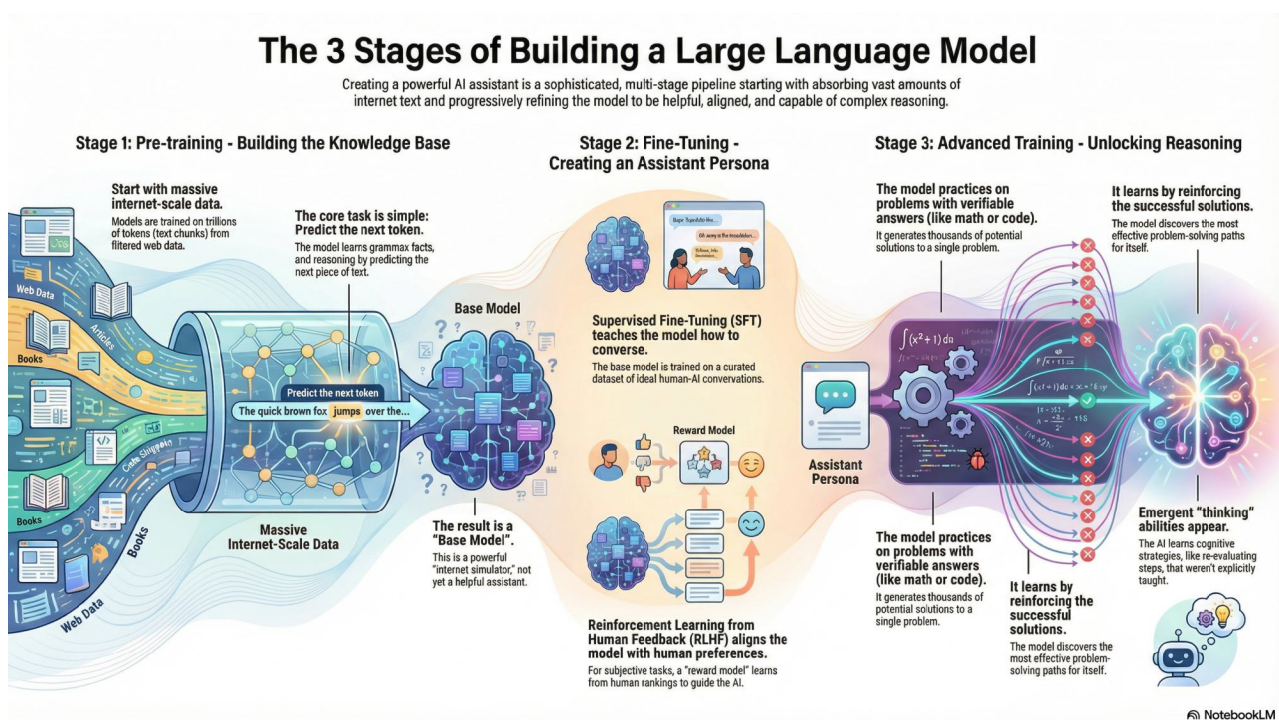


Figure 1.1: The three stages of building a large language model. Pre-training compresses the internet into a base model; fine-tuning with human feedback turns that base model into a helpful assistant; advanced training on verifiable problems rewards the model for reaching correct answers, from which deliberate reasoning strategies emerge.

Stage 1 — Pre-training builds the foundational knowledge base. Engineers take massive datasets extracted from sources like Common Crawl, representing a snapshot of the public internet — approximately fifteen trillion tokens of text. The core task given to the neural network during this phase is remarkably simple: predict the next token. Through trillions of repetitions the model learns grammar, facts, and rudimentary reasoning purely by predicting the next piece of text. The output of this enormous computational effort is, in essence, a lossy compressed representation of the internet — a web of statistical correlations that captures the patterns of human language.

Stage 2 — Supervised Fine-Tuning (SFT) and Reinforcement Learning transforms the base model from an internet document simulator into a conversational assistant. During SFT, engineers curate a dataset of ideal human-AI conversations and the model learns the format of instruction following. This is what transitions a base model like GPT-3 into InstructGPT, or a raw Llama checkpoint into a chat-ready assistant. The training targets three alignment goals known as the “Three H’s”: the model should be *Helpful* (actively solve the user’s task), *Honest* (provide accurate information and admit ignorance), and *Harmless* (refuse unsafe or illegal requests). After SFT, Reinforcement Learning from Human Feedback (RLHF) further refines the model: human evaluators rank multiple model outputs, a reward model learns from those rankings, and the main model is optimized to maximize its reward score. Through this reinforcement loop the model discovers the most effective ways to interact, aligning closely with human preferences.

Stage 3 — Advanced Training unlocks higher-order reasoning. Standard SFT models have limited working memory and must output a token immediately; they cannot pause to deliberate. Recent advances address this constraint by giving models “tokens to think” — internal reasoning chains that are generated before the final answer. This is the distinction between System 1 thinking (fast, intuitive, pattern-matching) and System 2 thinking (slow, deliberate, logical). Crucially, strategies like backtracking and self-correction are not explicitly programmed. They emerge organically during reinforcement learning when the model is rewarded only for correct final answers. This represents a fundamental shift from imitation to genuine discovery.

1.2 Tokenization: How Models See Text

Before any of this training can happen, text must be converted into a format the neural network can process. This is where tokenization enters the picture. A tokenizer breaks input text into discrete units called tokens — which may be whole words, subwords, or even individual characters — and maps each to a unique integer from the model’s vocabulary. GPT-4, for example, uses exactly 100,277 unique tokens.

Tokenization has practical consequences that business users encounter regularly. The now-famous “strawberry problem” illustrates this well: when asked how many r’s appear in the word “strawberry,” many models answer incorrectly because the tokenizer encodes the word as something like “Straw” and “berry.” The individual letters are essentially invisible to the model, so it fails at basic character-counting tasks. Understanding tokenization helps explain why LLMs struggle with spelling, letter counting, and certain text manipulations — even when they excel at higher-level language understanding. This is not a failure of reasoning but a direct consequence of how the model perceives text at the token level.

1.3 Hallucinations and the Swiss-Cheese Capability

Despite rigorous training, generative AI models are not omnipotent. Their intelligence is uneven, leading to what can be called a “Swiss cheese” capability profile. Models can be extraordinarily proficient at translation, coding, and creative writing, yet fail spectacularly at fact-checking, simple arithmetic comparisons, or counting letters.

This statistical nature also leads to hallucinations. Consider the “biography trap”: given the prompt “Who is Orson Kovats?” — a completely fictional name — the model will often confidently generate a detailed biography. It does this because it has read millions of biographies during pre-training and understands the statistical shape of biographical text. When forced to respond, it fills in the blanks of this recognizable pattern, prioritizing statistical fluency over factual accuracy. Understanding this failure mode is critical for business professionals, because it means that LLM outputs must always be verified against authoritative sources. The techniques covered in this book — retrieval-augmented generation, tool use, and grounded agents — are all designed to mitigate this fundamental limitation.

1.4 Multimodality: Beyond Text

The landscape of generative AI is no longer limited to text. We have entered the era of multimodality, where models can process images, audio, and video alongside language. Under the hood, it is all vectors: models process image pixels and sound waves using the same statistical prediction engines they use for text. Whether the input is a text string, an image, or an audio clip, the data is converted by a unified tokenizer into a sequence of tokens — text tokens, image “patch” tokens, or audio “spectrogram” tokens — and fed into the same transformer architecture as a mixed stream.

This architectural insight explains why the same retrieval-augmented generation pattern that works for text documents (covered in Chapters 8–10) also works for images (Chapter 11) and video (Chapter 11). The retrieval mechanism changes — CLIP embeddings replace text embeddings — but the retrieve-augment-generate pipeline is the same. Understanding multimodality as “the same architecture, different input modalities” demystifies what might otherwise seem like entirely separate technologies.

1.5 The Divided Ecosystem

Today the AI ecosystem is broadly divided into two camps. Proprietary models from companies like OpenAI (GPT-4, o1), Google (Gemini), and Anthropic (Claude) offer state-of-the-art performance but require API access and come with usage costs. Open-weight models from Meta (Llama), Mistral, Qwen, and others can be downloaded, inspected, and run locally — offering privacy, customization, and freedom from rate limits at the cost of requiring your own compute infrastructure.

For business practitioners, this divide creates a practical decision: use proprietary APIs for maximum capability and minimum operational overhead, or invest in open models for data privacy and cost control at scale. The `llm_cascade` package used throughout this book bridges this divide by supporting both camps — eight providers spanning proprietary and open-weight models, with automatic fallback when any single provider is unavailable. Chapter 5 covers this in detail.

1.6 From Chatbots to Agents

The progression of AI capabilities can be viewed across five distinct levels. Level 1 is conversational chatbots. Level 2 is reasoners capable of human-level problem solving. Level 3 is agents that can take autonomous actions. Level 4 is innovators that aid in invention. Level 5 is systems complex enough to do the work of entire organizations.

We are currently sitting between Level 2 and Level 3 — moving beyond models that merely talk to us and stepping into the era of agentic AI, where systems can think, plan, and act. Chapters 16 through 18 of this book trace this progression: from agentic systems that pair single-agent and multi-agent designs, to fully autonomous agents that plan their own research, and finally to tool-using agents connected to real-world services via the Model Context Protocol.

1.7 The Path Ahead

As you proceed through the hands-on exercises in the following chapters, keep this fundamental advice in mind: treat LLMs as tools, not oracles. You must always own and verify the output. Match the tool to the task — use advanced reasoning models when you need complex logic, and use standard models when you need conversational speed. By understanding the deeply statistical nature of tokenization, the journey from pre-training to reinforcement learning, and the boundaries of current capabilities, you are prepared to harness the true power of generative AI.

The chapters that follow put these concepts into practice. You will build RAG pipelines that ground model answers in retrieved evidence, fine-tune models with QLoRA to bake in new knowledge, construct agent systems that call real tools, and deploy these capabilities to production via MCP servers and no-code platforms. Every notebook is designed to be run on Google Colab — preferably with a T4 GPU, though CPU mode also works — so you can learn by doing, not just by reading.

For a related approach, see **LLM Foundations**.

1.8 Key Takeaways

1.9 Exercises

Easy: Explain in your own words why an LLM trained via next-token prediction can write coherent paragraphs but cannot reliably count the letters in a word. Relate your answer to tokenization.

Easy: Compare proprietary and open-weight models along three dimensions: cost, privacy, and capability. For each dimension, identify which camp has the advantage and why.

Medium: Given the “biography trap” example (the model inventing a biography for a fictional person), design a three-step verification protocol a business analyst should follow before trusting an LLM-generated factual claim.

Challenge: A healthcare startup wants to deploy an LLM for summarizing patient intake forms. Using the five-level AI capability framework from this chapter, identify which level this application requires, what risks exist at that level, and what guardrails (from the techniques previewed in this chapter) you would recommend.

1.10 References

- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30.
- Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35.
- Karpathy, A. (2024). *Intro to Large Language Models* [Video lecture]. YouTube.
- Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training. *OpenAI Technical Report*.
- Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., et al. (2023). Sparks of artificial general intelligence: Early experiments with GPT-4. *arXiv preprint arXiv:2303.12712*.

Chapter 2

LLM Foundations

Before building applications, you need to understand how LLMs process text. This chapter covers the four building blocks:

- **Tokenization** — how text is broken into tokens (the LLM’s “alphabet”)
- **Embeddings** — how tokens are mapped to high-dimensional vectors that capture meaning
- **TF-IDF** — a classical approach to measuring word importance (contrast with embeddings)
- **Attention** — the mechanism that lets transformers understand context and relationships between words

2.1 Tokenization: How Text Becomes Numbers

Before an LLM can do anything with text, the text has to be broken into **tokens** — small units (whole words, word pieces, or individual characters) that each get mapped to a unique integer ID. Different tokenizers split the same sentence in different ways, and those differences matter: vocabulary size, handling of case and whitespace, and how unfamiliar words get broken apart all affect what the model actually “sees.”

In this notebook we compare three tokenizers on the same short sentences: a classic **Keras** word tokenizer, the **LLaMA** tokenizer (used by the Open LLaMA family), and the **GPT-2** tokenizer (used by many OpenAI-family models). We finish with a quick look at legacy NLP tokenization (NLTK with stop-word removal and lemmatization) to contrast pre-LLM workflows, and with how a single character is represented as bits in UTF-8.

```
from tensorflow.keras.preprocessing.text import Tokenizer
from transformers import LlamaTokenizer, AutoTokenizer
```

```
# 1. Define the input text
data = [
    "The earth is spherical.",
    "The earth is a planet."
]

# 2. Initialize Tokenizer (simulating the vocabulary build)
tokenizer = Tokenizer(num_words=15, lower=True, split=' ')
tokenizer.fit_on_texts(data)

# 3. Convert text to Sequence of Integers (Token IDs)
ID_sequences = tokenizer.texts_to_sequences(data)

# Output the dictionary (Vocabulary) and the IDs
print("ID dictionary:", tokenizer.word_index)
print("ID sequences:", ID_sequences)
```

```
# 1. Load a pre-trained tokenizer
tokenizer = LlamaTokenizer.from_pretrained('openlm-research/open_llama_3b_v2')

# 2. Define the input string
prompt = 'Is the earth spherical?'

# 3. Convert text to tokens (input_ids)
input_ids = tokenizer(prompt, return_tensors="pt").input_ids

print(input_ids)
```

```
# 1. Load a larger pre-trained tokenizer
tokenizer_larger = AutoTokenizer.from_pretrained('gpt2')

# 2. Define the input string
prompt = 'Is the earth spherical?'

# 3. Convert text to tokens (input_ids)
input_ids = tokenizer_larger(prompt, return_tensors="pt").input_ids

print(input_ids)
```

2.1.0.0.1 Observations:

- *Case Sensitivity*: “Hello” vs “hello” results in different tokens.
- *Space Handling*: “hello world” (joined) vs “hello world” results in different token sequences. For instance, “hello world” might split into “hello” and “world” (with a leading space included in the token).
- *Vocabulary Size*: The compression trades off sequence length for vocabulary size. A larger vocabulary (e.g., 100k symbols) results in shorter sequences of integers

2.1.0.0.2 For comparison, let us do legacy tokenization

```
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
import string

# Download necessary NLTK data (run once)
try:
    nltk.data.find('tokenizers/punkt')
except LookupError:
    nltk.download('punkt')
# Add explicit download for 'punkt_tab' as suggested by the error traceback
try:
    # NLTK's find typically looks for specific files, but downloading 'punkt_tab' as
    # a collection
    # is suggested by the error. We will attempt to find a component of it, and if
    # not found,
    # proceed with the download.
    nltk.data.find('tokenizers/punkt_tab/english.pickle') # Check for a known file
    within punkt_tab
except LookupError:
    nltk.download('punkt_tab') # Explicitly download 'punkt_tab' collection

try:
    nltk.data.find('corpora/stopwords')
except LookupError:
    nltk.download('stopwords')
try:
    nltk.data.find('corpora/wordnet')
except LookupError:
    nltk.download('wordnet')

# 1. Define the input text
text = 'Is the earth spherical?'

print(f"Original text: '{text}'")

# 2. Tokenization (splitting into words)
tokens = word_tokenize(text)
print(f"\nTokens: {tokens}")

# 3. Lowercasing
lower_tokens = [word.lower() for word in tokens]
print(f"Lowercase tokens: {lower_tokens}")

# 4. Remove punctuation
punctuation_free_tokens = [word for word in lower_tokens if word not in
string.punctuation]
print(f"Punctuation-free tokens: {punctuation_free_tokens}")

# 5. Remove stop words
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in punctuation_free_tokens if word not in
stop_words]
print(f"Stop-word free tokens: {filtered_tokens}")

# 6. Lemmatization
lemmatizer = WordNetLemmatizer()
lemmatized_tokens = [lemmatizer.lemmatize(word) for word in filtered_tokens]
print(f"Lemmatized tokens: {lemmatized_tokens}")
```

2.1.0.0.3 Observations:

Legacy Tokenization code is long. Also, the outcome is not suitable for GenAI

2.1.1 Before we conclude:

2.1.1.1 Just for context, how do computers read alphabets in bits:

```
# 1. Define the input text
character = "a"

# 2. Encode to UTF-8
byte_data = character.encode('utf-8')

# 3. Retrieve the Integer Value (The "Emoji" or Symbol)
integer_value = byte_data[0] # Fix: Access the first byte to get its integer value

# 4. Convert to Binary string (The "1 0" representation)
bit_representation = format(integer_value, '08b')

print(f"Character: {character}")
print(f"Integer (0-255): {integer_value}")
print(f"UTF-8 Bits: {bit_representation}")
```

2.1.1.2 Key takeaways

- **Tokenization** converts raw text into integer IDs, which is the required first step before any LLM can process language.
- **Different tokenizers** (Keras word-level, LLaMA SentencePiece, GPT-2 BPE) split the same sentence into different token sequences and vocabulary sizes.
- **Case and whitespace matter** – “Hello” vs “hello” and single vs double spaces yield different token IDs in modern subword tokenizers.
- **Vocabulary size trades off** against sequence length: a larger vocab produces shorter integer sequences but consumes more memory.
- **Legacy NLP pipelines** (NLTK stop-word removal, lemmatization) strip information that generative models need and are unsuitable for LLM workflows.

2.1.1.3 Run the code

To run this notebook, copy the URL below into your browser’s address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~2 minutes on CPU

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/Tokenizer_simple_examples.ipynb

2.2 Word Embeddings: Teaching Machines the Meaning of Words

How does a computer understand that “king” and “queen” are related, or that “Paris” is to “France” what “Tokyo” is to “Japan”? The answer lies in **word embeddings** – a technique that converts words into numerical vectors (lists of numbers) that capture their meaning. Words that appear in similar contexts end up with similar vectors, which means we can actually *measure* how related two concepts are. In this notebook, we will train our own simple embeddings and then explore a powerful pre-trained model to see this in action.

```
!pip install gensim
```

2.2.1 Setting Up: Word2Vec with Gensim

We are using the `gensim` library, one of the most popular Python packages for working with word embeddings. Gensim gives us access to the **Word2Vec** algorithm (developed by Google in 2013) as well as a collection of pre-trained embedding models. Think of it as your toolkit for turning words into numbers that actually mean something.

```
from gensim.models import Word2Vec
import gensim.downloader as api
```

2.2.2 Step 1: Creating a Toy Corpus

Word2Vec learns by reading lots of text and figuring out which words tend to appear near each other. Below, we create a very small “corpus” (collection of sentences) for demonstration purposes. In the real world, you would feed Word2Vec millions of sentences – think all of Wikipedia or years of news articles. With only five short sentences, our model will not learn much, but it is enough to see *how* the process works.

```
# 1. Prepare your data (a list of lists of words)
# This small corpus is used for demonstration; a real corpus would be much larger.
sentences = [
    ['AI', 'for', 'business', 'course', 'is', 'fun'],
    ['AI', 'is', 'cool'],
    ['I', 'am', 'cool'],
    ['AI', 'uses', 'computers'],
    ['I', 'like', 'to', 'cook']
]
```

2.2.3 Step 2: Training the Word2Vec Model

Here we train Word2Vec on our tiny corpus. The key parameters control the learning process: `vector_size=100` means each word will be represented as a list of 100 numbers, `window=5` tells the model to look at words within 5 positions of each other, and `min_count=1` ensures we keep every word even if it only appears once. The model slides a window across each sentence and learns to predict nearby words – that is how it discovers meaning.

```
# 2. Train the Word2Vec model
# vector_size: Dimensionality of the word vectors (e.g., 100)
# window: Maximum distance between the current and predicted word within a sentence
# min_count: Ignores all words with total frequency lower than this
# workers: Number of CPU cores to use for training
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, workers=4)
```

2.2.4 Step 3: Examining a Word Vector

Once trained, every word in our vocabulary has its own vector – a list of 100 numbers (since we set `vector_size=100`). These numbers do not have an obvious human interpretation on their own, but their *relative positions* encode meaning. We are printing just the first 5 elements of the vector for “AI” to get a feel

for what these look like. In a well-trained model, words with similar meanings would have vectors pointing in similar directions.

```
# 3. Access the word vectors
# The trained vectors are accessible via the 'wv' attribute
word_vector = model.wv['AI']
print(f"Vector for 'AI' (first 5 elements): {word_vector[:5]}")
```

2.2.5 Step 4: Measuring Similarity Between Words

This is where embeddings get exciting. **Cosine similarity** measures how closely two word vectors point in the same direction, on a scale from -1 (opposite) to $+1$ (identical). If two words frequently appear in similar contexts, their vectors will be close and the cosine similarity will be high. Notice that with our tiny corpus, the similarity scores are not very meaningful – “AI” and “business” show roughly the same similarity as “AI” and “computers.” This is because Word2Vec needs *much* more data to learn real-world relationships.

```
# 4. Perform similarity tasks
# Find words similar to a given word
similar_words = model.wv.most_similar('AI', topn=3)
print(f"Words similar to 'AI': {similar_words}")

# Calculate cosine similarity between two words
print(f"Similarity between 'AI' and 'business': {model.wv.similarity('AI', 'business')}")
print(f"Similarity between 'AI' and 'computers': {model.wv.similarity('AI', 'computers')}")
print(f"Similarity between 'AI' and 'cook': {model.wv.similarity('AI', 'cook')}")
```

2.2.6 Using Pre-Trained Embeddings: GloVe

Instead of training from scratch on limited data, we can use **GloVe** (Global Vectors for Word Representation), a model pre-trained on billions of words from Wikipedia and news articles. Pre-trained embeddings are like hiring an expert who has already read the entire internet – they come with rich, nuanced understanding of word relationships baked in. The download may take a moment since we are pulling down 66 MB of learned word vectors.

```
# Load a pre-trained model (e.g., GloVe with 50 dimensions trained on Wikipedia)
# This download may take a few moments
wv2 = api.load('glove-wiki-gigaword-50')
```

2.2.7 Exploring GloVe Vectors

Now let us look at the vector for “computer” from the GloVe model. Notice that the numbers are much larger and more varied compared to our toy model – that is because GloVe was trained on vastly more data and has had the chance to learn meaningful patterns. These 50-dimensional vectors pack a surprising amount of semantic information into a compact numerical representation.

```
# Access the vector for a word
vector = wv2['computer']
print(f"Vector for 'computer' (first 5 elements): {vector[:5]}")
```

2.2.8 Real-World Similarity Scores

Now we see cosine similarity with a properly trained model, and the results make much more intuitive sense. “Computer” and “business” score about 0.70 – a strong relationship, since computers are widely used in business. “Cooking” and “computers” score only about 0.36 – related in the sense that they are both common English words, but semantically quite distant. These scores are the foundation of **semantic search**, recommendation systems, and many other AI applications you encounter daily.

```
# Calculate cosine similarity between two words
print(f"Similarity between 'computer' and 'business': {wv2.similarity('computer', 'business')}")
print(f"Similarity between 'cooking' and 'computers': {wv2.similarity('cooking', 'computers')}")
```

2.2.9 A Gotcha: Vocabulary Limitations

The cell below will throw a `KeyError` – and that is intentional. The GloVe model we loaded was trained on lowercase text, so it knows “ai” but not “AI” (uppercase). This is a common real-world issue: pre-trained models have a fixed vocabulary, and anything not in that vocabulary simply cannot be looked up. In practice, you would lowercase your input or choose a model with a larger vocabulary. It is a good reminder that these tools, while powerful, have practical limitations you need to be aware of.

```
print(f"Similarity between 'AI' and 'cook': {wv2.similarity('AI', 'cook')}")
```

2.2.10 Key takeaways

- **Embeddings** turn words into dense numerical vectors where proximity in vector space reflects similarity of meaning.
- **Word2Vec** learns these vectors by reading sentences and predicting which words appear near each other within a sliding context window.
- **Cosine similarity** between vectors lets you quantify how semantically related two words are on a scale from -1 to $+1$.
- **Training data matters** – a toy corpus of five sentences produces noisy similarities, while GloVe trained on billions of tokens captures intuitive relationships.
- **Pre-trained models** have fixed vocabularies, so out-of-vocabulary tokens (like uppercase “AI” in lowercase GloVe) raise errors and need preprocessing.

2.2.11 Run the code

To run this notebook, copy the URL below into your browser’s address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~3 minutes on CPU

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/Embedding_example.ipynb

2.3 TF-IDF: Finding the Words That Actually Matter

Before deep learning and fancy embeddings, there was **TF-IDF** (Term Frequency-Inverse Document Frequency) – and it is still remarkably useful today. Imagine you are searching through thousands of documents to find the most relevant ones. Common words like “the” and “is” appear everywhere and tell you nothing. TF-IDF solves this elegantly: it scores each word based on how important it is *to a specific document* relative to the entire collection. Words that are frequent in one document but rare across all documents get the highest scores. It remains a go-to baseline for text search, document classification, and keyword extraction.

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

2.3.1 Step 1: Creating Our Document Collection

We start with three tiny “documents” – just short phrases for clarity. In a real scenario, each document might be a product review, an email, or a news article. Notice that some words like “AI” and “cool” appear in multiple documents, while others like “am” appear in only one. TF-IDF will exploit exactly this pattern to determine which words are most distinctive for each document.

```
d0 = 'AI is cool'
d1 = 'I am cool'
d2 = 'AI'
string = [d0, d1, d2]
```

2.3.2 Step 2: Computing TF-IDF

Scikit-learn’s `TfidfVectorizer` handles everything in two steps. First, `fit` learns the vocabulary and computes the IDF (Inverse Document Frequency) values from the entire collection. Then, `transform` converts each document into a vector of TF-IDF scores. The result is a matrix where each row is a document and each column is a word – and each cell contains the TF-IDF score for that word in that document.

```
tfidf = TfidfVectorizer()
result = tfidf.fit_transform(string)
```

2.3.3 Step 3: Understanding IDF Values

The **IDF** (Inverse Document Frequency) score tells us how “rare” or distinctive a word is across the entire collection. Words that appear in many documents get lower IDF values, while words that appear in only one document get higher values. Here, “ai” and “cool” both appear in 2 out of 3 documents, so they get an IDF of about 1.29. Meanwhile, “am” and “is” each appear in only 1 document, earning a higher IDF of about 1.69. The rarer the word across documents, the more information it carries.

```
print('\nidf values:')
for ele1, ele2 in zip(tfidf.get_feature_names_out(), tfidf.idf_):
    print(ele1, ': ', ele2)
```

2.3.4 Step 4: The Full TF-IDF Matrix

Now we see the complete picture. The vocabulary index tells us which column corresponds to which word. The sparse matrix output shows only the non-zero entries (efficient for large datasets where most documents do not contain most words). The dense matrix view makes it easier to read: each row is a document, each column is a word, and the values are the final TF-IDF scores. Notice that document 3 (which contains only “AI”) gives “ai” a perfect score of 1.0 – it is the only word there, so it carries all the weight. The scores are also L2-normalized per row, meaning each document vector has a length of 1, which makes them ready for comparison.

```
print('\nWord indexes:')
print(tfidf.vocabulary_)
print('\ntf-idf value:')
print(result)
print('\ntf-idf values in matrix form:')
print(result.toarray())
```

2.3.5 Bonus: Handling Single-Character Words

If you noticed that the word “I” was missing from our results – good catch. By default, `TfidfVectorizer` uses a tokenizer that skips single-character words, since they are often just noise (punctuation, stray letters). But sometimes single-character words matter – like the pronoun “I” in English. The code below overrides the default by providing a custom `token_pattern` that includes single-character words. This is a practical lesson: always check what your preprocessing tools are doing behind the scenes, because defaults can quietly throw away information you care about.

```
### IF you made an astute observation that the word 'I' is not shown as a term. This
is because TfidfVectorizer by default uses a tokenizer that ignores single-character
words.
### This is often done to remove very short words or punctuation that might not
carry significant meaning.
## Below code explicitly requires TfidfVectorizer() to include single character word
in its bag of words.

tfidf_with_single_char = TfidfVectorizer(token_pattern=r'(?u)\b\w+\b')
result_with_single_char = tfidf_with_single_char.fit_transform(string)

print('\nidf values (that also include single-character words):')
for ele1, ele2 in zip(tfidf_with_single_char.get_feature_names_out(),
tfidf_with_single_char.idf_):
    print(ele1, ': ', ele2)

print('\nWord indexes:')
print(tfidf_with_single_char.vocabulary_)

print('\ntf-idf values in matrix form:')
print(result_with_single_char.toarray())
```

2.3.6 Key takeaways

- **TF-IDF** scores each word by how frequent it is in one document relative to how rare it is across the whole collection, surfacing distinctive terms.
- **IDF values** rise for words that appear in fewer documents, so “am” and “is” (in one doc each) score higher than “ai” and “cool” (in two).
- **Sklearn’s TfidfVectorizer** produces an L2-normalized sparse matrix in two lines, making each document vector ready for similarity comparison.
- **Default tokenizer** silently drops single-character words like “I” – override `token_pattern` when those tokens carry meaning.
- **Classic baselines** like TF-IDF remain practical for search, classification, and keyword extraction even in the era of deep embeddings.

2.3.7 Run the code

To run this notebook, copy the URL below into your browser’s address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~1 minute on CPU

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/TFIDF_example.ipynb

2.4 The Attention Mechanism: How AI Learns to Focus

The **attention mechanism** is the single most important innovation behind modern AI language models like ChatGPT, Gemini, and Claude. Before attention, neural networks processed text like a conveyor belt – one word at a time, left to right, hoping to remember everything along the way. Attention changed the game by letting the model look at *all* words simultaneously and decide which ones matter most for the task at hand. In this notebook, we build a simple sentiment classifier and use attention weights to see *exactly* which words the model focuses on when making its prediction.

2.4.1 Building a Sentiment Classifier with Attention

The code below does quite a lot in one cell, so let us walk through the big picture. We start with a small labeled dataset of movie reviews (positive = 1, negative = 0), duplicated 10 times to give the model more examples. The text goes through three stages: first, a `TextVectorization` layer converts words to numbers; then, an `Embedding` layer turns those numbers into meaningful vectors; and finally, an `Attention` layer lets the model weigh which words matter most. The output is a single number between 0 and 1 representing sentiment. We also capture the attention weights so we can visualize them later.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models

# 1. Dataset: 100 samples of real text

### Try improving data and see how the test prediction changes.
data = [
    ("the movie was great", 1), ("i loved the acting", 1), ("simply amazing", 1),
    ("a masterpiece of cinema", 1), ("really fun and exciting", 1),
    ("the plot was boring", 0), ("terrible acting script", 0), ("i hated it", 0),
    ("waste of time", 0), ("it was a bad experience", 0)
] * 10

texts = [item[0] for item in data]
labels = np.array([item[1] for item in data])

# 2. Text Preprocessing
max_tokens = 100
sequence_length = 5
vectorize_layer = layers.TextVectorization(max_tokens=max_tokens,
output_sequence_length=sequence_length)
vectorize_layer.adapt(texts)
X_train = vectorize_layer(texts)

# 3. Build Model
inputs = layers.Input(shape=(sequence_length,), name="input_layer")
embedding = layers.Embedding(max_tokens, 16, name="embedding_layer")(inputs)

# Built-in Keras Attention
# We set return_attention_scores=True to get the weights for visualization
attention_output, weights = layers.Attention(name="attention_layer")(
    [embedding, embedding], return_attention_scores=True
)

# Pool the results and classify
flat = layers.GlobalAveragePooling1D()(attention_output)
outputs = layers.Dense(1, activation='sigmoid', name="sentiment_output")(flat)

model = models.Model(inputs=inputs, outputs=[outputs, weights])

# 4. Use a list for loss. 'None' for the attention weights output.
model.compile(
    optimizer='adam',
    loss=['binary_crossentropy', None],
```

```

    metrics=['accuracy', None]
)

# 5. Train
print("Training model...")
### Try changing the no of epochs and see how the attention in the test data
changes.
model.fit(X_train, labels, epochs=100, verbose=0)
print("Training complete.\n")

```

2.4.2 Interpreting the Attention Weights

Now for the payoff – we feed a test sentence (“the movie was boring”) into our trained model and look at what the attention mechanism learned. The model correctly leans toward a negative sentiment. More interestingly, the attention weights reveal *why*: the word “boring” gets the highest weight (about 0.32), meaning the model focused most heavily on that word when making its decision. Words like “the” and “was” get much lower weights because they do not carry sentiment information. This interpretability is one of attention’s superpowers – unlike a black-box model, we can peek inside and see the reasoning.

```

# 6. Predict and Interpret
test_sentence = ["the movie was boring"]
X_test = vectorize_layer(test_sentence)
prediction, attention_weights = model.predict(X_test)

# Map numbers back to words
vocab = vectorize_layer.get_vocabulary()
words = [vocab[idx] for idx in X_test[0].numpy() if idx != 0]

print(f"Sentence: '{test_sentence[0]}'")
print(f"Sentiment: {'Positive' if prediction[0] > 0.5 else 'Negative'}
({prediction[0][0]:.4f})")
print("-" * 40)

# Average weights across the query dimension to see overall word importance
avg_weights = np.mean(attention_weights[0], axis=0)

for word, weight in zip(words, avg_weights):
    bar = "█" * int(weight * 40)
    print(f"{word:<10} | {weight:.4f} {bar}")

```

2.4.3 Does More Data Help? Training with a Richer Dataset

Our first model used only 10 unique reviews (duplicated). Now we scale up to 100 unique reviews – 50 positive and 50 negative – with much more varied language. This is closer to a real-world scenario where the model encounters diverse phrasing. After training on this richer dataset, notice how the sentiment prediction becomes more confident (further from 0.5) and the attention weights shift. The word “boring” still dominates, but the model now has a more nuanced understanding because it has seen many more examples of positive and negative language.

```

## Now try it with a slightly better data:

# 50 Positive Samples
pos_reviews = [
    ("the movie was great", 1), ("i loved the acting", 1), ("simply amazing", 1),
    ("a masterpiece of cinema", 1), ("really fun and exciting", 1), ("incredible
story line", 1),
    ("absolutely wonderful experience", 1), ("the best film ever", 1), ("highly
recommended movie", 1),
    ("superb performance by all", 1), ("i enjoyed every minute", 1), ("brilliant
directing", 1),
    ("a true classic film", 1), ("spectacular visuals", 1), ("i was impressed", 1),

```

```

    ("pure joy to watch", 1), ("it was very touching", 1), ("a beautiful story", 1),
    ("top notch acting", 1), ("very entertaining film", 1), ("the cinematography was
stunning", 1),
    ("captivating from start to finish", 1), ("refreshing and original", 1),
("greatest movie of year", 1),
    ("it made me happy", 1), ("an emotional journey", 1), ("wonderful cast", 1),
    ("powerful and moving", 1), ("delightful cinema", 1), ("i really liked it", 1),
    ("fantastic plot", 1), ("breathtaking scenes", 1), ("perfectly executed", 1),
    ("charming movie", 1), ("excellent work", 1), ("it was so good", 1),
    ("lovely film", 1), ("bold and brave story", 1), ("sweet and funny", 1),
    ("a must watch", 1), ("i felt inspired", 1), ("outstanding production", 1),
    ("very well made", 1), ("smart and witty", 1), ("uplifting ending", 1),
    ("cool characters", 1), ("impressive quality", 1), ("honest and raw", 1),
    ("magical feeling", 1), ("i loved it", 1)
]

# 50 Negative Samples
neg_reviews = [
    ("the plot was boring", 0), ("terrible acting script", 0), ("i hated it", 0),
    ("waste of time", 0), ("it was a bad experience", 0), ("very dull and slow", 0),
    ("worst movie ever", 0), ("extremely disappointed", 0), ("poorly directed", 0),
    ("the script was weak", 0), ("did not like it", 0), ("annoying characters", 0),
    ("boring story", 0), ("pathetic attempt at drama", 0), ("it was so noisy", 0),
    ("waste of money", 0), ("terrible directing", 0), ("ugly visuals", 0),
    ("i fell asleep", 0), ("the ending was bad", 0), ("not worth watching", 0),
    ("failed to impress", 0), ("silly plot", 0), ("boring and predictable", 0),
    ("i was very frustrated", 0), ("bad script writing", 0), ("horrible acting", 0),
    ("nothing makes sense", 0), ("clunky dialogue", 0), ("very mediocre", 0),
    ("disaster of a film", 0), ("low quality production", 0), ("uninspired film",
0),
    ("cheesy and cheap", 0), ("too long and boring", 0), ("i regret watching", 0),
    ("terrible movie", 0), ("complete failure", 0), ("badly paced", 0),
    ("offensive and loud", 0), ("pointless story", 0), ("awful experience", 0),
    ("it was painful", 0), ("messy plot", 0), ("boring and dry", 0),
    ("zero stars", 0), ("worst film of year", 0), ("really bad", 0),
    ("uninteresting and plain", 0), ("it was terrible", 0)
]

data = pos_reviews + neg_reviews
np.random.shuffle(data) # Good practice to shuffle!

texts = [item[0] for item in data]
labels = np.array([item[1] for item in data])

# 2. Text Preprocessing
max_tokens = 100
sequence_length = 5
vectorize_layer = layers.TextVectorization(max_tokens=max_tokens,
output_sequence_length=sequence_length)
vectorize_layer.adapt(texts)
X_train = vectorize_layer(texts)

# 3. Build Model
inputs = layers.Input(shape=(sequence_length,), name="input_layer")
embedding = layers.Embedding(max_tokens, 16, name="embedding_layer")(inputs)

# Built-in Keras Attention
# We set return_attention_scores=True to get the weights for visualization
attention_output, weights = layers.Attention(name="attention_layer")(
    [embedding, embedding], return_attention_scores=True
)

# Pool the results and classify

```

```

flat = layers.GlobalAveragePooling1D()(attention_output)
outputs = layers.Dense(1, activation='sigmoid', name="sentiment_output")(flat)

model = models.Model(inputs=inputs, outputs=[outputs, weights])

# 4. Use a list for loss. 'None' for the attention weights output.
model.compile(
    optimizer='adam',
    loss=['binary_crossentropy', None],
    metrics=['accuracy', None]
)

# 5. Train
print("Training model...")
#### Try changing the no of epochs and see how the attention in the test data
changes.
model.fit(X_train, labels, epochs=100, verbose=0)
print("Training complete.\n")

# 6. Predict and Interpret
test_sentence = ["the movie was boring"]
X_test = vectorize_layer(test_sentence)
prediction, attention_weights = model.predict(X_test)

# Map numbers back to words
vocab = vectorize_layer.get_vocabulary()
words = [vocab[idx] for idx in X_test[0].numpy() if idx != 0]

print(f"Sentence: '{test_sentence[0]}'")
print(f"Sentiment: {'Positive' if prediction[0] > 0.5 else 'Negative'}
({prediction[0][0]:.4f})")
print("-" * 40)

# Average weights across the query dimension to see overall word importance
avg_weights = np.mean(attention_weights[0], axis=0)

for word, weight in zip(words, avg_weights):
    bar = "█" * int(weight * 40)
    print(f"{word:<10} | {weight:.4f} {bar}")

```

2.4.4 What Happens Without Attention?

To truly appreciate what attention does, let us remove it entirely. The cell below builds the same model architecture but skips the attention layer – the embeddings go straight to pooling and classification. Without attention, the model treats every word as equally important. It can still learn sentiment to some degree (the prediction is still somewhat negative), but it loses the ability to *focus* on the words that matter most. Notice that we can no longer visualize per-word importance because there are no attention weights to inspect. This is why attention was such a breakthrough: it gives models both better performance *and* interpretability.

```

#### Let us try without the attention layer:

# 50 Positive Samples
pos_reviews = [
    ("the movie was great", 1), ("i loved the acting", 1), ("simply amazing", 1),
    ("a masterpiece of cinema", 1), ("really fun and exciting", 1), ("incredible
story line", 1),
    ("absolutely wonderful experience", 1), ("the best film ever", 1), ("highly
recommended movie", 1),
    ("superb performance by all", 1), ("i enjoyed every minute", 1), ("brilliant
directing", 1),
    ("a true classic film", 1), ("spectacular visuals", 1), ("i was impressed", 1),
    ("pure joy to watch", 1), ("it was very touching", 1), ("a beautiful story", 1),

```

```

    ("top notch acting", 1), ("very entertaining film", 1), ("the cinematography was
stunning", 1),
    ("captivating from start to finish", 1), ("refreshing and original", 1),
("greatest movie of year", 1),
    ("it made me happy", 1), ("an emotional journey", 1), ("wonderful cast", 1),
    ("powerful and moving", 1), ("delightful cinema", 1), ("i really liked it", 1),
    ("fantastic plot", 1), ("breathtaking scenes", 1), ("perfectly executed", 1),
    ("charming movie", 1), ("excellent work", 1), ("it was so good", 1),
    ("lovely film", 1), ("bold and brave story", 1), ("sweet and funny", 1),
    ("a must watch", 1), ("i felt inspired", 1), ("outstanding production", 1),
    ("very well made", 1), ("smart and witty", 1), ("uplifting ending", 1),
    ("cool characters", 1), ("impressive quality", 1), ("honest and raw", 1),
    ("magical feeling", 1), ("i loved it", 1)
]

# 50 Negative Samples
neg_reviews = [
    ("the plot was boring", 0), ("terrible acting script", 0), ("i hated it", 0),
    ("waste of time", 0), ("it was a bad experience", 0), ("very dull and slow", 0),
    ("worst movie ever", 0), ("extremely disappointed", 0), ("poorly directed", 0),
    ("the script was weak", 0), ("did not like it", 0), ("annoying characters", 0),
    ("boring story", 0), ("pathetic attempt at drama", 0), ("it was so noisy", 0),
    ("waste of money", 0), ("terrible directing", 0), ("ugly visuals", 0),
    ("i fell asleep", 0), ("the ending was bad", 0), ("not worth watching", 0),
    ("failed to impress", 0), ("silly plot", 0), ("boring and predictable", 0),
    ("i was very frustrated", 0), ("bad script writing", 0), ("horrible acting", 0),
    ("nothing makes sense", 0), ("clunky dialogue", 0), ("very mediocre", 0),
    ("disaster of a film", 0), ("low quality production", 0), ("uninspired film",
0),
    ("cheesy and cheap", 0), ("too long and boring", 0), ("i regret watching", 0),
    ("terrible movie", 0), ("complete failure", 0), ("badly paced", 0),
    ("offensive and loud", 0), ("pointless story", 0), ("awful experience", 0),
    ("it was painful", 0), ("messy plot", 0), ("boring and dry", 0),
    ("zero stars", 0), ("worst film of year", 0), ("really bad", 0),
    ("uninteresting and plain", 0), ("it was terrible", 0)
]

data = pos_reviews + neg_reviews
np.random.shuffle(data) # Good practice to shuffle!

texts = [item[0] for item in data]
labels = np.array([item[1] for item in data])

# 2. Text Preprocessing
max_tokens = 100
sequence_length = 5
vectorize_layer = layers.TextVectorization(max_tokens=max_tokens,
output_sequence_length=sequence_length)
vectorize_layer.adapt(texts)
X_train = vectorize_layer(texts)

# 3. Build Model
inputs = layers.Input(shape=(sequence_length,), name="input_layer")
embedding = layers.Embedding(max_tokens, 16, name="embedding_layer")(inputs)

# Built-in Keras Attention

### COMMENTED
# attention_output, weights = layers.Attention(name="attention_layer")(
#     [embedding, embedding], return_attention_scores=True
# )

# Pool the results and classify

```



```

## REPLACED this line to use embedding directly
# flat = layers.GlobalAveragePooling1D()(attention_output)
flat = layers.GlobalAveragePooling1D()(embedding)

outputs = layers.Dense(1, activation='sigmoid', name="sentiment_output")(flat)

# 5. Train
model.fit(X_train, labels, epochs=100, verbose=0)

## Weights not required
# model = models.Model(inputs=inputs, outputs=[outputs, weights])
model = models.Model(inputs, outputs)

#### model.compile is simpler without attention layer
# 4. Use a list for loss. 'None' for the attention weights output.
#model.compile(
#    optimizer='adam',
#    loss=['binary_crossentropy', None],
#    metrics=['accuracy', None]
#)
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# 5. Train
print("Training model...")
#### Try changing the no of epochs and see how the attention in the test data
changes.
model.fit(X_train, labels, epochs=100, verbose=0)
print("Training complete.\n")

# 6. Predict and Interpret
test_sentence = ["the movie was boring"]
X_test = vectorize_layer(test_sentence)

## Since we do not have attention_weights, this line also changes
## prediction, attention_weights = model.predict(X_test)
prediction = model.predict(X_test)

# Map numbers back to words
vocab = vectorize_layer.get_vocabulary()
words = [vocab[idx] for idx in X_test[0].numpy() if idx != 0]

print(f"Sentence: '{test_sentence[0]}'")
print(f"Sentiment: {'Positive' if prediction[0] > 0.5 else 'Negative'}
({prediction[0][0]:.4f})")
print("-" * 40)

#### ATTENTION WEIGHTS NOT AVAILABLE FOR THIS MODEL
# Average weights across the query dimension to see overall word importance
#avg_weights = np.mean(attention_weights[0], axis=0)

#for word, weight in zip(words, avg_weights):
#    bar = "█" * int(weight * 40)
#    print(f"{word:<10} | {weight:.4f} {bar}")

```

2.4.5 Key takeaways

- **Attention** lets a model look at every word in a sequence at once and learn which ones matter most for the task.
- **Attention weights** are inspectable, so you can see exactly which words (like “boring”) drove the sentiment prediction.

- **Dataset size and diversity** strongly affect confidence – expanding from 10 repeated reviews to 100 varied ones sharpens both predictions and attention patterns.
 - **Removing the attention layer** collapses per-word importance into uniform pooling, sacrificing both interpretability and the ability to focus on signal-carrying tokens.
 - **Keras provides** a drop-in `layers.Attention` that integrates with embeddings and returns weights via `return_attention_scores=True`.
-

2.4.6 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~5 minutes on T4 GPU

```
https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/Attention_simple_example.ipynb
```

2.5 Simple Transformers: Building a Sentiment Classifier from Scratch

Transformers are the architecture behind every modern large language model, from GPT to BERT to Gemini. At their core, transformers use a mechanism called **self-attention** that lets the model weigh the importance of every word relative to every other word in a sentence. This is fundamentally different from older architectures like RNNs, which process words one at a time in sequence.

The original transformer paper (“Attention Is All You Need”, 2017) introduced an **encoder-decoder** architecture. The encoder reads the input and builds a rich representation; the decoder generates the output. For classification tasks like sentiment analysis, we only need the encoder half – which is exactly what this notebook demonstrates.

In this notebook we build a transformer encoder from scratch using TensorFlow, train it on real IMDb movie reviews, and then compare our hand-built model to a pre-trained DistilBERT model loaded in a single line of code. The contrast is instructive: building from scratch teaches you what is happening inside; using a pre-trained model shows you how the industry actually works.

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np
```

2.5.1 Loading and Preparing the IMDb Dataset

We start by loading the IMDb movie review dataset, which contains 25,000 training and 25,000 test reviews, each labeled as positive or negative. The reviews are already tokenized into integer sequences – each integer maps to a word in a fixed vocabulary of 10,000 words. We pad (or truncate) every review to exactly 200 tokens so the transformer receives fixed-length inputs.

```
# 1. Load Real Data: IMDb Movie Reviews
# We limit to 10,000 words and 200 words per review for speed
vocab_size = 10000
maxlen = 200

(x_train, y_train), (x_val, y_val) =
tf.keras.datasets.imdb.load_data(num_words=vocab_size)
x_train = tf.keras.preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_val = tf.keras.preprocessing.sequence.pad_sequences(x_val, maxlen=maxlen)

# 2. Define the Transformer Encoder Block
class TransformerBlock(layers.Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super().__init__()
        self.att = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
        self.ffn = models.Sequential([
            layers.Dense(ff_dim, activation="relu"),
            layers.Dense(embed_dim),
        ])
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)
        self.dropout1 = layers.Dropout(rate)
        self.dropout2 = layers.Dropout(rate)

    def call(self, inputs, training=True):
        # Multi-Head Self-Attention
        attn_output = self.att(inputs, inputs)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output) # Residual connection

        # Feed Forward Network
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output) # Residual connection
```

```
# 3. Handle Token + Positional Embedding
class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, maxlen, vocab_size, embed_dim):
        super().__init__()
        self.token_emb = layers.Embedding(input_dim=vocab_size,
output_dim=embed_dim)
        self.pos_emb = layers.Embedding(input_dim=maxlen, output_dim=embed_dim)

    def call(self, x):
        maxlen = tf.shape(x)[-1]
        positions = tf.range(start=0, limit=maxlen, delta=1)
        positions = self.pos_emb(positions)
        x = self.token_emb(x)
        return x + positions # Adding position to meaning
```

2.5.2 The Transformer Block and Positional Embeddings

The code above defines two key components. The `TransformerBlock` implements multi-head self-attention followed by a feed-forward network, with residual connections and layer normalization – this is the exact architecture from the original paper. The `TokenAndPositionEmbedding` layer converts each token ID into a dense vector and adds a positional embedding so the model knows the order of words. Without positional embeddings, the transformer would treat “dog bites man” and “man bites dog” identically.

```
# 4. Build the Final Classification Model
embed_dim = 32 # Embedding size for each token
num_heads = 2 # Number of attention heads
ff_dim = 32 # Hidden layer size in feed forward network inside transformer

inputs = layers.Input(shape=(maxlen,))
embedding_layer = TokenAndPositionEmbedding(maxlen, vocab_size, embed_dim)
x = embedding_layer(inputs)
transformer_block = TransformerBlock(embed_dim, num_heads, ff_dim)
x = transformer_block(x)
x = layers.GlobalAveragePooling1D()(x) # Summarize the sequence
x = layers.Dropout(0.1)(x)
x = layers.Dense(20, activation="relu")(x)
outputs = layers.Dense(2, activation="softmax")(x)

model = models.Model(inputs=inputs, outputs=outputs)
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy",
metrics=["accuracy"])

# 5. Train on Real Data
print("Training Transformer on IMDB Dataset...")
model.fit(x_train, y_train, batch_size=32, epochs=2, validation_data=(x_val, y_val))
```

2.5.2.1 Building and Training the Classification Model

Here we assemble the full model: token+position embeddings feed into the transformer block, whose output is pooled into a single vector via global average pooling, then passed through a dense layer to produce a two-class (positive/negative) prediction. We train for just two epochs on the full IMDB training set. Even with this minimal configuration (32-dim embeddings, 2 attention heads), the model reaches around 87% validation accuracy. This demonstrates that the self-attention mechanism is remarkably effective even at small scale.

```
# 4. Predict on your Specific Test Reviews
test_reviews = [
    "this movie was an absolute masterpiece with brilliant acting",
    "i hated every minute of this film the plot was a total disaster"
]
```

```

# We must use the same word index mapping used for training
word_index = tf.keras.datasets.imdb.get_word_index()

def preprocess_text(texts):
    encoded_texts = []
    for text in texts:
        # Convert words to IMDb indices
        tokens = text.lower().split()
        sequence = [word_index.get(word, 0) + 3 for word in tokens] # +3 is an IMDb
dataset quirk
        encoded_texts.append(sequence)
    return tf.keras.preprocessing.sequence.pad_sequences(encoded_texts,
maxlen=maxlen)

X_test = preprocess_text(test_reviews)
predictions = model.predict(X_test)

# 5. Output Results
print("\n--- Manual Transformer Results ---")
for i, review in enumerate(test_reviews):
    sentiment = "Positive" if predictions[i][1] > 0.5 else "Negative"
    print(f"Review: {review}")
    print(f"Prediction: {sentiment} ({predictions[i][0]:.4f})\n")

```

2.5.3 Running Predictions on Custom Reviews

Now we test the trained model on two hand-crafted reviews – one clearly positive, one clearly negative. We have to manually tokenize these reviews using the same word index that IMDb uses, which highlights an important point: the tokenization scheme must match between training and inference. The model correctly identifies the sentiment of both reviews, confirming that our from-scratch transformer has learned meaningful representations.

```

import tensorflow as tf

# 1. Load the dataset (limiting to 10,000 most frequent words)
(x_train, y_train), _ = tf.keras.datasets.imdb.load_data(num_words=10000)

# 2. Get the word index (dictionary)
word_index = tf.keras.datasets.imdb.get_word_index()

# 3. Create a reverse word index to map integers back to words
# We shift by 3 because 0, 1, and 2 are reserved for <PAD>, <START>, and <UNK>
reverse_word_index = {value + 3: key for (key, value) in word_index.items()}
reverse_word_index[0] = "<PAD>"
reverse_word_index[1] = "<START>"
reverse_word_index[2] = "<UNK>"
reverse_word_index[3] = "<UNUSED>"

def decode_review(text_ids):
    return ' '.join([reverse_word_index.get(i, '?') for i in text_ids])

# Read and print the 5th to 10th records
# Python uses 0-based indexing, so the 5th record is index 4, and the 10th is index
9.
# The range function is exclusive of the stop value, so range(4, 10) will include
indices 4, 5, 6, 7, 8, 9.
print("Querying records from 5th to 10th:")
for i in range(4, 10):
    print(f"--- Record {i+1} ---") # Displaying record number from 1-based
perspective
    print(f"Label: {y_train[i]} (1 = Positive, 0 = Negative)")

```

```
print(f"Text: {decode_review(x_train[i][:50])}...") # Printing first 50 words
print("\n")
```

2.5.4 Exploring the Raw Training Data

Before moving on, let us decode a few training examples back into readable English. This step is important for understanding what the model actually sees – the `<UNK>` tokens show words that fell outside our 10,000-word vocabulary, and the `<START>` token marks the beginning of each review. Inspecting raw data is a good habit: it reveals preprocessing artifacts that can affect model performance.

```
### Why is the label inversion happening? NOT SURE

# 1. Load the dataset (limiting to 10,000 most frequent words)
(x_train, y_train), _ = tf.keras.datasets.imdb.load_data(num_words=10000)

# 2. Get the word index (dictionary)
word_index = tf.keras.datasets.imdb.get_word_index()

# 3. Create a reverse word index to map integers back to words
# We shift by 3 because 0, 1, and 2 are reserved for <PAD>, <START>, and <UNK>
reverse_word_index = {value + 3: key for (key, value) in word_index.items()}
reverse_word_index[0] = "<PAD>"
reverse_word_index[1] = "<START>"
reverse_word_index[2] = "<UNK>"
reverse_word_index[3] = "<UNUSED>"

def decode_review(text_ids):
    return ' '.join([reverse_word_index.get(i, '?') for i in text_ids])

# 4. Read and print few sample records
# for i in range(2):
for i in range(4,10):
    print(f"--- Record {i} ---")
    print(f"Label: {y_train[i]} (1 = Positive, 0 = Negative)")
    print(f"Text: {decode_review(x_train[i][:50])}...") # Printing first 50 words
    print("\n")
```

2.5.4.1 Verifying Labels Against Review Text

This cell repeats the decoding exercise with a slightly different range of records. Reading the decoded text alongside the labels helps us build intuition for what “positive” and “negative” mean in this dataset. The comment about label inversion is a useful reminder that real datasets sometimes have quirks that require investigation.

```
### Let's do it the easier way with BERT (Note that there is no training involved here)

from transformers import pipeline

# 1. The Single-Line Model: Load a pre-trained Transformer (DistilBERT)
# This handles tokenization, encoding, and the classification head automatically.
classifier = pipeline("sentiment-analysis", model="distilbert-base-uncased-finetuned-sst-2-english")

# 2. Test it on real data (Supervised Learning Inference)
test_reviews = [
    "This movie was an absolute masterpiece with brilliant acting.",
    "I hated every minute of this film; the plot was a total disaster."
]

results = classifier(test_reviews)
```

```
# 3. Print the results
for review, result in zip(test_reviews, results):
    print(f"Review: {review}")
    print(f"Result: {result['label']} (Confidence: {result['score']:.4f})\n")
```

2.5.5 The Easy Way: Pre-trained DistilBERT in One Line

After building a transformer from scratch, we now load a **pre-trained** DistilBERT model using Hugging Face's `pipeline` API. This single line of code downloads a model that was trained on millions of examples and fine-tuned specifically for sentiment analysis – no training required on our end. The confidence scores (0.9999 for positive, 0.9998 for negative) are dramatically higher than our from-scratch model, illustrating the power of transfer learning. In practice, you will almost always use pre-trained models; building from scratch is for understanding, not production.

2.5.6 Key takeaways

- **Transformer encoders** combine multi-head self-attention, feed-forward layers, residual connections, and layer normalization into a repeatable block.
- **Positional embeddings** are required because attention is order-agnostic – they tell the model where each token sits in the sequence.
- **A small from-scratch transformer** (32-dim embeddings, 2 heads) reaches roughly 87% accuracy on IMDb in just two epochs, showing attention's efficiency.
- **Tokenization consistency** between training and inference is essential – custom reviews must use the same IMDb word index the model was trained on.
- **Pre-trained DistilBERT** via Hugging Face's `pipeline` delivers near-perfect sentiment predictions in one line, illustrating why transfer learning dominates production.

2.5.7 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~5 minutes on T4 GPU

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/SimpleTransformers.ipynb

2.6 Attention in Image Analysis: Seeing What the Model Sees

The attention mechanism that makes transformers so powerful for text also works remarkably well for images. The **Vision Transformer (ViT)**, introduced by Google in 2020, applies the same self-attention architecture to image classification by splitting an image into a grid of patches and treating each patch like a “word” in a sentence.

This notebook demonstrates how attention extends from text to images. We load a pre-trained ViT model, feed it an image, and then **visualize the attention heatmap** to see which parts of the image the model focused on when making its prediction. The heatmap reveals that the model learns to attend to semantically meaningful regions – the animal’s face, distinctive features, or key objects – rather than processing the entire image uniformly.

Understanding attention visualization is valuable for model interpretability: it lets you verify that the model is making predictions for the right reasons, not just exploiting spurious correlations in the background.

```
import torch
from transformers import ViTImageProcessor, ViTForImageClassification
from PIL import Image
import matplotlib.pyplot as plt
import requests
import numpy as np
```

2.6.1 Loading the Vision Transformer and Required Libraries

We import PyTorch, the Hugging Face ViT model, and visualization libraries. The Vision Transformer (ViT) processes images by dividing them into 16x16 pixel patches, embedding each patch into a vector, and then running standard transformer self-attention over these patch embeddings. This is conceptually identical to how a text transformer processes word tokens – the key insight that bridges NLP and computer vision.

```
# 1. Load the Model and Processor
model_name = 'google/vit-base-patch16-224'
processor = ViTImageProcessor.from_pretrained(model_name)
model = ViTForImageClassification.from_pretrained(model_name,
output_attentions=True)

# 2. Load an Image (Using a clear cat image)
# url = 'http://images.cocodataset.org/val2017/000000039769.jpg' # Two cats on a
remote
#url = 'https://pawpals.ae/wp-content/uploads/2023/03/Shorthair-cat-1-1024x679.jpg'
#url = 'https://www.alleycat.org/wp-content/uploads/2018/08/SocialMediaLink_1200x
628_GenericCat-Centered.png'
url = 'https://d35441alu8djza.cloudfront.net/APHI/Blog/2023/September/small-breeds-
hero.jpg'
image = Image.open(requests.get(url, stream=True).raw)

# 3. Process and Predict
inputs = processor(images=image, return_tensors="pt")
outputs = model(**inputs)

# Find the label with the highest score
logits = outputs.logits
predicted_class_idx = logits.argmax(-1).item()
print(f"Predicted Class: {model.config.id2label[predicted_class_idx]}")

# 4. Extract Attention for that specific prediction
# We take the last layer's attention (Layer 11)
attentions = outputs.attentions[-1]

# Average the attention weights across all 'heads'
# We focus on the [CLS] token (index 0) which is the 'Decision Maker'
# The [CLS] token's attention toward the 196 image patches (index 1 to 197)
```

```

nh = attentions.shape[1] # Number of heads
attentions = attentions[0, :, 0, 1:].reshape(nh, -1)
val_attn = attentions.mean(dim=0) # Average across heads

# 5. Reshape and Resize Heatmap
grid_size = int(np.sqrt(val_attn.shape[0])) # 14x14
heatmap = val_attn.reshape(grid_size, grid_size).detach().numpy()

# 6. Plotting
fig, ax = plt.subplots(1, 2, figsize=(12, 6))

ax[0].imshow(image)
ax[0].set_title("Input Image")
ax[0].axis('off')

# Overlay the heatmap
ax[1].imshow(image)
ax[1].imshow(heatmap, cmap='magma', alpha=0.7, extent=(0, image.size[0],
image.size[1], 0))
ax[1].set_title(f"Attention Spotlight for:
{model.config.id2label[predicted_class_idx]}")
ax[1].axis('off')

plt.show()

```

2.6.2 Processing an Image and Extracting Attention Maps

This cell does the heavy lifting: we load the pre-trained ViT model (with `output_attentions=True` to capture the attention weights), download a test image, and run inference. After getting the predicted class, we extract the attention weights from the last transformer layer and focus on the `[CLS]` token – this is the special classification token whose attention pattern tells us which image patches were most important for the final prediction.

The resulting 14x14 heatmap is overlaid on the original image, creating an intuitive visualization. Bright regions in the heatmap indicate patches that received high attention – these are the areas the model considered most informative for its classification decision. This kind of visualization is a powerful tool for debugging and explaining model behavior to non-technical stakeholders.

2.6.3 Key takeaways

- **Vision Transformers** apply the same self-attention mechanism used on text to images, by splitting an image into 16x16 patches treated like word tokens.
- **Attention heatmaps** reveal which image regions the model focused on, turning a black-box prediction into a visual explanation.
- **The [CLS] token** aggregates information across patches, and its attention weights toward each patch drive the classification decision.
- **Pre-trained ViT** from Hugging Face (`google/vit-base-patch16-224`) can be loaded with `output_attentions=True` to expose these weights for analysis.
- **Interpretability matters** in production: visualizing attention helps verify a model is relying on meaningful features rather than spurious background cues.

2.6.4 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~3 minutes on T4 GPU

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/Attention_in_Image_Analysis.ipynb

Chapter 3

Prompt Engineering with Google AI Studio

Every interaction with a large language model begins with a prompt — the text you send in. The quality of that prompt often makes the difference between a useful answer and a generic one, or between hitting the right output format on the first try and iterating three times to get there. Modern models are forgiving enough that a one-line question usually works, but a little discipline about *how* you ask pays disproportionate dividends — especially when the output will be consumed by a downstream system rather than read by a human.

This chapter is deliberately hands-on rather than theoretical. We work through the core prompting techniques using **Google AI Studio** (<https://aistudio.google.com/>), a free browser-based playground that gives direct access to Gemini models without writing code. Everything here transfers directly to the API-based notebooks in the chapters that follow: when you call `llm.generate(prompt)` in Python, `prompt` is just the string you learned to construct in AI Studio. Settle into the playground first; move the patterns into code later.

3.1 Zero-Shot, Few-Shot, and the System Prompt

The simplest prompt is **zero-shot**: you describe the task, provide the input, and rely on the model’s pre-trained knowledge to produce the output.

```
Classify the text into neutral, negative, or positive.  
Text: I think the food was okay.  
Sentiment:
```

Zero-shot prompts work well for anything the model has seen many similar examples of during pre-training — sentiment classification, summarization, translation, simple Q&A. They fail when the task is domain-specific, when the output format matters, or when your notion of “correct” does not match the model’s defaults.

Few-shot prompting addresses the second and third problems by embedding a handful of input-output examples directly in the prompt, so the model can see what a good response looks like before generating its own.

```
Classify the text into neutral, negative, or positive.  
  
Example: The food was awesome.  
Sentiment: positive  
  
Text: I think the vacation is okay.  
Sentiment:
```

Research has surfaced a counterintuitive finding about why few-shot actually works in large models: **larger models learn mostly from the *format* of examples, not from the labels themselves**. Wei et al. (2023) showed that deliberately flipping the labels in few-shot examples barely hurts large-model accuracy on many tasks, while the structural consistency of the examples matters enormously. The practical implication for you is to put your energy into keeping the example format clean and consistent, rather than agonizing over which particular examples are most “informative.”

A third prompting scaffold is the **system prompt** — persistent context that applies to every user turn in a conversation. In AI Studio you set the system prompt in a dedicated field before starting a chat; in the Python API you pass it as a separate `system_instruction` argument. System prompts are where to put the persona, guardrails, and formatting requirements that should survive across an entire session, so they do not have to be repeated in every user turn.

3.2 The PICFAT Framework

For prompts that go beyond a one-line question, six components together cover almost everything you might need to specify. We call this checklist the **PICFAT framework**:

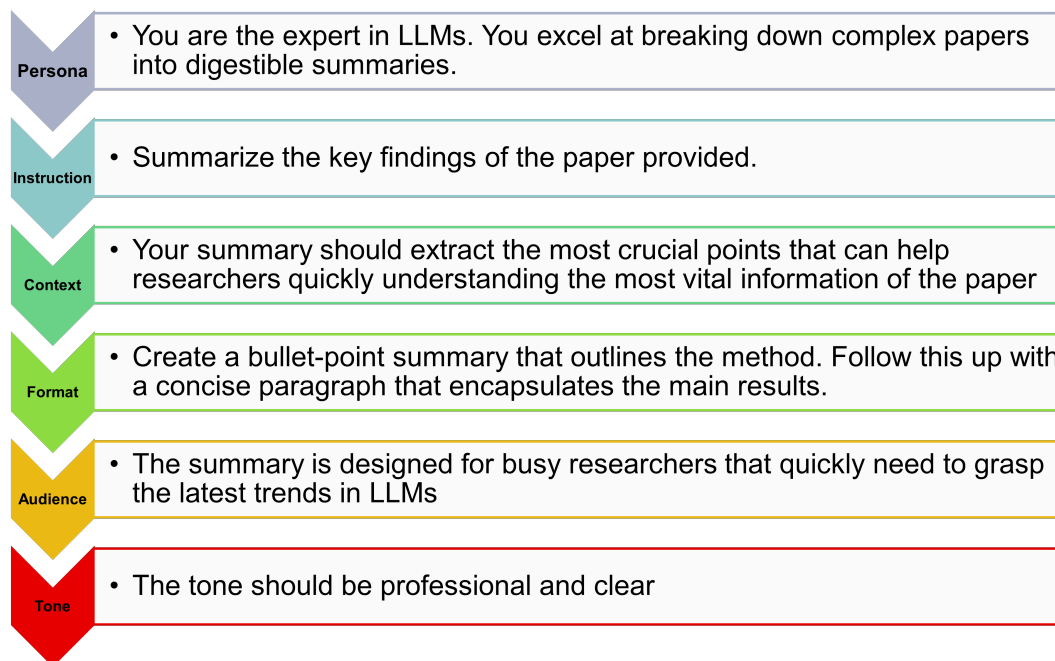


Figure 3.1: The PICFAT framework: Persona, Instruction, Context, Format, Audience, Tone. Only *Instruction* is strictly required; the other five are optional but each becomes valuable as the task grows in specificity.

- **Persona** — *identity and expertise*: who the AI is pretending to be. Sets the vocabulary, the depth of explanation, and the perspective.
- **Instruction** — *the core mission*: the specific, measurable goal. Prevents the model from drifting into related-but-irrelevant tasks.
- **Context** — *constraints and grounding*: background information or data the model should treat as authoritative.
- **Format** — *structure and output*: how the response should be organized. Saves you from reformatting afterwards (Markdown, JSON, tables, bullet lists).
- **Audience** — *the end user*: who will read the output. Calibrates assumed knowledge and complexity.
- **Tone** — *stylistic nuance*: the “vibe” of the response.

If your task requires data (a document to summarize, a table to analyze, a code snippet to review), that data goes in a seventh slot, typically fenced off with a code block or a clear delimiter so the model does not confuse the data with the instructions.

In practice, Persona and Format are the two you will add most often. Audience and Tone matter more the closer the output sits to content that will be read by humans, versus consumed by another system. The six-component prompt below is a useful reference template — note how each section maps cleanly to one of the PICFAT slots, with Markdown headings making the structure explicit:

```
# Persona
You are an expert in large language models. You excel at breaking down complex
papers into digestible summaries.

# Instruction
Summarize the key findings of the paper provided.
```

```

# Context
The summary should extract the most crucial points that help researchers quickly
understand the vital information.

# Format
Create a bullet-point summary that outlines the method, followed by a concise
paragraph that encapsulates the main results.

# Audience
Busy researchers who need to grasp the latest trends in LLMs.

# Tone
Professional and clear.

---

[Paste paper text here]

```

3.3 Prompting Reasoning Models

The PICFAT framework was shaped by what worked on earlier instruction-tuned models like GPT-3.5, Llama 3, and the original Gemini. A newer class of models — OpenAI’s o1, Gemini 2.5 with thinking enabled, DeepSeek-R1, and others — is trained to generate internal reasoning chains *before* the final answer. These **reasoning models** do not need you to tell them to “think step by step” — they have been explicitly rewarded for doing so during training. What they benefit from instead is a tighter specification of what counts as a correct answer and what the response must avoid.

- **Role:** Expert [Job Title] with [X] years of experience.
- **Task:** [Action Verb] a [Deliverable].
- **Background:** Use the attached [Data/Context] to ensure [Specific Goal].
- **Constraints:** Do not include [X]; prioritize [Y].
- **Style:** [Tone] writing for a [Audience] audience.
- **Output:** Present as a [Format].

Figure 3.2: The reasoning-model template: Role, Task, Background, Constraints, Style, Output. Compared to PICFAT, the emphasis shifts away from “how should you think” toward “what does a correct answer look like, and what must it avoid?”

- **Role** — who the AI is playing (usually more precise than Persona: “*Expert [Job Title] with [X] years of experience*”).
- **Task** — the action verb and the deliverable, stated directly.
- **Background** — data or context the model must use to ground the answer.
- **Constraints** — what to avoid and what to prioritize. This is the single most valuable slot for reasoning models.
- **Style** — the writing register for the intended audience.
- **Output** — the final deliverable’s shape (bullets, table, code, prose, JSON).

A concrete reasoning-model prompt that exercises all six slots:

```

Role: Senior AI architect with a talent for clear, analogical teaching.

Task: Deconstruct and explain the mechanism and value proposition of Retrieval-
Augmented Generation (RAG).

Background: Contrast RAG against "parametric knowledge" (what the model learned
during training) to show why RAG is necessary for reducing hallucinations and using
private data.

Constraints: Use a "first principles" reasoning approach. Do not just define the

```

acronym; explain the flow of data from query to vector database to augmented prompt.

Style: Analytical yet accessible, written for a product manager who understands tech but does not code.

Output: A structured breakdown including a "library vs. encyclopedia" analogy, a three-step technical workflow, and a brief "when to use RAG vs. fine-tuning" table.

A related research finding worth knowing: Kojima et al. (2022) discovered that simply appending “Let’s think step by step” to an ordinary prompt — with no examples at all — substantially improves performance on reasoning benchmarks when used with *non*-reasoning models. This “zero-shot chain-of-thought” trick revealed that step-by-step reasoning was already latent in pre-trained models and just needed the right surface trigger. Reasoning models effectively bake this trigger in at the training stage, which is why you no longer have to say it out loud.

3.4 Sandwich Prompting and Markdown Structure

When a prompt grows long — many paragraphs of context, multiple few-shot examples, or a large data block — the model’s attention tends to weigh the start and end of the prompt more heavily than the middle, a phenomenon often called “lost in the middle.” **Sandwich prompting** is the practical countermeasure: state the critical instruction at the beginning *and* repeat a condensed version of it at the end, sandwiching the long context in between.

Instruction (what to do, in one sentence).

[Long context / data / examples]

Reminder: (one-sentence restatement of the instruction).

This is especially helpful for extraction tasks where you are asking the model to pull specific fields out of a long document. Without the reminder, the model often drifts into summarizing or paraphrasing by the time it reaches the end of the context.

Related to structural discipline: **Markdown inside your prompt** helps the model distinguish between instructions, context, and data. Models have seen enough Markdown during pre-training that they respect its semantics.

Element	Syntax	Best use
Heading	<code>## Section</code>	Defining distinct blocks (Task, Context, Constraints)
Bold	<code>**Text**</code>	Non-negotiable rules
Code block	<code>```</code>	Isolating data or “don’t touch” text
Blockquote	<code>> Text</code>	Few-shot example inputs or outputs
Lists	<code>1.</code> or <code>*</code>	Step-by-step instructions or checklist-style requirements

When in doubt, over-structure rather than under-structure. The cost of extra Markdown is a handful of tokens; the benefit is often a dramatically cleaner response.

3.5 Controlling Behavior: Temperature and Friends

Beyond the prompt itself, every LLM exposes a handful of generation parameters that shape the output. In AI Studio these live in the right-hand panel.

Temperature (usually 0.0–1.0, sometimes up to 2.0) controls how random the model’s sampling is. Low values force the model to pick the most likely token at every step (deterministic, factual); high values encourage exploration (creative, diverse).

Range	Use case	Effect

Range	Use case	Effect
0.0–0.2	Factual Q&A, data extraction, code generation	Highly deterministic — the same prompt produces nearly identical outputs
0.3–0.5	General question answering, summarization	Balanced — mostly consistent with small variation
0.6–0.8	Creative writing, brainstorming, ideation	Noticeably diverse outputs run to run
0.9–1.0+	Maximum creativity experiments	Unpredictable; can drift into incoherence

Start at 0.2 for anything where correctness matters, and raise the temperature only when you specifically want variety.

The remaining parameters matter less often, but are worth knowing about:

- **Top-P** (nucleus sampling) is an alternative to temperature that truncates the vocabulary to the smallest set of tokens whose cumulative probability exceeds P. Leave it at the default (typically 0.95) unless you have a specific reason to tune it.
- **Max output tokens** is a hard ceiling on response length. Set this defensively to avoid run-away responses that burn through your daily quota.
- **Stop sequences** are strings that, when generated, cause the model to halt immediately. Useful when you want structured output to end at a known marker.

3.6 Grounding with Google Search

Even the best prompt cannot cure hallucination on topics the model has no training data for — recent events, private documents, or niche facts. **Grounding** is the general name for giving the model a real source of truth to consult alongside the prompt. In AI Studio, you can enable Gemini’s built-in Google Search grounding with a single toggle, and Gemini will quietly issue searches, read the retrieved snippets, and cite its sources in the response.

Chapters 8 through 11 of this book teach how to build grounding pipelines yourself using retrieval-augmented generation (RAG) over documents, databases, and images. AI Studio’s Google Search toggle is the easiest possible version of the same idea — the model retrieves, the model generates, but the retrieval happens server-side rather than in your own code. Treat it as a preview of what you will build from scratch later.

3.7 Hands-On Exercise in Google AI Studio

Open <https://aistudio.google.com/> (a Google account is all you need — no API key required for the playground itself) and work through these five steps:

1. **Zero-shot.** Pick a model — Gemini 2.5 Flash is a good default — and type a plain question. Note how long the response is and how confident the tone feels.
2. **Add a system prompt.** Set a Persona in the system-instruction field (e.g., “*You are a skeptical data analyst who always asks for supporting evidence*”) and rerun the same question. Notice how the response shifts.
3. **Few-shot.** Add two or three examples of input-output pairs at the top of your user prompt. Rerun. Does the model follow your format?
4. **Temperature slider.** Move the temperature from 0.1 to 0.9 and rerun a creative prompt (e.g., “*Write one tagline for an electric-bike startup*”). You should see the variance in outputs change dramatically.
5. **Search grounding.** Toggle on Google Search grounding and ask a question about an event from the last month. The response will include citation links — click through to verify the claims actually appear in the sources.

Once the patterns feel natural in AI Studio, move to the Python notebooks in Chapter 5 (LLM Providers and Fallback) and onward. Every system prompt, every few-shot example, every temperature value you set in the playground maps directly to an argument in `llm_cascade`’s `generate()` method.

3.8 Key Takeaways

3.9 Exercises

Easy: Take one of your own past prompts from ChatGPT or Gemini and rewrite it using the PICFAT framework. Run both versions in AI Studio and compare the responses.

Easy: Run the same creative prompt (“Write a short tagline for [your company]”) three times at temperatures 0.0, 0.5, and 1.0. Save the outputs and describe in a short paragraph what changes between them.

Medium: Construct a few-shot prompt for a classification task with at least three examples, then deliberately flip the labels in those examples (e.g., swap “positive” and “negative”). Observe whether the model still learns the task from your format, or whether it learns the inverted labels. Relate your finding to Wei et al. (2023).

Challenge: Write two prompts for the same task — one using PICFAT for a standard model and one using the Role/Task/Background/Constraints/Style/Output framework for a reasoning model. Run each against Gemini 2.5 Flash and Gemini 2.5 Pro (with thinking enabled) in AI Studio. Where are the outputs different in *kind*, not just in *quality*?

3.10 References and Further Reading

- Chen, W., & Chen, L. (2025). *Generative AI for Business: Frameworks, Techniques, and Governance*. GenAI Flows Publishing. ISBN 979-8-9997161-0-1. <https://genai4all.org/> — For an exhaustive theoretical treatment of prompt engineering, including production-scale templates and governance considerations, this companion textbook is the recommended deep dive.
- Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). Large language models are zero-shot reasoners. *Advances in Neural Information Processing Systems*, 35, 22199–22213.
- Wei, J., Wei, J., Tay, Y., Tran, D., Webson, A., Lu, Y., Chen, X., Liu, H., Zhou, D., Le, Q., & Ma, T. (2023). Larger language models do in-context learning differently. *arXiv preprint arXiv:2303.03846*.
- DAIR.AI. *Prompt Engineering Guide*. <https://www.promptingguide.ai/> — Community-maintained reference with code samples for every technique described above.
- Google Cloud. *What is Prompt Engineering?*. <https://cloud.google.com/discover/what-is-prompt-engineering> — Google’s own overview, with a focus on the Gemini model family used throughout this chapter.

Chapter 4

Setting Up API Keys

Most notebooks in this book call a large language model, and to do that you need at least one API key. The `llm_cascade` package looks for keys for eight providers and falls back automatically if the first one is rate-limited or returns an error — so configuring **any one** of them is enough to get every text notebook working, and configuring two or three gives you comfortable headroom when one hits a daily quota mid-class.

You do not need a credit card for any of the free-tier options. The table below lists the providers the book supports, with signup links and the environment variable names `llm_cascade` looks for:

Provider	Environment variable	Signup	Tier
Google Gemini	<code>GEMINI_API_KEY</code>	https://aistudio.google.com/apikey	Free — generous; handles text, images, and video
Groq	<code>GROQ_API_KEY</code>	https://console.groq.com	Free — very fast inference, with rate limits
OpenRouter	<code>OPENROUTER_API_KEY</code>	https://openrouter.ai/keys	Free-tier models available (identified by a <code>:free</code> suffix)
Hugging Face	<code>HF_TOKEN</code>	https://huggingface.co/settings/tokens	Free — serverless inference via the HF router
Ollama Cloud	<code>OLLAMA_API_KEY</code>	https://ollama.com/cloud	Free credits for cloud-hosted open-weight models
Cohere	<code>COHERE_API_KEY</code>	https://dashboard.cohere.com/api-keys	Free trial tier usable for class-scale experiments
OpenAI	<code>OPENAI_API_KEY</code>	https://platform.openai.com/api-keys	Paid (prepaid balance) — not required for this book
xAI (Grok)	<code>XAI_API_KEY</code>	https://console.x.ai/	Paid (prepaid balance) — not required for this book

If you are new to this and want a single recommendation: **start with Gemini**. It is the easiest to sign up for, has the most generous free tier, and is the only provider in the cascade that handles video natively (used by the Image RAG and Video RAG notebooks in Chapter 11). Adding Groq as a second key gives you a fast fallback when Gemini hits its daily quota.

4.1 Storing keys in Colab

If you are running notebooks in Google Colab — the recommended path for most readers — use Colab’s built-in **Secrets** panel so your keys are never pasted into the notebook itself:

1. Click the key icon in the left sidebar of a Colab notebook.
2. Click **Add new secret**.
3. Name the secret exactly as it appears in the *Environment variable* column above (for example, `GEMINI_API_KEY`).
4. Paste your key into the value field.
5. Toggle **Notebook access** on for each secret the current notebook should read.

The `llm_cascade` package auto-detects Colab secrets at runtime, so no code changes are needed once you have added them. Secrets persist across Colab sessions and are scoped to your Google account.

4.2 Storing keys locally

If you are running notebooks locally in VS Code or JupyterLab, create a plain-text file named `.env` in the project root with one line per key:

```
GEMINI_API_KEY=paste_your_actual_key_here  
GROQ_API_KEY=paste_your_actual_key_here
```

`llm_cascade` looks for `.env` in the current working directory and one level up, loading the keys into environment variables at import time. **Never commit `.env` to a public repository** — add it to your `.gitignore` before the first commit if you are starting a new project.

Chapter 5

Calling LLMs: Providers and Fallback

LLMs are accessed via APIs from providers like OpenAI, Google (Gemini), Groq, and others. Each has different models, pricing, rate limits, and capabilities.

This chapter introduces the `llm_cascade` package — a simple Python library that auto-detects your API keys and falls back to the next provider when one hits its quota. Every subsequent chapter in this book uses `llm_cascade` so you never have to worry about provider-specific code.

- **8 supported providers:** OpenAI, Gemini, Ollama, Grok (xAI), Groq, HuggingFace, Cohere, Open-Router
- **One line of code:** `llm = get_cascade()` — that's it
- **Automatic fallback:** if Gemini returns 429, the next call goes to Groq (or whichever is next)

5.1 The `llm_cascade` Package

Every notebook in this book uses `llm_cascade` for automatic LLM provider fallback. Here's a quick reference.

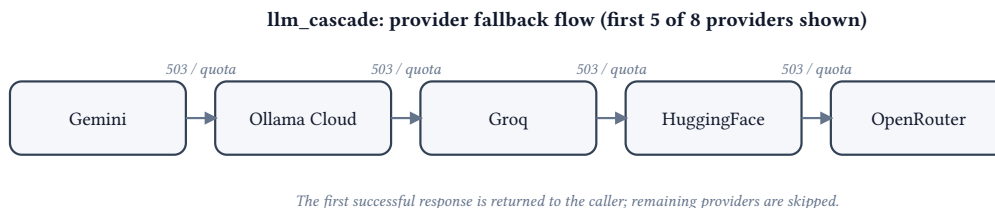


Figure 5.1: How `llm_cascade` cascades through providers: the first to succeed returns the response; a `503`, `429`, or quota-exceeded error on one provider falls through to the next. The cascade tries all 8 providers in order; only the first five are shown here.

As shown in [Figure 1](#), your notebook call never has to know which provider ultimately served the response — `response.provider` tells you after the fact.

5.1.1 Install

```
pip install git+https://github.com/KarAnalytics/llm_cascade.git
```

5.1.2 Quick start

```
from llm_cascade import get_cascade

llm = get_cascade()
response = llm.generate("What is Python?")
print(response.text)
print(response.provider)
"Gemini")
print(response.model)

# auto-detects your API keys
# falls back if one provider is down
# the answer
# which provider answered (e.g.,
# which model was used
```

5.1.3 Supported providers

Provider	Env Variable	Free Tier
OpenAI	<code>OPENAI_API_KEY</code>	Limited free credits
Gemini	<code>GEMINI_API_KEY</code>	500 req/day
Ollama Cloud	<code>OLLAMA_API_KEY</code>	Free tier
Grok (xAI)	<code>XAI_API_KEY</code>	\$25/month free
Groq	<code>GROQ_API_KEY</code>	30 req/min
HuggingFace	<code>HF_TOKEN</code>	Free inference
Cohere	<code>COHERE_API_KEY</code>	20 req/min
OpenRouter	<code>OPENROUTER_API_KEY</code>	Free models

Set any key in your `.env` file or Colab Secrets. The cascade tries them in order and falls back automatically on quota errors, auth failures, or server outages.

5.1.4 Override models per notebook

```
llm = get_cascade(models={"Gemini": "gemini-2.5-pro", "OpenAI": "gpt-4o"})
# or after creation:
llm.set_model("Gemini", "gemini-2.5-pro")
```

5.1.5 Embeddings (local, no API key needed)

```
embedding = llm.get_embedding("some text") # uses all-MiniLM-L6-v2 locally
```

Source: github.com/KarAnalytics/llm_cascade

5.2 Calling the OpenAI API

In this notebook we demonstrate how to make a basic API call to OpenAI's GPT models. OpenAI provides one of the most widely used commercial LLM APIs, offering models such as GPT-4 and GPT-4.1. You will learn how to authenticate with an API key, construct a request, and interpret the response object.

5.2.1 Setting Up the Client

We start by importing the OpenAI Python SDK and configuring the API key. The key is retrieved from the environment so that credentials are never hard-coded into the notebook. Once the client object is created, it manages authentication and connection pooling for every subsequent request.

```
import os
from google.colab import auth, userdata
os.environ["OPENAI_API_KEY"] = userdata.get("OPENAI_API_KEY")

from openai import OpenAI
client = OpenAI()
```

5.2.2 Sending a Chat Completion Request

Here we call the Responses API with the `gpt-4.1` model. The request includes a system message (which can steer the model's persona) and a user message containing our actual question. Parameters like `temperature` and `max_output_tokens` let us control creativity and response length.

```
response = client.responses.create(
    model="gpt-4.1",
    input=[
        {
            "role": "system",
            "content": [
                {
                    "type": "input_text",
                    "text": ""
                }
            ]
        },
        {
            "role": "user",
            "content": [
                {
                    "type": "input_text",
                    "text": "Explain RAG in simple words"
                }
            ]
        }
    ],
    text={
        "format": {
            "type": "text"
        }
    },
    reasoning={},
    tools=[],
    temperature=1,
    max_output_tokens=2048,
    top_p=1,
    store=True,
    include=["web_search_call.action.sources"]
)
```


5.2.3 Inspecting the Full Response

The API returns a rich JSON object that includes token usage statistics, model metadata, and the generated text. Dumping it with `json.dumps` helps us understand the structure and diagnose any issues with our request.

```
import json
print(json.dumps(response.model_dump(), indent=2))
```

5.2.4 Extracting the Generated Text

In most applications we only need the assistant's message text. The `response.output` attribute gives us direct access to the list of output messages without parsing raw JSON.

```
print(response.output)
```

5.2.5 Key takeaways

- **The OpenAI SDK** authenticates via the `OPENAI_API_KEY` environment variable (or a Colab secret) and exposes a `client` object that handles every request.
- **The Responses API** accepts a list of role-tagged messages (system, user) and returns a structured object with the generated text plus metadata.
- **Generation parameters** like `temperature`, `top_p`, and `max_output_tokens` let you dial creativity and cap response length.
- **Inspecting** `response.model_dump()` reveals token usage and model info, while `response.output` gives direct access to the assistant's message for production use.

5.2.6 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~2 minutes (requires API key)

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/OpenAI_API_call.ipynb

5.3 Calling the Google Gemini API

Google's Gemini family of models is accessible through the `google-genai` SDK. This notebook walks through installation, authentication, and making both streaming and non-streaming requests. We also show how to list available models so you can choose the right one for your task.

5.3.1 Setup

Please ensure you have imported a Gemini API key from AI Studio. You can do this directly in the Secrets tab on the left.

After doing so, please run the setup cell below.

5.3.1.1 Installing the SDK

We install the latest `google-genai` package. Keeping it up to date ensures access to the newest model releases and API features.

```
!pip install -U -q "google"
!pip install -U -q "google.genai"
```

5.3.1.2 Configuring the API Key

The Gemini API key is stored as a Colab secret and loaded into an environment variable. This pattern keeps credentials out of version control while making them available to the SDK.

```
import os
from google.colab import userdata
from google.colab import drive
from google import genai
from google.genai import types

#os.environ["GEMINI_API_KEY"] = userdata.get("FREEGOOGLE_API_KEY")
os.environ["GEMINI_API_KEY"] = userdata.get("GOOGLE_API_KEY")
```

5.3.1.3 Mounting Google Drive

If you have supporting files stored in Google Drive, mounting it provides direct filesystem access. This step is optional and only needed when your prompts reference uploaded documents.

```
drive.mount("/content/drive")
# Please ensure that uploaded files are available in the AI Studio folder or change
the working folder.
os.chdir("/content/drive/MyDrive/Google AI Studio")
```

5.3.2 Generated Code

5.3.2.1 Streaming a Response with Thinking

The `generate_content_stream` method returns chunks as they are produced, which is useful for long answers. We also enable the model's built-in thinking/reasoning capability at a medium level, letting it show intermediate reasoning steps.

```
def generate():
    client = genai.Client(
        # api_key=os.environ.get("GEMINI_API_KEY"),
        api_key=os.environ.get("GEMINI_API_KEY"),
    )

    model = "gemini-3-flash-preview"
    contents = [
        types.Content(
```

```

        role="user",
        parts=[
            types.Part.from_text(text="What is 7*8"),
        ],
    ),
]
generate_content_config = types.GenerateContentConfig(
    thinking_config=types.ThinkingConfig(
        thinking_level="MEDIUM",
    ),
)

for chunk in client.models.generate_content_stream(
    model=model,
    contents=contents,
    config=generate_content_config,
):
    print(chunk.text, end="")

if __name__ == "__main__":
    generate()

```

5.3.2.2 Listing Available Models

Google frequently adds new models. Calling `client.models.list()` returns every model your API key has access to, helping you stay current with the latest releases.

```

from google import genai

client = genai.Client()

for model in client.models.list():
    print(model.name)

```

5.3.2.3 A Simple Non-Streaming Call

For short answers, a single blocking call to `generate_content` is simpler than streaming. The response object contains the generated text, token counts, and safety ratings.

```

response = client.models.generate_content(
    model="gemini-3-flash-preview",
    contents="Explain how AI works in a few words",
)

print(response.text)

```

5.3.2.4 Switching Models and API Keys

You can swap models or API keys mid-session to compare outputs or manage rate limits. Here we switch to a different key and an older Gemini model to see how it responds to the same prompt.

```

os.environ["GEMINI_API_KEY"] = userdata.get("FREEGOOGLE_API_KEY")

response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="Are you slower than gemini-3?",
)

print(response.text)

```

```
response = client.models.generate_content(  
    model="gemini-2.5-flash",  
    contents="Which model is being called?",  
)  
  
print(response.text)
```

5.3.2.5 Key takeaways

- The **google-genai** SDK talks to Gemini models and reads its credentials from the `GEMINI_API_KEY` environment variable, typically populated from a Colab secret.
 - **Streaming vs. blocking**: use `generate_content_stream` to print tokens as they arrive, or `generate_content` for a single short response.
 - **Thinking mode** can be toggled via `ThinkingConfig` so Gemini reveals intermediate reasoning before the final answer.
 - `client.models.list()` enumerates every model your key can call, making it easy to switch between Gemini 2.5 and 3.x variants mid-session.
 - **Swapping API keys** at runtime lets you compare free-tier and paid-tier quotas or A/B-test different model versions.
-

5.3.2.6 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~2 minutes (requires API key)

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/Google_Studio_API_call.ipynb

5.4 Calling the Groq Cloud API

Groq provides ultra-fast inference for open-source LLMs using custom LPU hardware. This notebook shows how to authenticate with the Groq Python SDK, send chat completion requests, and compare different open-source models. Groq is an excellent choice when low latency is critical.

5.4.1 Setup and authentication

Install the `groq` Python SDK and read your API key from Colab's secret store. You can get a free key at console.groq.com – the free tier is generous enough for coursework and prototyping. Store it in Colab as a user secret named `GROQ_KEY` (left panel, key icon), and the cells below will read it into the `GROQ_API_KEY` environment variable that the SDK expects.

```
!pip install groq
```

```
import os
from groq import Groq
from google.colab import auth, userdata
os.environ["GROQ_API_KEY"] = userdata.get("GROQ_KEY")
```

5.4.2 Example 1: Structured evaluation with a system prompt

The Groq SDK mirrors OpenAI's chat-completions schema: you pass a `messages` list where each entry has a `role` (`system`, `user`, or `assistant`) and a `content` string. The `system` role sets the model's persona and rubric – here, an expert research reviewer that scores proposals on four transformativeness criteria. Passing `stream=True` returns chunks as they are generated, which we print as they arrive so the output feels interactive rather than arriving all at once at the end.

```
client = Groq()
completion = client.chat.completions.create(
    model="openai/gpt-oss-120b",
    messages=[
        {
            "role": "system",
            "content": "You are an expert reviewer at a leading national research agency. You read research proposals and give a score of 1 to 100 in terms of how transformative they are, in terms of the following criteria:\n\n1) Transdisciplinary\n2) Riskiness of the proposal\n3) Extends existing knowledge significantly (i.e., disruption) instead of extending it conservatively. \n4) Idea is novel and unique and has not been proposed before\n\nTo evaluate transformativeness of a proposal, consider the research done before the year it has been submitted. "
        },
        {
            "role": "user",
            "content": "Proposal: \nEconometrics has been distinguished from simply statistics as applied to economics as being\nprincipally concerned with establishing causes or, what is frequently taken by economists to be\nthe same thing, structural explanation. Economics has itself developed as a science of models.\nThe proposed research is a systematic philosophical investigation of models and causal structure\nin econometrics, and is part of a larger project on the logic of econometric inference and its place\nin the explanatory practices of economics. While the philosophy of science has recently devoted\nmuch attention to models, when it comes to the case of economics, most of the attention has been\npaid to purely theoretical models. The proposed research aims to redress the balance by\ninvestigating models of data (measurement and statistical) and especially how they relate to\ntheoretical models.\nThe intellectual merit of the proposed research rests in this: Econometrics is the core\nmethodology of applied economics, yet its deepest foundations have rarely been examined. It is\nimportant to examine its methods philosophically – that is, at a higher level of generality – in\norder a) to better understand the key issues confronting econometrics, which may provide a\nframework in which students and professional economists can more clearly understand the\nconceptual structure in which econometric methods function as an aid
```



```

schemas/message.js\nexport default {\n  name: \"message\", \n  title: \"Chat
Message\", \n  type: \"document\", \n  fields: [\n    {\n      name: \"text\", \n
title: \"Message Text\", \n      type: \"text\", \n    }, \n    {\n      name:
\"sender\", \n      title: \"Sender\", \n      type: \"reference\", \n      to: [{type:
\"user\"}], \n    }, \n    {\n      name: \"sentAt\", \n      title: \"Sent At\", \n
type: \"datetime\", \n      initialValue: () => new Date().toISOString(), \n    }, \n
    {\n      name: \"room\", \n      title: \"Chat Room\", \n      type: \"reference\", \n
to: [{type: \"room\"}], \n    }, \n  ], \n};\n\nYou'll also have a `user` and a
`room` document type (or you can embed those fields directly if you prefer a flatter
model).\n\n---\n\n## 3. Writing a message – the “save” part\n\n`js\nimport
sanityClient from '@sanity/client'\n\nconst client = sanityClient({\n  projectId:
'yourProjectId', \n  dataset: 'production', \n  token: 'sanity-write-token', // must
have write permissions\n  useCdn: false, // false = always hit the
API (important for fresh writes)\n})\n\n// Example payload from a UI form\nconst
newMessage = {\n  _type: 'message', \n  text: 'Hey, how are you?', \n  sender: {_type:
'reference', _ref: userId}, \n  room: {_type: 'reference', _ref: roomId}, \n
sentAt: new Date().toISOString(), \n}\n\n// Persist\nclient\n  .create(newMessage)\n
.then(res => console.log('Message saved\n\n✓', res))\n  .catch(err => console.error('✗ Save failed', err))\n\n\n> **Key
points**\n\n> * The client **must** be instantiated with a **write-capable token**.\n
If you use the public CDN token (or `useCdn: true` without a token), writes are
silently ignored. \n> * `useCdn: false` forces the request to go to the API instead
of the edge cache, ensuring you read the freshly written document right after
creating it. \n> * If you're using the **Sanity Studio** UI for testing, make sure
the “Create” button actually hits the same dataset you're querying in your
front-end.\n\n---\n\n## 4. Common reasons messages “don't appear” in GROQ\n\n|
Symptom | Likely Cause | Fix |\n|-----|-----|-----|\n| **Empty result**\n(`[]`) when you know you sent a message | a) Write never happened (missing token,
`useCdn:true`) <br> b) Message saved to a *different* dataset (`development` vs
`production`) <br> c) Wrong `_type` name (`Message` vs `message`) | Verify token
scopes (`write`), double-check dataset name, run `client.fetch('*[_type ==
\"message\"]')` in the same environment you used to write. |\n| **Message appears
only after a refresh** | Using the CDN cache (`useCdn:true`) for reads – CDN may
still hold the pre-write snapshot. | Set `useCdn:false` for reads that need
real-time consistency, or use the **listener** API (`client.listen`) to get push
updates. |\n| **Only the most recent message shows up** | Over-writing the same
document ID (e.g., you used `client.createOrReplace({_id: 'temp', ...})` each time). |
Use `client.create` (auto-generates a unique `_id`) or generate a UUID per message.
|\n| **Message shows up, but fields are missing** | Schema mismatch (e.g., you
omitted a required field, Sanity silently dropped it) | Check the schema's
`validation` rules; run `client.create` with the full payload and inspect the
returned document. |\n| **`listen` never fires** | Listening to the wrong dataset or
using a read-only token | Use the same project/dataset and a token with `read`
scope. |\n\n---\n\n## 5. Quick debugging checklist\n\n1. **Confirm you have a write
token** \n  ``bash\n    curl -H \"Authorization: Bearer <token>\" \"\n\nhttps://<projectId>.api.sanity.io/v2021-06-07/data/mutate/production?
returnIds=true\" \n\n    -d '{\"mutations\": [{\"create\": {\"_type\": \"message\",
\"text\": \"test\"}}]}' \n\n  You should get a `201` with the new document ID.
If you get `401` or `403`, the token is not authorized.\n\n2. **Inspect the dataset
directly** \n  - In **Sanity Studio**: open the “Message” list → you should see
the rows. \n  - Via **API**: `client.fetch('*[_type==\"message\"]{_id,text}')` →
should return an array.\n\n3. **Check the network tab** (browser dev tools) when you
click “Send”. \n  - **POST** to `https://<projectId>.api.sanity.io/v2021-06-07/
data/mutate/...` \n  - Response payload includes `createdId` or an error.\n\n4.
**Validate the GROQ query** \n  ``js\n    // Simple sanity check\n    client.fetch(`\n      *[_type == \"message\" && room._ref == $roomId] | order(sentAt
desc)[0...10]{\n        _id, \n        text, \n        \"senderName\": sender->name, \n
sentAt\n      }\n    `, {roomId})\n  `` \n  If this still returns `[]`, the data
really isn't there.\n\n5. **Look at the dataset name** \n  - In `sanity.json` you
may have ``api\": { \"projectId\": \"...\", \"dataset\": \"development\" }`. \n
- Your front-end code might be pointing at `\"production` – mismatch = no results.
\n\n---\n\n## 6. Example end-to-end flow (React + Sanity)\n\n`tsx\n//
ChatInput.tsx\nimport {useState} from 'react'\nimport {client} from './

```



```

sanityClient' // pre-configured client\n\nexport default function
ChatInput({roomId, userId}) {\n const [msg, setMsg] = useState('')\n\n const send
= async () => {\n if (!msg.trim()) return\n try {\n await
client.create({\n _type: 'message',\n text: msg,\n sender:
{ _type: 'reference', _ref: userId },\n room: { _type: 'reference', _ref:
roomId },\n sentAt: new Date().toISOString(),\n })\n
setMsg('') // clear UI\n } catch (e) {\n console.error('🔥
Could not save message', e)\n }\n }\n\n return (\n <div>\n <input
value={msg} onChange={e => setMsg(e.target.value)} />\n <button onClick={send}
>Send</button>\n </div>\n )\n }\n\n`tsx\n// ChatMessages.tsx\nimport
{useEffect, useState} from 'react'\nimport {client} from './sanityClient'\n\nexport
default function ChatMessages({roomId}) {\n const [messages, setMessages] =
useState([])\n\n // Initial fetch\n useEffect(() => {\n
client\n .fetch(`\n *[_type=="message" && room._ref == $roomId] |
order(sentAt desc)[0...50]{\n _id, text, sentAt,\n "senderName"\n
sender->name\n }, {roomId})\n .then(setMessages)\n },
[roomId])\n\n // Real-time updates\n useEffect(() => {\n const subscription =
client\n .listen(`*_type=="message" && room._ref == $roomId`,
{roomId})\n .subscribe(event => {\n if (event.transition === 'appear')
{\n setMessages(prev => [event.result, ...prev])\n }\n })\n
return () => subscription.unsubscribe()\n }, [roomId])\n\n return (\n <ul>\n
{messages.map(m => (\n <li key={m._id}>\n <strong>{m.senderName}</
strong> {m.text}\n </li>\n ))}\n </ul>\n )\n }\n\n`*\n\nIf the above
code works in a fresh project, then your issue is almost certainly a configuration
mismatch rather than a GROQ limitation.*\n\n---\n\n## 7. TL;DR Answer to Your
Question\n\n> ***"Are chat messages not saved in GROQ?"*** \n\nNo. GROQ doesn't
*save* anything. It can only *read* what has already been stored in Sanity (or any
GROQ-compatible backend). If your chat messages aren't appearing when you query
them, the problem lies in the *write path* – token permissions, dataset selection,
schema name, or using the CDN cache for reads.\n\nFix the write path, and then any
standard GROQ query (as shown above) will retrieve the messages instantly.\n\n---
\n\n### Need more help?\n\n* Which environment are you testing (Sanity Studio,
custom front-end, Next.js, etc.)? \n\n* Are you using the default public read-only
token or a custom write token? \n\n* Could you share the exact GROQ query you're
running and the code you use to create a message? \n\nDrop those details and I can
point out the precise line that's tripping you up. Happy chatting! 🚀"
}
],
temperature=1,
max_completion_tokens=8192,
top_p=1,
reasoning_effort="medium",
stream=True,
stop=None
)

for chunk in completion:
    print(chunk.choices[0].delta.content or "", end="")

```

5.4.4 Key takeaways

- **Groq Cloud** runs open-weight models (Llama, Mixtral, GPT-OSS) on custom LPU hardware, delivering the lowest-latency inference among mainstream providers.
- The **groq SDK** mirrors OpenAI's chat-completions schema, so migrating prompts between the two providers requires almost no code changes.
- **Streaming** with `stream=True` lets you print tokens as soon as they are produced, which showcases Groq's speed advantage.
- A **generous free tier** (with daily token quotas) makes Groq ideal for prototyping agentic workflows before committing to paid usage.
- **reasoning_effort** controls how much internal deliberation reasoning-capable models perform before emitting the final answer.

5.4.5 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~2 minutes (requires API key)

```
https://colab.research.google.com/github/KarAnalytics/code\_demos/blob/main/Groq\_Cloud\_API\_Call.  
ipynb
```


Chapter 6

Local LLMs

Throughout this book, every notebook has relied on cloud-hosted LLM providers — Gemini, OpenAI, Groq, and others — accessed over the internet through API calls. This is the default mode for most AI applications today, and for good reason: cloud providers offer state-of-the-art models with zero infrastructure overhead. But there is another way. You can download an open-weight language model, run it entirely on your own hardware, and never send a single byte of data to the cloud. This chapter explores why you might want to do that, how to set it up, and when local inference is the right choice for business applications.

6.1 Why Run LLMs Locally?

The most compelling reason to run models locally is **data privacy**. When you send a prompt to a cloud API, your data travels over the internet to a third-party server. For many business applications — legal document review, healthcare records analysis, proprietary code generation, financial modeling — this creates a compliance risk that legal and security teams will eventually flag. Regulations like GDPR in Europe and HIPAA in the United States impose strict requirements on where data can be processed, and sending it to a cloud LLM provider may violate those requirements. Running a model on your own machine or your company’s private server eliminates this concern entirely: the data never leaves your infrastructure.

Cost is the second major driver. Cloud APIs charge per token — every prompt and every response costs money. For experimental work, where you might run hundreds of variations of a prompt to find the right one, these costs add up quickly. A local model, once downloaded, can be queried millions of times for free (beyond the electricity to power your machine). This makes local models ideal for research, prototyping, and educational experimentation. As the instructor notes in class: “LM Studio or any other local LLM gives you the power of experimenting with AI. That’s something all of us should be doing after this course — have some kind of control over AI that can be done.”

Local models also work **offline**. On a plane, in a restricted environment, or simply with an unreliable internet connection, a local model keeps working. And because you control the entire stack — model weights, inference parameters, system prompt — you have complete freedom to experiment with configurations that cloud providers may not expose.

6.2 LM Studio

LM Studio is a desktop application that provides a visual interface for discovering, downloading, and running open-weight language models. It supports models in the GGUF format (a quantized format optimized for consumer hardware) and hosts a vast catalog of models from HuggingFace, the largest open-source model repository.

What makes LM Studio particularly useful for learning is its **Developer mode**. When enabled, LM Studio runs a local HTTP server on `localhost` that exposes an API endpoint compatible with the OpenAI API format. This means any code written to call OpenAI — including the notebooks in this book — can be redirected to your local model simply by changing the base URL and using any placeholder API key. The API key does not matter for local use because there is no authentication; you are talking to your own machine.

In a classroom setting, LM Studio offers a practical advantage: a shared models folder on a network drive allows the entire class to access downloaded models without each student needing to download multi-gigabyte files individually. Once a model is downloaded to the shared folder, any student can point their LM Studio instance to that directory and begin inference immediately.

6.3 Ollama

While LM Studio excels at interactive exploration, **Ollama** is the tool of choice for command-line workflows and production integration. Running a model is as simple as typing `ollama run llama3` in a terminal. Ollama manages model downloads, quantization, and serving automatically, and like LM Studio, it exposes an OpenAI-compatible API server that can be called from any application.

Ollama is available for Linux, macOS, and Windows, and it also offers a cloud service and a desktop application (in preview). Its command-line nature makes it well-suited for scripting, automation, and integration into CI/CD pipelines. Many practitioners use both tools: LM Studio for interactive experimentation and visual model browsing, Ollama for automated workflows and production deployments.

The `llm_cascade` package used throughout this book includes Ollama as a supported provider. By setting the `OLLAMA_API_KEY` environment variable (any placeholder value works for local use), the cascade will include your local Ollama instance in the provider rotation alongside cloud providers like Gemini and Groq. This means you can develop and test with a local model and automatically fall back to cloud providers when you need more capability.

6.4 Other Tools in the Ecosystem

The local LLM ecosystem extends well beyond LM Studio and Ollama. **llama.cpp** is the foundational C++ inference engine that both LM Studio and Ollama use internally — it is the reason these tools can run large models efficiently on consumer CPUs. For users who want maximum control, running `llama.cpp` directly offers the lowest overhead. **Jan** is another open-source desktop application similar to LM Studio, while **GPT4All** focuses on lightweight, privacy-preserving deployment. **AnythingLLM** combines a local model runtime with built-in RAG and document management, making it a self-contained system for document Q&A without any cloud dependency. For production serving at scale, **vLLM** provides high-throughput inference with features like continuous batching and PagedAttention.

6.5 Model Selection and Quantization

Choosing the right model for local use involves balancing intelligence against hardware requirements. Models are measured by their parameter count: a 1B parameter model is fast but limited, a 7B model is capable and runs well on most modern laptops, a 13B model requires a decent GPU or plenty of RAM, and a 70B model demands high-end hardware. For classroom use, the instructor recommends starting small: “Use very small models, like under 7B. Smaller models may not be very smart, but they’re fast.”

Quantization is the technique that makes local inference practical. The original model weights are stored in 16-bit or 32-bit floating point, but quantization compresses them to 4-bit or 8-bit integers with minimal quality loss. The GGUF format supports multiple quantization levels — `Q4_K_M` offers a good balance between quality and size, while `Q8_0` preserves more quality at the cost of larger files. A 7B model that would require 14 GB in FP16 can run in under 4 GB after Q4 quantization, fitting comfortably in laptop RAM.

Downloading a model is a one-time operation. As noted in class: “Once you load it, it’s there in your computer. You don’t have to download it every single time.” Larger models take longer to download but the transfer speed depends on your internet connection, not your computing resources.

6.6 When to Use Local vs. Cloud

The decision between local and cloud inference is not either-or — it depends on the use case. Local models are the right choice when data privacy is paramount, when you need offline capability, when you are experimenting intensively and want unlimited free inference, or when you are running batch processing on cost-sensitive workloads. Cloud models remain the better choice when you need maximum capability (GPT-4o, Claude, Gemini), when you need to scale to many concurrent users, when your team needs shared infrastructure, or when the latest model improvements matter more than cost.

In practice, many organizations adopt a **hybrid approach**: local models for development, testing, and privacy-sensitive workloads; cloud APIs for production deployment and maximum performance. The `llm_cascade` architecture used in this book naturally supports this pattern — local and cloud providers coexist in the same fallback chain, and the system automatically routes to whichever is available.

The companion notebook walks through setting up LM Studio, downloading a model, starting the local server, and calling it from Python — giving you firsthand experience with the local inference workflow.

For a related approach, see **Vibe Coding**.

6.7 Key Takeaways

6.8 Exercises

Easy: Install LM Studio or Ollama on your machine and download a small model (under 3B parameters). Send it a simple prompt and record the response time compared to a cloud API call.

Easy: Explain the difference between Q4_K_M and Q8_0 quantization levels. For a student laptop with 8 GB of RAM, which quantization level would you recommend for a 7B model, and why?

Medium: Set up Ollama as a provider in `llm_cascade` alongside Gemini. Configure the cascade so that Ollama is tried first and Gemini serves as fallback. Send five prompts and note which provider handles each request. Then disconnect from the internet and send five more prompts to verify that Ollama continues working offline.

Challenge: A law firm wants to deploy an LLM for reviewing confidential merger documents. They process approximately 200 documents per day, each around 5,000 tokens. Recommend a local deployment architecture: specify the model size, quantization level, hardware requirements, and whether LM Studio or Ollama is more appropriate. Justify your choices based on privacy, throughput, and cost.

6.9 References

- LM Studio. (2026). *LM Studio Documentation*. <https://lmstudio.ai/>
- Ollama. (2026). *Ollama Documentation*. <https://ollama.com/>
- Gerganov, G. (2023). *llama.cpp: Port of Facebook's LLaMA model in C/C++*. <https://github.com/ggerganov/llama.cpp>
- SitePoint. (2026). Guide to Local LLMs in 2026: Privacy, Tools & Hardware. <https://www.sitepoint.com/definitive-guide-local-llms-2026-privacy-tools-hardware/>
- Zealousys. (2026). LM Studio vs Ollama: Which Local LLM Tool Is Better in 2026? <https://zealousys.com/blog/lm-studio-vs-ollama/>

6.10 Running Local Models with LM Studio

LM Studio lets you download and run open-source language models entirely on your own machine. It exposes an OpenAI-compatible REST API, so you can reuse the same `openai` Python SDK you already know. This notebook demonstrates two approaches: using the OpenAI client pointed at your local server and using the native `lmstudio` Python package.

THIS CODE WON'T EXECUTE IN GOOGLE COLAB. You need to get lmstudio server running and then run this in VS Code or any other local IDE!

6.10.1 Importing Libraries

We import both the `lmstudio` SDK (for native access) and the `openai` SDK (for the OpenAI-compatible endpoint). Having both available lets you pick whichever interface fits your workflow.

```
import lmstudio as lms
## !pip install openai
from openai import OpenAI
import os
```

6.10.2 Checking the Working Directory

Printing the current working directory is a quick sanity check to confirm where any local files will be read from or written to.

```
# Get the current working directory
cwd = os.getcwd()
print("Current Working Directory:", cwd)
```

6.10.3 Calling LM Studio via the OpenAI-Compatible Endpoint

By pointing the OpenAI client at `http://127.0.0.1:1234/v1`, every call goes to LM Studio instead of OpenAI's servers. The `api_key` value is ignored by LM Studio but required by the SDK. Make sure you have a model loaded in LM Studio before running this cell.

```
# Initialize the client pointing to your local LM Studio server
client = OpenAI(
    base_url="http://127.0.0.1:1234/v1", # Ensure you add "/v1" to the URL
    api_key="lm-studio" # The SDK requires a key, but LM Studio ignores the specific
    value
)

response = client.chat.completions.create(
    model="gpt-oss-20B", # Ensure this matches the model loaded in LM Studio
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Tell me a joke."},
    ],
    temperature=0.7,
)

print(response.choices[0].message.content)
```

Another example

6.10.4 Using the Native LM Studio SDK

The `lmstudio` package connects to whichever model is currently loaded and provides a simpler `respond()` interface. This is convenient for quick experiments where you do not need fine-grained control over parameters.

```

model = lms.llm()

# 2. Write your prompt
prompt = "What is today's date?"

# 3. Get the response
print("Awaiting response...")
result = model.respond(prompt)

print(f"\nLLM Says: {result}")

```

6.10.5 Key takeaways

- **LM Studio** downloads and runs open-source LLMs entirely on your own machine, so prompts and documents never leave your device.
- **The OpenAI SDK works unchanged** once you set `base_url="http://127.0.0.1:1234/v1"`; the `api_key` is required syntactically but ignored by LM Studio.
- **Two SDK options:** the OpenAI-compatible client for portability, or the native `lmstudio` package with a lightweight `model.respond()` call.
- **Local execution** requires running this notebook in a local IDE (not Colab) with a model already loaded in the LM Studio desktop app.
- **Privacy and cost:** local inference has zero per-token cost and keeps sensitive data on-premises, at the price of hardware and slower throughput.

6.10.6 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~5 minutes (requires LM Studio running locally)

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/LMStudioAccess.ipynb

Chapter 7

Vibe Coding

In February 2025, Andrej Karpathy — co-founder of OpenAI and one of the most influential voices in AI — introduced a term that would quickly enter the mainstream vocabulary of software development: **vibe coding**. “There’s a new kind of coding I call ‘vibe coding,’” he wrote, “where you fully give in to the vibes, embrace exponentials, and forget that the code even exists.” The idea was provocative and deliberately so: instead of writing code line by line, you describe what you want in natural language and let an AI agent write, debug, and iterate on the code for you. You become the director; the AI becomes the programmer.

What began as a half-serious provocation has, by 2026, become a legitimate and widely adopted development practice. The tools have matured rapidly, the models have become remarkably capable at understanding intent and generating correct code, and a growing number of applications — from internal business tools to customer-facing products — have been built primarily or entirely through vibe coding. This chapter surveys the landscape, explains the workflow, examines the tools, and offers practical guidance for business professionals who want to build AI-powered applications without becoming software engineers.

7.1 What Vibe Coding Actually Means

Vibe coding is the practice of building software by describing your intent in natural language and letting an AI coding agent translate that intent into working code. You do not write Python functions, HTML templates, or SQL queries. Instead, you say things like “create a web form that takes a company name and returns the top five competitors from our database” or “build a Gradio interface for uploading Excel files and running SQL queries against them.” The AI agent generates the code, creates the files, installs dependencies, runs tests, and reports back. You review the output, point out problems, and iterate.

This is fundamentally different from using an LLM as a code autocomplete tool (though that is related). Autocomplete suggests the next line while you are actively writing code. Vibe coding delegates the entire implementation to the AI while you focus on intent, architecture, and quality review. The cognitive shift is significant: instead of thinking in syntax, you think in specifications.

The instructor’s experience from classroom sessions captures both the promise and the reality: “Vibe coding is easy, but it is very error-prone. If you know what the Python code is, knowing how to prototype, knowing how to debug a little bit — that’s the secret to successful vibe coding.” The skill is not writing code from scratch; the skill is describing what you want clearly enough for the AI to get it right, and knowing enough about code to recognize when it gets it wrong.

7.2 The Major Vibe Coding Tools

The vibe coding ecosystem in 2026 is competitive and fast-moving. Several tools have emerged as leaders, each with a different philosophy and target audience.

Claude Code is a terminal-based AI agent from Anthropic that reads your entire codebase, writes code, runs commands, creates files, and edits multiple files simultaneously. It excels at large-scale refactoring and architectural work — handling codebases of 50,000 or more lines of code with a roughly 75 percent success rate on complex multi-file tasks. Claude Code runs alongside your editor, not inside it, communicating via the command line. It is available through Claude Pro at approximately twenty dollars per month. Much of this book — the notebooks, the `llm_cascade` package, the MCP servers, and the book’s own build system — was developed with Claude Code.

Cursor is an AI-native IDE built as a fork of Visual Studio Code. It offers deep repository awareness, Composer mode for multi-file edits, and project-specific instruction files (`.cursorrules`) that tell the AI

how your project is structured. Cursor has the most polished user experience and the largest community among AI coding tools, with user ratings consistently around 4.9 out of 5. A free tier is available; the Pro plan costs approximately twenty dollars per month.

GitHub Copilot operates inside Visual Studio Code as an extension and is the most widely adopted AI coding assistant. Its strength is accessibility: if you already use VS Code (which most developers do), Copilot integrates seamlessly. It offers inline code completion, chat, and — as of 2025 — an Agent Mode for multi-file autonomous edits. Critically, Copilot is **free for students and educators** through GitHub Education, making it the natural starting point for classroom use. As the instructor notes: “The Copilot is free for students and instructors, so I got the free version for the education version, and I use it.”

Windsurf, originally built by Codeium and later acquired by OpenAI, is an AI-enhanced IDE centered around an autonomous agent called Cascade. Rather than waiting for explicit prompts, Cascade proactively gathers context from your project and executes multi-step tasks with minimal guidance. Its free tier is generous, making it a strong option for budget-conscious students and startups.

Antigravity is Google’s AI coding agent, available as a VS Code extension powered by Gemini models. The combination of Antigravity with Gemini Pro offers a capable vibe coding setup at no cost. The instructor has used this extensively: “You can use Antigravity with a Gemini Pro. That is a pretty powerful combination.” Several of the notebooks in this book were initially developed using Antigravity.

OpenAI Codex is OpenAI’s cloud-based coding agent, accessible from the ChatGPT interface or via API. It runs in a sandboxed environment where it can write code, execute it, run tests, and iterate — all without requiring a local development setup. This makes it particularly accessible for non-developers who want to build simple tools or automate tasks.

7.3 The Vibe Coding Workflow

A typical vibe coding session follows a conversational pattern. You begin with a clear description of what you want to build. The AI generates an initial implementation. You review the output — does it run? Does it do what you intended? You provide feedback, point out errors, or ask for modifications. The AI revises. You test again. This cycle repeats until the result meets your standards.

The workflow is iterative by nature. As observed in classroom sessions: “It is iterative. The secret to successful vibe coding is knowing how to prototype, knowing how to debug a little bit.” First attempts are rarely perfect, but each iteration gets closer. The key skill is not coding ability but **specificity of communication** — the more precisely you describe what you want, the more accurately the AI delivers.

A practical example: this book’s entire `llm_cascade` package started as a conversational description (“I want a Python package that auto-detects API keys for eight LLM providers and falls back to the next one when quota is hit”). Over multiple iterations, the agent generated the provider list, the fallback logic, the embedding support, the `setup.py`, and the README. The human guided architecture decisions and tested edge cases; the AI wrote the code.

7.4 The 90/10 Problem

Vibe coding has a well-known failure mode that practitioners call the 90/10 problem: the AI gets you 90 percent of the way in 10 percent of the time, but the remaining 10 percent — debugging edge cases, handling errors gracefully, integrating with existing systems — takes 90 percent of the effort. The initial generation is fast and impressive. The polish is where the real work happens.

This is why the instructor’s advice is critical: you need to understand enough about code to recognize when the AI has made a mistake. You do not need to be able to write the code from scratch, but you do need to be able to read it, understand its intent, and identify when something is wrong. For business professionals, this means investing time in understanding the basics of Python, SQL, and web development — not to become programmers, but to become effective AI directors.

7.5 Implications for Business Professionals

Vibe coding represents one of the most significant shifts in the relationship between business professionals and technology. Historically, building a software tool required either programming skills or the budget to hire a developer. Vibe coding lowers this barrier dramatically. An MBA student who can describe a business problem clearly can now prototype a working solution in hours, not weeks.

This does not eliminate the need for professional software engineers — production systems still require security auditing, performance optimization, scalability testing, and maintenance that go well beyond what vibe coding produces. But it does mean that the distance between “I have an idea” and “I have a working prototype” has collapsed to nearly zero. For product managers, consultants, entrepreneurs, and analysts, this is transformative.

Every notebook in this book was built using vibe coding tools. The fact that you are reading a functional, deployed, multi-chapter interactive textbook with 30+ runnable notebooks and a cloud-hosted MCP server on Google Cloud Run — all created through conversational AI interactions — is itself a demonstration of what vibe coding makes possible.

For a related approach, see **RAG from First Principles**.

7.6 Key Takeaways

7.7 Exercises

Easy: Choose one vibe coding tool (GitHub Copilot, Cursor, or Antigravity) and use it to generate a Python function that takes a list of product names and prices and returns the three most expensive items. Evaluate whether the output is correct on the first attempt.

Easy: Describe the 90/10 problem in your own words. Give a concrete example of a “last 10%” issue that an AI-generated prototype might have that would prevent it from being production-ready.

Medium: Using a vibe coding tool, build a simple Gradio interface that accepts a CSV file upload and displays basic summary statistics (mean, median, min, max for each numeric column). Document the number of iteration cycles required and the specific corrections you had to request.

Challenge: Use a vibe coding tool to build a multi-page Streamlit dashboard that connects to a SQLite database, displays interactive charts, and includes a natural-language query input that generates SQL. Track the full session: record each prompt you gave, each error the AI produced, and each correction cycle. Write a one-page retrospective analyzing where the AI excelled and where it struggled.

7.8 References

- Karpathy, A. (2025). *Vibe coding* [Social media post]. <https://x.com/karpathy/status/1886192184808149383>
- Nucamp. (2026). Top 10 Vibe Coding Tools in 2026. <https://www.nucamp.co/blog/top-10-vibe-coding-tools-in-2026-cursor-copilot-claude-code-more>
- Appwrite. (2026). Comparing the best vibe coding tools: Cursor, Claude Code, Windsurf, VS Code, Lovable and Bolt. <https://appwrite.io/blog/post/comparing-vibe-coding-tools>
- Lushbinary. (2026). AI Coding Agents 2026: Claude Code vs Antigravity vs Codex vs Cursor vs Kiro vs Copilot vs Windsurf. <https://lushbinary.com/blog/ai-coding-agents-comparison-cursor-windsurf-claude-copilot-kiro-2026/>
- GitHub. (2026). *GitHub Copilot Documentation*. <https://docs.github.com/en/copilot>
- Anthropic. (2026). *Claude Code Documentation*. <https://docs.anthropic.com/en/docs/claude-code>

Chapter 8

RAG from First Principles

Retrieval-Augmented Generation (RAG) is the most important technique for grounding LLM answers in your own data. Instead of hoping the LLM memorized the right facts during training, you *retrieve* relevant documents and *stuff* them into the prompt.

This chapter builds a complete RAG pipeline from scratch — no frameworks, no abstractions. You'll implement every step manually: chunking, embedding, vector search, prompt construction, and generation. This gives you the deepest understanding of what frameworks like LlamaIndex and LangChain do under the hood.

8.1 RAG from First Principles — No Framework, No Magic

Code courtesy: This code has been adapted from towardsai.net/book code materials accompanying this book: [Building LLMs for production](#)

In the previous chapter you may have seen RAG through a framework like LangChain, where a single function call hides all the plumbing. This notebook strips away every abstraction and builds the entire RAG pipeline by hand: chunk the text, embed the chunks, compute cosine similarity, retrieve the best matches, stuff them into a prompt, and call an LLM. By the end, you will understand every moving part well enough to build your own RAG system from scratch — or to debug one that someone else built with a framework.

The embedding step builds directly on the word-embedding ideas introduced in Chapter 2 — if the notion of mapping text to a vector space is still fuzzy, it is worth a quick detour back to the tokenization and embeddings demos before going further.

8.1.0.1 Install and Import

We install only two things beyond standard Python: `openai` and `google-genai` for LLM access, `scikit-learn` for cosine similarity math, and `llm_cascade` — a lightweight wrapper that auto-detects whichever API key you have configured and routes requests to the cheapest available provider. There is no LangChain, no vector database library, no orchestration framework. Everything else we build ourselves.

```
!pip install -qU git+https://github.com/KarAnalytics/llm_cascade.git openai google-genai scikit-learn
```

8.1.1 Step 1: Load and Chunk the Dataset

Every RAG pipeline starts with raw text. Ours comes from a CSV of blog articles about the LLaMA2 model. The challenge is that these articles are too long to embed as single units — embedding models work best on passages of a few hundred to a thousand characters. So we need to **chunk** them: break each article into fixed-size pieces that are small enough to embed meaningfully, yet large enough to carry coherent information.

8.1.1.1 Download the Data

We download two CSV files: one with the raw article text, and one with pre-computed embeddings (in case you want to skip the embedding step later). Both are hosted on GitHub so this notebook runs without any local file dependencies.

The dataset includes several articles from the TowardsAI blog, which provide an in-depth explanation of the LLaMA2 model. These articles will serve as our “knowledge base” — the documents that RAG will search through when answering questions.

8.1.1.2 Chunk the Articles

The `split_into_chunks` function below is intentionally simple: it slices text into fixed 1,024-character windows with no overlap. Each article gets split independently, and all chunks are collected into a single list. We then load them into a Pandas DataFrame so we can attach embedding vectors as a new column later. This is the “R” foundation of RAG — without good chunks, retrieval cannot find the right context.

```
!wget https://raw.githubusercontent.com/AlaFalaki/tutorial_notebooks/main/data/mini-llama-articles.csv
!wget https://raw.githubusercontent.com/AlaFalaki/tutorial_notebooks/main/data/mini-llama-articles-with_embeddings.csv
```

8.1.2 Step 2: Generate Embeddings

Now we convert each text chunk into a numerical vector. An **embedding** is a list of numbers (in our case, 384 numbers) that captures the *meaning* of a piece of text. Two chunks about similar topics will have vectors that point in roughly the same direction; two chunks about unrelated topics will point in very different directions. This is the mathematical trick that makes semantic search possible — instead of matching keywords, we match meaning.

The embedding model we use (`all-MiniLM-L6-v2`) runs entirely on your machine. No API call, no cost, no data leaving your laptop. It is small and fast, but surprisingly effective for retrieval tasks.

```
chunks = []

# Load the file as a CSV
with open("./mini-llama-articles.csv", mode="r", encoding="utf-8") as file:
    csv_reader = csv.reader(file)

    for idx, row in enumerate(csv_reader):
        if idx == 0:
            continue
        # Skip header row
        chunks.extend(split_into_chunks(row[1]))

# Convert the JSON list to a Pandas Dataframe
df = pd.DataFrame(chunks, columns=["chunk"])

df.keys()
```

After embedding, every row in our DataFrame now has two columns: the original text chunk and its 384-dimensional vector. This DataFrame is our vector database — just a table in memory. Production systems use dedicated vector stores (Pinecone, Weaviate, ChromaDB) for speed and persistence, but the underlying data structure is the same: text paired with its embedding.

8.1.3 Step 3: Embed the User’s Question

To find which chunks are relevant to a question, we need to put the question into the same vector space as the chunks. We embed the question using the exact same model (`all-MiniLM-L6-v2`), producing another 384-dimensional vector. Now we can compare apples to apples: the question vector lives in the same space as all the chunk vectors, and we can measure how “close” each chunk is to the question.

```
# Embeddings use a local HuggingFace model (all-MiniLM-L6-v2)
# No API key needed — runs locally via llm.get_embedding()
```

8.1.4 Step 4: Understand Cosine Similarity (A Quick Detour)

Before we search the full dataset, let us build intuition for how cosine similarity works. We will embed two sentences — one that clearly answers our question and one that is completely irrelevant — and compare their similarity scores to the question vector. If the math is working correctly, the relevant sentence should score close to 1.0 and the irrelevant one should score near 0.0.

Notice the scores: the irrelevant sentence (“The sky is blue”) scores near zero, while the relevant sentence (“LLaMA2 model has a total of 2B parameters”) scores close to 0.90. Cosine similarity ranges from -1 to 1 , where 1 means identical direction in vector space. A score of 0.90 is very high — it tells us the embedding model “understands” that both the question and the good source are talking about the same topic. This is semantic search in action: no keyword matching, just meaning.

```
# Define the user question, and convert it to embedding.
QUESTION = "How many parameters LLaMA2 model has?"
QUESTION_emb = llm.get_embedding(QUESTION)

len(QUESTION_emb)
```

8.1.5 Step 5: Retrieve the Best Chunks

Now we apply cosine similarity at scale. We compare the question embedding against *every* chunk embedding in our DataFrame, producing a similarity score for each one. The chunks with the highest scores are the ones whose meaning is closest to the question — these are our retrieved context. We pick the top 3 and will feed them into the LLM prompt in the next step.

Calculating the similarity of embedding representations can help us to find pieces of text that are close to each other. In the following sample you see how the Cosine Similarity metric can identify which sentence could be a possible answer for the given user question. Obviously, the unrelated answer will score lower.

```
BAD_SOURCE_emb = llm.get_embedding("The sky is blue.")
GOOD_SOURCE_emb = llm.get_embedding("LLaMA2 model has a total of 2B parameters.")
```

The indices [52, 89, 114] point to the three chunks whose embeddings are closest to our question. Let us inspect them to see if the retrieval makes sense — do they actually talk about LLaMA2 parameters? If they do, our embedding-based search is working. If they do not, we would need to revisit our chunking strategy or try a different embedding model.

8.1.5.1 LLM Response Function

With the relevant chunks retrieved, we need a way to call an LLM. The `gemini_response` function below uses `llm_cascade` to send a prompt to whichever provider is available — Gemini, Ollama, Groq, OpenAI, or others. It accepts a system prompt (to set the LLM's role) and a user prompt (containing the question and context). This is a thin wrapper; the real work happens in how we *construct* the prompt in the next step.

8.1.6 Step 6: Augment the Prompt and Generate

This is the “A” and “G” in RAG — **Augment** and **Generate**. We take the three retrieved chunks and insert them directly into the prompt between `<START_OF_CONTEXT>` and `<END_OF_CONTEXT>` tags. The system prompt tells the LLM to act as a context-grounded assistant that only answers AI-related questions. By embedding the retrieved text right in the prompt, we give the LLM specific evidence to draw from instead of relying on whatever it memorized during pre-training. The result should be a factual, grounded answer that cites information from the actual articles.

```
number_of_chunks_to_retrieve = 3

# Sort the scores
highest_index = np.argmax(cosine_similarities)

# Pick the N highest scored chunks
indices = np.argsort(cosine_similarities[0])[::-1][:number_of_chunks_to_retrieve]
print(indices)
```

8.1.6.1 Bonus: How RAG Defeats Knowledge Cutoff and Hallucination

Every LLM has a **knowledge cutoff** — the date when its training data ended. Ask about events or models released after that date, and the LLM will either refuse to answer or, worse, confidently make something up (hallucinate). RAG solves this elegantly: by stuffing fresh, factual context into the prompt, we override the model's stale parametric memory with up-to-date evidence.

The next two cells demonstrate this with a concrete example. We provide a chunk about LLaMA 4 (released after most models' training cutoff) and ask the same question with and without that context. Watch how the augmented version gives a precise, correct answer while the non-augmented version either admits ignorance or hallucinates outdated information.

8.1.6.2 LLM Response Function (via `llm_cascade`)

Instead of a hardcoded Gemini or OpenAI client, we use `llm.generate()` from `llm_cascade`. It automatically picks the first available provider and falls back if quota is exhausted. Each call prints which provider/model was used.

```
# Using llm_cascade instead of hardcoded Gemini client

def gemini_response(system_prompt, prompt):
    """Generate a response using llm_cascade (auto-fallback across providers)."""
    try:
        response = llm.generate(prompt, system_prompt=system_prompt)
        return response.text
```



```
except Exception as e:
    return f'Error: {e}'
```

8.1.7 Without Augmentation – the Control Experiment

Now we ask the exact same question but *without* providing any context. The LLM must rely entirely on its parametric knowledge – whatever it memorized during training. If its training data predates the LLaMA 4 release, it will either say “I don’t know” (the honest answer) or hallucinate incorrect details (the dangerous answer). Compare this output to the augmented version above to see exactly how much value the retrieval step adds.

8.1.7.1 What We Built – and What Comes Next

Let us recap the full pipeline we built from scratch:

1. **Chunk** – Split raw articles into 1,024-character pieces.
2. **Embed** – Convert each chunk into a 384-dimensional vector using a local sentence-transformer.
3. **Store** – Keep the chunks and their embeddings in a simple Pandas DataFrame (our DIY vector database).
4. **Query** – Embed the user’s question with the same model.
5. **Retrieve** – Use cosine similarity to find the top-k most relevant chunks.
6. **Augment** – Stuff those chunks into the LLM prompt as context.
7. **Generate** – Let the LLM produce a grounded answer.

Every RAG framework – LangChain, LlamaIndex, Haystack – automates these exact seven steps. Now that you have built them by hand, you understand what those frameworks are doing under the hood. In the next chapters, we will use LangChain to do this more concisely, and we will explore RAG beyond documents: with SQL databases and graph databases as knowledge sources.

```
from google import genai
from google.genai import types

# Use the Gemini API to answer the questions based on the retrieved pieces of text.
try:
    # Formulating the system prompt and condition the model to answer only AI-
    # related questions.
    system_prompt = """You are an assistant and expert in answering questions from a
    chunks of content.
                                Only answer AI-related question, else say that you cannot
    answer this question."""

    # User prompt with the user's question
    prompt = """
        Read the following informations that might contain the context you require
        to answer the question.
        You can use the informations starting from the <START_OF_CONTEXT> tag and
        end with the <END_OF_CONTEXT> tag.
        Here is the content:\n\n<START_OF_CONTEXT>\n{}\n<END_OF_CONTEXT>\n\n"
        Please provide an informative and accurate answer to the following question
        based on the available context.
        Be concise and take your time. \nQuestion: {}\nAnswer:"
    """

    # Add the retrieved pieces of text to the prompt.
    prompt = prompt.format("".join(df.chunk[indices]), QUESTION)

    response = gemini_response(system_prompt, prompt)
    print(response)

except Exception as e:
    print(f"An error occurred: {e}")
```

```
# Consider this as a retrieved chunk
# https://ai.meta.com/blog/llama-4-multimodal-intelligence/
```

```
Example_chunk = ""
# Llama 4 Technical Overview
```

Meta has introduced the Llama 4 model family, marking a significant advancement in open-weight multimodal AI with three distinct variants designed for different use cases. The Llama 4 Scout represents the more compact option with 17 billion active parameters utilizing 16 experts within a total parameter count of 109 billion, designed to fit on a single H100 GPU with Int4 quantization. The Llama 4 Maverick serves as the flagship model with 17 billion active parameters but employs 128 experts across 400 billion total parameters, fitting on a single H100 host and designed as the primary workhorse for general assistant and chat applications. The preview model Llama 4 Behemoth functions as a teacher model with 288 billion active parameters, 16 experts, and nearly 2 trillion total parameters, currently still in training but demonstrating state-of-the-art performance on mathematical and reasoning benchmarks.

The architecture represents Meta's first implementation of mixture-of-experts (MoE) design in the Llama series, fundamentally changing how the models process information by activating only a fraction of total parameters for each token. The MoE architecture alternates between dense and mixture-of-experts layers, with the Maverick model specifically using 128 routed experts alongside a shared expert, where each token is processed by the shared expert and routed to one of the 128 specialized experts. This design significantly improves compute efficiency during both training and inference while delivering higher quality outputs compared to dense models with equivalent computational budgets.

Native multimodality represents another breakthrough, achieved through early fusion architecture that seamlessly integrates text and vision tokens into a unified model backbone during pre-training. This approach enables joint training with massive amounts of unlabeled text, image, and video data, with the vision encoder based on MetaCLIP but trained separately in conjunction with a frozen Llama model to optimize adaptation to the language model. The models support up to 48 images during pre-training and demonstrate strong performance with up to 8 images in post-training scenarios, enabling sophisticated visual reasoning and understanding tasks across multiple input modalities.

The training infrastructure incorporates several technical innovations, including the MetaP technique for reliable hyperparameter optimization that transfers well across different model configurations and training scenarios. The models were trained on over 30 trillion tokens, representing more than double the training data of Llama 3, with support for 200 languages including over 100 languages with more than 1 billion tokens each. Training efficiency was maximized through FP8 precision without quality sacrifice, achieving 390 TFLOPs per GPU utilization on 32,000 GPUs during Behemoth pre-training, while a specialized mid-training phase enhanced core capabilities and enabled the industry-leading 10 million token context length for Scout.

Context length capabilities represent a dramatic advancement, with Llama 4 Scout supporting 10 million tokens compared to Llama 3's 128K limit through the innovative iRoPE architecture that employs interleaved attention layers without positional embeddings and inference-time temperature scaling of attention. This architecture enables superior length generalization and opens possibilities for multi-document summarization, extensive user activity parsing for personalization, and reasoning over vast codebases, with compelling performance demonstrated in retrieval tasks and cumulative negative log-likelihood evaluations over 10 million tokens of code.

The post-training pipeline underwent significant revision with a three-stage approach consisting of lightweight supervised fine-tuning, online reinforcement learning, and lightweight direct preference optimization. A critical insight involved removing over 50% of data tagged as easy using Llama models as judges, focusing training on harder examples to prevent over-constraining that could restrict exploration during online RL and lead to suboptimal performance in reasoning, coding, and mathematics. The continuous online RL strategy alternated between model training and adaptive data filtering to retain only medium-to-hard difficulty prompts, proving highly beneficial for compute and accuracy trade-offs.

Safety and bias mitigation received comprehensive attention through multi-layered approaches spanning pre-training data filtering, post-training policy conformance, and system-level safeguards. Meta introduced Generative Offensive Agent Testing (GOAT) to address traditional red-teaming limitations by simulating multi-turn interactions of medium-skilled adversarial actors, enabling more efficient vulnerability detection while allowing human experts to focus on novel adversarial areas. Particular emphasis was placed on addressing political and social bias inherent in internet training data, with Llama 4 demonstrating significant improvements over Llama 3 in presenting balanced perspectives on contentious issues without favoring particular viewpoints.

The distillation process for creating smaller models from Behemoth required novel approaches, including a dynamic loss function that weighted soft and hard targets throughout training, with codistillation during pre-training amortizing computational costs for the majority of training data. Post-training the 2 trillion parameter Behemoth model necessitated pruning 95% of supervised fine-tuning data compared to 50% for smaller models, followed by large-scale reinforcement learning focused on sampling hard prompts through pass@k analysis and constructing training curricula of increasing difficulty. The unprecedented scale required complete infrastructure overhaul, including optimized MoE parallelization and fully asynchronous online RL training framework that improved training efficiency by approximately 10x over previous generations through flexible GPU allocation and resource balancing across multiple models.

```
QUESTION = "How many parameters LLaMA 4 model has?"
```

```
# Formulating the system prompt
```

```
system_prompt = """ You are an assistant and expert in answering questions from a  
chunks of content.
```

```
    Only answer AI-related question, else say that you cannot answer  
this question."""
```

```
# Combining the system prompt with the user's question
```

```
prompt = """Read the following informations that might contain the context you  
require to answer the question.
```

```
    You can use the informations starting from the <START_OF_CONTEXT> tag  
and end with the <END_OF_CONTEXT> tag.
```

```
    Here is the content:\n\n<START_OF_CONTEXT>\n{}\n<END_OF_CONTEXT>\n\n".
```

```
    Please provide an informative and accurate answer to the following  
question based on the available context.
```

```
    Be concise and take your time. \nQuestion: {}\nAnswer:
```

```
    """
```

```
prompt = prompt.format(Example_chunk, QUESTION)
```

```
response = gemini_response(system_prompt,prompt)
```

```
print(response)
```

```
QUESTION = "How many parameters LLaMA 4 model has?"
```

```
# Formulating the system prompt
```

```
system_prompt = "You are an assistant and expert in answering questions."
```

```
# Combining the system prompt with the user's question
```

```
prompt = "Be concise and take your time to answer the following question.  
\nQuestion: {}\nAnswer:"
```

```
prompt = prompt.format(QUESTION)
```

```
response = gemini_response(system_prompt,prompt)
```

```
print(response)
```

8.1.7.2 Key takeaways

- **RAG is seven concrete steps:** chunk, embed, store, query-embed, retrieve, augment, generate – frameworks hide them, but each is plain Python.
 - **Cosine similarity on sentence-transformer embeddings** turns semantic search into a matrix operation: relevant chunks score near 1, unrelated ones near 0.
 - **A Pandas DataFrame is a valid vector store** for learning and small datasets; production systems just swap in Pinecone, Weaviate, or ChromaDB for speed and persistence.
 - **Augmenting the prompt with retrieved chunks** grounds the LLM in fresh facts, defeating both knowledge-cutoff gaps and hallucination in a single shot.
 - **Local embedding models** like `all-MiniLM-L6-v2` keep the “R” in RAG free and private, while `llm_cascade` handles provider fallback for the “G”.
-

8.1.7.3 Run the code

To run this notebook, copy the URL below into your browser’s address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~5 minutes (embedding generation + LLM calls)

```
https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/RAG_first_principles.ipynb
```


Chapter 9

RAG with Vector Databases

The previous chapter built RAG from scratch with numpy arrays. In practice, you use a **vector database** to store and search embeddings efficiently. This chapter explores ChromaDB — a popular open-source vector database — applied to real-world datasets.

You'll see how the same retrieve-augment-generate pattern scales from a few documents to hundreds, and how ChromaDB handles the embedding storage, indexing, and similarity search automatically.

9.1 RAG Example (North America): ChromaDB + LLM Cascade

Every large language model has a knowledge cutoff – a date beyond which it simply has not seen the world. Ask it about last quarter’s GDP figures or your company’s internal policies, and it will either confess ignorance or, worse, confidently hallucinate an answer. **Retrieval-Augmented Generation (RAG)** solves this by giving the model a research assistant: before answering, the system retrieves relevant documents from a knowledge base and injects them into the prompt as context. The model then generates its response grounded in real evidence rather than memory alone.

In this notebook we build a compact RAG pipeline over the **CIA World Factbook** – a public-domain dataset that contains structured intelligence about every country on earth, from population statistics to trade balances to energy production. We scope the demo to **North America only** so that indexing runs in under a minute on free-tier hardware, but the architecture generalizes to any document corpus.

What you will learn:

- How to ingest structured JSON and flatten it into retrieval-ready text documents
- How recursive chunking with overlap preserves context continuity at chunk boundaries
- How ChromaDB handles embedding and vector storage automatically, with no external API key required
- How grounding an LLM on retrieved context changes its answers – and where it still falls short

Why ChromaDB? Unlike cloud vector databases that require API keys and billing, ChromaDB runs entirely in-process. It pairs with a local sentence-transformer model (`all-MiniLM-L6-v2`) to embed your text, so the entire pipeline – from ingestion to retrieval to generation – works offline and for free. That makes it ideal for classroom demos where you want students focused on the concepts, not on credential management.

This notebook is the framework-assisted version of the pipeline you built by hand in Chapter 8’s `rag_first_principles.ipynb` – the seven manual steps are still there, but ChromaDB and LangChain hide most of the plumbing.

```
%pip install -q -U chromadb langchain-community langchain-text-splitters pandas
requests git+https://github.com/KarAnalytics/llm_cascade.git sentence-transformers
```

9.1.1 1) Imports and Environment Setup

Before we build anything, we need our toolkit. The libraries below each play a distinct role in the RAG pipeline: `pandas` for tabular inspection, `requests` for fetching raw data from GitHub, `chromadb` for vector storage and retrieval, and `RecursiveCharacterTextSplitter` from LangChain for intelligent chunking. Notice that we do not import any embedding library explicitly – ChromaDB will handle that internally when we configure it later.

API keys are resolved automatically. In Google Colab, you can store them in Secrets (e.g., `GEMINI_API_KEY`, `OPENAI_API_KEY`, `GROQ_API_KEY`). Outside Colab, the code falls back to environment variables. The `llm_cascade` library tries each configured provider in order, so you only need one working key to proceed.

```
import os
import pandas as pd
import requests
import chromadb
from langchain_text_splitters import RecursiveCharacterTextSplitter
```

```
from llm_cascade import get_cascade
```

```
llm = get_cascade()
```

```
def generate_text(prompt, system_prompt=None, **kwargs):
    response = llm.generate(prompt, system_prompt=system_prompt)
    return response.text, response.provider
```

```
def has_llm_provider():
    return True # llm_cascade auto-detects available providers
```

9.1.2 2) Build a Minimal RAG Pipeline

A RAG pipeline has four distinct phases, each solving a different problem. First, **data ingestion** fetches raw documents and converts them into plain text that an embedding model can understand. Second, **chunking** splits long documents into smaller, overlapping pieces so that retrieval can zero in on the most relevant passage rather than returning an entire country profile. Third, **vector storage** embeds each chunk into a high-dimensional vector and indexes it for fast similarity search. Fourth, **grounded generation** retrieves the top-K most relevant chunks for a user question and feeds them to the LLM as context, constraining its answer to what the evidence actually says.

We will build each phase as a standalone function, then wire them together at the end. This modular design makes it easy to swap components – for example, replacing ChromaDB with Pinecone, or changing the chunking strategy – without rewriting the rest of the pipeline.

9.1.2.1 Function: `_collect_text_lines`

The CIA World Factbook stores each country as deeply nested JSON – dictionaries inside dictionaries inside lists. A vector database, however, needs flat text strings to embed. This recursive helper walks the entire JSON tree, collects every leaf value, and pairs it with its full key path (e.g., `Economy.GDP.real growth rate.text: 2.1%`). The result is a list of human-readable lines that preserve the hierarchical context of each fact.

Why does this matter? If we simply dumped the raw JSON into ChromaDB, the embedding model would waste capacity on braces, brackets, and structural tokens that carry no semantic meaning. By flattening first, every token in the embedded text contributes to the meaning of the passage, which directly improves retrieval quality when a student later asks “What is Mexico’s GDP growth rate?”

```
# --- PHASE 1: DATA INGESTION (Pandas + GitHub) ---
def _collect_text_lines(obj, parent_key=""):
    """Recursively flatten nested JSON into readable key-value text lines."""
    lines = []

    if isinstance(obj, dict):
        for k, v in obj.items():
            next_key = f"{parent_key}.{k}" if parent_key else str(k)
            lines.extend(_collect_text_lines(v, next_key))
    elif isinstance(obj, list):
        for i, item in enumerate(obj):
            next_key = f"{parent_key}[{i}]" if parent_key else f"[{i}]"
            lines.extend(_collect_text_lines(item, next_key))
    else:
        if obj is None:
            return []
        text = str(obj).strip()
        if text:
            lines.append(f"{parent_key}: {text}")
    return lines
```

9.1.2.2 Function: `_get_country_json_paths`

The full CIA World Factbook covers over 250 countries and territories, which would take several minutes to download and index on free-tier hardware. For a classroom demo, speed matters more than completeness, so this function filters the GitHub repository tree to keep only files under `north-america/`. That gives us seven countries – enough to demonstrate every RAG concept while keeping the total pipeline runtime under 60 seconds.

The function uses the GitHub Trees API to list all files in the repository without downloading them, then applies the region filter. The sorted, deduplicated output ensures that every student in the room gets the same document order, which makes it easier to compare results during class discussion.

```
def _get_country_json_paths():
    """Discover country JSON files, restricted to North America."""
    tree_url = "https://api.github.com/repos/factbook/factbook.json/git/trees/
master?recursive=1"
    r = requests.get(tree_url, timeout=30)
    r.raise_for_status()
    tree = r.json().get("tree", [])

    allowed_prefixes = ("north-america/",)
    country_paths = []
    for item in tree:
        path = item.get("path", "")
        if item.get("type") == "blob" and path.endswith(".json") and "/" in path:
            if path.lower().startswith(allowed_prefixes):
                country_paths.append(path)

    # Deduplicate and sort for stable classroom demos.
    return sorted(set(country_paths))
```

9.1.2.3 Function: `get_world_data`

This is the orchestrator for the entire ingestion phase. It discovers country files, downloads each one, flattens the nested JSON into readable text using `_collect_text_lines`, and packages the result as a list of document dictionaries – each containing a `content` string and a `metadata` dict with the country code and source path. The metadata travels alongside the content through chunking and indexing, so that when we retrieve a chunk later, we always know which country it came from.

Think of this function as the “ETL” step of the RAG pipeline: it extracts raw data from an external source, transforms it into a format suitable for embedding, and loads it into memory ready for the next phase.

```
def get_world_data():
    """Fetch all country records and convert each into one retrieval-ready full-text
    document."""
    print("Step 1: Discovering country files from GitHub...")
    base_url = "https://raw.githubusercontent.com/factbook/factbook.json/master"
    country_paths = _get_country_json_paths()
    print(f"Discovered {len(country_paths)} country files.")

    docs = []
    for path in country_paths:
        try:
            r = requests.get(f"{base_url}/{path}", timeout=20)
            r.raise_for_status()
            payload = r.json()
        except Exception as exc:
            print(f"Skipping {path}: {exc}")
            continue

        lines = _collect_text_lines(payload)
        if not lines:
            continue

        country_code = path.split("/")[-1].replace(".json", "").upper()
        combined_text = (
            f"COUNTRY_CODE: {country_code}\n"
            f"SOURCE_PATH: {path}\n\n"
            + "\n".join(lines)
        )
```

```

docs.append(
    {
        "content": combined_text,
        "metadata": {
            "country": country_code,
            "path": path,
        },
    }
)

print(f"Loaded {len(docs)} country documents.")
return docs

```

9.1.2.4 Function: `chunk_documents`

A single country document can run to thousands of lines – far too long for an embedding model to capture in one vector. Chunking solves this by splitting each document into smaller, digestible pieces. We use LangChain’s `RecursiveCharacterTextSplitter`, which tries to break text at natural boundaries (paragraph breaks, then sentence boundaries, then word boundaries) before falling back to a hard character limit.

The two key parameters are `chunk_size` (800 characters by default) and `chunk_overlap` (100 characters). The overlap is crucial: without it, a fact that straddles two chunks would be split in half, and neither chunk would contain the complete information. Experimenting with these values is one of the most impactful tuning knobs in any RAG system.

```

## --- PHASE 2: RECURSIVE CHUNKING ---
def chunk_documents(docs, chunk_size=800, chunk_overlap=100):
    """Chunk long records so semantic retrieval can match smaller relevant spans."""
    print("Step 2: Chunking with overlap...")
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
    )

    chunks, metadatas, ids = [], [], []
    for doc in docs:
        split_texts = splitter.split_text(doc["content"])
        for i, text in enumerate(split_texts):
            chunks.append(text)
            metadatas.append(doc["metadata"])
            ids.append(f"{doc['metadata']['country']}_{i}")
    return chunks, metadatas, ids

# Small inspection check for students before moving on.
docs_preview = get_world_data()
print(f"Loaded {len(docs_preview)} source documents.")
print("Example metadata:", docs_preview[0]["metadata"] if docs_preview else "No documents loaded")

```

9.1.2.5 Checkpoint 1: Data and Chunking Reflection

Pause here and consider these questions before moving on. They highlight the design decisions that most affect RAG quality downstream.

1. Why do we transform structured JSON fields into one text document per country before retrieval? What would go wrong if we indexed the raw JSON directly?
2. How might changing `chunk_size` from 800 to 300 affect precision and recall? Would the answer to “What is Canada’s population?” improve or degrade?
3. What is the purpose of overlap, and what failure mode does it reduce?

9.1.2.6 Function: `build_vector_store`

This function wires together the embedding model selection, collection management, and batch upsert logic. We explicitly specify `all-MiniLM-L6-v2` as the embedding model – a compact, fast sentence-transformer that runs locally without any API key. It produces 384-dimensional vectors and handles English text well enough for our factbook data.

The function deletes any existing collection before creating a fresh one. This is a classroom convenience: it prevents the confusing situation where a student runs the notebook twice and ends up with duplicate chunks that skew retrieval results. In production, you would instead use versioned collections or incremental upserts with deduplication logic.

```
def build_vector_store(docs, db_path="./factbook_db", collection_name="world_facts",
batch_size=5000):
    """Create (or reset) a Chroma collection and upsert chunked records in
    batches."""
    chunks, metadatas, ids = chunk_documents(docs)

    chroma_client = chromadb.PersistentClient(path=db_path)

    # Explicitly specify the embedding model that converts text chunks into vectors
    from chromadb.utils.embedding_functions import
SentenceTransformerEmbeddingFunction
    embedding_fn = SentenceTransformerEmbeddingFunction(model_name="all-MiniLM-L6-
v2")
    print(" Embedding model: all-MiniLM-L6-v2 (local, no API key needed)")

    # Delete any existing collection to avoid embedding function conflicts, then
    create fresh
    try:
        chroma_client.delete_collection(name=collection_name)
    except Exception:
        pass
    collection = chroma_client.create_collection(
        name=collection_name,
        embedding_function=embedding_fn,
    )

    print("Step 3: Storing vectors in ChromaDB...")
    for start in range(0, len(ids), batch_size):
        stop = start + batch_size
        collection.upsert(
            documents=chunks[start:stop],
            metadatas=metadatas[start:stop],
            ids=ids[start:stop],
        )

    print(f"Stored {len(ids)} chunks in collection '{collection_name}'.")
    return collection

collection = build_vector_store(docs_preview)
```

9.1.2.7 2B) Retrieve Context and Generate a Grounded Answer

This is where the “retrieval” meets the “generation” in RAG. When a user asks a question, we first embed the question using the same sentence-transformer model and find the top-K nearest chunks in the vector store. Those chunks become the context paragraph in the LLM prompt, and a carefully worded system instruction tells the model to answer **only** from the provided context – if the answer is not there, it should say so rather than guessing.

This retrieve-then-generate pattern is the beating heart of every RAG system. The retrieval step acts as an evidence filter, dramatically narrowing the LLM’s attention from its entire training corpus down to a handful of relevant passages. The prompting constraint – “use **ONLY** the provided context” – further reduces hallucination by giving the model explicit permission to say “I don’t know.” Together, these two mechanisms

produce answers that are both relevant and verifiable: a human can inspect the retrieved chunks to confirm whether the model's claims are actually supported by the evidence.

9.1.2.8 Functions: `answer_with_rag` and `answer_without_rag`

These two functions are the core of our teaching comparison. `answer_with_rag` performs the full retrieve-then-generate pipeline: it queries ChromaDB for the top-K chunks, assembles them into a context block, and prompts the LLM to answer only from that evidence. `answer_without_rag` skips retrieval entirely and sends the bare question to the LLM, relying solely on whatever the model memorized during pre-training.

Running the same question through both functions reveals exactly what RAG adds – and what it costs. The RAG answer will be grounded in specific, verifiable data from the Factbook (exchange rates, population figures, trade balances), while the non-RAG answer will draw on the model's general knowledge, which may be outdated, vague, or simply wrong. Comparing them side by side is the single most effective way to build intuition for when RAG is worth the engineering overhead and when a bare LLM call suffices.

```
def answer_with_rag(
    question,
    collection,
    n_results=4,
    **kwargs,
):
    """Retrieve nearest chunks, then answer using the LLM cascade."""
    print(f"Step 4: Retrieving context for: '{question}'")
    results = collection.query(query_texts=[question], n_results=n_results)
    context = "\n\n".join(results["documents"][0])

    print("Step 5: Generating grounded answer...")
    system_prompt = "You are a careful analyst. Use ONLY the provided context to answer. If the answer is not in the context, say you don't know."
    user_prompt = f"CONTEXT:\n{context}\n\nQUESTION:\n{question}"
    answer_text, provider = generate_text(
        prompt=user_prompt,
        system_prompt=system_prompt,
    )
    print(f"Provider used: {provider}")
    return answer_text, context

def answer_without_rag(question, **kwargs):
    """Direct LLM answer without retrieval."""
    answer_text, provider = generate_text(prompt=question)
    print(f"Provider used: {provider}")
    return answer_text
```

9.1.2.9 2C) Run an End-to-End Query

Now we put it all together. The cell below runs one or more comparison questions through both pathways – with and without RAG – and prints the results side by side. Pay close attention to the [RETRIEVED CONTEXT PREVIEW] section: it shows you exactly which chunks the vector store returned. This is your audit trail. If the RAG answer is wrong, the first thing to check is whether the retrieved context actually contained the relevant information. A bad answer from good context is a generation problem (fix the prompt); a bad answer from irrelevant context is a retrieval problem (fix the chunking or the query).

Tip: try uncommenting the other comparison questions to see how the system handles hallucination tests (asking about fictional “Atlantis”), cross-country comparisons, and questions that blend factbook data with general world knowledge.

9.1.2.10 Function: `preview`

This small helper trims long model outputs and retrieved context so the classroom output stays readable. Without it, a single country's flattened text can fill several screens, burying the pedagogical signal in noise.

The `max_len` parameter defaults to 900 characters for answers and 1,200 for context, which is enough to see the key facts without scrolling endlessly.

```
comparison_questions = [
    #{
        # "type": "Hallucination test (out-of-scope)",
        # "question": "What is the current prime minister of Atlantis, and what is
        Atlantis GDP growth?",
    },
    {
        "type": "Local info extraction",
        "question": "Summarize Mexico's economy overview in 3 bullet points.",
    },
    #{
        # "type": "Local info extraction",
        # "question": "What is the population and GDP of Canada?",
    },
    #{
        # "type": "Local + public knowledge blend",
        # "question": "Explain how the United States trade policies may influence its
        North American neighbors.",
    },
]

def preview(text, max_len=900):
    text = text or ""
    return text[:max_len] + ("..." if len(text) > max_len else "")

if "collection" not in globals():
    print("Error: Build the Chroma vector store first before running this
    comparison.")
elif not has_llm_provider():
    print("Error: No LLM API key configured. Set at least one API key in Colab
    Secrets.")
else:
    try:
        for i, item in enumerate(comparison_questions, start=1):
            q_type = item["type"]
            q = item["question"]
            print("=" * 90)
            print(f"Q{i}. {q_type}")
            print(f"QUESTION: {q}")
            print()

            direct_answer = answer_without_rag(q)
            print("[WITHOUT RAG]")
            print(preview(direct_answer))

            rag_answer, rag_context = answer_with_rag(q, collection, n_results=4)
            print("[WITH RAG]")
            print(preview(rag_answer))

            print("[RETRIEVED CONTEXT PREVIEW]")
            print(preview(rag_context, max_len=1200))
            print()
    except Exception as e:
        print(f"Error: {e}")
```

9.1.2.11 Checkpoint 4: Evaluation Reflection

Now that you have seen the with-RAG and without-RAG answers side by side, reflect on what the comparison reveals about each approach's strengths and weaknesses.

1. Compare the RAG answer against the retrieved context preview. Which specific claims in the answer are directly supported by the retrieved chunks, and which (if any) appear to be the model filling in gaps?
2. What additional evidence would improve confidence in the answer? Would retrieving more chunks (`n_results=8` instead of 4) help, or would it introduce noise?
3. Design one out-of-scope query (about a country not in our index, or a topic the Factbook does not cover) and predict how each approach – with RAG and without RAG – should respond.

9.1.2.12 Function: `preview`

This small helper trims long model outputs and retrieved context so the classroom output stays readable.

Why it matters: it keeps the demo focused on interpretation instead of overwhelming students with raw text.

9.1.3 Key takeaways

- **RAG pipelines** have four phases – ingestion, chunking, vector storage, and grounded generation – and each can be swapped independently without rewriting the others.
 - **Flattening nested JSON** into key-path text lines gives embeddings semantic tokens to work with instead of wasting capacity on structural punctuation.
 - **Recursive chunking with overlap** prevents facts from being split across boundaries, and tuning `chunk_size` and `chunk_overlap` is the single biggest retrieval-quality knob.
 - **ChromaDB plus `all-MiniLM-L6-v2`** runs entirely locally, so the full ingest-to-answer loop works offline with no API keys or billing.
 - **Comparing with-RAG and without-RAG** answers side by side exposes hallucination, and the retrieved-context preview is your audit trail for diagnosing wrong answers.
-

9.1.4 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~5 minutes (data download + embedding + LLM)

```
https://colab.research.google.com/github/KarAnalytics/code\_demos/blob/main/RAG\_countries\_NA.ipynb
```

9.2 RAG over Many Documents with ChromaDB

Learning goals:

- Understand data ingestion and document construction
- Learn recursive chunking for retrieval quality
- Build a vector index with ChromaDB
- Ground LLM generation using retrieved context

This is the full-scale sibling of `rag_countries_chromadb.ipynb` (North America only) and of Chapter 8's from-scratch RAG — same seven-step pipeline, just scaled to every country on earth so the embedding and retrieval costs actually start to bite.

```
# Install modern package names used by current LangChain ecosystem.
# Note: text splitters are now in a dedicated package: langchain-text-splitters
%pip install -q -U google-genai chromadb langchain-community langchain-text-splitters pandas requests git+https://github.com/KarAnalytics/llm_cascade.git sentence-transformers
```

9.2.1 1) Imports and Environment Setup

This section imports core libraries and handles API key lookup.

In Google Colab, you can store your API keys in Secrets (e.g., `GEMINI_API_KEY`, `OPENAI_API_KEY`, `GRQQ_API_KEY`, etc.). Outside Colab, the code falls back to environment variables.

```
import os
import pandas as pd
import requests
import chromadb
from langchain_text_splitters import RecursiveCharacterTextSplitter
```

```
from llm_cascade import get_cascade

llm = get_cascade()
```

9.2.2 2) Build a Minimal RAG Pipeline

Pipeline phases:

1. Data ingestion from GitHub JSON documents
2. Chunking with overlap to preserve context continuity
3. Vector storage and semantic retrieval in ChromaDB
4. Grounded generation with Gemini

9.2.2.1 Function: `_collect_text_lines`

This recursive helper converts nested JSON into flat text lines that can be embedded and retrieved.

Why it matters: vector databases work on text, so structured JSON must be turned into readable, searchable content first.

```
# --- PHASE 1: DATA INGESTION (Pandas + GitHub) ---
def _collect_text_lines(obj, parent_key=""):
    """Recursively flatten nested JSON into readable key-value text lines."""
    lines = []

    if isinstance(obj, dict):
        for k, v in obj.items():
            next_key = f"{parent_key}.{k}" if parent_key else str(k)
            lines.extend(_collect_text_lines(v, next_key))
    elif isinstance(obj, list):
        for i, item in enumerate(obj):
```

```

        next_key = f"{parent_key}[{i}]" if parent_key else f"[{i}]"
        lines.extend(_collect_text_lines(item, next_key))
    else:
        if obj is None:
            return []
        text = str(obj).strip()
        if text:
            lines.append(f"{parent_key}: {text}")
    return lines

```

9.2.2.2 Function: `_get_country_json_paths`

This function discovers which country JSON files exist in the GitHub repository.

Why it matters: the demo stays flexible because it does not rely on a hard-coded country list.

```

def _get_country_json_paths():
    """Discover all country JSON files in the factbook repo (no hard-coded country
    list)."""
    tree_url = "https://api.github.com/repos/factbook/factbook.json/git/trees/
    master?recursive=1"
    r = requests.get(tree_url, timeout=30)
    r.raise_for_status()
    tree = r.json().get("tree", [])

    country_paths = []
    for item in tree:
        path = item.get("path", "")
        if item.get("type") == "blob" and path.endswith(".json") and "/" in path:
            if not path.lower().startswith(("source/", "tools/", "scripts/")):
                country_paths.append(path)

    # Deduplicate and sort for stable classroom demos.
    return sorted(set(country_paths))

```

9.2.2.3 Function: `get_world_data`

This function fetches each country record, flattens it into text, and packages it with metadata.

Why it matters: it creates the retrieval-ready documents that the rest of the RAG pipeline depends on.

```

def get_world_data():
    """Fetch all country records and convert each into one retrieval-ready full-text
    document."""
    print("Step 1: Discovering country files from GitHub...")
    base_url = "https://raw.githubusercontent.com/factbook/factbook.json/master"
    country_paths = _get_country_json_paths()
    print(f"Discovered {len(country_paths)} country files.")

    docs = []
    for path in country_paths:
        try:
            r = requests.get(f"{base_url}/{path}", timeout=20)
            r.raise_for_status()
            payload = r.json()
        except Exception as exc:
            print(f"Skipping {path}: {exc}")
            continue

```



```

lines = _collect_text_lines(payload)
if not lines:
    continue

country_code = path.split("/")[-1].replace(".json", "").upper()
combined_text = (
    f"COUNTRY_CODE: {country_code}\n"
    f"SOURCE_PATH: {path}\n\n"
    + "\n".join(lines)
)

docs.append(
    {
        "content": combined_text,
        "metadata": {
            "country": country_code,
            "path": path,
        },
    },
)

print(f"Loaded {len(docs)} country documents.")
return docs

```

9.2.2.4 Function: `chunk_documents`

This function splits long country documents into overlapping chunks before indexing.

Why it matters: smaller chunks improve retrieval precision, while overlap reduces the chance of losing context at chunk boundaries.

```

## --- PHASE 2: RECURSIVE CHUNKING ---
def chunk_documents(docs, chunk_size=800, chunk_overlap=100):
    """Chunk long records so semantic retrieval can match smaller relevant spans."""
    print("Step 2: Chunking with overlap...")
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
    )

    chunks, metadatas, ids = [], [], []
    for doc in docs:
        split_texts = splitter.split_text(doc["content"])
        for i, text in enumerate(split_texts):
            chunks.append(text)
            metadatas.append(doc["metadata"])
            ids.append(f"{doc['metadata']['country']}_{i}")
    return chunks, metadatas, ids

# Small inspection check for students before moving on.
docs_preview = get_world_data()
print(f"Loaded {len(docs_preview)} source documents.")
print("Example metadata:", docs_preview[0]["metadata"] if docs_preview else "No documents loaded")

```

9.2.2.5 Checkpoint 1: Data and Chunking Reflection

1. Why do we transform structured JSON fields into one text document per country before retrieval?
2. How might changing `chunk_size` from 800 to 300 affect precision and recall?
3. What is the purpose of overlap, and what failure mode does it reduce?

9.2.2.6 2A) Build and Populate the Vector Store

In this phase, we transform chunks into retrievable vector records inside ChromaDB.

Teaching note: for reproducible classroom runs, we clear the collection before re-inserting documents.

9.2.2.7 Function: `build_vector_store`

This function creates the Chroma collection, clears older demo records, and writes new chunk batches into the vector store.

Why it matters: it is the bridge between text preprocessing and semantic retrieval.

****This code ran for 30 minutes for me, so be warned. Try GPU/TPU in runtime to speedup. ****

```
def build_vector_store(docs, db_path="./factbook_db", collection_name="world_facts",
    batch_size=5000):
    """Create (or reset) a Chroma collection and upsert chunked records in
    batches."""
    chunks, metadatas, ids = chunk_documents(docs)

    chroma_client = chromadb.PersistentClient(path=db_path)
    collection = chroma_client.get_or_create_collection(name=collection_name)

    # Clear older records so each class run starts cleanly.
    existing = collection.get(include=[])
    if existing.get("ids"):
        collection.delete(ids=existing["ids"])

    print("Step 3: Storing vectors in ChromaDB...")
    for start in range(0, len(ids), batch_size):
        stop = start + batch_size
        collection.upsert(
            documents=chunks[start:stop],
            metadatas=metadatas[start:stop],
            ids=ids[start:stop],
        )

    print(f"Stored {len(ids)} chunks in collection '{collection_name}'.")
    return collection

collection = build_vector_store(docs_preview)
```

9.2.2.8 Checkpoint 2: Vector Store Reflection

1. Why do we clear old records before upserting during a classroom demo?
2. What reproducibility issue appears if we repeatedly insert without cleanup?
3. If this were production, when would you keep historical vectors instead?

9.2.2.9 2B) Retrieve Context and Generate a Grounded Answer

This phase demonstrates the core RAG idea: retrieve relevant evidence first, then condition the LLM on that evidence.

Prompting rule used here: answer only from retrieved context; otherwise say you do not know.

9.2.2.10 Functions: `answer_with_rag` and `answer_without_rag`

These two functions create the core teaching comparison in the notebook.

`answer_with_rag` retrieves evidence before generation. `answer_without_rag` sends the question directly to Gemini with no retrieved context.

```
def answer_with_rag(question, collection, n_results=4):
    '''Retrieve nearest chunks, then ask LLM to answer using only retrieved
    context.'''
```

```

print(f"Step 4: Retrieving context for: '{question}'")
results = collection.query(query_texts=[question], n_results=n_results)
context = (chr(10) + chr(10)).join(results['documents'][0])

print('Step 5: Generating grounded answer with LLM...')
system_prompt = 'You are a careful analyst. Use ONLY the provided context to answer. If the answer is not in the context, say you don' + chr(39) + 't know.'
prompt = 'CONTEXT:' + chr(10) + context + chr(10) + chr(10) + 'QUESTION:' + chr(10) + question
response = llm.generate(prompt, system_prompt=system_prompt)
return response.text, context

def answer_without_rag(question):
    '''Direct LLM answer without retrieval for comparison against RAG.'''
    response = llm.generate(question)
    return response.text

```

9.2.2.11 Checkpoint 3: Retrieval and Prompting Reflection

1. Why do we retrieve first and generate second in a RAG system?
2. What is the effect of changing `n_results` from 2 to 5?
3. How does the prompt reduce hallucinations, and where can it still fail?

9.2.2.12 2C) Run an End-to-End Query

You can now test questions and inspect both the answer and retrieved evidence.

Tip: print retrieved context in class to discuss why the model answered the way it did.

9.2.2.13 Function: `preview`

This small helper trims long model outputs and retrieved context so the classroom output stays readable.

Why it matters: it keeps the demo focused on interpretation instead of overwhelming students with raw text.

```

comparison_questions = [
    {
        "type": "Hallucination test (out-of-scope)",
        "question": "What is the current prime minister of Atlantis, and what is Atlantis GDP growth?",
    },
    {
        "type": "Local info extraction",
        "question": "Summarize Japan's economy overview in 3 bullet points.",
    },
    {
        "type": "Local info extraction",
        "question": "Compare inflation-related details for Brazil and Canada.",
    },
    {
        "type": "Local + public knowledge blend",
        "question": "Explain how Japan's economic profile may influence its role in global supply chains today.",
    },
]

def preview(text, max_len=900):
    text = text or ""
    return text[:max_len] + ("..." if len(text) > max_len else "")

if "collection" not in globals():

```

```

    print("Error: Build the Chroma vector store in Cell 9 before running this
comparison.")
else:
    try:
        for i, item in enumerate(comparison_questions, start=1):
            q_type = item["type"]
            q = item["question"]
            print("\n" + "=" * 90)
            print(f"Q{i}. {q_type}")
            print(f"QUESTION: {q}\n")

            direct_answer = answer_without_rag(q)
            print("[WITHOUT RAG]")
            print(preview(direct_answer))

            rag_answer, rag_context = answer_with_rag(q, collection, n_results=4)
            print("\n[WITH RAG]")
            print(preview(rag_answer))

            print("\n[RETRIEVED CONTEXT PREVIEW]")
            print(preview(rag_context, max_len=1200))
    except Exception as e:
        print(f"Error: {e}")

```

9.2.2.14 Checkpoint 4: Evaluation Reflection

1. Compare the final answer against the retrieved context. Which statements are directly supported?
2. What additional evidence would improve confidence in the answer?
3. Design one out-of-scope query and explain how the system should respond.

9.2.3 3) Teaching Notes and Suggested Exercises

Exercises for class discussion:

- Change `chunk_size` and `chunk_overlap`; evaluate answer quality
- Increase `n_results` and observe grounding changes
- Add more countries and test comparative questions
- Intentionally ask out-of-scope questions to inspect hallucination control

9.2.4 Key takeaways

- **Scaling from regional to global** (200+ countries) shows how embedding cost and runtime grow linearly with the corpus – plan for GPU/TPU when indexing at scale.
- **Dynamic discovery** via the GitHub Trees API keeps the pipeline flexible, so adding or removing countries does not require editing a hard-coded list.
- **Clearing the collection** before each upsert keeps classroom runs reproducible, but in production you would version collections or deduplicate incrementally instead.
- **Out-of-scope questions** (like “Atlantis”) reveal whether the “answer only from context” prompt actually prevents hallucination or just softens it.
- **Tuning `n_results` and chunk size** trades precision for recall – more chunks add evidence but also noise that can dilute the grounded answer.

9.2.5 Run the code

To run this notebook, copy the URL below into your browser’s address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~10 minutes (200+ countries, embedding heavy)

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/RAG_allcountries_ChromaDB.ipynb

9.3 RAG with a Featherweight LLM

Can Retrieval-Augmented Generation work with a model so small it barely qualifies as a “large” language model? In this notebook we find out. We wire up **SmolLM2-135M** — a HuggingFace model with only 135 million parameters — into a full LangChain RAG pipeline backed by ChromaDB. The point is not to get GPT-4-quality answers; it is to prove that the *pattern* of RAG (retrieve relevant context, stuff it into a prompt, generate) works at any scale. If you understand this notebook, you understand the skeleton that every production RAG system is built on — the only difference is swapping in a bigger brain.

The retrieval half of this pipeline is the same cosine-similarity-over-embeddings dance we implemented by hand in Chapter 8’s `rag_first_principles.ipynb`; what we are stress-testing here is whether a tiny generator can still make use of it.

9.3.1 Install Dependencies

The cell below installs everything we need in one shot. **LangChain** provides the orchestration glue — it connects the language model, the vector store, and the retrieval logic into a single pipeline. **ChromaDB** (`langchain-chroma`) is an open-source vector database that will store our document embeddings. **HuggingFace Transformers** gives us access to the SmolLM2 model itself, and **sentence-transformers** provides the embedding model that converts text into vectors. Think of this as assembling a toolkit: LangChain is the workbench, ChromaDB is the filing cabinet, and the HuggingFace libraries are the engine.

9.3.2 1. Load the “Micro” Model

Here is where the experiment gets interesting. **SmolLM2-135M-Instruct** has only 135 million parameters — roughly 500 times smaller than GPT-3 and over a thousand times smaller than GPT-4. We chose it deliberately: it is small enough to run on a free Colab GPU in seconds, yet it has been instruction-tuned, so it understands the format of a question-and-answer prompt. We load it in half-precision (`float16`) and force it onto the GPU with `device_map="cuda"` to keep inference fast.

The `max_new_tokens=50` cap is another deliberate choice. A 135M-parameter model tends to ramble or repeat itself if you let it generate too much text. Capping the output at 50 tokens forces it to be concise — and lets us see quickly whether it actually absorbed the retrieved context or just hallucinated. Finally, we wrap the HuggingFace pipeline in `HuggingFacePipeline` so that LangChain can call it like any other LLM.

9.3.3 2. Upload Your Document

RAG needs a knowledge source. Here we ask you to upload a plain-text file — this could be a Wikipedia article, a company memo, or any prose document. The file is read into a single Python string that we will chunk and embed in the next step. In a production system this upload step would be replaced by a document ingestion pipeline, but for a classroom demo, drag-and-drop keeps things simple.

9.3.4 3. Chunk, Embed, and Build the RAG Pipeline

This single cell does three things that form the backbone of any RAG system.

Chunking. The `CharacterTextSplitter` breaks the uploaded text into 1,000-character pieces with no overlap. Chunking matters because embedding models and LLMs have limited context windows — feeding them the entire document at once would either truncate it or dilute the signal. Smaller chunks mean each vector represents a focused topic, which improves retrieval precision.

Embedding and storing. Each chunk is converted into a 384-dimensional vector by `all-MiniLM-L6-v2`, a lightweight sentence-transformer. These vectors are stored in **ChromaDB**, an in-memory vector database. When a question arrives, ChromaDB will compare the question’s vector against every chunk vector using cosine similarity and return the closest matches.

Pipeline assembly. `RetrievalQA.from_chain_type` ties the retriever to the LLM. We set `k=1` — retrieve only the single best-matching chunk — to keep things fast and to make it crystal clear which piece of context the model is working from. The “stuff” chain type simply concatenates the retrieved chunk into the prompt; other chain types exist for multi-document summarization, but “stuff” is the simplest and most transparent.

9.3.5 4. Query the Pipeline

Now we put the pipeline to the test. When you call `rag.invoke(query)`, LangChain performs three steps behind the scenes: (1) embed the question, (2) retrieve the best-matching chunk from ChromaDB, and (3) pass both the chunk and the question to SmolLM2 for generation. Look at the output carefully — you will likely see the model echo the LangChain prompt template and the retrieved context before its answer. This is a quirk of very small models: they do not always know where the “instructions” end and the “answer” begins. A larger model would suppress the template and give you a clean response.

9.3.6 5. A Second Query

We run a second question to see how the model handles a different type of information need. Notice that the retrieval step may or may not pull a different chunk this time — it depends on which chunk’s embedding is closest to this new question. The quality of the answer will vary: sometimes the tiny model nails it, sometimes it trails off or repeats itself. That inconsistency is the price of using a 135M-parameter model — and it is exactly why production systems use larger models for generation while keeping the same retrieval architecture you see here.

9.3.7 Takeaways: What Did We Learn?

The most important lesson from this notebook is that **RAG is an architecture, not a model**. The retrieve-then-generate pattern works whether the generator has 135 million parameters or 1.8 trillion. The retrieval side (chunking, embedding, vector search) was actually quite good here — cosine similarity found the right context every time. It was the *generation* side that struggled, because a tiny model lacks the reasoning capacity to synthesize a clean, well-structured answer.

In a real deployment, you would keep ChromaDB (or a production vector store like Pinecone or Weaviate) and swap SmolLM2 for a capable model like GPT-4o or Gemini. The retrieval pipeline stays the same; only the last mile — the LLM call — changes. That modularity is the whole point of the LangChain abstraction, and it is why understanding this featherweight demo transfers directly to building enterprise-grade RAG systems.

```
## Large chunks mean less work for the vector DB
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
docs = text_splitter.create_documents([raw_text])

embedding_model = HuggingFaceEmbeddings(model_name='sentence-transformers/all-
MiniLM-L6-v2')
db = Chroma.from_documents(docs, embedding_model)

# --- 3. The RAG Pipeline ---
# Use k=1 to make it even faster (only looks at the single best match)
rag = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type='stuff',
    retriever=db.as_retriever(search_kwargs={"k": 1})
)
```

```
## --- 4. Run the Query ---
query = "What is the main topic of the document?"
response = rag.invoke(query)

print(f"\n--- Answer ---\n{response['result']}")
```

```
# --- 5. Run the Query ---
query = "When did the event begin, when will it end?"
response = rag.invoke(query)

print(f"\n--- Answer ---\n{response['result']}")
```

9.3.8 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~8 minutes on T4 GPU (local model loading)

```
https://colab.research.google.com/github/KarAnalytics/code\_demos/blob/main/Simple\_RAG\_using\_feat\_weightAI.ipynb
```


9.4 LanceDB: A Local Vector Database

Vectors, the output data format of Neural Network models, can effectively encode information and serve a pivotal role in AI applications such as knowledge base, semantic search, Retrieval Augmented Generation (RAG) and more.

LanceDB is an open-source, serverless vector database that is embedded directly into your application. In this guide, we will walk you through how to set up LanceDB locally within minutes and use the Python client library to store and search vectors.

This notebook takes the in-memory Pandas DataFrame from Chapter 8's `rag_first_principles.ipynb` and swaps it for a persistent vector database — the chunking and cosine-similarity ideas are identical; the storage layer is what changes.

9.4.1 Install Dependencies

First, we will install the required libraries: `lancedb`, `sentence-transformers`, `pandas`, and `pydantic`.

```
!pip install lancedb sentence-transformers pandas pydantic
```

9.4.2 Set Up Database and Embedding Model

To create a local LanceDB vector database, simply connect to a local directory (like a `.db` file but for folders). We will also define our embedding model using the `sentence-transformers` registry within LanceDB.

9.4.3 Importing Libraries

We import LanceDB along with `pandas`, `PyArrow`, and the LanceDB embeddings registry. The Pydantic integration lets us define a typed schema that LanceDB uses to validate and auto-embed data.

```
import lancedb
import pandas as pd
import pyarrow as pa
from lancedb.embeddings import get_registry
from lancedb.pydantic import LanceModel, Vector
import os
from google.colab import userdata
```

9.4.4 Connecting to the Database and Selecting an Embedding Model

LanceDB stores data in a local directory, so there is no server to manage. We select the `all-MiniLM-L6-v2` sentence-transformer as our embedding model; it produces 384-dimensional vectors and offers a good balance of speed and quality.

```
# Connects to a local directory (like a .db file but for folders)
db = lancedb.connect("./lancedb_is_research")

# Select the embedding model
registry = get_registry().get("sentence-transformers")
model = registry.create(name="all-MiniLM-L6-v2")
```

9.4.5 Setting the Hugging Face Token

While optional today, providing a Hugging Face token avoids rate limits and ensures reliable access to sentence-transformer model weights.

```
### Creating and using the HUGGINGFACE_TOKEN is optional (as of now in colab), but I
would recommend that you do it, as it is used whenever SENTENCE TRANSFORMERS
models will be called.
os.environ["HF_TOKEN"] = userdata.get('HUGGINGFACE_TOKEN')
```

9.4.6 Define the Schema

In LanceDB, we define the schema to specify how the data should be structured and embedded. Here, we tell LanceDB to automatically embed the 'Abstract' column.

9.4.7 Loading and Previewing the Data

We load the ISResearch dataset directly from GitHub. Each row contains a paper's title, abstract, publication year, journal, and URL. The abstracts will be embedded and stored as vectors.

```
class Papers(LanceModel):
    id: int
    Year: int
    Title: str
    Abstract: str = model.SourceField()
    URL: str
    JournalFN: str
    # The vector field is automatically populated by the model above
    vector: Vector(model.ndims()) = model.VectorField()
```

9.4.8 Creating the Table and Inserting Data

When we call `table.add(df)`, LanceDB automatically generates embedding vectors for every abstract using the model we registered in the schema. This single line handles both embedding and storage, which keeps the pipeline simple.

9.4.9 Prepare and Load Data

Load our existing dataset, `ISResearch.csv`, into a Pandas DataFrame. Then, create the table and add the data. LanceDB handles the embedding generation automatically during the insertion process.

```
# Load your CSV (ensure ISResearch.csv is in your directory)
df = pd.read_csv("https://raw.githubusercontent.com/KarAnalytics/datasets/refs/heads/master/ISResearch.csv")

df.head()
```

9.4.10 Running a Semantic Search

We pass a natural-language query and LanceDB embeds it with the same model, then finds the closest vectors using approximate nearest-neighbor search. The results come back as a pandas DataFrame sorted by distance, making downstream analysis straightforward.

```
# Create the table and add the data
# LanceDB handles the embedding generation during the 'add' process
table = db.create_table("ISResearch", schema=Papers, mode="overwrite")
table.add(df)

print(f"Migration Complete. Total papers in DB: {len(table)}")
```

9.4.11 Semantic Search

Now we can perform semantic searches. LanceDB seamlessly embeds the query automatically and returns a Pandas DataFrame for easy viewing.

```
query = "Which papers mention blockchain or decentralized systems?"

# We search using raw text; LanceDB embeds the query automatically
results = table.search(query).limit(5).to_pandas()

print("\n--- Top 5 Semantic Search Results ---")
print(results[['Title', 'Year', '_distance']])
```

```
# Optional: To show the URL of the top result
print(f"\nTop Match URL: {results.iloc[0]['URL']}")
```

9.4.12 Key takeaways

- **LanceDB** is serverless and embedded – it stores vectors in a local directory, so there is no database server to run or manage.
 - **Pydantic schemas** with `SourceField` and `VectorField` let LanceDB auto-embed a text column on insert, collapsing embedding and storage into a single `table.add()` call.
 - **Raw-text queries** are embedded with the same registered model at search time, so you never have to hand-embed queries or maintain a separate inference path.
 - **Results as pandas DataFrames** (with a `_distance` column) make it trivial to sort, filter, and join semantic search output with the rest of an analytics workflow.
 - **all-MiniLM-L6-v2** produces 384-dim vectors and hits a sweet spot of speed versus quality for demos over a few thousand documents.
-

9.4.13 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~3 minutes

```
https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/LanceDB_vectorDB.ipynb
```

9.5 Pinecone: A Managed Vector Database

Vectors, the output data format of Neural Network models, can effectively encode information and serve a pivotal role in AI applications such as knowledge base, semantic search, Retrieval Augmented Generation (RAG) and more.

Pinecone is a managed, cloud-native vector database designed for high performance and scalability. In this guide, we will walk you through how to use Pinecone to store and search vectors in the cloud.

Where Chapter 8's from-scratch notebook kept everything in a Pandas DataFrame, Pinecone pushes the vectors to a managed cloud index — same retrieval logic, very different operational footprint (API keys, regions, per-request latency).

9.5.1 Install Dependencies

First, we will install the required libraries: `pinecone-client`, `sentence-transformers`, `pandas`, and `python-dotenv`.

```
!pip install pinecone sentence-transformers pandas python-dotenv
```

9.5.2 Initialize Pinecone Client

To use Pinecone, you need an API key from the Pinecone console. Create yet another account in Pinecone and generate an API key that can be added to the google colab secrets.

9.5.3 Importing Pinecone and Environment Setup

We import the Pinecone client along with `dotenv` for managing environment variables. Keeping the API key in a secret store rather than in code is essential for security.

```
import os
from dotenv import load_dotenv
from pinecone import Pinecone, ServerlessSpec
from google.colab import userdata
```

9.5.4 Authenticating with Pinecone

The Pinecone API key is loaded from Colab secrets and passed to the client constructor. We also set an optional Hugging Face token for the sentence-transformer model used later.

```
os.environ["PINECONE_API_KEY"] = userdata.get('PINECONE_KEY')

### Creating and using the HUGGINGFACE_TOKEN is optional (as of now in colab), but I
would recommend that you do it, as it is used whenever SENTENCE TRANSFORMERS models
will be called.
os.environ["HF_TOKEN"] = userdata.get('HUGGINGFACE_TOKEN')

# Initialize the Pinecone client
pc = Pinecone(api_key=os.environ["PINECONE_API_KEY"])
```

9.5.5 Set Up Embedding Model

We will use `sentence-transformers` to embed the text data. This model generates 384-dimensional embeddings.

9.5.6 Loading the Embedding Model

We use `all-MiniLM-L6-v2` from `sentence-transformers`, the same model used in the LanceDB example. Consistency in the embedding model is important because vectors from different models are not comparable.

```
from sentence_transformers import SentenceTransformer

# Initialize the embedding model
model = SentenceTransformer('all-MiniLM-L6-v2')
```

9.5.7 Create a Pinecone Index

We need an index in Pinecone to store our vectors. The dimension parameter must match the size of embeddings generated by our model (384-dimensions).

9.5.8 Creating a Pinecone Index

An index in Pinecone is the container for your vectors. The dimension must match the embedding model output (384), and we choose cosine similarity as the distance metric. The serverless spec means Pinecone manages all infrastructure for you.

```
index_name = "is-research"

# Check if the index exists, and if not, create it
if index_name not in pc.list_indexes().names():
    pc.create_index(
        name=index_name,
        dimension=384, # Match the 'all-MiniLM-L6-v2' output dimension
        metric='cosine',
        spec=ServerlessSpec(
            cloud='aws',
            region='us-east-1'
        )
    )

# Connect to the index
index = pc.Index(index_name)
```

9.5.9 Prepare and Load Data

Load our existing dataset, `ISResearch.csv`, into a Pandas DataFrame. Since Pinecone operates on the cloud, we will embed the abstracts locally and push batches of `(id, vector, metadata)` directly to the index.

9.5.10 Embedding and Upserting Data

Unlike LanceDB, Pinecone requires you to generate embeddings yourself before uploading. We encode all abstracts locally, package each vector with its metadata, and upsert in batches of 100 to stay within payload limits.

```
import pandas as pd

# Load your CSV
df = pd.read_csv("https://raw.githubusercontent.com/KarAnalytics/datasets/refs/heads/master/ISResearch.csv")

df.head()

# Generate embeddings for all abstracts
abstracts = df['Abstract'].tolist()
embeddings = model.encode(abstracts)

# Prepare data in the format Pinecone requires: a list of tuples (id, vector, metadata_dict)
vectors_to_upsert = []
for i, row in df.iterrows():
    vectors_to_upsert.append({
        "id": str(row['id']),
```

```

        "values": embeddings[i].tolist(),
        "metadata": {
            "Title": row['Title'],
            "Year": row['Year'],
            "URL": row['URL'],
            "Journal": row['JournalFN']
        }
    })

# Upsert (insert or update) the vectors into Pinecone in batches to avoid payload
limits
batch_size = 100
for i in range(0, len(vectors_to_upsert), batch_size):
    batch = vectors_to_upsert[i:i + batch_size]
    index.upsert(vectors=batch)

print(f"Migration Complete. Upserted {len(vectors_to_upsert)} vectors to Pinecone.")

```

9.5.11 Querying the Index

To search, we embed the query with the same model and send the vector to Pinecone. The `top_k` parameter controls how many nearest neighbors to return, and `include_metadata=True` brings back the paper titles and URLs alongside the similarity scores.

9.5.12 Semantic Search

Now we can perform semantic searches. First, we generate an embedding for our search query and then we send it to Pinecone to retrieve the top matches along with their metadata.

```

query = "Which papers mention blockchain or decentralized systems?"

# Embed the query text
query_vector = model.encode([query])[0].tolist()

# Search the Pinecone index
results = index.query(
    vector=query_vector,
    top_k=5,
    include_metadata=True # Ensure we retrieve the Title, Year, and URL
)

print("\n--- Top 5 Semantic Search Results ---")
for match in results['matches']:
    metadata = match['metadata']
    print(f"Score: {match['score']:.4f} | Year: {metadata.get('Year')} | Title: {metadata.get('Title')}")

# Optional: To show the URL of the top result
top_match = results['matches'][0]['metadata']
print(f"\nTop Match URL: {top_match.get('URL')}")

```

9.5.13 Key takeaways

- **Pinecone** is a managed cloud vector database – you get an API key, create an index, and Pinecone handles infrastructure, scaling, and availability for you.
- **Index dimension must match** the embedding model output (384 for `all-MiniLM-L6-v2`), and the metric (cosine here) is fixed at index creation time.
- **Manual embedding** is required – unlike LanceDB’s auto-embed, Pinecone expects you to encode text locally and upsert `(id, vector, metadata)` tuples in batches.
- **Metadata** travels with each vector and is returned at query time via `include_metadata=True`, enabling filter-plus-semantic hybrid queries without a second lookup.

- **Serverless spec** on AWS means no servers to provision, but the tradeoff is API keys, network latency, and per-request cost versus a local embedded store.
-

9.5.14 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~5 minutes (requires Pinecone API key)

```
https://colab.research.google.com/github/KarAnalytics/code\_demos/blob/main/Pinecone\_vectorDB.ipynb
```


Chapter 10

RAG with Structured Data

RAG isn't limited to text documents. When your data lives in a **SQL database** or a **graph database**, the “retrieval” step becomes a query (SQL or Cypher) rather than a vector search.

This chapter covers two variants:

- **DBMS RAG** — the LLM generates SQL queries against a SQLite database, executes them, and answers based on the results
- **Graph RAG** — the LLM generates Cypher queries against a Kùzu graph database for relational/network data

Both follow the same pattern: retrieve evidence → stuff into prompt → generate answer. Only the retrieval mechanism changes.

10.1 RAG with a Local DBMS: Text-to-SQL using SQLite

What happens when the knowledge you need is not buried in documents, but neatly organized in database tables? In previous chapters we retrieved text chunks using vector similarity. Here we take a fundamentally different approach: instead of searching for *similar text*, we ask the LLM to write a *SQL query* that fetches the exact data we need. The database does the heavy lifting — joins, filters, aggregations — and the LLM simply translates English into SQL, then translates the results back into English.

How it works:

1. Build a SQLite database from the classic **Supplier-Parts** schema (DDL from a `.sql` file, data from an `.xlsx` file).
2. Extract the database schema as structured context — this is the “retrieval” step.
3. Given a natural-language question, the LLM generates a SQL query.
4. Execute the SQL and feed the results back to the LLM for a natural-language answer.

Learning goals:

- Understand how RAG applies beyond vector databases
- See how schema metadata serves as retrieval context
- Practice grounding LLM output on real query results
- Compare answers with and without database context

Provider setup: Uses `llm_cascade` for automatic LLM provider selection.

Store your LLM API keys in Colab Secrets (or a local `.env` file). Supported keys: `GEMINI_API_KEY`, `OPENAI_API_KEY`, `GROQ_API_KEY`, `HF_TOKEN`, `COHERE_API_KEY`, `XAI_API_KEY`, `OLLAMA_API_KEY`, `OPENROUTER_API_KEY`.

If the text-RAG pipeline from Chapter 8’s `rag_first_principles.ipynb` is still fresh, keep it in mind as a contrast — there, retrieval meant fuzzy vector search over chunks; here retrieval is a precise SQL query, and the “chunks” are just the schema the LLM needs to write that query.

```
%pip install -q -U openpyxl git+https://github.com/KarAnalytics/llm_cascade.git
sentence-transformers
```

10.1.1 1) Imports and Provider Helpers

SQLite is built into Python — no extra database driver needed.

We use `openpyxl` to load data from the Excel file.

The provider helpers below are identical to the pattern used in the RAG countries notebook: Gemini first, Ollama Cloud fallback, keys from Colab Secrets or `.env`.

```
import os
import sqlite3
import re
from pathlib import Path

import openpyxl

from llm_cascade import get_cascade

llm = get_cascade()

def generate_text(prompt, system_prompt=None, **kwargs):
    response = llm.generate(prompt, system_prompt=system_prompt)
    return response.text, response.provider

def has_llm_provider():
    return True # llm_cascade auto-detects available providers
```

```
print("Provider ready (llm_cascade)")
```

10.1.2 2) Build the SQLite Database from External Files

We load the **Supplier-Parts** database from two companion files hosted in a public GitHub repo:

File	Source	Purpose
SUPPLIER_PARTS_DDL.sql	GitHub	CREATE TABLE statements (schema definition)
SUPPLIER_PARTS.xlsx	GitHub	Data for each table (one sheet per table)

The next cell downloads both files automatically. If you want to try your own database, uncomment Option B in the cell to upload local `.sql` and `.xlsx` files instead.

The demo database has three tables:

Table	Purpose
SUPPLIERS	Supplier ID, name, status, and city
PARTS	Part ID, name, color, weight, and city
SHIPMENTS	Which supplier ships which part, with quantity

This is the classic Supplier-Parts schema used in database textbooks — small enough to fit in a single prompt, but rich enough to demonstrate joins, aggregations, and foreign-key relationships.

```
import urllib.request

DB_PATH = 'supplier_parts.db'
DDL_FILE = 'SUPPLIER_PARTS_DDL.sql'
XLSX_FILE = 'SUPPLIER_PARTS.xlsx'

# =====
# Option A (default): download the demo files from GitHub
# =====
# This is the default so the notebook runs on a fresh Colab without needing
# any file uploads. Students can swap in their own DDL and data files by
# using Option B below instead.

DDL_URL = 'https://raw.githubusercontent.com/KarAnalytics/datasets/master/SUPPLIER_PARTS_DDL.sql'
XLSX_URL = 'https://github.com/KarAnalytics/datasets/raw/master/SUPPLIER_PARTS.xlsx'

print(f'Downloading {DDL_FILE}...')
urllib.request.urlretrieve(DDL_URL, DDL_FILE)
print(f'Downloading {XLSX_FILE}...')
urllib.request.urlretrieve(XLSX_URL, XLSX_FILE)
print('Downloaded both files.')
print()

# =====
# Option B: upload your own files instead
# =====
# Uncomment the lines below if you want to try a different database. Upload
# your .sql (DDL) file and .xlsx (data) file to the Colab workspace first,
# then update DDL_FILE / XLSX_FILE above to match your filenames.
#
# from google.colab import files
# uploaded = files.upload() # a file picker will appear
# DDL_FILE = list(uploaded.keys())[0] # e.g., 'my_schema.sql'
# XLSX_FILE = list(uploaded.keys())[1] # e.g., 'my_data.xlsx'
```

```

# --- Build the SQLite database from the files ---
if Path(DB_PATH).exists():
    Path(DB_PATH).unlink() # remove stale DB from previous runs

conn = sqlite3.connect(DB_PATH)
cur = conn.cursor()

# 1) Execute DDL from the .sql file
ddl_text = Path(DDL_FILE).read_text(encoding='utf-8')
cur.executescript(ddl_text)
print(f'Schema loaded from {DDL_FILE}')

# 2) Load data from Excel (one sheet per table)
wb = openpyxl.load_workbook(XLSX_FILE, read_only=True, data_only=True)

# Insertion order matters because of foreign-key constraints:
# SUPPLIERS and PARTS first, then SHIPMENTS.
table_order = ['SUPPLIERS', 'PARTS', 'SHIPMENTS']

for table_name in table_order:
    ws = wb[table_name]
    rows = list(ws.iter_rows(values_only=True))
    headers = rows[0] # first row is column names
    data_rows = rows[1:] # remaining rows are data
    placeholders = ', '.join('?' * len(headers))
    cur.executemany(
        f'INSERT INTO {table_name} VALUES ({placeholders})',
        data_rows,
    )
    print(f' {table_name}: {len(data_rows)} rows inserted')

wb.close()
conn.commit()
print(f'\nDatabase "{DB_PATH}" ready.')

```

10.1.3 3) Schema Extraction — the “Retrieval” in DBMS RAG

This is the conceptual heart of the notebook, so pause and think about it carefully. In document RAG, the retrieval step searches a vector store for text chunks similar to the question. In **DBMS RAG**, we do something very different: we retrieve the **database schema itself** — the `CREATE TABLE` statements that describe what tables exist, what columns they have, and how they relate to each other.

Why the schema and not the data? Because the schema is small and fixed-size regardless of how many rows the database has. A database with 10 rows and a database with 10 billion rows have the same schema. By giving the LLM the schema (plus a few sample rows so it understands data formats), we equip it to write valid SQL queries that will fetch the exact data needed — without ever stuffing millions of rows into a prompt.

Think of it this way: in document RAG, retrieval *is* the answer (the chunks contain the evidence). In DBMS RAG, retrieval gives the LLM a *map* of the data, and the LLM uses that map to write a SQL query that goes and gets the actual evidence.

```

def get_schema_context(db_path, sample_rows=3):
    """Extract CREATE TABLE statements and sample rows as a text block."""
    con = sqlite3.connect(db_path)
    cur = con.cursor()

    # Get all CREATE TABLE DDL
    cur.execute("SELECT name, sql FROM sqlite_master WHERE type='table' ORDER BY name")

```

```

tables = cur.fetchall()

parts = []
for table_name, ddl in tables:
    parts.append(f"-- Table: {table_name}")
    parts.append(ddl + ";")

    # Sample rows
    cur.execute(f"SELECT * FROM [{table_name}] LIMIT {int(sample_rows)}")
    rows = cur.fetchall()
    col_names = [desc[0] for desc in cur.description]
    parts.append(f"-- Sample rows ({table_name}): columns = {col_names}")
    for row in rows:
        parts.append(f"-- {row}")
    parts.append("")

con.close()
return "\n".join(parts)

schema_context = get_schema_context(DB_PATH)
print(schema_context)

```

10.1.4 4) Text-to-SQL Generation

This is the core RAG step: the LLM receives the schema context and the user's natural-language question, then produces a SQL query. Notice the two-phase design: the first LLM call *generates* SQL (it does not answer the question directly), and we will *execute* that SQL against the real database before asking the LLM to compose a final answer. This separation is key — the LLM never sees the raw data during SQL generation, only the schema. It is essentially writing a recipe for how to find the answer.

Safety note: We only allow `SELECT` statements. The `extract_sql` function strips markdown fences and validates that the query starts with `SELECT` before execution, preventing any accidental `DROP TABLE` or `UPDATE` commands. In a production system you would also run queries with a read-only database connection.

```

SQL_SYSTEM_PROMPT = (
    "You are an expert SQL analyst. Given the database schema below, write a "
    "single SQLite-compatible SELECT query that answers the user's question.\n"
    "Return ONLY the SQL query – no explanation, no markdown fences, no comments.\n"
    "If the question cannot be answered from the schema, reply with: SELECT "
    "'NOT_ANSWERABLE' AS result;"
)

def extract_sql(raw_text):
    """Strip markdown fences or extra prose, returning just the SQL statement."""
    match = re.search(r"```(?:sql)?\s*\n?(.*?)```", raw_text, re.DOTALL |
re.IGNORECASE)
    if match:
        return match.group(1).strip()
    return raw_text.strip().rstrip(";")

def generate_sql(question, schema, **kwargs):
    """Ask the LLM to produce a SELECT query for the given question."""
    prompt = f"DATABASE SCHEMA:\n{schema}\n\nQUESTION:\n{question}"
    raw_response, provider = generate_text(
        prompt=prompt,
        system_prompt=SQL_SYSTEM_PROMPT,
    )
    sql = extract_sql(raw_response)

```

```

        return sql, provider

def execute_sql(db_path, sql):
    """Run a read-only SQL query and return column names + rows."""
    normalized = sql.strip().upper()
    if not normalized.startswith("SELECT"):
        return [], [], "Blocked: only SELECT queries are allowed."

    con = sqlite3.connect(db_path)
    try:
        cur = con.execute(sql)
        columns = [desc[0] for desc in cur.description]
        rows = cur.fetchall()
        return columns, rows, None
    except Exception as exc:
        return [], [], str(exc)
    finally:
        con.close()

print("Text-to-SQL functions ready.")

```

10.1.5 5) Answer Generation – Grounding on Query Results

After executing the SQL, we feed the actual query results back to the LLM and ask it to compose a natural-language answer. This is the second LLM call, and it follows the exact same pattern as document RAG: retrieved evidence goes into the prompt, and the model must base its answer on that evidence alone. The difference is that in document RAG the evidence was text chunks found by vector search; here the evidence is database rows returned by a SQL query.

This two-call architecture is what makes DBMS RAG so powerful for numerical questions. When you ask “What is the total quantity of all shipments?”, the LLM does not have to add numbers in its head (which it is bad at). Instead, it writes `SELECT SUM(Quantity) FROM SHIPMENTS`, the database computes the exact answer, and the LLM simply wraps that number in a sentence. The math is always right because SQL does the math.

```

ANSWER_SYSTEM_PROMPT = (
    "You are a helpful business analyst. Use ONLY the provided SQL results to answer "
    "the user's question. If the results are empty or the query returned "
    "NOT_ANSWERABLE, "
    "say you cannot determine the answer from the available data."
)

def answer_with_dbms_rag(question, db_path=DB_PATH, **kwargs):
    """Full DBMS-RAG pipeline: schema retrieval -> SQL generation -> execution ->
    answer."""
    schema = get_schema_context(db_path)

    print(f" [Step 1] Schema retrieved ({len(schema)} chars)")

    sql, sql_provider = generate_sql(question, schema)
    print(f" [Step 2] SQL generated (provider: {sql_provider})")
    print(f" {sql}")

    columns, rows, error = execute_sql(db_path, sql)
    if error:
        print(f" [Step 3] SQL execution error: {error}")
        return f"SQL failed: {error}", sql, sql_provider

    print(f" [Step 3] SQL executed – {len(rows)} row(s) returned")

```

```

result_text = f"Columns: {columns}\n"
for row in rows:
    result_text += f" {row}\n"

prompt = (
    f"QUESTION:\n{question}\n\n"
    f"SQL QUERY USED:\n{sql}\n\n"
    f"QUERY RESULTS:\n{result_text}"
)
answer, ans_provider = generate_text(
    prompt=prompt,
    system_prompt=ANSWER_SYSTEM_PROMPT,
)
print(f" [Step 4] Answer generated (provider: {ans_provider})")

return answer, sql, sql_provider

def answer_without_rag(question, **kwargs):
    """Direct LLM answer with no database context."""
    answer, provider = generate_text(prompt=question)
    return answer, provider

print("RAG answer pipeline ready.")

```

10.1.6 6) Run End-to-End Examples

Time to see the full pipeline in action. We test several questions and compare two approaches side by side:

- With DBMS RAG: schema retrieval, SQL generation, execution, and a grounded answer based on real query results.
- Without RAG: the LLM answers the question directly with no database access.

Pay close attention to the “without RAG” answers. The model has no way to know that total shipment quantity is 5,300 — it has never seen this particular database. So it either asks for data it does not have (the honest response) or invents a plausible-sounding but completely wrong number (hallucination). The RAG version, by contrast, always gives the exact answer because the SQL engine computed it.

```

questions = [
    "What is the total quantity of all shipments?",
    #"Which supplier has shipped the most parts? Show their name and total quantity.",
    #"List all parts supplied by supplier S1, with quantities.",
    #"How many distinct parts are shipped from London?",
    #"What is the average shipment quantity per supplier located in Paris?",
]

def preview(text, max_len=800):
    text = text or ""
    return text[:max_len] + ("..." if len(text) > max_len else "")

if not has_llm_provider():
    print("Error: No LLM API key configured. Set at least one API key in Colab Secrets.")
else:
    for i, q in enumerate(questions, start=1):
        print("\n" + "=" * 80)
        print(f"Q{i}. {q}\n")

```

```

try:
    answer_rag, sql_used, prov = answer_with_dbms_rag(q)
    print("\n [WITH DBMS RAG]")
    print(preview(answer_rag))
except Exception as e:
    print(f" RAG error: {e}")

try:
    answer_direct, prov_direct = answer_without_rag(q)
    print(f"\n [WITHOUT RAG] (provider: {prov_direct})")
    print(preview(answer_direct))
except Exception as e:
    print(f" Direct error: {e}")

```

10.1.6.1 Checkpoint: Reflection Questions

1. Compare the RAG and non-RAG answers. Which ones show hallucinated numbers?
2. What happens if you ask a question about data that is not in the database (e.g., supplier revenue or profit)?
3. What are the risks of letting an LLM generate SQL against a production database?
4. How does this DBMS-RAG pattern compare to the vector-store RAG we used with ChromaDB?

10.1.7 7) How DBMS RAG Works – and How It Compares to Document RAG

This notebook implements RAG using a **SQL database as the knowledge source** instead of chunked documents. The pipeline has **two retrieval phases**, both appended to prompts before going to the LLM – so it's structurally similar to document RAG, but with a twist.

10.1.7.1 The pipeline (per question)

```

Question
|
+--> Step 1: Retrieve SCHEMA (not data)
|     get_schema_context() pulls CREATE TABLE statements
|     + 3 sample rows per table
|
+--> Step 2: LLM generates SQL
|     Prompt = schema + question -> LLM -> SELECT ...
|
+--> Step 3: Execute the SQL on the database
|     Runs against SQLite, returns columns + rows
|
+--> Step 4: LLM writes the final answer
|     Prompt = question + SQL + query results -> LLM -> natural language answer

```

Two LLM calls per question:

- **Text-to-SQL** (`generate_sql`) – schema goes in, SQL query comes out
- **Answer generation** (inside `answer_with_dbms_rag`) – query results go in, natural-language answer comes out

10.1.7.2 Is it different from document RAG?

Same core principle, different retrieval mechanism. Both patterns follow: retrieve -> stuff into prompt -> generate answer . The differences:

	Document RAG	DBMS RAG (this notebook)
What's retrieved first?	Text chunks similar to the question	The database schema (DDL + sample rows)
How is retrieval done?	Vector similarity (cosine) on embeddings	Just dump all CREATE TABLE statements

	Document RAG	DBMS RAG (this notebook)
Is question content matched against stored data?	Yes, directly (embedding search)	No — retrieval is schema, not data
When does the actual data enter the prompt?	Immediately (chunks = data)	In a second LLM call, after SQL executes
Scales with data size?	Gets slower (more chunks to search)	No — schema is fixed size even if the DB has billions of rows
Handles aggregations?	Poorly (LLM must do math in its head)	Perfectly (SQL SUM , COUNT , AVG are exact)

10.1.7.3 The key insight

In document RAG, retrieval and grounding happen in the same step — the chunks you retrieve ARE the evidence. Whatever the vector search brings back is what the LLM sees.

In DBMS RAG, retrieval happens twice:

- First retrieval = schema only (what tables exist, what columns, what types)
- Second “retrieval” = the SQL query results (the actual evidence, fetched dynamically)

This is why it’s so much better for structured data:

- Document RAG has to hope the right chunks got retrieved. If your question is “total sales in Q3,” it retrieves chunks that mention Q3 sales — but the LLM has to add them up, and may miss rows or hallucinate.
- DBMS RAG doesn’t hope. It asks the LLM to write `SELECT SUM(amount) WHERE quarter='Q3'`, runs it, and shows the exact answer.

10.1.7.4 What does `get_schema_context()` actually retrieve?

Two things:

- `CREATE TABLE` statements from SQLite’s `sqlite_master` metadata table. This is cheap — it returns the same few thousand characters regardless of whether the tables have 100 rows or 100 million.
- 3 sample rows per table. This gives the LLM a feel for the data format (e.g., “oh, dates are stored as strings like `2025-Q1`, supplier IDs are integers, colors are lowercase”). These sample rows are the closest thing to “chunks” in DBMS RAG.

10.1.7.5 The second LLM call — where “actual retrieval” happens

After `execute_sql()` runs and returns rows, the notebook stuffs those rows INTO the prompt for the second LLM call:

```
prompt = (
    f'QUESTION:\n{question}\n\n'
    f'SQL QUERY USED:\n{sql}\n\n'
    f'QUERY RESULTS:\n{result_text}'
)
```

This is exactly the same pattern as document RAG — retrieved evidence + question → LLM → answer. The only difference is that the “retrieval” was done via a SQL query instead of a vector search.

10.1.7.6 TL;DR

Is DBMS RAG different from document RAG? Yes in how retrieval is done, no in the overall pattern.

- Document RAG: `vector_search(question) -> chunks -> prompt -> LLM -> answer`
- DBMS RAG: `read_schema() -> prompt -> LLM -> SQL -> execute SQL -> results -> prompt -> LLM -> answer`

Both pipelines ground the LLM’s final answer on retrieved evidence. DBMS RAG just retrieves via SQL execution instead of embedding similarity — which makes it perfect for structured/numeric data where

aggregations and joins matter, and completely impractical for unstructured text where there's no schema to query.

10.1.7.7 Where this notebook sits in the RAG family

Each type of RAG uses a different retrieval mechanism but the same *pattern* (retrieve evidence → stuff into prompt → ground the answer):

Type	Best for	Retrieval mechanism	Example notebook
Document RAG	Unstructured text (articles, PDFs)	Vector similarity	LlamaIndex_RAG , LangChain_demo
DBMS RAG (this one)	Structured tabular data	Text-to-SQL + execution	DBMS_RAG_SQLite
Graph RAG	Relational / network data	Text-to-Cypher + execution	GRAPH_RAG_Trade
Image / Video RAG	Visual content	CLIP embedding similarity	Image_RAG , Video_RAG

10.1.7.8 Key takeaways

- RAG is **not limited to vector databases** — any structured data source can serve as retrieved context.
- In DBMS RAG, the **schema** is the retrieval context and **SQL execution** is the grounding mechanism.
- Grounding the final answer on actual query results prevents hallucination of specific numbers.
- Loading schema from `.sql` files and data from `.xlsx` files mirrors real-world enterprise workflows.

10.1.7.9 Exercises

- Add a new table (e.g., `PROJECTS` with a project-supplier-part relationship) and ask questions that require joins across it.
- Try asking ambiguous questions and observe how the LLM interprets them.
- Modify `SQL_SYSTEM_PROMPT` to request query explanations alongside the SQL.
- Compare the quality of SQL generated by different models in `llm_cascade` (Gemini vs. Groq vs. Ollama).
- Swap in a completely different database by uncommenting Option B in cell 5 and uploading your own `.sql` and `.xlsx` files.
- Discuss: when would you prefer DBMS RAG over vector-store RAG for a business application?

10.1.8 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. (If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/DBMS_RAG_SQLite.ipynb

10.2 Graph RAG: Retrieval-Augmented Generation with Graph Databases

Some data is inherently about *relationships*. Who trades with whom? Which countries are connected through supply chains? What is the shortest path between two trading partners? Relational databases can answer these questions, but the SQL gets ugly fast — you need self-joins, recursive CTEs, and complex subqueries. Graph databases are purpose-built for this kind of data: they store entities as **nodes** and connections as **edges**, and they answer relationship queries in a fraction of the code.

In this notebook we build a **Graph RAG** pipeline using **Kuzu**, a lightweight embedded graph database. The pattern mirrors what we did with SQL RAG — extract the schema, let the LLM write a query, execute it, ground the answer on results — but the query language is **Cypher** instead of SQL, and the data model is a graph instead of tables. The result is a system that excels at questions like “Which African countries export to both the USA and China?” — queries that would require multiple joins in SQL but are a single, readable pattern match in Cypher.

How it works:

1. Build a graph database from international trade data (countries as nodes, trade flows as edges).
2. Extract the **graph schema** (node labels, relationship types, properties) as LLM context.
3. The LLM generates a **Cypher query** from a natural-language question.
4. Execute the Cypher query and feed results back to the LLM for a natural-language answer.
5. **Visualize** the retrieved subgraph to see the data structure.

Learning goals:

- Understand why graph databases are better than SQL for relationship-heavy data
- Learn the basics of **Cypher**, the graph query language
- See how graph schema serves the same role as SQL DDL in DBMS RAG
- Experience the limitations of LLM-generated Cypher (and how to debug it)

Provider setup: Uses `llm_cascade` for automatic LLM provider selection.

Store your LLM API keys in Colab Secrets (or a local `.env` file). Supported keys: `GEMINI_API_KEY`, `OPENAI_API_KEY`, `GROQ_API_KEY`, `HF_TOKEN`, `COHERE_API_KEY`, `XAI_API_KEY`, `OLLAMA_API_KEY`, `OPENROUTER_API_KEY`.

Keep the vector-RAG notebooks from Chapters 8–9 and the SQL RAG demo in `dbms_rag_sqlite.ipynb` in mind as you read — Graph RAG is the third point in the same triangle, swapping vector search or SQL execution for Cypher pattern matching.

```
!pip install -q -U kuzu networkx matplotlib pandas git+https://github.com/KarAnalytics/llm_cascade.git sentence-transformers
```

10.2.1 1) Imports and Provider Helpers

We use **Kuzu** as our embedded graph database — it runs in-process (no server to install) and supports the **Cypher** query language, the same language used by Neo4j and other production graph databases. For visualization we use **NetworkX** and **Matplotlib** to render subgraphs as node-and-edge diagrams. The `llm_cascade` library handles LLM provider selection automatically, just as it did in the SQL RAG notebook.

```
import os
import re
from pathlib import Path

import kuzu
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt

from llm_cascade import get_cascade

llm = get_cascade()
```

```
def generate_text(prompt, system_prompt=None, **kwargs):
    response = llm.generate(prompt, system_prompt=system_prompt)
    return response.text, response.provider

def has_llm_provider():
    return True # llm_cascade auto-detects available providers

print("Provider ready (llm_cascade)")
```

10.2.2 2) Build the Graph Database

International trade data is a natural fit for a graph. Each **country** is a node with properties like name, continent, and population. Each **trade flow** is a directed edge from exporter to importer, carrying properties like the year and dollar value of exports. This structure makes it trivial to ask questions like “Who does Germany export to?” (follow outgoing edges) or “Which countries are trade hubs connecting two continents?” (find nodes with many incoming and outgoing edges).

We load the data from two CSV files hosted on GitHub:

File	Role	Columns
country_names_simple.csv	Node Data	ID, Country, Continent, Population
edges_trade_2014to17_simple.csv	Relationship Data	Year, Source, Target, Type, export_val

The code below downloads both files, creates the Kuzu schema (one node table and one relationship table), and loads the data. Notice the parallel to how we built the SQLite database in the previous notebook — the steps are conceptually identical (define schema, load data), just expressed in graph terms instead of relational terms.

```
import shutil
import os
from pathlib import Path
import pandas as pd # Ensure pandas is imported

DB_FOLDER = "trade_db"
COUNTRY_CSV_URL = "https://raw.githubusercontent.com/KarAnalytics/InternationalTrade2017/main/country_names_simple.csv"
TRADE_CSV_URL = "https://raw.githubusercontent.com/KarAnalytics/InternationalTrade2017/main/edges_trade_2014to17_simple.csv"

LOCAL_COUNTRY_CSV = "country_names_simple.csv"
LOCAL_TRADE_CSV = "edges_trade_2014to17_simple.csv"

# 1. Clean up existing DB for a fresh run
db_path = Path(DB_FOLDER)

# Close any existing connections before attempting to remove files
# This helps release file locks from previous runs
if 'conn' in globals() and conn is not None: # Use globals() to check for existence across cell executions
    try:
        conn.close()
        print("Closed existing Kùzu connection.")
    except Exception as e:
        print(f"Error closing connection: {e}")
if 'db' in globals() and db is not None:
    try:
        db.close()
        print("Closed existing Kùzu database.")
```

```

except Exception as e:
    print(f"Error closing database: {e}")

if db_path.exists():
    if db_path.is_dir():
        print(f"Removing existing database directory: {db_path}")
        shutil.rmtree(db_path)
    elif db_path.is_file():
        print(f"Removing existing database file: {db_path}")
        os.remove(db_path)

# Also explicitly clean up the .wal file if it exists, as Kùzu sometimes leaves it
# outside the main folder.
wal_file_path = Path(str(db_path) + ".wal")
if wal_file_path.is_file():
    print(f"Removing existing WAL file: {wal_file_path}")
    os.remove(wal_file_path)

# 2. Initialize Kùzu
db = kuzu.Database(DB_FOLDER)
conn = kuzu.Connection(db)

# 3. Create Schema
print("Defining schema...")
conn.execute("CREATE NODE TABLE Country(id STRING, name STRING, continent STRING,
population INT64, PRIMARY KEY (id))")
conn.execute("CREATE REL TABLE TRADE(FROM Country TO Country, year INT64, type
STRING, export_val DOUBLE)")

# 4. Load Node Data
print(f"Downloading and loading nodes from {COUNTRY_CSV_URL}...")
# Download CSV locally first as Kùzu COPY does not support URLs directly
df_country = pd.read_csv(COUNTRY_CSV_URL)
df_country.to_csv(LOCAL_COUNTRY_CSV, index=False)
conn.execute(f"COPY Country FROM \"{LOCAL_COUNTRY_CSV}\" (HEADER=TRUE)")
Path(LOCAL_COUNTRY_CSV).unlink() # Clean up local file

# 5. Load Relationship Data
# Note: Our CSV column order is (Year, Source, Target, Type, export_val).
# Kùzu's COPY REL expects the FROM/TO columns to match the relationship definition.
# We'll use pandas to rearrange the columns to (Source, Target, Year, Type,
export_val)
# so Kùzu can match 'Source' to 'FROM' and 'Target' to 'TO'.
print(f"Downloading and loading relationships from {TRADE_CSV_URL}...")
df_trade = pd.read_csv(TRADE_CSV_URL)
cols = ['Source', 'Target', 'Year', 'Type', 'export_val']
df_trade[cols].to_csv(LOCAL_TRADE_CSV, index=False)
conn.execute(f"COPY TRADE FROM \"{LOCAL_TRADE_CSV}\" (HEADER=TRUE)")
Path(LOCAL_TRADE_CSV).unlink() # Clean up local file

print("\nGraph Database ready.")

```

10.2.3 3) Graph Schema Extraction — the “Retrieval” Step

Just as SQL RAG retrieves `CREATE TABLE` statements, Graph RAG retrieves the **graph schema**: what node labels exist, what properties they have, and what relationship types connect them. This schema plays the exact same role as the DDL in DBMS RAG — it is a compact description of the data’s structure that the LLM needs in order to write valid queries.

The key difference is representational. A SQL schema says “there is a table called SHIPMENTS with columns SID, PID, and Quantity.” A graph schema says “there is a relationship type TRADE from Country to Country with properties year, type, and export_val.” Both convey the same kind of information — what entities exist and how they connect — but the graph schema makes the connectivity explicit. The LLM reads this schema

and learns: “Ah, countries are connected by TRADE edges, each edge has a year and a dollar value, and I can traverse these edges with Cypher patterns.”

```
def get_graph_schema(conn):
    """Extract graph schema for LLM context."""
    schema_parts = ["GRAPH SCHEMA:"]

    # Nodes
    schema_parts.append("Nodes:")
    nodes = conn.execute("CALL SHOW_TABLES() RETURN *").get_as_df()
    node_tables = nodes[nodes['type'] == 'NODE']['name'].tolist()
    for nt in node_tables:
        props = conn.execute(f"CALL TABLE_INFO('{nt}') RETURN *").get_as_df()
        prop_desc = ", ".join([f"{r['name']} ({r['type']})" for _, r in
                                props.iterrows()])
        schema_parts.append(f"  - Label: {nt} | Properties: {prop_desc}")

    # Relationships
    schema_parts.append("Relationships:")
    rels = nodes[nodes['type'] == 'REL']['name'].tolist()
    for rt in rels:
        props = conn.execute(f"CALL TABLE_INFO('{rt}') RETURN *").get_as_df()
        prop_desc = ", ".join([f"{r['name']} ({r['type']})" for _, r in
                                props.iterrows()])
        # Get connectivity
        # Kùzu SHOW_TABLES output for REL tables doesn't directly show FROM/TO in
        # some versions,
        # but we know it's Country to Country here. For a dynamic demo, we'd query
        # it.
        schema_parts.append(f"  - Type: {rt} | Properties: {prop_desc} |
        Connections: (Country)-[:{rt}]->(Country)")

    return "\n".join(schema_parts)

schema_context = get_graph_schema(conn)
print(schema_context)
```

10.2.4 4) Text-to-Cypher RAG Pipeline

Now we build the heart of Graph RAG: translating natural-language questions into **Cypher** queries. Cypher is to graph databases what SQL is to relational databases — a declarative query language — but its syntax is designed around pattern matching. Instead of writing `SELECT ... FROM ... JOIN ... ON ...`, you write patterns like `(a:Country)-[:TRADE]->(b:Country)` that visually resemble the graph structure you are searching for. This makes Cypher particularly intuitive for relationship queries: “find all countries that Germany exports to” becomes `MATCH (deu:Country {id:'deu'})-[:TRADE]->(partner:Country) RETURN partner.name`.

The pipeline follows the same two-call pattern as SQL RAG: the first LLM call generates the Cypher query, we execute it against Kuzu, and the second LLM call turns the raw results into a natural-language answer. The `extract_cypher` helper strips any markdown formatting the LLM might add, and we cap results at 50 rows to keep the final prompt manageable.

```
CYPHER_SYSTEM_PROMPT = (
    "You are a Cypher query expert. Given the graph schema below, write a read-only\n"
    "\n"
    "Cypher query to answer the user's question.\n"
    "- Output ONLY the raw Cypher query code.\n"
    "- Do not include markdown fences, triple backticks, or any explanation.\n"
    "- Return a maximum of 50 relevant nodes or relationships to keep context\n"
    "manageable.\n"
    "- If the question cannot be answered, return: MATCH (n) RETURN 'NOT_ANSWERABLE'\n"
    "AS result LIMIT 1;"
)
```

```

)

def extract_cypher(raw_text):
    # Strip markdown fences if any
    match = re.search(r"```(?:cypher)?\s*\n?(.*?)```", raw_text, re.DOTALL |
re.IGNORECASE)
    if match: return match.group(1).strip()
    return raw_text.strip().rstrip(";")

def execute_cypher(conn, query):
    try:
        # Execute and get results
        res = conn.execute(query)
        cols = res.get_column_names()
        rows = []
        while res.has_next():
            rows.append(res.get_next())
        return cols, rows, None
    except Exception as e:
        return [], [], str(e)

def answer_with_graph_rag(question, conn):
    schema = get_graph_schema(conn)
    prompt = f"SCHEMA:\n{schema}\n\nQUESTION:\n{question}"

    # 1. Generate Cypher
    raw_cypher, prov_c = generate_text(prompt, system_prompt=CYPHER_SYSTEM_PROMPT)
    query = extract_cypher(raw_cypher)

    print(f" [Cypher Generated]: {query}")

    # 2. Execute
    cols, rows, error = execute_cypher(conn, query)
    if error: return f"Query Error: {error}", query

    # 3. Grounded Answer
    results_text = f"Columns: {cols}\nRows: {rows[:10]} (truncated)"
    ans_prompt = f"QUESTION: {question}\n\nCYPHER QUERY: {query}\n\nRESULTS:
{results_text}"
    answer, prov_a = generate_text(ans_prompt, system_prompt="Answer concisely based
ONLY on the graph results.")
    return answer, query

```

10.2.5 5) Graph Visualization

One of the most compelling advantages of graph databases is that their data has a natural visual representation. Nodes are circles, edges are arrows, and you can literally *see* the structure of your data. The function below queries the immediate trade partners of a given country and renders the result as a directed graph using NetworkX. Edge labels show export values in dollars, making it easy to spot which trade relationships are the most significant. This kind of visualization is much harder to produce from relational data, where the connections are implicit in foreign keys rather than explicit in the data model.

```

def visualize_subgraph(conn, target_country_id, depth=1):
    """Visualize immediate trade partners of a country."""
    query = f"""
MATCH (s:Country {{id: '{target_country_id}'}})-[r:TRADE]->(t:Country)
RETURN s.name, t.name, r.export_val, r.year
LIMIT 20
"""
    _, rows, _ = execute_cypher(conn, query)

    G = nx.DiGraph()

```



```

for row in rows:
    s_name, t_name, val, year = row
    G.add_edge(s_name, t_name, weight=val, label=f"{year}")

plt.figure(figsize=(10, 6))
pos = nx.spring_layout(G, k=0.5)
nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=2000,
font_size=10, arrowsize=20)

# Edge labels showing a bit of info
edge_labels = {(u, v): f'${d["weight"]:.0f}' for u, v, d in G.edges(data=True)}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=8)

plt.title(f"Trade Connections for {target_country_id} (Sample)")
plt.show()

```

10.2.6 6) Run Examples

Let us put the pipeline to work. We ask the LLM a question that requires traversing the graph — finding Germany's top trade partners by export value in 2014. The pipeline will generate a Cypher query with `ORDER BY` and `LIMIT` clauses, execute it, and return a grounded answer. We also visualize Germany's trade subgraph so you can see the network of connections firsthand.

```

if not has_llm_provider():
    print("Configuration Error: Please set API keys.")
else:
    q1 = "Who were the top 3 trade partners of Germany (deu) in 2014 by export value?"
    print(f"\nQuestion: {q1}")
    ans, cypher = answer_with_graph_rag(q1, conn)
    print(f"Answer: {ans}")

    print("\nVisualizing subgraph for Germany...")
    visualize_subgraph(conn, "deu")

```

10.2.7 When Graph RAG Gets It Wrong

Graph RAG is not foolproof — and this section demonstrates exactly where it breaks down. The LLM must generate syntactically valid Cypher *and* use the correct property values. If the LLM writes `{name: 'USA'}` when the database stores `'United States'`, the query returns zero results even though the data is there. This is a fundamental challenge of text-to-query RAG: the LLM is guessing at data values it has never seen. Including sample data in the schema context (as we did in SQL RAG) can help, but it does not guarantee correctness. The next two queries illustrate this problem.

```

q2 = "Find countries in Africa that exported to both the USA (usa) and China (chn) in 2017."
print(f"\nQuestion: {q2}")
ans, cypher = answer_with_graph_rag(q2, conn)
print(f"Answer: {ans}")

```

The query above returned zero results, claiming no African countries export to both the USA and China. That is clearly wrong — we know from common sense that many African nations trade with both superpowers. The problem is that the LLM used `{name: 'USA'}` and `{name: 'China'}` in the Cypher query, but the database stores these countries as `'United States'` and `'China'` (or uses ISO codes). A simple name mismatch caused the entire query to silently return nothing. This is a critical lesson: **zero results does not mean the answer is zero — it may mean the query is wrong.**

```

q3 = "Which countries in Africa imported from USA (usa) in 2014."
print(f"\nQuestion: {q3}")

```



```
ans, cypher = answer_with_graph_rag(q3, conn)
print(f"Answer: {ans}")
```

10.2.8 Debugging: Writing Cypher by Hand

The results above looked suspicious, so let us do what any good analyst would do: write the queries ourselves and see what the data actually says. The cells below use the correct country name ('United States' instead of 'USA') and confirm that the data is there — 49 African countries imported from the US in 2014, and 48 African countries exported to both the US and China in 2017. The LLM's Cypher was structurally correct but used the wrong property values, a common and instructive failure mode.

```
### Query: Which countries in Africa imported from USA (usa) in 2014.

# The Cypher query adjusted for your database schema:
# - Node labels are 'Country'
# - The property for the country name is 'name'
# - The relationship is 'TRADE'
query = """
MATCH (usa:Country)-[t:TRADE]->(af:Country)
WHERE usa.name = 'United States'
      AND af.continent = 'Africa'
      AND t.year = 2014
      AND t.export_val > 0
RETURN af.name AS countryName ;
"""

# Execute the query and return the results as a Pandas DataFrame
# This will display as a formatted table in the notebook output
results_df = conn.execute(query).get_as_df()
print(results_df.head())
print(results_df.shape)

### USA exports to almost all African countries each year, this is something I knew
in advance.
```

```
### QUERY: Find countries in Africa that exported to both the USA (usa) and China
(chn) in 2017.

# The Cypher query adjusted for your database schema:
# - Node labels are 'Country'
# - The property for the country name is 'name'
# - The relationship is 'TRADE'
query = """
MATCH (usa:Country {name: 'United States'}), (chn:Country {name: 'China'})
MATCH (af:Country {continent: 'Africa'})-[t1:TRADE {year: 2017}]->(usa)
MATCH (af)-[t2:TRADE {year: 2017}]->(chn)
WHERE t1.export_val > 0 AND t2.export_val > 0
RETURN af.name AS countryName
"""

# Execute the query and return the results as a Pandas DataFrame
# This will display as a formatted table in the notebook output
results_df = conn.execute(query).get_as_df()
print(results_df.head())
print(results_df.shape)

### Almost every African country exports to USA as well as China each year, this is
something I knew in advance.
```

10.2.9 7) Reflection and Key Takeaways

Why graphs? Graph databases shine when your questions are about *connections* — who trades with whom, what paths exist between entities, which nodes are central to a network. The Cypher query language makes these patterns readable and concise. In SQL, a two-hop traversal requires nested joins; in Cypher, it is a single line: `(a) -[:TRADE] -> (b) -[:TRADE] -> (c)`.

Graph schema = SQL DDL. Just as DBMS RAG retrieves `CREATE TABLE` statements, Graph RAG retrieves node labels and relationship types. Both serve the same purpose: giving the LLM enough structural knowledge to write valid queries. The schema is compact and fixed-size regardless of how much data the graph contains.

The value mismatch problem. As we saw, the LLM may generate syntactically correct Cypher that uses wrong property values (e.g., `'USA'` vs. `'United States'`). This is an active area of research. Solutions include: adding sample node data to the schema context, using fuzzy matching on property values, or adding a validation step that checks query results before presenting them to the user.

Exercises to try:

1. Ask the LLM about multi-hop trade paths (e.g., “Which countries connect Germany to Brazil through a single intermediary?”).
2. Modify the schema context to include sample node values and see if the LLM produces better queries.
3. Compare the same question answered by SQL RAG (if you had the trade data in a relational table) vs. Graph RAG. Which query is more readable?
4. Discuss: for what kinds of business data would you choose a graph database over a relational one?

10.2.10 Key takeaways

- **Graph databases** model relationships natively – Cypher patterns like `(a) -[:TRADE] -> (b)` replace multi-join SQL for connection-heavy questions.
- **Graph schema retrieval** (node labels, relationship types, properties) plays the same role for Graph RAG as DDL plays for SQL RAG – compact structural context for query generation.
- **Text-to-Cypher** follows the same two-call pattern as text-to-SQL: generate query, execute, then ground the natural-language answer on real results.
- **Value mismatches** (LLM writes `'USA'` when data stores `'United States'`) silently return zero rows – zero results does NOT always mean zero; always sanity-check.
- **Visualization** of retrieved subgraphs with NetworkX turns abstract query output into an interpretable picture of the underlying data.

10.2.11 Run the code

To run this notebook, copy the URL below into your browser’s address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~5 minutes (graph construction + Cypher generation)

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/GRAPH_RAG_Trade.ipynb

Chapter 11

RAG with Images and Video

RAG can also work with visual content. Instead of embedding text, we embed **images** (and video frames) using CLIP — a model that maps both text and images into the same vector space.

This chapter builds:

- **Image RAG** — embed product images with CLIP, retrieve by text query (“red circular product”), answer with a multimodal LLM
- **Video RAG** — extract keyframes from a video, embed them with CLIP, retrieve relevant frames, and describe them

The same retrieve-augment-generate pattern applies — just with pixels instead of paragraphs.

11.1 RAG with Images: Image Retrieval and Multimodal Q&A

This notebook demonstrates **Retrieval-Augmented Generation (RAG)** where the knowledge source is a collection of **images** rather than text documents or a database.

How it works:

1. Build a small image dataset (synthetic product catalog images generated in-notebook — no external data needed).
2. Compute **CLIP embeddings** for each image to create a searchable vector index.
3. Given a natural-language query, compute a **text embedding** and retrieve the most relevant images.
4. Send the retrieved images to a **multimodal LLM** which generates a grounded, natural-language answer.

Learning goals:

- Understand how RAG extends to visual/multimodal data
- See how CLIP jointly embeds text and images into a shared vector space
- Practice grounding LLM output on retrieved image evidence
- Compare answers with and without image context

Provider setup: Gemini is tried first; if quota is exhausted, Ollama Cloud is used automatically.

Store your LLM API keys in Colab Secrets (or a local `.env` file). Supported keys: `GEMINI_API_KEY`, `OPENAI_API_KEY`, `GROQ_API_KEY`, `HF_TOKEN`, `COHERE_API_KEY`, `XAI_API_KEY`, `OLLAMA_API_KEY`, `OPENROUTER_API_KEY`.

This is the same retrieve-then-generate pattern from Chapter 8's `rag_first_principles.ipynb` applied to a new modality — CLIP is essentially the embedding idea from Chapter 2, extended so that pictures and sentences land in the same vector space.

```
!pip install -q google-genai Pillow==11.0.0 transformers==4.46.3 git+https://github.com/KarAnalytics/llm_cascade.git
```

11.1.1 1) Imports and Provider Helpers

We use **CLIP** (from Hugging Face `transformers`) to embed images and text into the same vector space, and **PIL/Pillow** for image handling.

The provider helpers below follow the same pattern used in the DBMS RAG notebook: Gemini first, Ollama Cloud fallback, keys from Colab Secrets or `.env`.

```
import numpy as np
from pathlib import Path

from PIL import Image, ImageDraw, ImageFont

from llm_cascade import get_cascade

llm = get_cascade()

def generate_text(prompt, system_prompt=None):
    response = llm.generate(prompt, system_prompt=system_prompt)
    return response.text, response.provider

def has_llm_provider():
    return True # llm_cascade auto-detects available providers

print("Provider ready (llm_cascade)")
```

11.1.2 2) Generate a Synthetic Image Dataset

Since we do not have external image data, we create a small **synthetic product catalog** consisting of simple colored shapes with labels. Each image represents a product with specific visual attributes (shape, color, size, pattern).

This keeps the demo self-contained while illustrating the full Image RAG pipeline.

Product	Shape	Color	Pattern
Widget A	Circle	Red	Solid
Widget B	Square	Blue	Striped
Gadget X	Triangle	Green	Solid
Gadget Y	Circle	Yellow	Dotted
Tool M	Rectangle	Purple	Solid
Tool N	Diamond	Orange	Striped

```
import math

# --- Product catalog definition ---
PRODUCTS = [
    {"name": "Widget A", "shape": "circle", "color": "#E74C3C", "color_name": "red", "pattern": "solid", "price": 12.99, "category": "widgets"},
    {"name": "Widget B", "shape": "square", "color": "#3498DB", "color_name": "blue", "pattern": "striped", "price": 24.50, "category": "widgets"},
    {"name": "Gadget X", "shape": "triangle", "color": "#2ECC71", "color_name": "green", "pattern": "solid", "price": 8.75, "category": "gadgets"},
    {"name": "Gadget Y", "shape": "circle", "color": "#F1C40F", "color_name": "yellow", "pattern": "dotted", "price": 15.00, "category": "gadgets"},
    {"name": "Tool M", "shape": "rectangle", "color": "#9B59B6", "color_name": "purple", "pattern": "solid", "price": 34.99, "category": "tools"},
    {"name": "Tool N", "shape": "diamond", "color": "#E67E22", "color_name": "orange", "pattern": "striped", "price": 19.25, "category": "tools"},
    {"name": "Widget C", "shape": "circle", "color": "#1ABC9C", "color_name": "teal", "pattern": "solid", "price": 11.50, "category": "widgets"},
    {"name": "Gadget Z", "shape": "square", "color": "#E91E63", "color_name": "pink", "pattern": "dotted", "price": 29.99, "category": "gadgets"},
]

def draw_product_image(product, size=256):
    """Generate a synthetic product image with shape, color, pattern, and label."""
    img = Image.new("RGB", (size, size), "white")
    draw = ImageDraw.Draw(img)
    margin = 40
    cx, cy = size // 2, size // 2
    r = size // 2 - margin # radius or half-size

    color = product["color"]
    shape = product["shape"]

    # Draw the shape
    if shape == "circle":
        draw.ellipse([cx - r, cy - r, cx + r, cy + r], fill=color, outline="black", width=2)
    elif shape == "square":
        draw.rectangle([cx - r, cy - r, cx + r, cy + r], fill=color, outline="black", width=2)
    elif shape == "rectangle":
        draw.rectangle([cx - r, cy - r // 2, cx + r, cy + r // 2], fill=color, outline="black", width=2)
```

```

elif shape == "triangle":
    points = [(cx, cy - r), (cx - r, cy + r), (cx + r, cy + r)]
    draw.polygon(points, fill=color, outline="black", width=2)
elif shape == "diamond":
    points = [(cx, cy - r), (cx + r, cy), (cx, cy + r), (cx - r, cy)]
    draw.polygon(points, fill=color, outline="black", width=2)

# Add pattern overlay
if product["pattern"] == "striped":
    for y in range(0, size, 12):
        draw.line([(0, y), (size, y)], fill="white", width=2)
elif product["pattern"] == "dotted":
    for dx in range(margin, size - margin, 20):
        for dy in range(margin, size - margin, 20):
            draw.ellipse([dx - 3, dy - 3, dx + 3, dy + 3], fill="white")

# Add product name label at top
try:
    font = ImageFont.truetype("arial.ttf", 16)
except (OSError, IOError):
    font = ImageFont.load_default()
bbox = draw.textbbox((0, 0), product["name"], font=font)
tw = bbox[2] - bbox[0]
draw.text(((size - tw) // 2, 5), product["name"], fill="black", font=font)

# Add price label at bottom
price_text = f"${product['price']:.2f}"
bbox = draw.textbbox((0, 0), price_text, font=font)
tw = bbox[2] - bbox[0]
draw.text(((size - tw) // 2, size - 25), price_text, fill="black", font=font)

return img

# Generate and save all product images
IMAGE_DIR = Path("product_images")
IMAGE_DIR.mkdir(exist_ok=True)

image_paths = []
for product in PRODUCTS:
    img = draw_product_image(product)
    filename = product["name"].lower().replace(" ", "_") + ".png"
    path = IMAGE_DIR / filename
    img.save(path)
    image_paths.append(path)

print(f"Generated {len(image_paths)} product images in '{IMAGE_DIR}/'")
print("Files:", [p.name for p in image_paths])

```

11.1.2.1 Preview the Generated Images

Before we index anything, it is worth taking a moment to look at our dataset. The images below are intentionally simple – colored geometric shapes with text labels – but they carry enough visual variety (shape, color, pattern, price) to make retrieval interesting. In a production system these would be real product photographs, but synthetic images let us control every attribute and verify that CLIP picks up on the right visual cues.

```

import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 4, figsize=(14, 7))
for ax, product, path in zip(axes.flat, PRODUCTS, image_paths):
    img = Image.open(path)

```

```

ax.imshow(img)
ax.set_title(f"{product['name']} ({product['color_name']})", fontsize=10)
ax.axis("off")
plt.suptitle("Synthetic Product Catalog", fontsize=14, fontweight="bold")
plt.tight_layout()
plt.show()

```

11.1.3 3) Build the Image Index with CLIP Embeddings

CLIP (Contrastive Language-Image Pre-training) is the key technology that makes Image RAG possible. Unlike traditional image classifiers that output a fixed set of labels, CLIP maps both text and images into the **same 512-dimensional vector space**. It was trained on hundreds of millions of image-caption pairs from the internet, learning to push matching image-text pairs close together and non-matching pairs apart.

Why does this matter for RAG? Because once images and text live in a shared space, we can measure the similarity between a user's natural-language query and every image in our catalog using ordinary cosine similarity – no manual tagging, no predefined categories. The model “understands” that the text “red circular product” should be close to an image of a red circle, even though those are entirely different data modalities. This is analogous to how ChromaDB uses text embeddings in document RAG, but here the bridge spans from language to vision.

We compute an embedding for each product image below, normalize them to unit vectors, and store the result as our searchable image index.

```

from transformers import CLIPProcessor, CLIPModel
import torch

# Load CLIP model (downloads ~600 MB on first run)
CLIP_MODEL_NAME = "openai/clip-vit-base-patch32"
clip_model = CLIPModel.from_pretrained(CLIP_MODEL_NAME)
clip_processor = CLIPProcessor.from_pretrained(CLIP_MODEL_NAME)

print(f"CLIP model loaded: {CLIP_MODEL_NAME}")
print(f"Embedding dimension: {clip_model.config.projection_dim}")

def embed_images(image_paths, model, processor):
    """Compute CLIP embeddings for a list of image files."""
    images = [Image.open(p).convert("RGB") for p in image_paths]
    inputs = processor(images=images, return_tensors="pt", padding=True)
    with torch.no_grad():
        embeddings = model.get_image_features(**inputs)
    # Normalize to unit vectors for cosine similarity
    embeddings = embeddings / embeddings.norm(dim=-1, keepdim=True)
    return embeddings.numpy()

def embed_text(query, model, processor):
    """Compute CLIP embedding for a text query."""
    inputs = processor(text=[query], return_tensors="pt", padding=True,
truncation=True)
    with torch.no_grad():
        embeddings = model.get_text_features(**inputs)
    embeddings = embeddings / embeddings.norm(dim=-1, keepdim=True)
    return embeddings.numpy()

# Build the image index
image_embeddings = embed_images(image_paths, clip_model, clip_processor)
print(f"Image index built: {image_embeddings.shape[0]} images,
{image_embeddings.shape[1]}-dim embeddings")

```


With our index built – 8 images, each represented as a 512-dimensional unit vector – we are ready to perform cross-modal search. Notice that the embedding functions above handle text and images through different CLIP encoders (`get_image_features` vs. `get_text_features`), but both produce vectors in the same space. That shared geometry is what lets us compare apples (text) to oranges (images) using a single dot product.

11.1.4 4) Image Retrieval – the “Retrieval” in Image RAG

Now that every product image has been mapped to a 512-dimensional vector, we can perform retrieval. Given a text query, we embed it with CLIP’s text encoder and compute **cosine similarity** between the query vector and every image vector in our index. Because we normalized all embeddings to unit length earlier, this cosine similarity reduces to a simple dot product – a single matrix multiplication gives us a ranked list of all images.

A word about the similarity scores you will see below: CLIP similarities tend to be modest in absolute terms (often 0.2-0.4 for good matches) because the 512-dimensional space is vast and the model must accommodate enormous variety. What matters is the **relative ordering** – the top-ranked image should be the most visually relevant to the query, even if its raw score looks low. Think of the score not as a confidence percentage but as a ranking signal.

This is the image equivalent of vector similarity search in document RAG.

```
def retrieve_images(query, image_embeddings, image_paths, products, model,
processor, top_k=3):
    """Retrieve the top-k most relevant images for a text query."""
    query_emb = embed_text(query, model, processor)
    # Cosine similarity (embeddings are already normalized)
    similarities = (query_emb @ image_embeddings.T).flatten()
    # Get top-k indices
    top_indices = similarities.argsort()[::-1][:top_k]

    results = []
    for idx in top_indices:
        results.append({
            "rank": len(results) + 1,
            "product": products[idx],
            "path": image_paths[idx],
            "similarity": float(similarities[idx]),
        })
    return results

# Test retrieval
test_query = "red circular product"
results = retrieve_images(test_query, image_embeddings, image_paths, PRODUCTS,
clip_model, clip_processor)

print(f"Query: '{test_query}'\n")
for r in results:
    print(f"  #{r['rank']} {r['product']['name']} "
          f"({r['product']['color_name']} {r['product']['shape']}) "
          f"– similarity: {r['similarity']:.4f}")

# Visualize retrieval results
def show_retrieval_results(query, results):
    """Display retrieved images side by side."""
    n = len(results)
    fig, axes = plt.subplots(1, n, figsize=(5 * n, 5))
    if n == 1:
        axes = [axes]
    for ax, r in zip(axes, results):
        img = Image.open(r["path"])
        ax.imshow(img)
```

```

        ax.set_title(f"#{r['rank']} {r['product']} {r['name']}\nsim={r['similarity']:.3f}", fontsize=11)
        ax.axis("off")
        fig.suptitle(f"Query: \"{query}\"", fontsize=13, fontweight="bold")
        plt.tight_layout()
        plt.show()

show_retrieval_results(test_query, results)

```

11.1.5 5) Multimodal Answer Generation

This is the “Generation” half of Image RAG — the step where a multimodal LLM actually looks at the retrieved images and produces a grounded answer. Unlike document RAG where the LLM receives text chunks, here the model receives **actual image pixels** alongside product metadata and the user’s question.

We use `llm_cascade`’s `generate_multimodal()` method, which accepts image paths (or bytes, or PIL images) and routes them to the first available vision-capable provider. Each provider has its own serialization format — Gemini wraps images in `Part.from_bytes` objects while OpenAI-compatible APIs encode them as base64 data URIs in an `image_url` block — but the cascade hides those details so the notebook just passes a list of images. If Gemini is rate-limited or returns a 503, the cascade automatically falls back to the next vision-capable provider (Ollama Cloud, HuggingFace, or OpenRouter). See the *Optional* section at the end of the notebook for what the raw Gemini call looks like if you were to write it by hand.

```

ANSWER_SYSTEM_PROMPT = (
    "You are a helpful product catalog assistant. Use ONLY the provided images and "
    "product metadata to answer the user's question. If the retrieved products do "
    "not "
    "contain enough information, say so. Do not invent products that are not shown."
)

def build_metadata_context(results):
    """Build a text description of retrieved products for the prompt."""
    lines = ["Retrieved products (ranked by visual similarity):"]
    for r in results:
        p = r["product"]
        lines.append(
            f" #{r['rank']} {p['name']}: {p['color_name']} {p['shape']}, "
            f"{p['pattern']} pattern, ${p['price']:.2f}, category={p['category']} "
            f"(similarity={r['similarity']:.3f})"
        )
    return "\n".join(lines)

def generate_multimodal_answer(question, results):
    """Send retrieved images + metadata to a multimodal LLM via llm_cascade.

    The cascade tries vision-capable providers in order
    (Gemini -> Ollama -> HuggingFace -> OpenRouter) and falls back
    automatically on quota, 503, or overload errors.
    """
    metadata_text = build_metadata_context(results)
    image_paths = [r["path"] for r in results]
    prompt = f"{metadata_text}\n\nQUESTION: {question}"
    response = llm.generate_multimodal(
        prompt, images=image_paths, system_prompt=ANSWER_SYSTEM_PROMPT)
    return response.text, response.provider

print("Multimodal answer generation ready (via llm_cascade).")

```

11.1.6 6) Full Image RAG Pipeline

Now we connect the two halves of the system – retrieval and generation – into a single end-to-end function. Given a natural-language question, the pipeline embeds the query with CLIP, retrieves the most relevant product images, and then hands those images (along with their metadata) to a multimodal LLM for a grounded answer.

We also define a “without RAG” baseline that sends the same question directly to the LLM with no image context at all. Comparing the two reveals a striking pattern: without retrieval, the model has no knowledge of our product catalog and either refuses to answer or invents products that do not exist. This is the same hallucination problem we saw in document RAG, but it becomes even more vivid when the “ground truth” is visual.

```
def answer_with_image_rag(question, top_k=3):
    """Full Image RAG pipeline: embed query -> retrieve images -> multimodal
    answer."""
    print(f" [Step 1] Embedding query with CLIP ...")
    results = retrieve_images(
        question, image_embeddings, image_paths, PRODUCTS,
        clip_model, clip_processor, top_k=top_k,
    )
    print(f" [Step 2] Retrieved {len(results)} images:")
    for r in results:
        print(f"          #{r['rank']} {r['product']['name']}
(sim={r['similarity']:.3f})")

    print(f" [Step 3] Generating multimodal answer ...")
    answer, provider = generate_multimodal_answer(question, results)
    print(f" [Step 4] Answer generated (provider: {provider})")

    return answer, results, provider

def answer_without_rag(question):
    """Direct LLM answer with no image context."""
    answer, provider = generate_text(prompt=question)
    return answer, provider

print("Full Image RAG pipeline ready.")
```

11.1.7 7) Run End-to-End Examples

We test several questions and compare:

- **With Image RAG:** CLIP retrieval + multimodal grounded answer
- **Without RAG:** direct LLM answer (no product images or metadata)

Watch for hallucinations in the “without RAG” answers – the model has no access to our product catalog.

```
questions = [
    "What red products do you have?",
    # "Show me the cheapest circular product.",
    # "Which products have a striped pattern?",
    # "I need a green product. What are my options?",
    "What is the most expensive product in the catalog?",
]

def preview(text, max_len=800):
    text = text or ""
    return text[:max_len] + ("..." if len(text) > max_len else "")
```

```

if not has_llm_provider():
    print("Error: No LLM API key configured. Set at least one API key in Colab
Secrets.")
else:
    for i, q in enumerate(questions, start=1):
        print("\n" + "=" * 80)
        print(f"Q{i}. {q}\n")

        try:
            answer_rag, retrieved, prov = answer_with_image_rag(q)
            show_retrieval_results(q, retrieved)
            print("\n [WITH IMAGE RAG]")
            print(preview(answer_rag))
        except Exception as e:
            print(f" RAG error: {e}")

        try:
            answer_direct, prov_direct = answer_without_rag(q)
            print(f"\n [WITHOUT RAG] (provider: {prov_direct})")
            print(preview(answer_direct))
        except Exception as e:
            print(f" Direct error: {e}")

```

11.1.7.1 Checkpoint: Reflection Questions

1. Compare the RAG and non-RAG answers. Which ones hallucinate products that don't exist?
2. How does CLIP's cross-modal embedding enable text-to-image search without manual tagging?
3. What happens when you query for something not in the catalog (e.g., "black hexagonal product")?
4. How does this Image RAG pattern compare to the DBMS RAG and vector-store RAG approaches?

11.1.8 8) Interactive Query (Optional)

Now it is your turn. The cell below lets you ask your own questions against the product image catalog. Try queries that test different aspects of CLIP's cross-modal understanding: ask about colors, shapes, patterns, price ranges, or combinations of attributes. You might also try querying for something that does not exist in the catalog (e.g., "black hexagonal product") to see how the system handles out-of-distribution queries – it will still return the "closest" matches, but with lower similarity scores.

```

# Change this question to anything you want to ask about the product catalog.
my_question = "What purple or blue products do you have, and which is cheaper?"

if has_llm_provider():
    print(f"Question: {my_question}\n")
    try:
        answer, retrieved, provider = answer_with_image_rag(my_question)
        show_retrieval_results(my_question, retrieved)
        print(f"\nAnswer ({provider}): \n{answer}")
    except Exception as e:
        print(f"Error: {e}")
else:
    print("No LLM API key configured. Set at least one API key in Colab Secrets.")

```

11.1.9 9) Teaching Notes and Exercises

Key takeaways:

- RAG extends naturally to images and other modalities via **cross-modal embeddings** like CLIP.
- CLIP embeds text and images into a **shared vector space**, enabling text-to-image retrieval without manual tagging.
- The retrieval step uses the same cosine similarity as document RAG, but operates on image embeddings.
- Multimodal LLMs can directly "see" retrieved images, producing answers grounded on visual evidence.

- Even with synthetic data, the pipeline demonstrates real-world patterns used in e-commerce, medical imaging, and visual search.

Exercises:

- Replace the synthetic images with real product photos (e.g., from a Kaggle dataset) and observe how retrieval quality changes.
- Experiment with different CLIP model sizes (`clip-vit-large-patch14`) and compare retrieval accuracy.
- Add more products with similar colors/shapes and test whether the system can still distinguish them.
- Implement a **re-ranking** step where the LLM re-orders retrieved images by relevance before answering.
- Discuss: when would you prefer Image RAG over text-based RAG for a business application?

11.1.10 Key takeaways

- **Image RAG** extends retrieval-augmented generation from text to visual data by swapping text embeddings for multimodal ones.
- **CLIP** maps images and text into a shared 512-dimensional vector space, letting you search an image catalog with natural-language queries.
- **Cosine similarity** over normalized CLIP embeddings is all you need for retrieval – a single matrix multiply produces the ranked list.
- **Multimodal LLMs** like Gemini can ingest the retrieved image pixels directly, producing answers grounded in what the model actually “sees”.
- **Without RAG**, the LLM has no catalog knowledge and will either refuse or hallucinate products that do not exist.

11.1.11 Optional: Raw Gemini Multimodal API Call

Above, `llm.generate_multimodal()` hid the provider-specific details of sending an image to an LLM. If you want to see what the cascade is doing internally, or you need to call Gemini’s multimodal API directly to use a Gemini-only feature, here is what that looks like.

Each provider has its own serialization format. Gemini’s Python SDK wraps image bytes in a `Part.from_bytes` object, while OpenAI-compatible providers (OpenRouter, Ollama Cloud, HuggingFace) expect base64-encoded data URIs inside a JSON `image_url` block. The cascade converts between these on your behalf; the direct call below skips that abstraction.

```
# Direct Gemini multimodal call -- the same shape the cascade uses internally
# for Gemini, but without fallback to other providers.
from google import genai
from google.genai import types as genai_types
from llm_cascade.providers import _get_key

gemini_key = _get_key("GEMINI_API_KEY")
if not gemini_key:
    print("Skipping: GEMINI_API_KEY not configured.")
else:
    question = "What color is the product shown?"
    image_bytes = Path(image_paths[0]).read_bytes()

    client = genai.Client(api_key=gemini_key)
    parts = [
        genai_types.Part.from_text(text=question),
        genai_types.Part.from_bytes(data=image_bytes, mime_type="image/png"),
    ]
    resp = client.models.generate_content(
        model="gemini-2.5-flash",
        contents=parts,
        config=genai_types.GenerateContentConfig(
            thinking_config=genai_types.ThinkingConfig(thinking_budget=0),
        ),
    )
```

```
print(f"Question: {question}")
print(f"Image: {Path(image_paths[0]).name}")
print(f"\nAnswer:\n{resp.text}")
```

11.1.12 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~5 minutes on T4 GPU (CLIP model loading + LLM calls)

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/Image_RAG.ipynb

11.2 RAG with Video: Visual Question-Answering over Video Frames

This notebook demonstrates **Retrieval-Augmented Generation (RAG)** where the knowledge source is a **video**. Instead of searching documents or databases, we extract frames from a video, index them with CLIP embeddings, and retrieve the most relevant frames to answer natural-language questions.

How it works:

1. Generate a short **synthetic video** (animated product showcase — no external data needed).
2. Extract **keyframes** at regular intervals.
3. Compute **CLIP embeddings** for each frame to create a searchable index.
4. Given a natural-language question, retrieve the most relevant frame(s) via cosine similarity.
5. Send retrieved frames to a **multimodal LLM** for a grounded, natural-language answer.

Learning goals:

- Understand how RAG applies to temporal/video data
- See how frame extraction converts video into an image retrieval problem
- Practice grounding LLM output on visual evidence from specific moments in a video
- Compare answers with and without video context

Store whichever API keys you have in Colab Secrets (or a local `.env` file): `GEMINI_API_KEY`, `OLLAMA_API_KEY`, `XAI_API_KEY`, `GROQ_API_KEY`, `HF_TOKEN`, `COHERE_API_KEY`, `OPENROUTER_API_KEY`

This notebook builds directly on the sibling `image_rag.ipynb` — once you convert a video into a set of keyframes, Video RAG reduces to Image RAG with a timestamp attached to every hit.

```
!pip install -q google-genai Pillow==11.0.0 transformers==4.46.3 matplotlib opencv-  
python-headless git+https://github.com/KarAnalytics/llm_cascade.git
```

11.2.1 1) Imports and Provider Helpers (7-Vendor Cascade)

We use **OpenCV** to create and read video, **CLIP** to embed frames and text queries into the same vector space, and **PIL/Pillow** for image display. If you have already worked through the Image RAG notebook, you will recognize most of these imports – and that is the point. Video RAG is fundamentally the same pattern as Image RAG, applied to a different modality. The main new ingredient is OpenCV, which handles the mechanics of extracting individual frames from a video file.

```
# =====  
# Imports + LLM Cascade  
# =====  
  
import math  
import numpy as np  
from pathlib import Path  
  
import cv2  
from PIL import Image, ImageDraw, ImageFont  
  
from llm_cascade import get_cascade  
  
llm = get_cascade()  
  
def has_llm_provider():  
    """True if llm_cascade has at least one provider configured."""  
    return True # get_cascade() already validates  
  
def generate_text(prompt, system_prompt=None):  
    """Send a text prompt through llm_cascade.  
  
    Returns (response_text, provider_string).
```

```

"""
resp = llm.generate(prompt, system_prompt=system_prompt)
return resp.text, f"{resp.provider}/{resp.model}"

print("LLM cascade ready.")

```

```

LLM Cascade - available providers:
+ Gemini          model=gemini-2.5-flash
+ Ollama          model=kimi-k2.5:cloud
+ Groq           model=llama-3.3-70b-versatile
+ HuggingFace     model=meta-llama/Llama-3.3-70B-Instruct
+ Cohere          model=command-a-03-2025
+ OpenRouter      model=meta-llama/llama-3.3-70b-instruct:free
+ OpenAI          model=gpt-4o-mini
Not configured (skipped):
- Grok (xAI)      (set XAI_API_KEY)
LLM cascade ready.

```

11.2.2 2) Generate a Synthetic Video

We create a short animated video that simulates a **product showcase reel**. The video has distinct scenes, each showing a different product with visual attributes that change over time.

This keeps the demo self-contained (no external video files needed) while providing enough visual variety to test retrieval.

Time (sec)	Scene	Content
0–3	Title card	“Acme Corp Product Showcase” on blue background
3–7	Product 1	Red circle — “Widget Alpha” with price
7–11	Product 2	Green triangle — “Gadget Beta” with price
11–15	Product 3	Blue square — “Tool Gamma” with price
15–19	Product 4	Orange diamond — “Device Delta” with price
19–22	Closing card	“Thank you! Visit acme-corp.example.com ”

```

VIDEO_PATH = "product_showcase.mp4"
FPS = 10
WIDTH, HEIGHT = 640, 480

# Scene definitions
SCENES = [
    {
        "name": "title",
        "start": 0, "end": 3,
        "label": "Acme Corp\nProduct Showcase 2026",
        "bg_color": (40, 60, 120),
        "shape": None,
        "product": None,
    },
    {
        "name": "widget_alpha",
        "start": 3, "end": 7,
        "label": "Widget Alpha – $12.99",
        "bg_color": (255, 245, 238),
        "shape": "circle",
        "shape_color": (231, 76, 60),
        "product": {"name": "Widget Alpha", "price": 12.99, "color": "red", "shape": "circle"},
    },

```



```

    },
    {
        "name": "gadget_beta",
        "start": 7, "end": 11,
        "label": "Gadget Beta – $8.75",
        "bg_color": (240, 255, 240),
        "shape": "triangle",
        "shape_color": (46, 204, 113),
        "product": {"name": "Gadget Beta", "price": 8.75, "color": "green", "shape":
"triangle"}},
    },
    {
        "name": "tool_gamma",
        "start": 11, "end": 15,
        "label": "Tool Gamma – $34.99",
        "bg_color": (235, 245, 255),
        "shape": "square",
        "shape_color": (52, 152, 219),
        "product": {"name": "Tool Gamma", "price": 34.99, "color": "blue", "shape":
"square"}},
    },
    {
        "name": "device_delta",
        "start": 15, "end": 19,
        "label": "Device Delta – $19.25",
        "bg_color": (255, 248, 230),
        "shape": "diamond",
        "shape_color": (230, 126, 34),
        "product": {"name": "Device Delta", "price": 19.25, "color": "orange",
"shape": "diamond"}},
    },
    {
        "name": "closing",
        "start": 19, "end": 22,
        "label": "Thank you!\nVisit acme-corp.example.com",
        "bg_color": (40, 60, 120),
        "shape": None,
        "product": None,
    },
]

```

```

def draw_frame_pil(scene, t, width=WIDTH, height=HEIGHT):
    """Draw a single video frame for a given scene and time offset using PIL."""
    bg = scene["bg_color"]
    img = Image.new("RGB", (width, height), bg)
    draw = ImageDraw.Draw(img)

    # Load font
    try:
        font_large = ImageFont.truetype("/usr/share/fonts/truetype/dejavu/
DejaVuSans-Bold.ttf", 28)
        font_small = ImageFont.truetype("/usr/share/fonts/truetype/dejavu/
DejaVuSans.ttf", 22)
    except (OSError, IOError):
        try:
            font_large = ImageFont.truetype("arial.ttf", 28)
            font_small = ImageFont.truetype("arial.ttf", 22)
        except (OSError, IOError):
            font_large = ImageFont.load_default()
            font_small = font_large

    cx, cy = width // 2, height // 2

```

```

if scene["shape"] is None:
    # Title or closing card – centered text
    text_color = (255, 255, 255)
    lines = scene["label"].split("\n")
    total_h = len(lines) * 40
    y_start = cy - total_h // 2
    for j, line in enumerate(lines):
        bbox = draw.textbbox((0, 0), line, font=font_large)
        tw = bbox[2] - bbox[0]
        draw.text(((width - tw) // 2, y_start + j * 40), line, fill=text_color,
font=font_large)
    else:
        # Product scene – animated shape (gentle bounce)
        bounce = int(15 * math.sin(t * 3))
        shape_cy = cy - 30 + bounce
        r = 80
        color = scene["shape_color"]

        if scene["shape"] == "circle":
            draw.ellipse([cx - r, shape_cy - r, cx + r, shape_cy + r], fill=color,
outline=(0, 0, 0), width=3)
        elif scene["shape"] == "square":
            draw.rectangle([cx - r, shape_cy - r, cx + r, shape_cy + r], fill=color,
outline=(0, 0, 0), width=3)
        elif scene["shape"] == "triangle":
            pts = [(cx, shape_cy - r), (cx - r, shape_cy + r), (cx + r, shape_cy +
r)]
            draw.polygon(pts, fill=color, outline=(0, 0, 0), width=3)
        elif scene["shape"] == "diamond":
            pts = [(cx, shape_cy - r), (cx + r, shape_cy), (cx, shape_cy + r), (cx -
r, shape_cy)]
            draw.polygon(pts, fill=color, outline=(0, 0, 0), width=3)

        # Product label at top
        bbox = draw.textbbox((0, 0), scene["label"], font=font_small)
        tw = bbox[2] - bbox[0]
        draw.text(((width - tw) // 2, 20), scene["label"], fill=(30, 30, 30),
font=font_small)

        # Timestamp at bottom-right
        ts_text = f"t={t:.1f}s"
        draw.text((width - 90, height - 30), ts_text, fill=(120, 120, 120),
font=font_small)

    return img

def generate_video(path, scenes, fps, width=WIDTH, height=HEIGHT):
    """Generate an MP4 video from scene definitions."""
    fourcc = cv2.VideoWriter_fourcc(*"mp4v")
    writer = cv2.VideoWriter(path, fourcc, fps, (width, height))

    for scene in scenes:
        duration = scene["end"] - scene["start"]
        n_frames = int(duration * fps)
        for i in range(n_frames):
            t = scene["start"] + i / fps
            pil_img = draw_frame_pil(scene, t, width, height)
            # Convert PIL (RGB) -> OpenCV (BGR)
            frame = cv2.cvtColor(np.array(pil_img), cv2.COLOR_RGB2BGR)
            writer.write(frame)

```

```
writer.release()
```

```
generate_video(VIDEO_PATH, SCENES, FPS)
total_duration = SCENES[-1]["end"]
file_size_kb = Path(VIDEO_PATH).stat().st_size / 1024
print(f"Video generated: '{VIDEO_PATH}'")
print(f"  Duration: {total_duration}s | FPS: {FPS} | Resolution: {WIDTH}x{HEIGHT} |
Size: {file_size_kb:.0f} KB")
```

```
Video generated: 'product_showcase.mp4'
Duration: 22s | FPS: 10 | Resolution: 640x480 | Size: 260 KB
```

11.2.3 3) Extract Keyframes from the Video

Here is the crucial insight behind Video RAG: we do not try to process the video as a continuous stream. Instead, we **sample keyframes** at regular intervals and treat each one as an independent image. This converts the temporal video retrieval problem into the familiar image retrieval problem we already solved in the Image RAG notebook.

Why not process the full video? A 22-second video at 10 FPS contains 220 frames, most of which are nearly identical to their neighbors. Processing every frame would waste compute and storage without improving retrieval quality. By extracting one frame per second, we capture each scene while keeping our index compact. For longer or faster-paced videos you would adjust this sampling rate – the tradeoff is always between coverage (catching brief but important moments) and efficiency (keeping the index manageable).

```
def extract_keyframes(video_path, interval_sec=1.0):
    """Extract frames from a video at a fixed time interval.

    Returns a list of dicts: {"timestamp": float, "image": PIL.Image}
    """
    cap = cv2.VideoCapture(video_path)
    fps = cap.get(cv2.CAP_PROP_FPS)
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    duration = total_frames / fps
    frame_interval = int(fps * interval_sec)

    frames = []
    frame_idx = 0
    while True:
        cap.set(cv2.CAP_PROP_POS_FRAMES, frame_idx)
        ret, frame = cap.read()
        if not ret:
            break
        # Convert BGR -> RGB -> PIL
        rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        pil_img = Image.fromarray(rgb)
        timestamp = frame_idx / fps
        frames.append({"timestamp": timestamp, "image": pil_img})
        frame_idx += frame_interval

    cap.release()
    return frames, duration

keyframes, video_duration = extract_keyframes(VIDEO_PATH, interval_sec=1.0)
print(f"Extracted {len(keyframes)} keyframes from {video_duration:.1f}s video (1
frame/sec)")
print(f"Timestamps: [{f'[{kf[\"timestamp\"]:.1f}]s' for kf in keyframes}]")
```

```

Extracted 22 keyframes from 22.0s video (1 frame/sec)
Timestamps: ['0.0s', '1.0s', '2.0s', '3.0s', '4.0s', '5.0s', '6.0s', '7.0s', '8.0s',
'9.0s', '10.0s', '11.0s', '12.0s', '13.0s', '14.0s', '15.0s', '16.0s', '17.0s',
'18.0s', '19.0s', '20.0s', '21.0s']

```

11.2.3.1 Preview Keyframes

The grid below shows every other keyframe from our 22-second video. You can see the distinct scenes: a title card, four product showcases (each with a different colored shape), and a closing card. Each of these frames will become an entry in our CLIP index, tagged with its timestamp so we can tell the user not just *what* was found but *when* in the video it appeared.

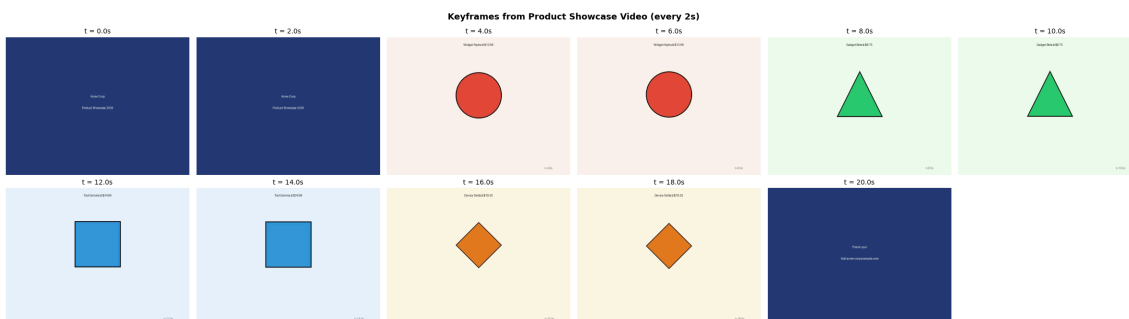
```

import matplotlib.pyplot as plt

# Show a subset of keyframes (every other frame to save space)
display_frames = keyframes[::2]
n = len(display_frames)
cols = min(n, 6)
rows = math.ceil(n / cols)

fig, axes = plt.subplots(rows, cols, figsize=(4 * cols, 3.5 * rows))
axes_flat = axes.flat if hasattr(axes, 'flat') else [axes]
for i, ax in enumerate(axes_flat):
    if i < n:
        ax.imshow(display_frames[i]["image"])
        ax.set_title(f"t = {display_frames[i]['timestamp']:.1f}s", fontsize=10)
    ax.axis("off")
fig.suptitle("Keyframes from Product Showcase Video (every 2s)", fontsize=13,
fontweight="bold")
plt.tight_layout()
plt.show()

```



11.2.4 4) Build the Frame Index with CLIP Embeddings

Just like in Image RAG, we use **CLIP** to embed each keyframe into a 512-dimensional vector. At query time we embed the user’s text query with the same model and find the closest frame(s) by cosine similarity. The embedding and retrieval code is nearly identical to what we wrote for static images – because once you extract keyframes, a video is just a collection of images.

The key difference from Image RAG is that each frame also carries a **timestamp**. This temporal metadata transforms a simple “which image matches?” answer into a richer “at what point in the video can you find this?” answer. Timestamps are what make Video RAG genuinely useful for applications like meeting recording search, surveillance review, or training video navigation.

```

from transformers import CLIPProcessor, CLIPModel
import torch

CLIP_MODEL_NAME = "openai/clip-vit-base-patch32"
clip_model = CLIPModel.from_pretrained(CLIP_MODEL_NAME)
clip_processor = CLIPProcessor.from_pretrained(CLIP_MODEL_NAME)

```

```
print(f"CLIP model loaded: {CLIP_MODEL_NAME}")
print(f"Embedding dimension: {clip_model.config.projection_dim}")
```

```
config.json: 0.00B [00:00, ?B/s]
```

```
pytorch_model.bin: 0%|          | 0.00/605M [00:00<?, ?B/s]
```

```
preprocessor_config.json: 0%|          | 0.00/316 [00:00<?, ?B/s]
```

```
tokenizer_config.json: 0%|          | 0.00/592 [00:00<?, ?B/s]
```

```
vocab.json: 0.00B [00:00, ?B/s]
```

```
merges.txt: 0.00B [00:00, ?B/s]
```

```
tokenizer.json: 0.00B [00:00, ?B/s]
```

```
special_tokens_map.json: 0%|          | 0.00/389 [00:00<?, ?B/s]
```

```
CLIP model loaded: openai/clip-vit-base-patch32
Embedding dimension: 512
```

```
def embed_frames(keyframes, model, processor):
    """Compute CLIP embeddings for a list of keyframe images."""
    images = [kf["image"].convert("RGB") for kf in keyframes]
    inputs = processor(images=images, return_tensors="pt", padding=True)
    with torch.no_grad():
        embeddings = model.get_image_features(**inputs)
        embeddings = embeddings / embeddings.norm(dim=-1, keepdim=True)
    return embeddings.numpy()

def embed_text(query, model, processor):
    """Compute CLIP embedding for a text query."""
    inputs = processor(text=[query], return_tensors="pt", padding=True,
truncation=True)
    with torch.no_grad():
        embeddings = model.get_text_features(**inputs)
        embeddings = embeddings / embeddings.norm(dim=-1, keepdim=True)
    return embeddings.numpy()

# Build the frame index
frame_embeddings = embed_frames(keyframes, clip_model, clip_processor)
print(f"Frame index built: {frame_embeddings.shape[0]} frames,
{frame_embeddings.shape[1]}-dim embeddings")
```

```
Frame index built: 22 frames, 512-dim embeddings
```

Our frame index is now ready: 22 keyframes, each represented as a 512-dimensional CLIP vector. Notice how the code above is almost identical to the image embedding code from the Image RAG notebook – the only difference is that our “images” come from video frames rather than from files on disk. This reuse of the same CLIP infrastructure across modalities is a powerful design pattern: once you can convert any data source into images, you get cross-modal search for free.

11.2.5 5) Frame Retrieval – the “Retrieval” in Video RAG

Given a text query, we:

1. Embed the query with CLIP’s text encoder.
2. Compute **cosine similarity** against all frame embeddings.
3. Return the top-k most similar frames with their timestamps.

The timestamp tells us *when* in the video the answer can be found.

```
def get_scene_for_timestamp(timestamp, scenes):
    """Look up which scene a timestamp belongs to."""
    for scene in scenes:
        if scene["start"] <= timestamp < scene["end"]:
            return scene
    return scenes[-1] # default to last scene

def retrieve_frames(query, frame_embeddings, keyframes, model, processor, top_k=3):
    """Retrieve the top-k most relevant video frames for a text query."""
    query_emb = embed_text(query, model, processor)
    similarities = (query_emb @ frame_embeddings.T).flatten()
    top_indices = similarities.argsort()[::-1][:top_k]

    results = []
    for idx in top_indices:
        ts = keyframes[idx]["timestamp"]
        scene = get_scene_for_timestamp(ts, SCENES)
        results.append({
            "rank": len(results) + 1,
            "frame_index": int(idx),
            "timestamp": ts,
            "scene_name": scene["name"],
            "product": scene.get("product"),
            "image": keyframes[idx]["image"],
            "similarity": float(similarities[idx]),
        })
    return results

def show_retrieved_frames(query, results):
    """Display retrieved frames side by side."""
    n = len(results)
    fig, axes = plt.subplots(1, n, figsize=(6 * n, 5))
    if n == 1:
        axes = [axes]
    for ax, r in zip(axes, results):
        ax.imshow(r["image"])
        label = f"#{r['rank']} t={r['timestamp']:.1f}"
        s = f"s\n{r['scene_name']}\nsim={r['similarity']:.3f}"
        ax.set_title(label, fontsize=10)
        ax.axis("off")
    fig.suptitle(f'Query: "{query}"', fontsize=13, fontweight="bold")
    plt.tight_layout()
    plt.show()

# Test retrieval
test_query = "red circle product"
results = retrieve_frames(test_query, frame_embeddings, keyframes, clip_model,
clip_processor)

print(f"Query: '{test_query}'\n")
for r in results:
```

```

prod = r['product']
prod_str = f"{prod['name']} ({prod['color']} {prod['shape']})" if prod else "(no product)"
print(f"  #{r['rank']} t={r['timestamp']:.1f}s | {r['scene_name']} | {prod_str} | sim={r['similarity']:.4f}")

show_retrieved_frames(test_query, results)

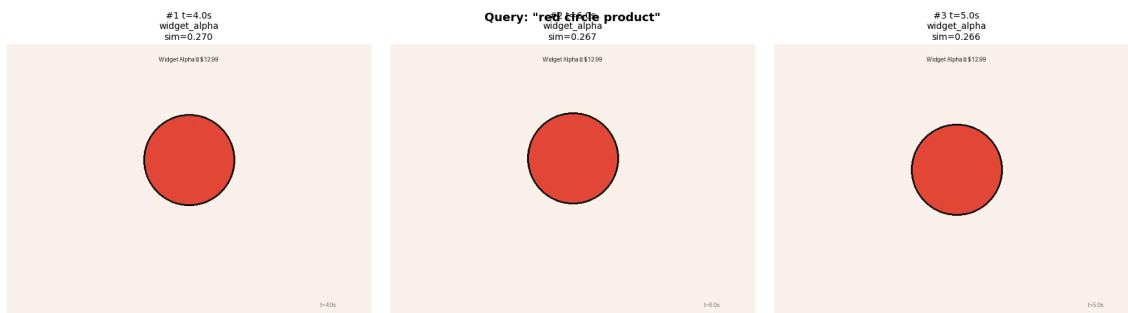
```

Query: 'red circle product'

```

#1 t=4.0s | widget_alpha | Widget Alpha (red circle) | sim=0.2704
#2 t=6.0s | widget_alpha | Widget Alpha (red circle) | sim=0.2670
#3 t=5.0s | widget_alpha | Widget Alpha (red circle) | sim=0.2658

```



11.2.6 6) Answer Generation — Multimodal LLM with Retrieved Frames

The multimodal LLM receives:

- The **retrieved frames** (as images)
- **Metadata** about each frame (timestamp, scene, product info)
- The user's **question**

It generates an answer grounded on what it can see in the frames and the associated timestamps.

We pass the frames through `llm_cascade`'s `generate_multimodal()` method, which handles the serialization differences between providers (Gemini uses `Part.from_bytes`; OpenAI-compatible providers use base64 data URIs) and automatically falls back if the first provider is rate-limited or overloaded. A “without RAG” variant below answers with no video context at all, so we can compare.

The cascade here sends **frames**, not the whole video file — that is the path that works across all four vision-capable providers. See the *Optional* section at the end of the notebook for a direct Gemini API call that takes the entire MP4 file as a single input. Gemini is the only provider in the cascade that accepts video bytes natively, which is why `generate_multimodal()` is frame-based for parity.

```

def build_frame_context(results):
    """Build a text description of retrieved frames."""
    lines = ["Retrieved video frames (ranked by visual similarity):"]
    for r in results:
        prod = r["product"]
        prod_str = (
            f"{prod['name']}: {prod['color']} {prod['shape']}, ${prod['price']:.2f}"
            if prod else "(title/closing card)"
        )
        lines.append(
            f"  #{r['rank']} timestamp={r['timestamp']:.1f}s, "
            f"scene={r['scene_name']}, "
            f"product={prod_str}, similarity={r['similarity']:.3f}"
        )
    return "\n".join(lines)

```

```

VIDEO_RAG_SYSTEM_PROMPT = (
    "You are a helpful video analyst. You are given keyframes extracted from a  

    product "  

    "showcase video, along with their timestamps and metadata. Answer the user's  

    question "  

    "based on what you can SEE in the frames and the provided metadata. Reference  

    specific "  

    "timestamps when relevant (e.g., 'at t=5.0s'). If the frames do not contain  

    enough "  

    "information, say so. Do not invent products or details not shown."  

)

def generate_answer_with_frames(question, results):
    """Send retrieved frames + metadata to a multimodal LLM via llm_cascade."""
    context_text = build_frame_context(results)
    prompt = f"{context_text}\n\nQUESTION: {question}"
    # Pass PIL frames directly; the cascade handles per-provider encoding.
    frames = [r["image"] for r in results]
    resp = llm.generate_multimodal(
        prompt, images=frames, system_prompt=VIDEO_RAG_SYSTEM_PROMPT)
    return resp.text, f"{resp.provider}/{resp.model}"

def generate_answer_without_rag(question):
    """Direct LLM answer with NO video frames."""
    return generate_text(question)

print("Answer generation functions ready (llm_cascade).")

```

```

Answer generation functions ready (llm_cascade).

```

11.2.7 7) Full Video RAG Pipeline

Putting it all together into a single callable function. The pipeline mirrors what we built for Image RAG – embed the query, retrieve relevant frames, generate an answer – but now each result includes a timestamp that grounds the answer in the video’s timeline. This is the complete Video RAG loop: from a natural-language question to a temporally grounded, visually informed answer.

```

def answer_with_video_rag(question, top_k=3):
    """Full Video RAG: extract frames -> CLIP retrieval -> multimodal LLM answer."""
    print(f" [Step 1] Embedding query with CLIP ...")
    results = retrieve_frames(
        question, frame_embeddings, keyframes, clip_model, clip_processor,
        top_k=top_k,
    )
    print(f" [Step 2] Retrieved {len(results)} frame(s):")
    for r in results:
        prod_name = r['product']['name'] if r['product'] else '(card)'
        print(f"          #{r['rank']} t={r['timestamp']:.1f}s - {prod_name} (sim={r['similarity']:.3f})")

    print(f" [Step 3] Generating multimodal answer ...")
    answer, provider = generate_answer_with_frames(question, results)
    print(f" [Step 4] Answer generated (provider: {provider})")
    return answer, results, provider

print("Video RAG pipeline ready.")

```


Video RAG pipeline ready.

11.2.8 8) Run End-to-End Examples: With RAG vs. Without RAG

We test questions about the product showcase video and compare:

- **With Video RAG:** CLIP retrieves the relevant frame(s), multimodal LLM answers with visual + temporal grounding
- **Without RAG:** LLM answers with no video context — it knows nothing about our product showcase

Watch how the without-RAG answers either refuse to answer (no context) or hallucinate product details.

```
questions = [
    "What is the red product shown in the video and how much does it cost?",
    #"At what point in the video does the green triangle appear?",
    #"What is the most expensive product in this showcase video?",
    #"What shape is the orange product?",
    "What text appears at the beginning of the video?",
]

def preview(text, max_len=800):
    text = text or ""
    return text[:max_len] + ("..." if len(text) > max_len else "")

if not has_llm_provider():
    print("Error: No LLM API key configured. Set at least one API key in Colab Secrets.")
else:
    for i, q in enumerate(questions, start=1):
        print("\n" + "=" * 80)
        print(f"Q{i}. {q}")
        print("=" * 80)

        # --- WITH VIDEO RAG ---
        print("\n--- WITH VIDEO RAG ---")
        try:
            answer_rag, retrieved, prov = answer_with_video_rag(q)
            show_retrieved_frames(q, retrieved)
            print(f"\n Answer (provider: {prov}):")
            print(preview(answer_rag))
        except Exception as e:
            print(f" RAG error: {e}")

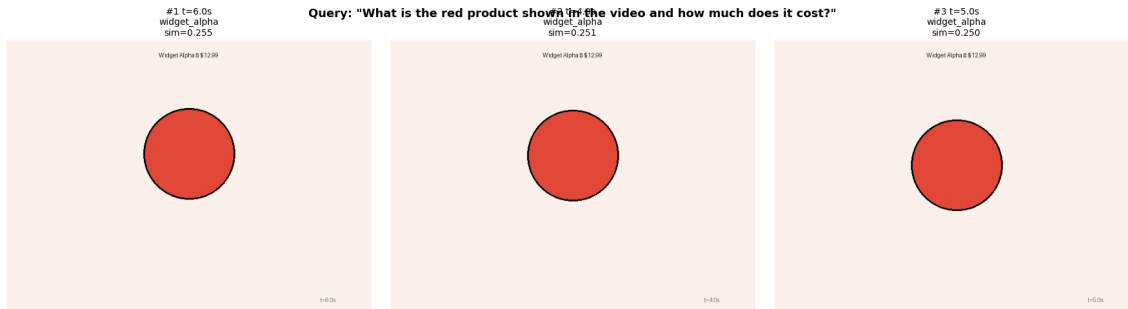
        # --- WITHOUT RAG ---
        print("\n--- WITHOUT RAG (LLM knowledge only) ---")
        try:
            answer_direct, prov_direct = generate_answer_without_rag(q)
            print(f"\n Answer (provider: {prov_direct}):")
            print(preview(answer_direct))
        except Exception as e:
            print(f" Direct error: {e}")

    print()
```

```
=====
Q1. What is the red product shown in the video and how much does it cost?
=====
```

```
--- WITH VIDEO RAG ---
```

```
[Step 1] Embedding query with CLIP ...
[Step 2] Retrieved 3 frame(s):
        #1 t=6.0s – Widget Alpha (sim=0.255)
        #2 t=4.0s – Widget Alpha (sim=0.251)
        #3 t=5.0s – Widget Alpha (sim=0.250)
[Step 3] Generating multimodal answer ...
[Multimodal response from Gemini / gemini-2.5-flash]
[Step 4] Answer generated (provider: Gemini/gemini-2.5-flash)
```



Answer (provider: Gemini/gemini-2.5-flash):
The red product shown in the video is a "Widget Alpha", as indicated by the metadata and text overlay in the frames (e.g., at t=6.0s). It costs \$12.99.

--- WITHOUT RAG (LLM knowledge only) ---

[Response from Gemini / gemini-2.5-flash]

Answer (provider: Gemini/gemini-2.5-flash):

I'm sorry, I cannot answer the question about the red product and its cost. The provided context does not include any information about a video or a red product. Therefore, I'm unable to identify what you're referring to or provide details about its cost.

=====

Q2. What text appears at the beginning of the video?

=====

--- WITH VIDEO RAG ---

[Step 1] Embedding query with CLIP ...

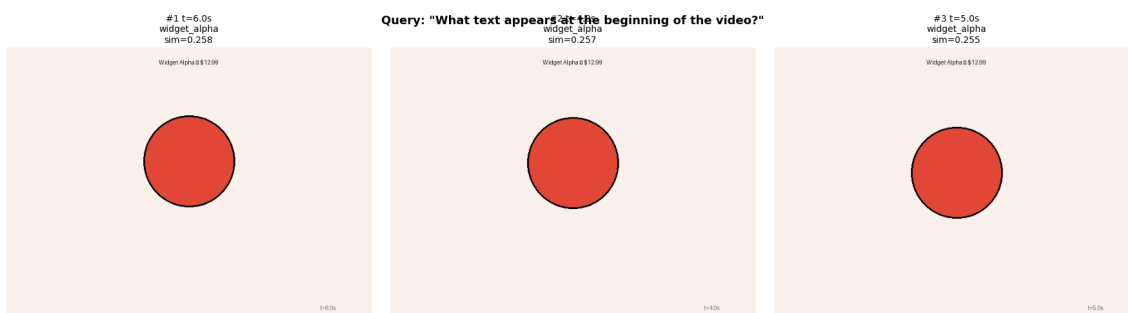
[Step 2] Retrieved 3 frame(s):

#1 t=6.0s – Widget Alpha (sim=0.258)
#2 t=4.0s – Widget Alpha (sim=0.257)
#3 t=5.0s – Widget Alpha (sim=0.255)

[Step 3] Generating multimodal answer ...

[Multimodal response from Gemini / gemini-2.5-flash]

[Step 4] Answer generated (provider: Gemini/gemini-2.5-flash)



Answer (provider: Gemini/gemini-2.5-flash):

The provided frames show the text "Widget Alpha \$12.99" at the top of the screen at timestamps $t=4.0s$, $t=5.0s$, and $t=6.0s$. However, these frames do not capture the very beginning of the video, so I cannot determine what text appeared at the absolute beginning.

```
--- WITHOUT RAG (LLM knowledge only) ---
[Response from Gemini / gemini-2.5-flash]
Answer (provider: Gemini/gemini-2.5-flash):
The video begins with the text "How do humans learn to speak?"
```

11.2.8.1 Checkpoint: Reflection Questions

1. **Temporal grounding:** Did the RAG answers include timestamps? How useful is knowing *when* in the video something appears?
2. **Without-RAG failure:** The LLM has never seen our synthetic video — what kind of responses did it give? Refusal? Hallucination?
3. **Frame sampling:** We sampled 1 frame/second. What would happen with 1 frame every 5 seconds? Every 0.5 seconds?
4. **Comparison with Image RAG:** What extra complexity does the temporal dimension add over static Image RAG?

11.2.9 9) Interactive Query (Optional)

Ask your own questions about the product showcase video.

```
# Change this to any question about the product showcase video
my_question = "How many products are shown in total, and what are their prices?"

if has_llm_provider():
    print(f"Question: {my_question}\n")

    print("=" * 60)
    print("WITH VIDEO RAG:")
    print("=" * 60)
    try:
        answer, retrieved, provider = answer_with_video_rag(my_question, top_k=4)
        show_retrieved_frames(my_question, retrieved)
        print(f"\nAnswer ({provider}): \n{answer}")
    except Exception as e:
        print(f"Error: {e}")

    print("\n" + "=" * 60)
    print("WITHOUT RAG:")
    print("=" * 60)
    try:
        answer_direct, prov_direct = generate_answer_without_rag(my_question)
        print(f"\nAnswer ({prov_direct}): \n{answer_direct}")
    except Exception as e:
        print(f"Error: {e}")
else:
    print("No LLM API key configured. Set at least one API key in Colab Secrets.")
```

Question: How many products are shown in total, and what are their prices?

=====

WITH VIDEO RAG:

=====

```
[Step 1] Embedding query with CLIP ...
[Step 2] Retrieved 4 frame(s):
#1 t=14.0s - Tool Gamma (sim=0.246)
#2 t=11.0s - Tool Gamma (sim=0.243)
```

```
#3 t=13.0s - Tool Gamma (sim=0.243)
#4 t=12.0s - Tool Gamma (sim=0.241)
[Step 3] Generating multimodal answer ...
[Gemini] unavailable, trying next...
Error: Error code: 404 - {'error': {'message': 'model "qwen2.5-vl:cloud" not found',
'type': 'not_found_error', 'param': None, 'code': None}}
```

=====

WITHOUT RAG:

=====

[Response from Gemini / gemini-2.5-flash]

Answer (Gemini/gemini-2.5-flash):
Let's count them! There are 5 products shown in total.

Here are the products and their prices:

1. ****Hand Cream:**** \$15
2. ****Body Wash:**** \$25
3. ****Body Lotion:**** \$22
4. ****Body Scrub:**** \$30
5. ****Bath Bomb Set:**** \$18

11.2.10 10) Teaching Notes and Exercises

Key takeaways:

- **Video RAG = frame extraction + Image RAG + temporal metadata.** The video is decomposed into keyframes that are indexed and retrieved just like static images.
- **Timestamps add a temporal dimension** that Image RAG lacks — answers can reference *when* in the video something occurs.
- **Frame sampling rate** is a key design choice: too sparse and you miss content; too dense and you waste compute and storage.
- The same **CLIP + multimodal LLM** stack from Image RAG works here — the main new component is the frame extraction step.
- **Without-RAG comparison** is especially stark for video: the LLM has never seen the video, so it either refuses or hallucinates.

Exercises:

- Download a short YouTube clip (e.g., a product review) and replace the synthetic video. How does CLIP handle real-world video frames?
- Experiment with different frame sampling rates (0.5s, 2s, 5s) and measure the impact on retrieval accuracy.
- Add **scene change detection** instead of fixed-interval sampling — extract frames only when the visual content changes significantly.
- Combine video frames with **audio transcription** (e.g., Whisper) for a multimodal RAG that uses both visual and spoken content.
- Discuss: In what business scenarios is Video RAG most valuable? (e.g., surveillance review, training video search, meeting recording Q&A, sports analytics)

11.2.11 Key takeaways

- **Video RAG** reduces to Image RAG once you extract keyframes — the same CLIP + cosine-similarity pipeline handles retrieval.
- **Frame sampling rate** trades coverage for efficiency; one frame per second is a reasonable default for most content.
- **Timestamps** attached to each frame let the LLM ground its answer in *when* something happens, not just *what* appears.
- **Multimodal LLMs** receive the actual retrieved frames as images, enabling answers that read both visuals and on-screen text.

- **Without RAG**, the model has never seen your video and will refuse or hallucinate – retrieval is what supplies the evidence.

11.2.12 Optional: Raw Gemini Video API Call

The `llm.generate_multimodal()` method sends **frames** to whichever vision-capable provider is available – Gemini, Ollama Cloud, HuggingFace, or OpenRouter. But Gemini has a unique capability: it can accept the **entire video file** as input (not just extracted frames) and do its own internal frame sampling. This works because Gemini’s native API supports video MIME types like `video/mp4` directly.

The code below shows how that looks. It’s Gemini-only – OpenAI-compatible endpoints do not accept video bytes, which is why the cascade restricts video inputs to Gemini. You might prefer this approach for short videos where letting Gemini choose its own frames beats a fixed one-frame-per-second strategy.

```
# Direct Gemini call on the whole video file -- Gemini-only feature that the
# cascade does not exercise for other providers.
from google import genai
from google.genai import types as genai_types
from llm_cascade.providers import _get_key

gemini_key = _get_key("GEMINI_API_KEY")
if not gemini_key:
    print("Skipping: GEMINI_API_KEY not configured.")
else:
    question = "What products appear in this video and in what order?"
    video_bytes = Path(VIDEO_PATH).read_bytes()

    client = genai.Client(api_key=gemini_key)
    parts = [
        genai_types.Part.from_text(text=question),
        genai_types.Part.from_bytes(data=video_bytes, mime_type="video/mp4"),
    ]
    resp = client.models.generate_content(
        model="gemini-2.5-flash",
        contents=parts,
        config=genai_types.GenerateContentConfig(
            thinking_config=genai_types.ThinkingConfig(thinking_budget=0),
        ),
    )
    print(f"Question: {question}")
    print(f"Video: {VIDEO_PATH} ({Path(VIDEO_PATH).stat().st_size // 1024} KB)")
    print(f"\nAnswer:\n{resp.text}")
```

Question: What products appear in this video and in what order?
Video: product_showcase.mp4 (260 KB)

Answer:
Here are the products that appear in the video, in order of appearance:

1. **Widget Alpha** (a red circle)
2. **Gadget Beta** (a green triangle)
3. **Tool Gamma** (a blue square)
4. **Device Delta** (an orange diamond)

11.2.13 Run the code

To run this notebook, copy the URL below into your browser’s address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/Video_RAG.ipynb

Chapter 12

LlamaIndex

LlamaIndex is a framework that abstracts the entire RAG pipeline into a few lines of code. Where earlier chapters built everything manually (chunking, embedding, retrieval, prompting), LlamaIndex handles it all with `SimpleDirectoryReader` + `VectorStoreIndex` + `query_engine`.

This chapter demonstrates LlamaIndex with three fictional company document sets, showing how the framework automatically routes questions to the correct company's documents.

12.1 LlamaIndex RAG Demo: Document Q&A with a Framework



Estimated run time: ~5 minutes (document ingestion + LLM calls)

This notebook demonstrates how **LlamaIndex** simplifies building RAG pipelines. Instead of manually chunking documents, computing embeddings, and querying vector stores, LlamaIndex handles the plumbing so you can focus on the application logic.

How it works:

1. Create a small set of **synthetic company documents** (no external data needed).
2. Use LlamaIndex to **ingest, chunk, embed, and index** the documents in one step.
3. Given a natural-language question, LlamaIndex **retrieves** relevant chunks and sends them to an LLM.
4. Compare answers **with RAG** (LlamaIndex pipeline) vs. **without RAG** (direct LLM call).

Learning goals:

- Understand what a RAG framework (LlamaIndex) does under the hood
- See how LlamaIndex abstracts document loading, chunking, embedding, and retrieval
- Compare the framework approach to the manual RAG we built in earlier notebooks
- Observe how RAG grounding eliminates hallucination on private/synthetic data

Provider setup: This notebook uses the `llm_cascade` package, which auto-detects your API keys and falls back to the next provider if one is unavailable. Supported providers: OpenAI, Gemini, Ollama, Grok (xAI), Groq, HuggingFace, Cohere, OpenRouter.

Store any of these API keys in Colab Secrets (or a local `.env` file): `OPENAI_API_KEY`, `GEMINI_API_KEY`, `OLLAMA_API_KEY`, `XAI_API_KEY`, `GROQ_API_KEY`, `HF_TOKEN`, `COHERE_API_KEY`, `OPENROUTER_API_KEY`

Think of LlamaIndex as a convenience wrapper around the seven manual steps you wrote out in Chapter 8 and the ChromaDB/LanceDB/Pinecone pipelines from Chapter 9 — same chunking, same embeddings, same retrieval, just expressed in two or three lines instead of fifty.

```
!pip install -q -U llama-index llama-index-llms-openai-like llama-index-llms-gemini
llama-index-embeddings-huggingface llama-index-readers-github google-genai openai
git+https://github.com/KarAnalytics/llm_cascade.git sentence-transformers
```

12.1.1 1) Imports and Provider Helpers (7-Vendor Cascade)

We configure **LlamaIndex** to use whichever LLM provider you have available. The cascade tries each provider in order and falls through on quota errors, ensuring the notebook works regardless of which API keys you have set up.

For **embeddings**, we use a local HuggingFace model (`all-MiniLM-L6-v2`) so that embedding always works regardless of which LLM API key you have. This is a deliberate design choice: by keeping the embedding model local, we avoid the situation where a rate-limited API blocks the entire pipeline. The LLM is the only component that needs an external API call.

```
from pathlib import Path
from llm_cascade.providers import PROVIDERS, _load_env, _get_key,
_is_retrievable_error

_load_env()
```



```
def get_available_providers():
    return [p for p in PROVIDERS if _get_key(p['key_env'])]

def has_llm_provider():
    return len(get_available_providers()) > 0

# Print status
available = get_available_providers()
if available:
    print('Providers configured (in fallback order):')
    for p in available:
        print(f"  + {p['name']:<16} model = {p['default_model']}")
else:
    print('WARNING: No API keys found.')
```

```
WARNING: No API keys found.
```

12.1.2 2) Configure LlamaIndex LLM and Embeddings

LlamaIndex needs two things configured before it can build a RAG pipeline:

- An **LLM** for generating answers (we pick the first available provider from our cascade)
- An **embedding model** for converting text to vectors (we use a local HuggingFace model – no API key needed)

This cell configures LlamaIndex's global `Settings` object so all downstream components use our chosen models. The `CascadeLLM` wrapper you see below is necessary because LlamaIndex expects an LLM object that conforms to its own interface, but our `llm_cascade` package has its own API. The wrapper bridges the two, translating LlamaIndex's `complete()` calls into `llm_cascade`'s `generate()` calls. Without this adapter, we would be locked into a single provider and lose the automatic fallback behavior that makes these notebooks robust across different API key configurations.

```
from llama_index.core import Settings
from llama_index.core.llms import CustomLLM, LLMMetadata, CompletionResponse,
CompletionResponseGen
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llm_cascade import get_cascade
from typing import Any

# ---- Embedding model (local, no API key needed) ----
Settings.embed_model = HuggingFaceEmbedding(model_name="sentence-transformers/all-
MiniLM-L6-v2")
print("Embedding model: sentence-transformers/all-MiniLM-L6-v2 (local)")

# ---- Custom LlamaIndex LLM that wraps llm_cascade for automatic fallback ----
class CascadeLLM(CustomLLM):
    """LlamaIndex LLM that delegates to llm_cascade (8-provider fallback)."""
    context_window: int = 8000
    num_output: int = 1024
    model_name: str = "llm_cascade"

    @property
    def metadata(self) -> LLMMetadata:
        return LLMMetadata(
            context_window=self.context_window,
            num_output=self.num_output,
            model_name=self.model_name,
        )

    def complete(self, prompt: str, **kwargs: Any) -> CompletionResponse:
        cascade = get_cascade(verbose=False)
```

```

        response = cascade.generate(prompt)
        return CompletionResponse(text=response.text, additional_kwargs={"provider":
response.provider, "model": response.model})

    def stream_complete(self, prompt: str, **kwargs: Any) -> CompletionResponseGen:
        # Non-streaming fallback
        resp = self.complete(prompt, **kwargs)
        def gen():
            yield resp
        return gen()

Settings.llm = CascadeLLM()
print("LLM: CascadeLLM (auto-fallback across available providers)")

```

```
modules.json: 0%|          | 0.00/349 [00:00<?, ?B/s]
```

```
config_sentence_transformers.json: 0%|          | 0.00/116 [00:00<?, ?B/s]
```

```
README.md: 0.00B [00:00, ?B/s]
```

```
sentence_bert_config.json: 0%|          | 0.00/53.0 [00:00<?, ?B/s]
```

```
config.json: 0%|          | 0.00/612 [00:00<?, ?B/s]
```

```
model.safetensors: 0%|          | 0.00/90.9M [00:00<?, ?B/s]
```

```
Loading weights: 0%|          | 0/103 [00:00<?, ?it/s]
```

```
BertModel LOAD REPORT from: sentence-transformers/all-MiniLM-L6-v2
```

Key	Status		
-----+-----+-----+-----			
embeddings.position_ids	UNEXPECTED		

Notes:

- UNEXPECTED :can be ignored when loading from different task/architecture; not ok if you expect identical arch.

```
tokenizer_config.json: 0%|          | 0.00/350 [00:00<?, ?B/s]
```

```
tokenizer_config.json: 0%|          | 0.00/350 [00:00<?, ?B/s]
```

```
vocab.txt: 0.00B [00:00, ?B/s]
```

```
tokenizer.json: 0.00B [00:00, ?B/s]
```

```
special_tokens_map.json: 0%|          | 0.00/112 [00:00<?, ?B/s]
```

```
config.json: 0%|          | 0.00/190 [00:00<?, ?B/s]
```

```
Embedding model: sentence-transformers/all-MiniLM-L6-v2 (local)
LLM: CascadeLLM (auto-fallback across available providers)
```

12.1.3 3) Create Synthetic Company Documents

We generate **fictional documents for three companies** that the LLM has never seen in training. This ensures that correct answers can **only** come from RAG retrieval, not from parametric memory. Each company's documents live in a **separate directory**, simulating real-world data silos.

Directory	Company	Documents
acme_docs/	Acme Analytics Inc. (BI tools, Lawrence KS)	Overview, Products, Financials, Hand-book, Case Study
globex_docs/	Globex Cybersecurity Corp. (cybersecurity, Boston)	Overview, Products, Incident Report
nextera_docs/	Nextera Green Solutions (sustainability consulting, Denver)	Overview, Services, Impact Report

Later, we will use LlamaIndex **Data Loaders** (`SimpleDirectoryReader`) to ingest all three directories — demonstrating how LlamaIndex can pull information from multiple, separate data sources and unify them in a single searchable index.

```
DOCUMENTS = {
    "company_overview.txt": """ACME ANALYTICS INC. – COMPANY OVERVIEW

Acme Analytics Inc. was founded in 2019 by Dr. Sarah Chen and Marcus Rivera
in Lawrence, Kansas. The company specializes in AI-powered business intelligence
tools for mid-market companies (100–5,000 employees).

Headquarters: 1420 Jayhawk Boulevard, Lawrence, KS 66045
Employees: 287 (as of January 2026)
Annual Revenue (2025): $42.3 million
Funding: Series B ($18M raised in March 2023 from Midwest Ventures)

Mission: "To democratize data analytics so every business decision is informed
by evidence, not intuition."

The company operates three offices: Lawrence (HQ), Chicago (sales), and
Austin (engineering). CEO: Dr. Sarah Chen. CT0: Marcus Rivera. CF0: Linda Park."""

    "products_and_services.txt": """ACME ANALYTICS – PRODUCTS AND SERVICES

1. InsightBoard Pro (Flagship Product)
- Real-time dashboard platform with natural language query interface
- Pricing: $49/user/month (Standard), $89/user/month (Enterprise)
- Supports PostgreSQL, MySQL, Snowflake, BigQuery, and Redshift
- 1,847 active enterprise customers as of Q4 2025

2. DataPipe ETL
- Automated data pipeline builder with 200+ pre-built connectors
- Pricing: starts at $500/month for up to 10 million records/day
- Launched in September 2024

3. PredictIQ
- ML-powered forecasting add-on for InsightBoard Pro
- Pricing: $29/user/month (requires InsightBoard Pro subscription)
- Uses proprietary time-series model trained on retail and logistics data
- Beta launched March 2025, GA release planned for June 2026

All products include 24/7 email support. Enterprise plans include dedicated
account manager and 99.9% SLA."""

    "q4_2025_financials.txt": """ACME ANALYTICS – Q4 2025 FINANCIAL RESULTS
(CONFIDENTIAL)
```

Period: October 1 – December 31, 2025

Revenue: \$12.8 million (Q4) / \$42.3 million (FY 2025)
Gross Margin: 78.2%
Operating Profit: \$1.9 million (Q4) / \$5.1 million (FY 2025)
Net Income: \$1.4 million (Q4)
Cash on Hand: \$14.7 million
Burn Rate: Company is cash-flow positive since Q2 2025

Key Metrics:

- ARR (Annual Recurring Revenue): \$48.6 million (up 34% YoY)
- Net Revenue Retention: 118%
- Customer Acquisition Cost (CAC): \$8,200
- Customer Lifetime Value (LTV): \$67,400
- LTV/CAC Ratio: 8.2x

Headcount grew from 241 to 287 employees during 2025.

R&D spending: 31% of revenue. Sales & Marketing: 28% of revenue.""",

"employee_handbook_excerpt.txt": ""ACME ANALYTICS – EMPLOYEE HANDBOOK (EXCERPT)

PAID TIME OFF (PTO):

- All full-time employees receive 22 days of PTO per year (accrued monthly).
- PTO increases to 27 days after 3 years of service.
- Unused PTO can be carried over (max 5 days) or paid out at year-end.
- Sick leave: 10 days per year (separate from PTO).

REMOTE WORK POLICY:

- Engineering and Data Science teams: fully remote eligible.
- Sales and Customer Success: hybrid (minimum 2 days/week in office).
- All employees may work remotely up to 4 weeks/year from any US location.
- International remote work requires VP approval and tax review.

PROFESSIONAL DEVELOPMENT:

- Annual learning budget: \$2,500 per employee.
- Conference attendance: up to 2 conferences per year with manager approval.
- Tuition reimbursement: up to \$5,250/year for degree programs.""",

"customer_case_study.txt": ""CUSTOMER CASE STUDY: BIGRETAIL CORP

Company: BigRetail Corp (1,200 retail stores across 38 states)

Challenge: Siloed data across POS, inventory, and CRM systems made it impossible for regional managers to get timely insights.

Solution: Deployed InsightBoard Pro Enterprise + DataPipe ETL

- Connected 14 data sources in 3 weeks using DataPipe
- 340 regional managers now use InsightBoard daily
- Natural language queries replaced manual SQL report requests

Results (after 6 months):

- 62% reduction in time-to-insight (from 4 days to 1.5 days average)
- \$3.2 million saved in inventory carrying costs
- 23% increase in regional manager satisfaction scores
- SQL report request backlog eliminated entirely

Quote from BigRetail CIO Janet Torres:

\\"InsightBoard Pro transformed how our managers interact with data. They went from waiting days for a report to asking questions in plain English and getting answers in seconds.\""",
}

```
# Write documents to disk  
DOC_DIR = Path("acme_docs")
```

```
DOC_DIR.mkdir(exist_ok=True)

for filename, content in DOCUMENTS.items():
    (DOC_DIR / filename).write_text(content, encoding="utf-8")

print(f"Created {len(DOCUMENTS)} documents in '{DOC_DIR}/':")
for f in sorted(DOC_DIR.iterdir()):
    print(f"  {f.name} ({f.stat().st_size} bytes)")
```

```
Created 5 documents in 'acme_docs/':
  company_overview.txt (715 bytes)
  customer_case_study.txt (924 bytes)
  employee_handbook_excerpt.txt (828 bytes)
  products_and_services.txt (915 bytes)
  q4_2025_financials.txt (709 bytes)
```

```
# --- Company 2: Globex Cybersecurity Corp. → globex_docs/ ---
```

```
GLOBEX_DOCUMENTS = {
    "globex_overview.txt": """GLOBEX CYBERSECURITY CORP. – COMPANY OVERVIEW

Globex Cybersecurity Corp. was founded in 2017 by former NSA analyst
James Whitfield and AI researcher Dr. Priya Narayanan in Boston, Massachusetts.
The company provides AI-driven cybersecurity solutions for financial services
and healthcare organizations.

Headquarters: 88 Federal Street, Suite 1200, Boston, MA 02110
Employees: 412 (as of January 2026)
Annual Revenue (2025): $78.5 million
Funding: Series C ($45M raised in June 2024 from CyberVentures Capital
and Goldman Sachs Growth Equity)

Mission: "To make enterprise-grade cybersecurity autonomous, adaptive,
and accessible to every regulated industry."

The company operates four offices: Boston (HQ), Washington D.C. (government
relations), Tel Aviv (R&D), and Singapore (APAC sales).
CEO: James Whitfield. CTO: Dr. Priya Narayanan. CISO: Robert Tanaka.""",
    "globex_products.txt": """GLOBEX CYBERSECURITY – PRODUCTS AND SERVICES

1. ShieldAI Platform (Flagship Product)
  - AI-powered threat detection and automated incident response
  - Uses behavioral analysis to detect zero-day attacks in real time
  - Pricing: $120/endpoint/year (Standard), $195/endpoint/year (Premium)
  - 623 enterprise customers across 18 countries as of Q4 2025
  - Average detection time: 2.3 minutes (industry average: 197 days)

2. VaultGuard
  - Data encryption and access management platform for regulated industries
  - HIPAA, SOX, and PCI-DSS compliant out of the box
  - Pricing: $35,000/year (up to 500 users), $60,000/year (unlimited users)
  - Launched January 2023

3. ThreatIntel Live
  - Real-time threat intelligence feed with dark web monitoring
  - Pricing: $8,500/month (includes API access and analyst dashboard)
  - Aggregates data from 2,400+ sources updated every 15 minutes
  - Integrates with Splunk, CrowdStrike, Palo Alto, and Microsoft Sentinel

All products include a dedicated Security Operations Center (SOC) team for
```

```
Premium customers. Globex holds ISO 27001 and SOC 2 Type II certifications.""",  
  
    "globex_incident_report.txt": """"GLOBEX CYBERSECURITY – 2025 THREAT LANDSCAPE  
REPORT (PUBLIC)
```

```
Published: February 2026
```

```
KEY FINDINGS FROM GLOBEX'S CUSTOMER BASE (2025):
```

```
Total incidents analyzed: 14,237 across 623 customers  
- Ransomware attacks: up 42% YoY (3,891 incidents)  
- Phishing campaigns: up 28% YoY (5,102 incidents)  
- Supply chain attacks: up 67% YoY (1,245 incidents – fastest-growing category)  
- Insider threats: relatively stable at 1,834 incidents
```

```
ShieldAI Platform Performance:
```

```
- Blocked 97.3% of attacks before any data exfiltration  
- Mean time to detect (MTTD): 2.3 minutes  
- Mean time to respond (MTTR): 8.7 minutes (automated response)  
- False positive rate: 0.4% (down from 1.2% in 2024)
```

```
NOTABLE CASE: MERIDIAN HEALTH SYSTEM
```

```
- 340-bed hospital network targeted by Medusa ransomware variant  
- ShieldAI detected anomalous lateral movement within 47 seconds  
- Automated containment isolated 3 affected servers in under 2 minutes  
- Zero patient data exfiltrated; zero downtime for clinical systems  
- Estimated savings: $12M+ (average healthcare ransomware cost per Ponemon)
```

```
INDUSTRY PREDICTIONS FOR 2026:
```

```
- AI-generated phishing emails will become indistinguishable from human-written  
- Quantum-resistant encryption adoption will begin in financial services  
- Average cost of a data breach expected to exceed $5.2 million globally""",  
}
```

```
GLOBEX_DIR = Path("globex_docs")  
GLOBEX_DIR.mkdir(exist_ok=True)
```

```
for filename, content in GLOBEX_DOCUMENTS.items():  
    (GLOBEX_DIR / filename).write_text(content, encoding="utf-8")  
  
print(f"Created {len(GLOBEX_DOCUMENTS)} documents in '{GLOBEX_DIR}/':")  
for f in sorted(GLOBEX_DIR.iterdir()):  
    print(f"  {f.name} ({f.stat().st_size} bytes)")
```

```
Created 3 documents in 'globex_docs':  
  globex_incident_report.txt (1321 bytes)  
  globex_overview.txt (864 bytes)  
  globex_products.txt (1131 bytes)
```

```
# --- Company 3: Nextera Green Solutions → nextera_docs/ ---
```

```
NEXTERA_DOCUMENTS = {  
    "nextera_overview.txt": """"NEXTERA GREEN SOLUTIONS – COMPANY OVERVIEW
```

```
Nextera Green Solutions was founded in 2020 by climate scientist Dr. Elena  
Vasquez and former McKinsey partner David Okafor in Denver, Colorado.  
The company provides sustainability consulting and carbon accounting software  
to help companies meet ESG (Environmental, Social, Governance) targets.
```

```
Headquarters: 1700 Lincoln Street, Suite 3400, Denver, CO 80203  
Employees: 156 (as of January 2026)
```

Annual Revenue (2025): \$23.8 million

Funding: Series A (\$12M raised in November 2023 from Climate Capital Partners and Patagonia Ventures)

Mission: "To give every company a clear, data-driven path to net zero – turning climate commitments into measurable action."

The company operates two offices: Denver (HQ) and London (European operations).

CEO: Dr. Elena Vasquez. COO: David Okafor. VP Engineering: Kenji Tanaka.

Nextera is a certified B Corporation and a member of the UN Global Compact."",

"nextera_services.txt": ""NEXTERA GREEN SOLUTIONS – SERVICES AND PLATFORM

1. CarbonLens Platform (Flagship Product)

- Automated Scope 1, 2, and 3 carbon emissions tracking
- Integrates with ERP systems (SAP, Oracle, NetSuite) to pull real data
- AI-powered recommendations for emission reduction strategies
- Pricing: \$2,500/month (Standard, up to \$500M revenue companies)
\$6,500/month (Enterprise, unlimited + custom reporting)
- 312 active customers as of Q4 2025

2. ESG Reporting Suite

- Automated generation of sustainability reports (GRI, SASB, TCFD, CDP)
- Audit-ready documentation with full data lineage
- Pricing: \$18,000/year (bundled with CarbonLens Enterprise)
- Standalone: \$24,000/year
- Launched April 2024

3. Sustainability Strategy Consulting

- Science-Based Targets initiative (SBTi) alignment services
- Supply chain decarbonization roadmaps
- Pricing: Project-based (\$75,000 – \$250,000 per engagement)
- Team of 28 consultants with backgrounds in climate science and industry

4. Carbon Offset Marketplace

- Verified carbon credits from 45 projects across 12 countries
- Blockchain-verified chain of custody for each credit
- Transaction fee: 4.5% per credit purchase
- Beta launched August 2025

All platform customers receive quarterly sustainability benchmarking reports comparing their performance against industry peers."",

"nextera_impact_report.txt": ""NEXTERA GREEN SOLUTIONS – 2025 IMPACT REPORT

Published: January 2026

COMPANY IMPACT METRICS:

- Total customer carbon emissions tracked: 47.3 million metric tons CO2e
- Verified emission reductions enabled: 6.8 million metric tons CO2e (14.4%)
- Customers who improved their CDP score after using Nextera: 89%
- Average time to produce ESG report reduced from 14 weeks to 3.5 weeks

NOTABLE CLIENT RESULTS:

1. SummitBrew Coffee (specialty coffee chain, 380 locations)

- Reduced Scope 3 supply chain emissions by 31% in 18 months
- Switched 62% of logistics to electric vehicles using Nextera roadmap
- First coffee chain to achieve SBTi-validated near-term target

2. Pacific Northwest Lumber Co. (timber and building materials)

- Used CarbonLens to discover 40% of emissions came from transportation
- Optimized shipping routes: saved 8,200 metric tons CO2e and \$1.4M/year

- Achieved carbon-neutral certification for 2025 operations
3. Meridian Financial Group (regional bank, \$8B assets)
- Implemented financed emissions tracking (PCAF methodology)
 - Identified \$2.1B in loan portfolio exposed to high transition risk
 - Developed green lending program: \$340M in sustainability-linked loans

AWARDS AND RECOGNITION (2025):

- Fast Company "Most Innovative Companies" – Energy category (#7)
- B Corp "Best for the World" – Environment category
- Colorado Governor's Award for Clean Energy Innovation""

}

```
NEXTERA_DIR = Path("nextera_docs")
NEXTERA_DIR.mkdir(exist_ok=True)
```

```
for filename, content in NEXTERA_DOCUMENTS.items():
    (NEXTERA_DIR / filename).write_text(content, encoding="utf-8")

print(f"Created {len(NEXTERA_DOCUMENTS)} documents in '{NEXTERA_DIR}/':")
for f in sorted(NEXTERA_DIR.iterdir()):
    print(f" {f.name} ({f.stat().st_size} bytes)")
```

```
Created 3 documents in 'nextera_docs/':
nextera_impact_report.txt (1434 bytes)
nextera_overview.txt (938 bytes)
nextera_services.txt (1394 bytes)
```

12.1.4 4) Ingest Documents Using LlamaIndex Data Loaders (via LlamaHub)

A key feature of LlamaIndex is its **Data Loaders** available via [LlamaHub](#). Data Loaders can pull information from **hundreds of sources** – Google Docs, Notion, **GitHub**, Slack, databases, PDFs, or simple `.txt` files. Think of LlamaHub as a plugin ecosystem: instead of writing custom ingestion code for each data source, you install a loader and point it at your data.

Here we use `SimpleDirectoryReader` (the built-in Data Loader for local files) to load documents from **three separate company directories** into a single searchable index. Under the hood, `SimpleDirectoryReader` reads each file, wraps it in a `LlamaIndex Document` object with metadata (file name, path), and passes it to `VectorStoreIndex`. The index then automatically chunks each document, computes embeddings using our configured HuggingFace model, and stores everything in an in-memory vector store. All of that – chunking, embedding, indexing – happens in a single `VectorStoreIndex.from_documents()` call.

Want to try the GitHub Data Loader? LlamaHub also offers `GithubRepositoryReader`, which reads files directly from a GitHub repository – no `git clone` needed. See the commented-out code below to try it yourself. You just need a free GitHub token (any GitHub account, no special permissions needed for public repos): <https://github.com/settings/tokens?type=beta>

Loader	Source	Package	Used here?
<code>SimpleDirectoryReader</code>	Local <code>.txt</code> files	<code>llama-index</code> (built-in)	Yes (default)
<code>GithubRepositoryReader</code>	GitHub repo	<code>llama-index-readers-github</code> (LlamaHub)	Commented-out example
<code>GoogleDriveReader</code>	Google Drive	<code>llama-index-readers-google</code>	Available via LlamaHub
<code>NotionPageReader</code>	Notion pages	<code>llama-index-readers-notion</code>	Available via LlamaHub
<code>ConfluenceReader</code>	Atlassian Confluence	<code>llama-index-readers-confluence</code>	Available via LlamaHub
<code>S3Reader</code>	AWS S3 buckets	<code>llama-index-readers-s3</code>	Available via LlamaHub

Loader	Source	Package	Used here?
...and 300+ more			

```

from llama_index.core import SimpleDirectoryReader, VectorStoreIndex

# =====
# DEFAULT: Load documents from local directories using SimpleDirectoryReader
# =====

LOCAL_DIRS = {
    "acme_docs": "Acme Analytics (BI tools)",
    "globex_docs": "Globex Cybersecurity (security)",
    "nextera_docs": "Nextera Green Solutions (sustainability)",
}

all_documents = []
for dir_name, label in LOCAL_DIRS.items():
    loader = SimpleDirectoryReader(dir_name)
    docs = loader.load_data()
    print(f" Loaded: '{dir_name}/' → {len(docs)} docs  [{label}]")
    all_documents.extend(docs)

print(f"Total documents loaded from {len(LOCAL_DIRS)} directories:
{len(all_documents)}")

# =====
# OPTIONAL: Try the GitHub Data Loader instead (LlamaHub)
#
# This reads the SAME documents directly from a GitHub repo – no git clone
# needed. To try it:
# 1. Generate a free GitHub token: https://github.com/settings/tokens?type=beta
#    (any GitHub account works; no special permissions needed for public repos)
# 2. Add it to your .env file: GITHUB_TOKEN=github_pat_xxxx
#    Or in Colab: add a Secret named GITHUB_TOKEN
# 3. Uncomment the code below and comment out the SimpleDirectoryReader above
# =====

# from llama_index.readers.github import GithubRepositoryReader, GithubClient
#
# github_client = GithubClient(github_token=_get_key("GITHUB_TOKEN"), verbose=False)
#
# GITHUB_DIRS = {
#     "acme_docs": "Acme Analytics (BI tools)",
#     "globex_docs": "Globex Cybersecurity (security)",
#     "nextera_docs": "Nextera Green Solutions (sustainability)",
# }
#
# all_documents = []
# for dir_path, label in GITHUB_DIRS.items():
#     reader = GithubRepositoryReader(
#         github_client=github_client,
#         owner="KarAnalytics", # GitHub username
#         repo="llamaindexdata", # repo name
#         filter_directories=[dir_path],
#         GithubRepositoryReader.FilterType.INCLUDE),
#         verbose=False,
#     )
#     docs = reader.load_data(branch="main")
#     print(f" GitHub Loader: '{dir_path}/' → {len(docs)} docs  [{label}]")
#     all_documents.extend(docs)
#
# -----

```

```
# Build a SINGLE vector index across all three companies
# -----
index = VectorStoreIndex.from_documents(all_documents, show_progress=True)
print(f"Vector index built with {len(all_documents)} chunks")

# Create a query engine
query_engine = index.as_query_engine(similarity_top_k=3)
print("Query engine ready – ask questions about ANY of the three companies!")
```

```
WARNING:llama_index.core.readers.file.base:`llama-index-readers-file` package not
found, some file readers will not be available if not provided by the
`file_extractor` parameter.
WARNING:llama_index.core.readers.file.base:`llama-index-readers-file` package not
found, some file readers will not be available if not provided by the
`file_extractor` parameter.
WARNING:llama_index.core.readers.file.base:`llama-index-readers-file` package not
found, some file readers will not be available if not provided by the
`file_extractor` parameter.
WARNING:llama_index.core.readers.file.base:`llama-index-readers-file` package not
found, some file readers will not be available if not provided by the
`file_extractor` parameter.
WARNING:llama_index.core.readers.file.base:`llama-index-readers-file` package not
found, some file readers will not be available if not provided by the
`file_extractor` parameter.
WARNING:llama_index.core.readers.file.base:`llama-index-readers-file` package not
found, some file readers will not be available if not provided by the
`file_extractor` parameter.
WARNING:llama_index.core.readers.file.base:`llama-index-readers-file` package not
found, some file readers will not be available if not provided by the
`file_extractor` parameter.
WARNING:llama_index.core.readers.file.base:`llama-index-readers-file` package not
found, some file readers will not be available if not provided by the
`file_extractor` parameter.
WARNING:llama_index.core.readers.file.base:`llama-index-readers-file` package not
found, some file readers will not be available if not provided by the
`file_extractor` parameter.
WARNING:llama_index.core.readers.file.base:`llama-index-readers-file` package not
found, some file readers will not be available if not provided by the
`file_extractor` parameter.
```

```
Loaded: 'acme_docs/' → 5 docs [Acme Analytics (BI tools)]
Loaded: 'globex_docs/' → 3 docs [Globex Cybersecurity (security)]
Loaded: 'nextera_docs/' → 3 docs [Nextera Green Solutions (sustainability)]
Total documents loaded from 3 directories: 11
```

```
Applying transformations: 0%|          | 0/1 [00:00<?, ?it/s]
```

```
Generating embeddings: 0%|          | 0/11 [00:00<?, ?it/s]
```

```
Vector index built with 11 chunks
Query engine ready – ask questions about ANY of the three companies!
```

Notice how few lines of code that took. Two calls – `SimpleDirectoryReader` to load, `VectorStoreIndex.from_documents()` to index – replaced all the manual chunking, embedding, and vector store insertion we did in earlier notebooks. This is the core value proposition of LlamaIndex: it abstracts the RAG plumbing so you can focus on choosing the right data sources and tuning retrieval quality.

12.1.5 5) What Just Happened? Peeking Under the Hood

The two lines of code above – `SimpleDirectoryReader` and `VectorStoreIndex.from_documents()` – did a remarkable amount of work. They loaded 11 text files, split them into chunks, computed a 384-dimensional embedding for each chunk using our local HuggingFace model, and organized those embeddings into a searchable index. If you built RAG manually in the earlier notebooks, you know that each of those steps required explicit code: reading files, calling a text splitter, batching through an embedding API, and inserting into a vector store.

Let us inspect what `LlamaIndex` created so we can verify that the abstraction is doing sensible things. The cell below runs a test retrieval query and shows which chunks came back, including their similarity scores and source files. This transparency is important: frameworks are powerful, but you should always be able to look inside and confirm that the right documents are being retrieved for the right reasons.

```
# Inspect the chunks LlamaIndex created
retriever = index.as_retriever(similarity_top_k=3)

# Test retrieval without LLM – just see what chunks come back
test_query = "How much revenue did Acme make in Q4 2025?"
retrieved_nodes = retriever.retrieve(test_query)

print(f"Query: '{test_query}'")
print(f"Retrieved {len(retrieved_nodes)} chunks:\n")
for i, node in enumerate(retrieved_nodes, 1):
    source = node.metadata.get('file_name', 'unknown')
    score = node.score
    text_preview = node.text[:200].replace('\n', ' ')
    print(f"  Chunk #{i} (score={score:.4f}, source={source})")
    print(f"    {text_preview}...\n")
```

Query: 'How much revenue did Acme make in Q4 2025?'

Retrieved 3 chunks:

Chunk #1 (score=0.6737, source=q4_2025_financials.txt)
ACME ANALYTICS – Q4 2025 FINANCIAL RESULTS (CONFIDENTIAL) Period: October 1 – December 31, 2025 Revenue: \$12.8 million (Q4) / \$42.3 million (FY 2025)
Gross Margin: 78.2% Operating Profi...

Chunk #2 (score=0.4163, source=company_overview.txt)
ACME ANALYTICS INC. – COMPANY OVERVIEW Acme Analytics Inc. was founded in 2019 by Dr. Sarah Chen and Marcus Rivera in Lawrence, Kansas. The company specializes in AI-powered business intelligence too...

Chunk #3 (score=0.3867, source=employee_handbook_excerpt.txt)
ACME ANALYTICS – EMPLOYEE HANDBOOK (EXCERPT) PAID TIME OFF (PTO): - All full-time employees receive 22 days of PTO per year (accrued monthly). - PTO increases to 27 days after 3 years of service. - U...

12.1.6 6) Query with RAG vs. Without RAG

Now we compare:

- **With RAG (LlamaIndex):** The query engine retrieves relevant chunks, sends them + the question to the LLM, and returns a grounded answer.
- **Without RAG (Direct LLM):** We ask the same question directly to the LLM with no context. Since the data is fictional, the LLM either refuses or hallucinates.

For the “without RAG” call we use `generate_text()` from our 7-vendor cascade.

```
from llm_cascade import get_cascade

_llm_raw = get_cascade(verbose=False)
```

```
def generate_text_no_rag(prompt, system_prompt=None):
    """Direct LLM call via llm_cascade (no RAG context)."""
    response = _llm_raw.generate(prompt, system_prompt=system_prompt)
    return response.text, response.provider

print("Direct LLM function ready (for without-RAG comparison).")
```

```
Direct LLM function ready (for without-RAG comparison).
```

12.1.7 7) Run End-to-End Examples: With RAG vs. Without RAG

We test questions about **three fictional companies** that no LLM has seen in training. Notice how LlamaIndex **routes each question to the correct company's documents** — even though all three are in the same index. The source attribution shows which file answered each question.

- With RAG: correct, specific answers grounded on the right company's documents
- Without RAG: refusals, hedging, or hallucinated numbers

```
questions = [
    # Acme Analytics questions
    "What was Acme Analytics' revenue in Q4 2025?",
    "How many days of PTO do new employees get at Acme?",
    # Globex Cybersecurity questions
    "What is Globex Cybersecurity's flagship product and how fast does it detect threats?",
    "How did Globex help Meridian Health System during a ransomware attack?",
    # Nextera Green Solutions questions
    "Who founded Nextera Green Solutions and where is it headquartered?",
    "How much carbon emissions reduction did Nextera enable for its customers in 2025?",
]

def preview(text, max_len=800):
    text = str(text) if text else ""
    return text[:max_len] + ("..." if len(text) > max_len else "")

if not has_llm_provider():
    print("Error: No LLM API key configured. Set at least one API key in Colab Secrets.")
else:
    for i, q in enumerate(questions, start=1):
        print("=" * 80)
        print(f"Q{i}. {q}")
        print("=" * 80)

        # --- WITH RAG (LlamaIndex) ---
        print("--- WITH RAG (LlamaIndex) ---")
        try:
            response = query_engine.query(q)
            print(" Answer:")
            print(preview(response.response))
            sources = set()
            for node in response.source_nodes:
                sources.add(node.metadata.get("file_name", "unknown"))
            print(f" Sources: {' '.join(sources)}")
        except Exception as e:
            print(f" RAG error: {e}")
```

```
# --- WITHOUT RAG ---
print("--- WITHOUT RAG (LLM knowledge only) ---")
try:
    answer_direct, prov_direct = generate_text_no_rag(q)
    print(f" Answer (provider: {prov_direct}):")
    print(preview(answer_direct))
except Exception as e:
    print(f" Direct error: {e}")

print()
```

```
=====
Q1. What was Acme Analytics' revenue in Q4 2025?
=====
```

```
--- WITH RAG (LlamaIndex) ---
```

```
Answer:
```

```
Acme Analytics' revenue in Q4 2025 was $12.8 million.
```

```
Sources: q4_2025_financials.txt, products_and_services.txt, company_overview.txt
```

```
--- WITHOUT RAG (LLM knowledge only) ---
```

```
Answer (provider: Gemini):
```

```
I don't have enough information to answer that question. Here's why:
```

```
* **Real-time financial data is private:** Companies' specific revenue figures,
especially for future quarters like Q4 2025, are proprietary and not publicly
available in a general knowledge base like mine.
```

```
* **Future events are speculative:** Even if Acme Analytics were a real, publicly
traded company, Q4 2025 is in the future. Any revenue projections would be
estimates, not definitive figures, and only the company itself would be making
those.
```

```
**To find revenue information for a real company, you would typically look for:**
```

```
* **Official company earnings reports:** These are usually released quarterly and
annually.
```

```
* **Financial news websites:** Reputable financial news outlets often report on
company earnings.
```

```
* **In...
```

12.1.7.1 Checkpoint: Reflection Questions

1. **Hallucination check:** Did the without-RAG answers invent specific revenue numbers or employee counts for a fictional company?
2. **Source attribution:** LlamaIndex tells us *which document* the answer came from. Why is this important for trust?
3. **Chunking:** How does the chunk size affect retrieval quality? What if a key fact spans two chunks?
4. **Framework vs. manual:** Compare this LlamaIndex approach to the manual RAG we built in the VectorDB and DBMS_RAG notebooks. What did the framework handle for us?

12.1.8 8) Interactive Query (Optional)

Now it is your turn to explore. Change the question in the cell below to anything you want to ask about the three fictional companies. Try questions that span multiple documents (e.g., “Compare Acme’s revenue to Globex’s products”) or questions about specific details buried deep in a single document. Pay attention to the source attributions – they tell you which file LlamaIndex retrieved each answer from, which is a useful trust signal in any RAG system.

```
# Change this to any question about Acme Analytics
my_question = "What is Acme's LTV/CAC ratio and what does it mean?"
```

```

if has_llm_provider():
    print(f"Question: {my_question}\n")

    print("=" * 60)
    print("WITH RAG (LlamaIndex):")
    print("=" * 60)
    try:
        response = query_engine.query(my_question)
        print(f"\n{response.response}")
        sources = set()
        for node in response.source_nodes:
            sources.add(node.metadata.get('file_name', 'unknown'))
        print(f"\nSources: {'', '.join(sources)}")
    except Exception as e:
        print(f"Error: {e}")

    print("\n" + "=" * 60)
    print("WITHOUT RAG:")
    print("=" * 60)
    try:
        answer_direct, prov = generate_text_no_rag(my_question)
        print(f"\n[{prov}] {answer_direct}")
    except Exception as e:
        print(f"Error: {e}")
else:
    print("Set at least one API key first.")

```

Question: What is Acme's LTV/CAC ratio and what does it mean?

=====

WITH RAG (LlamaIndex):

=====

Based on the financial results provided, Acme's **LTV/CAC ratio is 8.2x** (or 8.2).

What this means:

- **LTV (Lifetime Value)** of \$67,400 represents the total revenue Acme expects to generate from a single customer over the entire duration of their relationship.
- **CAC (Customer Acquisition Cost)** of \$8,200 represents the sales and marketing spend required to acquire one new customer.

An 8.2x ratio indicates that for every **\$1** Acme spends to acquire a customer, the company expects to earn **\$8.20** back from that customer over time. This suggests highly efficient customer acquisition economics, where the long-term value generated significantly exceeds the upfront investment required to win the business.

Sources: q4_2025_financials.txt, company_overview.txt, employee_handbook_excerpt.txt

=====

WITHOUT RAG:

=====

[Gemini] Let's break down Acme's LTV/CAC ratio, what it means, and why it's so important.

First, a quick refresher on the two components:

- * **LTV (Lifetime Value):** The total revenue a business can expect to generate from a single customer account over the entire period of their relationship.
- * **CAC (Customer Acquisition Cost):** The total cost associated with acquiring a new customer. This includes marketing spend, sales salaries, tools, etc., divided by the number of new customers acquired over a given period.

****What is Acme's LTV/CAC ratio?****

To answer this, we need the *actual numbers* for Acme's LTV and CAC. Since I don't have access to Acme's internal financial data, I can't give you a specific numerical ratio.

****However, I can tell you how to calculate it and give you examples of what it *might* be.****

****Formula:****

$$\text{LTV/CAC Ratio} = \frac{\text{Lifetime Value (LTV)}}{\text{Customer Acquisition Cost (CAC)}}$$

****Example Scenarios for Acme:****

Let's imagine some possibilities for Acme:

- * ****Scenario 1: Acme has an LTV of $\$3,000$ and a CAC of $\$1,000$.****
* **LTV/CAC Ratio = $\$3,000 / \$1,000 = 3:1$ (or just 3)**
- * ****Scenario 2: Acme has an LTV of $\$500$ and a CAC of $\$250$.****
* **LTV/CAC Ratio = $\$500 / \$250 = 2:1$ (or just 2)**
- * ****Scenario 3: Acme has an LTV of $\$1,500$ and a CAC of $\$2,000$.****
* **LTV/CAC Ratio = $\$1,500 / \$2,000 = 0.75:1$ (or just 0.75)**

****What Does Acme's LTV/CAC Ratio Mean?****

The LTV/CAC ratio is a crucial metric that tells Acme (and its investors) **how much value it gets back for every dollar it spends to acquire a new customer.**

Here's a breakdown of what different ratios generally signify:

1. ****LTV/CAC < 1:1 (e.g., 0.75:1)****
* ****Meaning for Acme:**** Acme is spending more money to acquire a customer than that customer will ever generate in revenue.
* ****Implication:**** This is a highly problematic and unsustainable business model. Acme is losing money on every customer acquired. It needs to drastically reduce its CAC, increase its LTV, or both, to survive.
2. ****LTV/CAC $\approx$ 1:1 (e.g., 1.1:1)****
* ****Meaning for Acme:**** Acme is just barely breaking even on its customer acquisition efforts.
* ****Implication:**** While not losing money, Acme isn't generating much profit from new customers. There's little room for error, and the business likely won't scale profitably. It suggests a need for optimization in either LTV or CAC.
3. ****LTV/CAC between 2:1 and 3:1 (e.g., 2.5:1)****
* ****Meaning for Acme:**** Acme is generating a reasonable return on its acquisition spend. For every dollar spent, it gets back $\$2$ to $\$3$ over the customer's lifetime.
* ****Implication:**** This is generally considered a healthy and sustainable ratio for many businesses, especially for newer companies or those in competitive markets. It indicates a solid foundation for growth.
4. ****LTV/CAC > 3:1 (e.g., 4:1, 5:1, or even higher)****
* ****Meaning for Acme:**** Acme is highly efficient in its customer acquisition,

generating significant returns for every dollar invested. For every dollar spent, it gets back \$3 or more.

* **Implication:** This is often considered an excellent ratio. It means Acme has a very profitable customer acquisition machine. It might even suggest that Acme could *afford* to spend more on marketing and sales to accelerate growth, as long as the ratio remains healthy. Sometimes, an extremely high ratio can mean Acme isn't investing enough in growth and could be leaving market share on the table.

In Summary, for Acme:

The LTV/CAC ratio is a **key indicator of business health and scalability.**

* It helps Acme understand the **profitability** of its customer acquisition channels.

* It guides decisions on **marketing spend**, sales strategy, product development, and customer retention efforts.

* **A higher ratio is generally better**, but an *extremely* high ratio might mean Acme is under-investing in growth.

* **The ideal ratio can vary by industry** (e.g., SaaS often targets 3:1 or higher due to high margins and recurring revenue, while e-commerce might be happy with 2:1 for repeat purchases).

Without Acme's specific numbers, we can only discuss the implications of the ratio, but knowing *how* to interpret it is crucial for any business.

12.1.9 9) Bonus: Customizing the RAG Pipeline

LlamaIndex is highly configurable, and understanding the knobs you can turn is essential for moving from a demo to a production system. Here we show how to tweak three important parameters:

- **Chunk size** – smaller chunks yield more precise retrieval (the system finds the exact paragraph), while larger chunks give the LLM more surrounding context per chunk. The tradeoff is granularity vs. context.
- **Top-k** – how many chunks to retrieve. More chunks provide broader coverage but increase the prompt length (and cost). Too few chunks risk missing relevant information.
- **System prompt** – customize the LLM's behavior and persona for your specific use case.

The cell below rebuilds the index with a different chunk size and top-k, then runs the same question to show how these parameters affect the answer.

```
from llama_index.core.node_parser import SentenceSplitter

# Re-index with smaller chunks and more overlap
splitter = SentenceSplitter(chunk_size=256, chunk_overlap=50)
small_chunk_index = VectorStoreIndex.from_documents(
    all_documents,
    transformations=[splitter],
    show_progress=True,
)

# Query with more chunks retrieved
custom_engine = small_chunk_index.as_query_engine(similarity_top_k=5)

test_q = "What is the pricing for InsightBoard Pro?"
print(f"Question: {test_q}\n")

if has_llm_provider():
    try:
        response = custom_engine.query(test_q)
        print(f"Answer (small chunks, top_k=5):")
        print(response.response)
        print(f"\nChunks used: {len(response.source_nodes)}")
```



```

    for i, node in enumerate(response.source_nodes, 1):
        print(f"  #{i} score={node.score:.4f} | {node.metadata.get('file_name',
'')}")
        print(f"      {node.text[:100].replace(chr(10), ' ')}...")
    except Exception as e:
        print(f"Error: {e}")

```

Applying transformations: 0%| | 0/1 [00:00<?, ?it/s]

Generating embeddings: 0%| | 0/15 [00:00<?, ?it/s]

Question: What is the pricing for InsightBoard Pro?

Answer (small chunks, top_k=5):
InsightBoard Pro has two pricing tiers:
* **Standard:** \$49/user/month
* **Enterprise:** \$89/user/month

Chunks used: 5
#1 score=0.7414 | products_and_services.txt
ACME ANALYTICS – PRODUCTS AND SERVICES 1. InsightBoard Pro (Flagship Product)
- Real-time dashbo...
#2 score=0.4986 | globex_products.txt
GLOBEX CYBERSECURITY – PRODUCTS AND SERVICES 1. ShieldAI Platform (Flagship
Product) - AI-powere...
#3 score=0.3897 | globex_products.txt
ThreatIntel Live - Real-time threat intelligence feed with dark web
monitoring - Pricing: \$8,5...
#4 score=0.3663 | nextera_services.txt
NEXTERA GREEN SOLUTIONS – SERVICES AND PLATFORM 1. CarbonLens Platform
(Flagship Product) - Auto...
#5 score=0.3621 | customer_case_study.txt
CUSTOMER CASE STUDY: BIGRETAIL CORP Company: BigRetail Corp (1,200 retail
stores across 38 states) ...

12.1.10 10) Teaching Notes and Exercises

Key takeaways:

- **LlamaIndex** abstracts the RAG pipeline: document loading, chunking, embedding, indexing, retrieval, and LLM prompting — all in a few lines of code.
- Under the hood, it does the **same thing** as our manual RAG notebooks (embed → retrieve → generate), but with less boilerplate.
- **Source attribution** (which document chunk answered the question) comes for free.
- The framework is **highly configurable**: chunk size, overlap, top-k, embedding model, LLM, and prompt templates can all be swapped.
- Using **synthetic/private data** makes the with-vs-without-RAG comparison very clear — the LLM literally cannot answer without retrieval.

LlamaIndex vs. Manual RAG — When to use which:

Aspect	Manual RAG	LlamaIndex
Learning	Better for understanding fundamentals	Hides implementation details
Speed to prototype	Slower	Much faster
Customization	Full control	Configurable, but within framework
Production readiness	Build it yourself	Batteries included

Exercises:

- Add more documents (e.g., a product roadmap, a competitor analysis) and see how retrieval adapts.
- Experiment with different chunk sizes (128, 512, 1024) and compare answer quality.
- Try using a different embedding model (e.g., BAAI/bge-small-en-v1.5) and compare retrieval accuracy.
- Add a **metadata filter** so queries only search specific document types (e.g., only financials).

12.1.11 Key takeaways

- **LlamaIndex** collapses the manual RAG pipeline (load, chunk, embed, index, retrieve, prompt) into two or three lines of code.
- **Data Loaders** from LlamaHub let you ingest from local files, GitHub, Google Drive, Notion, S3, and 300+ other sources with a uniform API.
- **Source attribution** comes for free – each answer carries the file names of the chunks that produced it, which is critical for trust.
- **Configuration knobs** like `chunk_size`, `chunk_overlap`, and `similarity_top_k` let you tune the precision-vs-context tradeoff without rewriting plumbing.
- **Synthetic private documents** make the with-vs-without-RAG contrast vivid: the LLM cannot possibly know these companies from training.

12.1.12 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/LlamaIndex_RAG.ipynb

12.2 LlamaIndex RAG: A Simpler Single-Company Demo

This notebook is the **beginner-friendly version** of our LlamaIndex RAG pipeline. Rather than working with multiple data sources or complex configurations, we focus on a single fictional company (Acme Analytics) with just five short documents. This makes it easy to see exactly what LlamaIndex does at each step without getting lost in data complexity.

If you are new to RAG frameworks, start here. The notebook walks through the complete LlamaIndex workflow – loading documents, building a vector index, querying with retrieval-augmented generation, and comparing RAG answers against a direct LLM call. Once you are comfortable with these basics, move on to the full `llamaindex_rag.ipynb` sibling, which handles multiple companies, custom chunking, and more advanced configuration knobs.

12.2.1 LlamaIndex RAG Demo: Document Q&A with a Framework

This notebook demonstrates how **LlamaIndex** simplifies building RAG pipelines. Instead of manually chunking documents, computing embeddings, and querying vector stores, LlamaIndex handles the plumbing so you can focus on the application logic.

How it works:

1. Create a small set of **synthetic company documents** (no external data needed).
2. Use LlamaIndex to **ingest, chunk, embed, and index** the documents in one step.
3. Given a natural-language question, LlamaIndex **retrieves** relevant chunks and sends them to an LLM.
4. Compare answers **with RAG** (LlamaIndex pipeline) vs. **without RAG** (direct LLM call).

Learning goals:

- Understand what a RAG framework (LlamaIndex) does under the hood
- See how LlamaIndex abstracts document loading, chunking, embedding, and retrieval
- Compare the framework approach to the manual RAG we built in earlier notebooks
- Observe how RAG grounding eliminates hallucination on private/synthetic data

Provider setup: This notebook uses the `llm_cascade` package, which auto-detects your API keys and falls back to the next provider if one is unavailable. Supported providers: OpenAI, Gemini, Ollama, Grok (xAI), Groq, HuggingFace, Cohere, OpenRouter.

Store any of these API keys in Colab Secrets (or a local `.env` file): `OPENAI_API_KEY`, `GEMINI_API_KEY`, `OLLAMA_API_KEY`, `XAI_API_KEY`, `GROQ_API_KEY`, `HF_TOKEN`, `COHERE_API_KEY`, `OPENROUTER_API_KEY`

```
!pip install -q -U llama-index llama-index-llms-openai-like llama-index-llms-gemini
llama-index-embeddings-huggingface google-genai openai git+https://github.com/
KarAnalytics/llm_cascade.git
```

12.2.1.1 1) Imports and Provider Helpers (7-Vendor Cascade)

We configure **LlamaIndex** to use whichever LLM provider you have available. The cascade tries each provider in order and falls through on quota errors.

For **embeddings**, we use a local HuggingFace model (`all-MiniLM-L6-v2`) so that embedding always works regardless of which LLM API key you have.

```
from pathlib import Path
from llm_cascade.providers import PROVIDERS, _load_env, _get_key,
_is_retriable_error

_load_env()

def get_available_providers():
    return [p for p in PROVIDERS if _get_key(p['key_env'])]

def has_llm_provider():
    return len(get_available_providers()) > 0
```

```
# Print status
available = get_available_providers()
if available:
    print('Providers configured (in fallback order):')
    for p in available:
        print(f" + {p['name']:<16} model = {p['default_model']}")
else:
    print('WARNING: No API keys found.')
```

12.2.1.2 2) Configure LlamaIndex LLM and Embeddings

LlamaIndex needs two things:

- An **LLM** for generating answers (we pick the first available provider from our cascade)
- An **embedding model** for converting text to vectors (we use a local HuggingFace model — no API key needed)

This cell configures LlamaIndex's global `Settings` so all downstream components use our chosen models.

```
from llama_index.core import Settings
from llama_index.core.llms import CustomLLM, LLMMetadata, CompletionResponse,
CompletionResponseGen
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.llms.cascade import get_cascade
from typing import Any

# ---- Embedding model (local, no API key needed) ----
Settings.embed_model = HuggingFaceEmbedding(model_name="sentence-transformers/all-
MiniLM-L6-v2")
print("Embedding model: sentence-transformers/all-MiniLM-L6-v2 (local)")

# ---- Custom LlamaIndex LLM that wraps llama_index.llms.cascade for automatic fallback ----
class CascadeLLM(CustomLLM):
    """LlamaIndex LLM that delegates to llama_index.llms.cascade (8-provider fallback)."""
    context_window: int = 8000
    num_output: int = 1024
    model_name: str = "llm_cascade"

    @property
    def metadata(self) -> LLMMetadata:
        return LLMMetadata(
            context_window=self.context_window,
            num_output=self.num_output,
            model_name=self.model_name,
        )

    def complete(self, prompt: str, **kwargs: Any) -> CompletionResponse:
        cascade = get_cascade(verbose=False)
        response = cascade.generate(prompt)
        return CompletionResponse(text=response.text, additional_kwargs={"provider":
response.provider, "model": response.model})

    def stream_complete(self, prompt: str, **kwargs: Any) -> CompletionResponseGen:
        # Non-streaming fallback
        resp = self.complete(prompt, **kwargs)
        def gen():
            yield resp
        return gen()

Settings.llm = CascadeLLM()
print("LLM: CascadeLLM (auto-fallback across available providers)")
```

12.2.1.3 3) Create Synthetic Company Documents

We generate a small set of **fictional company documents** that the LLM has never seen in training. This ensures that correct answers can **only** come from RAG retrieval, not from parametric memory.

Document	Topic
Company Overview	Acme Analytics Inc. — founding, mission, HQ
Products & Services	Three product lines with pricing
Q4 2025 Financial Results	Revenue, profit, growth numbers
Employee Handbook (excerpt)	PTO policy, remote work rules
Customer Case Study	BigRetail Corp deployment results

```
DOCUMENTS = {
  "company_overview.txt": """ACME ANALYTICS INC. – COMPANY OVERVIEW

Acme Analytics Inc. was founded in 2019 by Dr. Sarah Chen and Marcus Rivera
in Lawrence, Kansas. The company specializes in AI-powered business intelligence
tools for mid-market companies (100–5,000 employees).

Headquarters: 1420 Jayhawk Boulevard, Lawrence, KS 66045
Employees: 287 (as of January 2026)
Annual Revenue (2025): $42.3 million
Funding: Series B ($18M raised in March 2023 from Midwest Ventures)

Mission: "To democratize data analytics so every business decision is informed
by evidence, not intuition."

The company operates three offices: Lawrence (HQ), Chicago (sales), and
Austin (engineering). CEO: Dr. Sarah Chen. CTO: Marcus Rivera. CFO: Linda Park."""

  "products_and_services.txt": """ACME ANALYTICS – PRODUCTS AND SERVICES

1. InsightBoard Pro (Flagship Product)
  - Real-time dashboard platform with natural language query interface
  - Pricing: $49/user/month (Standard), $89/user/month (Enterprise)
  - Supports PostgreSQL, MySQL, Snowflake, BigQuery, and Redshift
  - 1,847 active enterprise customers as of Q4 2025

2. DataPipe ETL
  - Automated data pipeline builder with 200+ pre-built connectors
  - Pricing: starts at $500/month for up to 10 million records/day
  - Launched in September 2024

3. PredictIQ
  - ML-powered forecasting add-on for InsightBoard Pro
  - Pricing: $29/user/month (requires InsightBoard Pro subscription)
  - Uses proprietary time-series model trained on retail and logistics data
  - Beta launched March 2025, GA release planned for June 2026

All products include 24/7 email support. Enterprise plans include dedicated
account manager and 99.9% SLA."""

  "q4_2025_financials.txt": """ACME ANALYTICS – Q4 2025 FINANCIAL RESULTS
(CONFIDENTIAL)

Period: October 1 – December 31, 2025

Revenue:          $12.8 million (Q4) / $42.3 million (FY 2025)
Gross Margin:     78.2%
Operating Profit: $1.9 million (Q4) / $5.1 million (FY 2025)
Net Income:       $1.4 million (Q4)
```

Cash on Hand: \$14.7 million
Burn Rate: Company is cash-flow positive since Q2 2025

Key Metrics:

- ARR (Annual Recurring Revenue): \$48.6 million (up 34% YoY)
- Net Revenue Retention: 118%
- Customer Acquisition Cost (CAC): \$8,200
- Customer Lifetime Value (LTV): \$67,400
- LTV/CAC Ratio: 8.2x

Headcount grew from 241 to 287 employees during 2025.

R&D spending: 31% of revenue. Sales & Marketing: 28% of revenue.""",

"employee_handbook_excerpt.txt": ""ACME ANALYTICS – EMPLOYEE HANDBOOK (EXCERPT)

PAID TIME OFF (PTO):

- All full-time employees receive 22 days of PTO per year (accrued monthly).
- PTO increases to 27 days after 3 years of service.
- Unused PTO can be carried over (max 5 days) or paid out at year-end.
- Sick leave: 10 days per year (separate from PTO).

REMOTE WORK POLICY:

- Engineering and Data Science teams: fully remote eligible.
- Sales and Customer Success: hybrid (minimum 2 days/week in office).
- All employees may work remotely up to 4 weeks/year from any US location.
- International remote work requires VP approval and tax review.

PROFESSIONAL DEVELOPMENT:

- Annual learning budget: \$2,500 per employee.
- Conference attendance: up to 2 conferences per year with manager approval.
- Tuition reimbursement: up to \$5,250/year for degree programs.""",

"customer_case_study.txt": ""CUSTOMER CASE STUDY: BIGRETAIL CORP

Company: BigRetail Corp (1,200 retail stores across 38 states)

Challenge: Siloed data across POS, inventory, and CRM systems made it impossible for regional managers to get timely insights.

Solution: Deployed InsightBoard Pro Enterprise + DataPipe ETL

- Connected 14 data sources in 3 weeks using DataPipe
- 340 regional managers now use InsightBoard daily
- Natural language queries replaced manual SQL report requests

Results (after 6 months):

- 62% reduction in time-to-insight (from 4 days to 1.5 days average)
- \$3.2 million saved in inventory carrying costs
- 23% increase in regional manager satisfaction scores
- SQL report request backlog eliminated entirely

Quote from BigRetail CIO Janet Torres:

\\"InsightBoard Pro transformed how our managers interact with data. They went from waiting days for a report to asking questions in plain English and getting answers in seconds.\""",
}

Write documents to disk

DOC_DIR = Path("acme_docs")

DOC_DIR.mkdir(exist_ok=True)

for filename, content in DOCUMENTS.items():

(DOC_DIR / filename).write_text(content, encoding="utf-8")

print(f"Created {len(DOCUMENTS)} documents in '{DOC_DIR}/':")

```
for f in sorted(DOC_DIR.iterdir()):
    print(f" {f.name} ({f.stat().st_size} bytes)")
```

12.2.1.4 4) Ingest Documents into LlamaIndex

This is where LlamaIndex shines — **one line** to load all documents, and **one line** to build a searchable index.

Under the hood, LlamaIndex:

1. Reads each `.txt` file
2. Splits text into chunks (default ~1024 tokens with overlap)
3. Computes embeddings for each chunk using our HuggingFace model
4. Stores them in an in-memory vector index

```
from llama_index.core import SimpleDirectoryReader, VectorStoreIndex

# Step 1: Load documents from the folder
documents = SimpleDirectoryReader(str(DOC_DIR)).load_data()
print(f"Loaded {len(documents)} document chunks")

# Step 2: Build vector index (embeds + indexes all chunks)
index = VectorStoreIndex.from_documents(documents, show_progress=True)
print(f"Vector index built with {len(documents)} chunks")

# Step 3: Create a query engine
query_engine = index.as_query_engine(similarity_top_k=3)
print("Query engine ready (top_k=3 retrieval)")
```

12.2.1.5 5) What Just Happened? Peeking Under the Hood

Let's inspect what LlamaIndex created — how many chunks, what the embeddings look like, and how retrieval works before we query the LLM.

```
# Inspect the chunks LlamaIndex created
retriever = index.as_retriever(similarity_top_k=3)

# Test retrieval without LLM — just see what chunks come back
test_query = "How much revenue did Acme make in Q4 2025?"
retrieved_nodes = retriever.retrieve(test_query)

print(f"Query: '{test_query}'")
print(f"Retrieved {len(retrieved_nodes)} chunks:\n")
for i, node in enumerate(retrieved_nodes, 1):
    source = node.metadata.get('file_name', 'unknown')
    score = node.score
    text_preview = node.text[:200].replace('\n', ' ')
    print(f"  Chunk #{i} (score={score:.4f}, source={source})")
    print(f"    {text_preview}...\n")
```

12.2.1.6 6) Query with RAG vs. Without RAG

Now we compare:

- **With RAG (LlamaIndex):** The query engine retrieves relevant chunks, sends them + the question to the LLM, and returns a grounded answer.
- **Without RAG (Direct LLM):** We ask the same question directly to the LLM with no context. Since the data is fictional, the LLM either refuses or hallucinates.

For the “without RAG” call we use `generate_text()` from our 7-vendor cascade.

```

from llm_cascade import get_cascade

_llm_raw = get_cascade(verbose=False)

def generate_text_no_rag(prompt, system_prompt=None):
    """Direct LLM call via llm_cascade (no RAG context)."""
    response = _llm_raw.generate(prompt, system_prompt=system_prompt)
    return response.text, response.provider

print('Direct LLM function ready (for without-RAG comparison).')

```

12.2.1.7 7) Run End-to-End Examples: With RAG vs. Without RAG

We test questions about Acme Analytics — a **fictional company** that no LLM has seen in training. This makes the with/without RAG comparison very clear:

- With RAG: correct, specific answers grounded on our documents
- Without RAG: refusals, hedging, or hallucinated numbers

```

questions = [
    # "What was Acme Analytics' revenue in Q4 2025?",
    "How many days of PTO do new employees get at Acme?",
    # "What products does Acme Analytics offer and what do they cost?",
    # "How much did BigRetail Corp save using Acme's products?",
    # "Who founded Acme Analytics and where is the company headquartered?",
]

def preview(text, max_len=800):
    text = str(text) if text else ""
    return text[:max_len] + ("..." if len(text) > max_len else "")

if not has_llm_provider():
    print("Error: No LLM API key configured. Set at least one API key in Colab Secrets.")
else:
    for i, q in enumerate(questions, start=1):
        print("=" * 80)
        print(f"Q{i}. {q}")
        print("=" * 80)

        # --- WITH RAG (LlamaIndex) ---
        print("--- WITH RAG (LlamaIndex) ---")
        try:
            response = query_engine.query(q)
            print("  Answer:")
            print(preview(response.response))
            sources = set()
            for node in response.source_nodes:
                sources.add(node.metadata.get("file_name", "unknown"))
            print(f"  Sources: {' '.join(sources)}")
        except Exception as e:
            print(f"  RAG error: {e}")

        # --- WITHOUT RAG ---
        print("--- WITHOUT RAG (LLM knowledge only) ---")
        try:
            answer_direct, prov_direct = generate_text_no_rag(q)
            print(f"  Answer (provider: {prov_direct}):")
            print(preview(answer_direct))

```



```
except Exception as e:
    print(f" Direct error: {e}")

print()
```

12.2.1.7.1 Checkpoint: Reflection Questions

1. **Hallucination check:** Did the without-RAG answers invent specific revenue numbers or employee counts for a fictional company?
2. **Source attribution:** LlamaIndex tells us *which document* the answer came from. Why is this important for trust?
3. **Chunking:** How does the chunk size affect retrieval quality? What if a key fact spans two chunks?
4. **Framework vs. manual:** Compare this LlamaIndex approach to the manual RAG we built in the VectorDB and DBMS_RAG notebooks. What did the framework handle for us?

12.2.1.8 Key takeaways

- **Two-line RAG** – `SimpleDirectoryReader` plus `VectorStoreIndex.from_documents` is enough to turn a folder of `.txt` files into a working Q&A system.
- **Global Settings** centralize your LLM and embedding choices so every downstream component inherits them automatically.
- **Local embeddings** via HuggingFace's `all-MiniLM-L6-v2` keep the pipeline API-key-free for the vector step, isolating cost to the LLM call.
- **Source nodes** returned alongside every answer show which chunk was retrieved, making hallucinations easy to spot.
- **Synthetic Acme data** guarantees the LLM has zero prior knowledge, so any correct specifics must be coming from retrieval.

12.2.1.9 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

Estimated run time: ~3 minutes

```
https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/LlamaIndex_RAG_
simple_single_company.ipynb
```


Chapter 13

LangChain

LangChain is a modular toolkit for building LLM applications. Unlike LlamaIndex (which is opinionated about how RAG should work), LangChain gives you explicit control over each step: document loaders, text splitters, embeddings, vector stores, and chains.

This chapter covers:

- **LCEL (LangChain Expression Language)** — the modern pipe syntax for composing chains
- **RAG with FAISS** — building a retrieval chain step by step
- **Agents with tools** — LLMs that decide which functions to call, implemented with LCEL pipes

13.1 LangChain RAG Demo: Document Q&A with a Framework

This notebook demonstrates how **LangChain** simplifies building RAG pipelines. LangChain is one of the most popular frameworks for building LLM-powered applications, providing modular components for document loading, text splitting, embeddings, vector stores, and retrieval chains.

How it works:

1. Load synthetic company documents from three separate directories (no external data needed).
2. Use LangChain to split, embed, and store the documents in a FAISS vector store. NOTE that FAISS is not a vector database. It is an open-source library for efficient, high-speed similarity search and clustering of dense vectors. Unlike ChromaDB or Pinecone, which are managed or structured vector databases, FAISS is a low-level, high-performance library that stores vectors primarily in-memory.
3. Build a retrieval chain that fetches relevant chunks and sends them to an LLM.
4. Compare answers with RAG (LangChain pipeline) vs. without RAG (direct LLM call).

Learning goals:

- Understand LangChain's modular architecture: Loaders → Splitters → Embeddings → Vector Stores → Chains
- See how LangChain compares to LlamaIndex for building RAG pipelines
- Observe how RAG grounding eliminates hallucination on private/synthetic data

Provider setup: This notebook uses the `llm_cascade` package, which auto-detects your API keys and falls back to the next provider if one is unavailable. Supported providers: OpenAI, Gemini, Ollama, Grok (xAI), Groq, HuggingFace, Cohere, OpenRouter.

Store any of these API keys in Colab Secrets (or a local `.env` file): `OPENAI_API_KEY`, `GEMINI_API_KEY`, `OLLAMA_API_KEY`, `XAI_API_KEY`, `GROQ_API_KEY`, `HF_TOKEN`, `COHERE_API_KEY`, `OPENROUTER_API_KEY`

LangChain is the other major framework in this space — if you already worked through Chapter 12's LlamaIndex notebooks, think of this as the same problem solved by a more explicit, composable toolkit where you wire each step together yourself.

```
!pip install -q -U langchain langgraph langchain-community langchain-huggingface
langchain-text-splitters faiss-cpu google-genai openai langchain-google-genai
git+https://github.com/KarAnalytics/llm_cascade.git sentence-transformers
```

13.1.1 Imports and Provider Helpers (8-Vendor Cascade)

We configure the same 8-provider cascade used across all our RAG notebooks. LangChain can work with any OpenAI-compatible API, so our cascade integrates naturally — we point LangChain's `ChatOpenAI` class at whichever provider is available, using its OpenAI-compatible endpoint. This means the same LangChain code works whether you are running against OpenAI, Gemini, Groq, or any other provider in the cascade.

```
from pathlib import Path
from llm_cascade.providers import PROVIDERS, _load_env, _get_key,
_is_retrievable_error

_load_env()

def get_available_providers():
    return [p for p in PROVIDERS if _get_key(p['key_env'])]

def has_llm_provider():
    return len(get_available_providers()) > 0

# Print status
available = get_available_providers()
if available:
    print('Providers configured (in fallback order):')
    for p in available:
        print(f"  {p['name']:<16} model = {p['default_model']}")
```

```

else:
    print('WARNING: No API keys found.')

```

```

Providers configured (in fallback order):
+ Gemini          model = gemini-2.5-flash
+ Ollama          model = kimi-k2.5:cloud
+ Groq            model = llama-3.3-70b-versatile
+ HuggingFace     model = meta-llama/Llama-3.3-70B-Instruct
+ Cohere          model = command-a-03-2025
+ OpenRouter      model = meta-llama/llama-3.3-70b-instruct:free
+ OpenAI          model = gpt-4o-mini

```

13.1.2 Configure LangChain LLM and Embeddings

LangChain needs two components before we can build a retrieval chain:

- An LLM for generating answers (we use `ChatOpenAI` pointed at the first available provider from our cascade)
- An embedding model for converting text to vectors (local HuggingFace model – no API key needed)

One thing to notice here: LangChain uses `ChatOpenAI` as its LLM wrapper even when talking to non-OpenAI providers. This works because most LLM providers now expose OpenAI-compatible API endpoints. We simply swap the `base_url` and `api_key` to point at Gemini, Groq, or whatever provider is available. The rest of LangChain's pipeline does not need to know or care which model is behind the API.

```

from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.chat_models import ChatOpenAI

# ---- Embedding model (local, no API key needed) -----
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
print("Embedding model: sentence-transformers/all-MiniLM-L6-v2 (local)")

# ---- LLM with runtime fallback (cascades if quota is exhausted) -----
llm_candidates = []
for provider in get_available_providers():
    api_key = _get_key(provider["key_env"])
    try:
        if provider["style"] == "gemini":
            from langchain_google_genai import ChatGoogleGenerativeAI
            candidate = ChatGoogleGenerativeAI(
                google_api_key=api_key,
                model=provider["default_model"],
            )
        else:
            candidate = ChatOpenAI(
                api_key=api_key,
                base_url=provider["base_url"],
                model=provider["default_model"],
            )
        llm_candidates.append((provider, candidate))
        print(f" + {provider['name']:<16} ({provider['default_model']})")
    except Exception as e:
        print(f" - {provider['name']:<16} skipped: {e}")

if llm_candidates:
    primary_provider, primary_llm = llm_candidates[0]
    fallback_llms = [c for _, c in llm_candidates[1:]]

    if fallback_llms:
        llm = primary_llm.with_fallbacks(fallback_llms)
        print(f"\nPrimary LLM: {primary_provider['name']}")

```

```

({primary_provider['default_model']})")
    print(f"Fallbacks: {' ', '.join(p['name'] for p, _ in llm_candidates[1:])}")
    print("(If primary hits quota, automatically tries the next provider)")
    else:
        llm = primary_llm
        print(f"\nLLM provider: {primary_provider['name']}")
    ({primary_provider['default_model']})")
    print("(No fallback providers configured)")

    llm_provider = primary_provider
else:
    llm = None
    primary_llm = None
    llm_provider = None
    print("ERROR: Could not configure any LLM provider. Check your API keys.")

```

```
modules.json: 0%|          | 0.00/349 [00:00<?, ?B/s]
```

```
config_sentence_transformers.json: 0%|          | 0.00/116 [00:00<?, ?B/s]
```

```
README.md: 0.00B [00:00, ?B/s]
```

```
sentence_bert_config.json: 0%|          | 0.00/53.0 [00:00<?, ?B/s]
```

```
config.json: 0%|          | 0.00/612 [00:00<?, ?B/s]
```

```
model.safetensors: 0%|          | 0.00/90.9M [00:00<?, ?B/s]
```

```
Loading weights: 0%|          | 0/103 [00:00<?, ?it/s]
```

```
BertModel LOAD REPORT from: sentence-transformers/all-MiniLM-L6-v2
```

Key	Status		
-----+-----+-----+-----			
embeddings.position_ids	UNEXPECTED		

Notes:

- UNEXPECTED :can be ignored when loading from different task/architecture; not ok if you expect identical arch.

```
tokenizer_config.json: 0%|          | 0.00/350 [00:00<?, ?B/s]
```

```
vocab.txt: 0.00B [00:00, ?B/s]
```

```
tokenizer.json: 0.00B [00:00, ?B/s]
```

```
special_tokens_map.json: 0%|          | 0.00/112 [00:00<?, ?B/s]
```

```
config.json: 0%|          | 0.00/190 [00:00<?, ?B/s]
```

```
Embedding model: sentence-transformers/all-MiniLM-L6-v2 (local)
+ Gemini          (gemini-2.5-flash)
```

```
/tmp/ipykernel_2361/1873978192.py:20: LangChainDeprecationWarning: The class
`ChatOpenAI` was deprecated in LangChain 0.0.10 and will be removed in 1.0. An
updated version of the class exists in the `langchain-openai` package and should be
used instead. To use it run `pip install -U `langchain-openai` and import as `from
`langchain_openai import ChatOpenAI``.
  candidate = ChatOpenAI(
```

```
+ Ollama          (kimi-k2.5:cloud)
+ Groq            (llama-3.3-70b-versatile)
+ HuggingFace     (meta-llama/Llama-3.3-70B-Instruct)
+ Cohere          (command-a-03-2025)
+ OpenRouter      (meta-llama/llama-3.3-70b-instruct:free)
+ OpenAI          (gpt-4o-mini)
```

Primary LLM: Gemini (gemini-2.5-flash)
 Fallbacks: Ollama, Groq, HuggingFace, Cohere, OpenRouter, OpenAI
 (If primary hits quota, automatically tries the next provider)

13.1.3 Create Synthetic Company Documents

We generate fictional documents for three companies that the LLM has never seen in training. This ensures that correct answers can only come from RAG retrieval, not from parametric memory. Each company's documents live in a separate directory, simulating real-world data silos.

Directory	Company	Documents
acme_docs/	Acme Analytics Inc. (BI tools, Lawrence KS)	Overview, Products, Financials, Handbook, Case Study
globex_docs/	Globex Cybersecurity Corp. (cybersecurity, Boston)	Overview, Products, Incident Report
nextera_docs/	Nextera Green Solutions (sustainability consulting, Denver)	Overview, Services, Impact Report

Later, we will use LlamaIndex **Data Loaders** (`SimpleDirectoryReader`) to ingest all three directories — demonstrating how LlamaIndex can pull information from multiple, separate data sources and unify them in a single searchable index.

```
DOCUMENTS = {
    "company_overview.txt": """ACME ANALYTICS INC. – COMPANY OVERVIEW

Acme Analytics Inc. was founded in 2019 by Dr. Sarah Chen and Marcus Rivera
in Lawrence, Kansas. The company specializes in AI-powered business intelligence
tools for mid-market companies (100–5,000 employees).

Headquarters: 1420 Jayhawk Boulevard, Lawrence, KS 66045
Employees: 287 (as of January 2026)
Annual Revenue (2025): $42.3 million
Funding: Series B ($18M raised in March 2023 from Midwest Ventures)

Mission: "To democratize data analytics so every business decision is informed
by evidence, not intuition."

The company operates three offices: Lawrence (HQ), Chicago (sales), and
Austin (engineering). CEO: Dr. Sarah Chen. CTO: Marcus Rivera. CFO: Linda Park.""",
    "products_and_services.txt": """ACME ANALYTICS – PRODUCTS AND SERVICES

1. InsightBoard Pro (Flagship Product)
  - Real-time dashboard platform with natural language query interface
  - Pricing: $49/user/month (Standard), $89/user/month (Enterprise)
```

- Supports PostgreSQL, MySQL, Snowflake, BigQuery, and Redshift
- 1,847 active enterprise customers as of Q4 2025

2. DataPipe ETL

- Automated data pipeline builder with 200+ pre-built connectors
- Pricing: starts at \$500/month for up to 10 million records/day
- Launched in September 2024

3. PredictIQ

- ML-powered forecasting add-on for InsightBoard Pro
- Pricing: \$29/user/month (requires InsightBoard Pro subscription)
- Uses proprietary time-series model trained on retail and logistics data
- Beta launched March 2025, GA release planned for June 2026

All products include 24/7 email support. Enterprise plans include dedicated account manager and 99.9% SLA."",

"q4_2025_financials.txt": ""ACME ANALYTICS – Q4 2025 FINANCIAL RESULTS (CONFIDENTIAL)

Period: October 1 – December 31, 2025

Revenue: \$12.8 million (Q4) / \$42.3 million (FY 2025)
 Gross Margin: 78.2%
 Operating Profit: \$1.9 million (Q4) / \$5.1 million (FY 2025)
 Net Income: \$1.4 million (Q4)
 Cash on Hand: \$14.7 million
 Burn Rate: Company is cash-flow positive since Q2 2025

Key Metrics:

- ARR (Annual Recurring Revenue): \$48.6 million (up 34% YoY)
- Net Revenue Retention: 118%
- Customer Acquisition Cost (CAC): \$8,200
- Customer Lifetime Value (LTV): \$67,400
- LTV/CAC Ratio: 8.2x

Headcount grew from 241 to 287 employees during 2025.

R&D spending: 31% of revenue. Sales & Marketing: 28% of revenue."",

"employee_handbook_excerpt.txt": ""ACME ANALYTICS – EMPLOYEE HANDBOOK (EXCERPT)

PAID TIME OFF (PTO):

- All full-time employees receive 22 days of PTO per year (accrued monthly).
- PTO increases to 27 days after 3 years of service.
- Unused PTO can be carried over (max 5 days) or paid out at year-end.
- Sick leave: 10 days per year (separate from PTO).

REMOTE WORK POLICY:

- Engineering and Data Science teams: fully remote eligible.
- Sales and Customer Success: hybrid (minimum 2 days/week in office).
- All employees may work remotely up to 4 weeks/year from any US location.
- International remote work requires VP approval and tax review.

PROFESSIONAL DEVELOPMENT:

- Annual learning budget: \$2,500 per employee.
- Conference attendance: up to 2 conferences per year with manager approval.
- Tuition reimbursement: up to \$5,250/year for degree programs."",

"customer_case_study.txt": ""CUSTOMER CASE STUDY: BIGRETAIL CORP

Company: BigRetail Corp (1,200 retail stores across 38 states)

Challenge: Siloed data across POS, inventory, and CRM systems made it impossible for regional managers to get timely insights.

Solution: Deployed InsightBoard Pro Enterprise + DataPipe ETL

- Connected 14 data sources in 3 weeks using DataPipe
- 340 regional managers now use InsightBoard daily
- Natural language queries replaced manual SQL report requests

Results (after 6 months):

- 62% reduction in time-to-insight (from 4 days to 1.5 days average)
- \$3.2 million saved in inventory carrying costs
- 23% increase in regional manager satisfaction scores
- SQL report request backlog eliminated entirely

Quote from BigRetail CIO Janet Torres:

```
\ "InsightBoard Pro transformed how our managers interact with data.
They went from waiting days for a report to asking questions in plain
English and getting answers in seconds.\ """,
}
```

```
# Write documents to disk
DOC_DIR = Path("acme_docs")
DOC_DIR.mkdir(exist_ok=True)

for filename, content in DOCUMENTS.items():
    (DOC_DIR / filename).write_text(content, encoding="utf-8")

print(f"Created {len(DOCUMENTS)} documents in '{DOC_DIR}/':")
for f in sorted(DOC_DIR.iterdir()):
    print(f"  {f.name} ({f.stat().st_size} bytes)")
```

```
Created 5 documents in 'acme_docs/':
company_overview.txt (715 bytes)
customer_case_study.txt (924 bytes)
employee_handbook_excerpt.txt (828 bytes)
products_and_services.txt (915 bytes)
q4_2025_financials.txt (709 bytes)
```

```
# --- Company 2: Globex Cybersecurity Corp. → globex_docs/ ---
```

```
GLOBEX_DOCUMENTS = {
    "globex_overview.txt": """GLOBEX CYBERSECURITY CORP. – COMPANY OVERVIEW
```

```
Globex Cybersecurity Corp. was founded in 2017 by former NSA analyst
James Whitfield and AI researcher Dr. Priya Narayanan in Boston, Massachusetts.
The company provides AI-driven cybersecurity solutions for financial services
and healthcare organizations.
```

```
Headquarters: 88 Federal Street, Suite 1200, Boston, MA 02110
Employees: 412 (as of January 2026)
Annual Revenue (2025): $78.5 million
Funding: Series C ($45M raised in June 2024 from CyberVentures Capital
and Goldman Sachs Growth Equity)
```

```
Mission: "To make enterprise-grade cybersecurity autonomous, adaptive,
and accessible to every regulated industry."
```

```
The company operates four offices: Boston (HQ), Washington D.C. (government
relations), Tel Aviv (R&D), and Singapore (APAC sales).
CEO: James Whitfield. CTO: Dr. Priya Narayanan. CISO: Robert Tanaka.""",
```

```
    "globex_products.txt": """GLOBEX CYBERSECURITY – PRODUCTS AND SERVICES
```

1. ShieldAI Platform (Flagship Product)
 - AI-powered threat detection and automated incident response
 - Uses behavioral analysis to detect zero-day attacks in real time
 - Pricing: \$120/endpoint/year (Standard), \$195/endpoint/year (Premium)
 - 623 enterprise customers across 18 countries as of Q4 2025
 - Average detection time: 2.3 minutes (industry average: 197 days)
2. VaultGuard
 - Data encryption and access management platform for regulated industries
 - HIPAA, SOX, and PCI-DSS compliant out of the box
 - Pricing: \$35,000/year (up to 500 users), \$60,000/year (unlimited users)
 - Launched January 2023
3. ThreatIntel Live
 - Real-time threat intelligence feed with dark web monitoring
 - Pricing: \$8,500/month (includes API access and analyst dashboard)
 - Aggregates data from 2,400+ sources updated every 15 minutes
 - Integrates with Splunk, CrowdStrike, Palo Alto, and Microsoft Sentinel

All products include a dedicated Security Operations Center (SOC) team for Premium customers. Globex holds ISO 27001 and SOC 2 Type II certifications."",

"globex_incident_report.txt": ""GLOBEX CYBERSECURITY – 2025 THREAT LANDSCAPE REPORT (PUBLIC)

Published: February 2026

KEY FINDINGS FROM GLOBEX'S CUSTOMER BASE (2025):

Total incidents analyzed: 14,237 across 623 customers

- Ransomware attacks: up 42% YoY (3,891 incidents)
- Phishing campaigns: up 28% YoY (5,102 incidents)
- Supply chain attacks: up 67% YoY (1,245 incidents – fastest-growing category)
- Insider threats: relatively stable at 1,834 incidents

ShieldAI Platform Performance:

- Blocked 97.3% of attacks before any data exfiltration
- Mean time to detect (MTTD): 2.3 minutes
- Mean time to respond (MTTR): 8.7 minutes (automated response)
- False positive rate: 0.4% (down from 1.2% in 2024)

NOTABLE CASE: MERIDIAN HEALTH SYSTEM

- 340-bed hospital network targeted by Medusa ransomware variant
- ShieldAI detected anomalous lateral movement within 47 seconds
- Automated containment isolated 3 affected servers in under 2 minutes
- Zero patient data exfiltrated; zero downtime for clinical systems
- Estimated savings: \$12M+ (average healthcare ransomware cost per Ponemon)

INDUSTRY PREDICTIONS FOR 2026:

- AI-generated phishing emails will become indistinguishable from human-written
 - Quantum-resistant encryption adoption will begin in financial services
 - Average cost of a data breach expected to exceed \$5.2 million globally""
- }

```
GLOBEX_DIR = Path("globex_docs")
GLOBEX_DIR.mkdir(exist_ok=True)
```

```
for filename, content in GLOBEX_DOCUMENTS.items():
    (GLOBEX_DIR / filename).write_text(content, encoding="utf-8")
```

```
print(f"Created {len(GLOBEX_DOCUMENTS)} documents in '{GLOBEX_DIR}/':")
for f in sorted(GLOBEX_DIR.iterdir()):
    print(f"  {f.name} ({f.stat().st_size} bytes)")
```

```
Created 3 documents in 'globex_docs/':
globex_incident_report.txt (1321 bytes)
globex_overview.txt (864 bytes)
globex_products.txt (1131 bytes)
```

```
# --- Company 3: Nextera Green Solutions → nextera_docs/ ---
```

```
NEXTERA_DOCUMENTS = {
    "nextera_overview.txt": """"NEXTERA GREEN SOLUTIONS – COMPANY OVERVIEW

Nextera Green Solutions was founded in 2020 by climate scientist Dr. Elena
Vasquez and former McKinsey partner David Okafor in Denver, Colorado.
The company provides sustainability consulting and carbon accounting software
to help companies meet ESG (Environmental, Social, Governance) targets.

Headquarters: 1700 Lincoln Street, Suite 3400, Denver, CO 80203
Employees: 156 (as of January 2026)
Annual Revenue (2025): $23.8 million
Funding: Series A ($12M raised in November 2023 from Climate Capital Partners
and Patagonia Ventures)

Mission: "To give every company a clear, data-driven path to net zero –
turning climate commitments into measurable action."

The company operates two offices: Denver (HQ) and London (European operations).
CEO: Dr. Elena Vasquez. COO: David Okafor. VP Engineering: Kenji Tanaka.
Nextera is a certified B Corporation and a member of the UN Global Compact.""",

    "nextera_services.txt": """"NEXTERA GREEN SOLUTIONS – SERVICES AND PLATFORM

1. CarbonLens Platform (Flagship Product)
  - Automated Scope 1, 2, and 3 carbon emissions tracking
  - Integrates with ERP systems (SAP, Oracle, NetSuite) to pull real data
  - AI-powered recommendations for emission reduction strategies
  - Pricing: $2,500/month (Standard, up to $500M revenue companies)
              $6,500/month (Enterprise, unlimited + custom reporting)
  - 312 active customers as of Q4 2025

2. ESG Reporting Suite
  - Automated generation of sustainability reports (GRI, SASB, TCFD, CDP)
  - Audit-ready documentation with full data lineage
  - Pricing: $18,000/year (bundled with CarbonLens Enterprise)
  - Standalone: $24,000/year
  - Launched April 2024

3. Sustainability Strategy Consulting
  - Science-Based Targets initiative (SBTi) alignment services
  - Supply chain decarbonization roadmaps
  - Pricing: Project-based ($75,000 – $250,000 per engagement)
  - Team of 28 consultants with backgrounds in climate science and industry

4. Carbon Offset Marketplace
  - Verified carbon credits from 45 projects across 12 countries
  - Blockchain-verified chain of custody for each credit
  - Transaction fee: 4.5% per credit purchase
  - Beta launched August 2025

All platform customers receive quarterly sustainability benchmarking reports
comparing their performance against industry peers.""",

    "nextera_impact_report.txt": """"NEXTERA GREEN SOLUTIONS – 2025 IMPACT REPORT
```

Published: January 2026

COMPANY IMPACT METRICS:

- Total customer carbon emissions tracked: 47.3 million metric tons CO2e
- Verified emission reductions enabled: 6.8 million metric tons CO2e (14.4%)
- Customers who improved their CDP score after using Nextera: 89%
- Average time to produce ESG report reduced from 14 weeks to 3.5 weeks

NOTABLE CLIENT RESULTS:

1. SummitBrew Coffee (specialty coffee chain, 380 locations)
 - Reduced Scope 3 supply chain emissions by 31% in 18 months
 - Switched 62% of logistics to electric vehicles using Nextera roadmap
 - First coffee chain to achieve SBTi-validated near-term target
2. Pacific Northwest Lumber Co. (timber and building materials)
 - Used CarbonLens to discover 40% of emissions came from transportation
 - Optimized shipping routes: saved 8,200 metric tons CO2e and \$1.4M/year
 - Achieved carbon-neutral certification for 2025 operations
3. Meridian Financial Group (regional bank, \$8B assets)
 - Implemented financed emissions tracking (PCAF methodology)
 - Identified \$2.1B in loan portfolio exposed to high transition risk
 - Developed green lending program: \$340M in sustainability-linked loans

AWARDS AND RECOGNITION (2025):

- Fast Company "Most Innovative Companies" – Energy category (#7)
 - B Corp "Best for the World" – Environment category
 - Colorado Governor's Award for Clean Energy Innovation"
- ```
}
```

```
NEXTERA_DIR = Path("nextera_docs")
```

```
NEXTERA_DIR.mkdir(exist_ok=True)
```

```
for filename, content in NEXTERA_DOCUMENTS.items():
 (NEXTERA_DIR / filename).write_text(content, encoding="utf-8")

print(f"Created {len(NEXTERA_DOCUMENTS)} documents in '{NEXTERA_DIR}/':")
for f in sorted(NEXTERA_DIR.iterdir()):
 print(f" {f.name} ({f.stat().st_size} bytes)")
```

```
Created 3 documents in 'nextera_docs/':
nextera_impact_report.txt (1434 bytes)
nextera_overview.txt (938 bytes)
nextera_services.txt (1394 bytes)
```

### 13.1.4 Load Documents from Three Company Directories

LangChain provides **Document Loaders** – modular components that load data from various sources. Here we use `DirectoryLoader` to load `.txt` files from three separate company directories, each representing a different fictional company. This is the LangChain equivalent of LlamaIndex's `SimpleDirectoryReader`, but notice how much more explicit it is: you specify the loader class (`TextLoader`), the file glob pattern, and the directory path separately. LangChain favors this explicit, composable style over LlamaIndex's "batteries included" approach.

```
from langchain_community.document_loaders import DirectoryLoader, TextLoader

Load documents from three separate company directories
DOC_DIRS = {
 "acme_docs": "Acme Analytics (BI tools)",
 "globex_docs": "Globex Cybersecurity (security)",
```

```

 "nextera_docs": "Nextera Green Solutions (sustainability)",
}

all_docs = []
for dir_name, label in DOC_DIRS.items():
 loader = DirectoryLoader(dir_name, glob="**/*.txt", loader_cls=TextLoader,
 loader_kwargs={"encoding": "utf-8"})

 docs = loader.load()
 print(f" Loaded: '{dir_name}/' \u2192 {len(docs)} docs [{label}]")
 all_docs.extend(docs)

print(f"\nTotal documents loaded: {len(all_docs)}")

```

```

Loaded: 'acme_docs/' \u2192 5 docs [Acme Analytics (BI tools)]
Loaded: 'globex_docs/' \u2192 3 docs [Globex Cybersecurity (security)]
Loaded: 'nextera_docs/' \u2192 3 docs [Nextera Green Solutions (sustainability)]

```

```
Total documents loaded: 11
```

### 13.1.5 Split, Embed, and Index with FAISS

This is where LangChain's modular architecture becomes visible. Each step in the pipeline is an explicit, swappable component:

- Text Splitter (`RecursiveCharacterTextSplitter`) – breaks documents into smaller chunks, trying to split at paragraph boundaries before falling back to sentence and character boundaries
- Embeddings (`HuggingFaceEmbeddings`) – converts each chunk into a dense vector
- Vector Store (FAISS) – stores and indexes the vectors for fast similarity search

This is the same embed-then-store-then-retrieve pattern as `LlamaIndex`, but here each step is a separate object you can inspect, swap, or configure independently. Want to switch from FAISS to ChromaDB? Just change one line. Want a different chunking strategy? Swap the splitter. That modularity is LangChain's defining philosophy.

```

from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import FAISS

Step 1: Split documents into chunks
text_splitter = RecursiveCharacterTextSplitter(
 chunk_size=1000,
 chunk_overlap=200,
 separators=["\n--\n", "\n\n", "\n", " "], # split on section dividers first
)
chunks = text_splitter.split_documents(all_docs)
print(f"Split {len(all_docs)} documents into {len(chunks)} chunks")

Step 2 & 3: Embed chunks and store in FAISS vector store
vectorstore = FAISS.from_documents(chunks, embeddings)
print(f"FAISS vector store built with {len(chunks)} vectors")

Create a retriever (top-k=3)
retriever = vectorstore.as_retriever(search_kwargs={"k": 3})
print("Retriever ready (top_k=3)")

```

```

Split 11 documents into 15 chunks
FAISS vector store built with 15 vectors
Retriever ready (top_k=3)

```

### 13.1.6 Peek Under the Hood – Test Retrieval

Before querying the LLM, let us see what chunks the retriever returns for a test question. This is exactly the same kind of inspection we did with LlamaIndex – verifying that the right documents are being retrieved before we trust the LLM to generate an answer from them. Understanding what the retriever returns is the single most important debugging step in any RAG pipeline, because if retrieval is wrong, no amount of LLM sophistication can save the answer.

```
Test retrieval without LLM – just see what chunks come back
test_query = "How much revenue did Acme make in Q4 2025?"
retrieved_docs = retriever.invoke(test_query)

print(f"Query: '{test_query}'")
print(f"Retrieved {len(retrieved_docs)} chunks:\n")
for i, doc in enumerate(retrieved_docs, 1):
 source = doc.metadata.get('source', 'unknown')
 text_preview = doc.page_content[:200].replace('\n', ' ')
 print(f" Chunk #{i} (source={source})")
 print(f" {text_preview}...\n")
```

Query: 'How much revenue did Acme make in Q4 2025?'  
Retrieved 3 chunks:

Chunk #1 (source=acme\_docs/q4\_2025\_financials.txt)  
ACME ANALYTICS – Q4 2025 FINANCIAL RESULTS (CONFIDENTIAL) Period: October 1 – December 31, 2025 Revenue: \$12.8 million (Q4) / \$42.3 million (FY 2025)  
Gross Margin: 78.2% Operating Profi...

Chunk #2 (source=acme\_docs/company\_overview.txt)  
ACME ANALYTICS INC. – COMPANY OVERVIEW Acme Analytics Inc. was founded in 2019 by Dr. Sarah Chen and Marcus Rivera in Lawrence, Kansas. The company specializes in AI-powered business intelligence too...

Chunk #3 (source=acme\_docs/employee\_handbook\_excerpt.txt)  
ACME ANALYTICS – EMPLOYEE HANDBOOK (EXCERPT) PAID TIME OFF (PTO): - All full-time employees receive 22 days of PTO per year (accrued monthly). - PTO increases to 27 days after 3 years of service. - U...

### 13.1.7 Build a Retrieval Chain (LCEL)

LangChain’s modern approach uses **LCEL (LangChain Expression Language)** to compose chains with the `|` pipe operator. If you have used Unix shell pipes (`cat file | grep pattern | sort`), the mental model is identical: data flows left to right through a series of transformations.

Here is how to read the chain we build below:

1. The retriever fetches relevant chunks from the FAISS vector store
2. `format_docs` concatenates those chunks into a single text block
3. The prompt template combines the chunks + user question into a structured prompt
4. The LLM generates an answer from that prompt
5. `StrOutputParser()` extracts the text from the LLM’s response object

Each `|` connects one step’s output to the next step’s input. This is more flexible and composable than the older `RetrievalQA` chain – you can insert, remove, or rearrange steps just by editing the pipe expression.

We also set up a direct LLM call function for the “without RAG” comparison.

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser
```

```

def format_docs(docs):
 """Join retrieved documents into a single context string."""
 return (chr(10) * 2 + "---" + chr(10) * 2).join(doc.page_content for doc in
docs)

The prompt template is used by the RAG chain below
rag_prompt = ChatPromptTemplate.from_template(
 """Answer the question based ONLY on the following context.
 If the context doesn't contain the answer, say "I don't have enough information."

 Context:
 {context}

 Question: {question}

 Answer: ""
)

=====
RAG Chain with LCEL (LangChain Expression Language) -- the modern way
=====
LCEL uses the pipe operator `|` to chain Runnables together, similar to
Unix pipes (`cat file.txt | grep foo | wc -l`).
#
HOW TO READ THE PIPE SYNTAX:
#
result = input | step1 | step2 | step3
#
means: input goes into step1, step1's output goes into step2, etc.
Each `|` is "then pass to".
#
Reading top-to-bottom when you call chain.invoke("your question"):
#
1. The DICT at the top runs TWO things in parallel with the same input:
- "context": retriever | format_docs
(fetches docs, then joins them into a string)
- "question": RunnablePassthrough()
(just passes the question through unchanged)
Output: {"context": "...retrieved text...", "question": "your question"}
#
2. | rag_prompt
Fills the {context} and {question} placeholders in the prompt template.
#
3. | llm
Sends the filled prompt to the LLM. Returns an AIMessage object.
#
4. | StrOutputParser()
Extracts .content from the AIMessage so you get a plain string back.

chain = (
 {"context": retriever | format_docs, "question": RunnablePassthrough()}
 | rag_prompt
 | llm
 | StrOutputParser()
)
print("LCEL RAG chain ready: retriever | format_docs | prompt | llm | parser")

=====
ALTERNATIVE: Custom class WITHOUT LangChain (shown for comparison)
=====

```



```

Here's what the same RAG pipeline looks like as a plain Python class,
without using LCEL or LangChain's chaining features. It works, but...
#
WHY LCEL IS USEFUL COMPARED TO THIS CUSTOM CLASS:
- Streaming tokens: for token in chain.stream(q): ... (works automatically
in LCEL)
The custom class would need to be rewritten to handle
streaming.
- Batch processing: chain.batch([q1, q2, q3]) (parallel execution
automatic)
The custom class would need threading/async code added
manually.
- Automatic fallback: chain.with_fallbacks([backup]) (one line)
The custom class would need try/except blocks everywhere.
- Tracing/debugging: works with LangSmith out of the box
The custom class would need manual logging at each step.
- Async support: chain.ainvoke(q) (free)
The custom class would need an async version written
separately.
#
Uncomment the block below to use the custom class approach instead of LCEL.
#
class SimpleRAGChain:
def __init__(self, retriever, prompt, llm):
self.retriever = retriever
self.prompt = prompt
self.llm = llm
#
def invoke(self, question):
Step 1: retrieve relevant docs from vector store
docs = self.retriever.invoke(question)
Step 2: join docs into a single context string
context = format_docs(docs)
Step 3: fill the prompt template
formatted = self.prompt.format(context=context, question=question)
Step 4: send to the LLM
response = self.llm.invoke(formatted)
Step 5: extract the text from the AIMessage object
return response.content
#
chain = SimpleRAGChain(retriever, rag_prompt, llm)

```

LCEL RAG chain ready: retriever | format\_docs | prompt | llm | parser

### 13.1.8 With RAG vs. Without RAG – Three Companies

We test questions about three fictional companies that no LLM has seen in training. Notice how LangChain retrieves from the correct company's documents automatically – even though all three are in the same FAISS index. The vector similarity naturally routes each question to the right documents without any explicit routing logic. This is one of the elegant properties of embedding-based retrieval: semantic similarity does the routing for you.

```

questions = [
 "What was Acme Analytics' revenue in Q4 2025?",
 "How many days of PTO do new employees get at Acme?",
 "What is Globex Cybersecurity's flagship product and how fast does it detect
 threats?",
 "How did Globex help Meridian Health System during a ransomware attack?",
 "Who founded Nextera Green Solutions and where is it headquartered?",
 "How much carbon emissions reduction did Nextera enable for its customers in
 2025?",
]

```



```

def preview(text, max_len=800):
 text = str(text) if text else ""
 return text[:max_len] + ("..." if len(text) > max_len else "")

if not has_llm_provider():
 print("Error: No LLM API key configured. Set at least one API key in Colab Secrets.")
else:
 for i, q in enumerate(questions, start=1):
 print("=" * 80)
 print(f"Q{i}. {q}")
 print("=" * 80)

 try:
 answer = chain.invoke(q)
 print("Answer:")
 print(preview(answer))
 # Show which documents were retrieved
 source_docs = retriever.invoke(q)
 sources = set(doc.metadata.get("source", "unknown") for doc in
source_docs)
 print(f"Sources: {' , '.join(sources)}")
 except Exception as e:
 if _is_retriable_error(e):
 print(" Rate limited -- skipping")
 else:
 print(f" Error: {e}")

 print()

```

```

=====
Q1. Who founded Nextera Green Solutions and where is it headquartered?
=====
Answer:
Nextera Green Solutions was founded by **Dr. Elena Vasquez** and **David Okafor**.
It is headquartered at **1700 Lincoln Street, Suite 3400, Denver, CO 80203** in
Denver, Colorado.
Sources: nextera_docs/nextera_services.txt, nextera_docs/nextera_overview.txt,
nextera_docs/nextera_impact_report.txt

```

### 13.1.9 LangChain vs. LlamaIndex – Comparison

| Aspect           | LangChain                                      | LlamaIndex                                 |
|------------------|------------------------------------------------|--------------------------------------------|
| Philosophy       | Modular toolkit – you compose the pipeline     | Opinionated framework – batteries included |
| Document loading | DirectoryLoader, TextLoader, etc.              | SimpleDirectoryReader                      |
| Text splitting   | Explicit<br>RecursiveCharacterTextSplitter     | Automatic (configurable)                   |
| Vector store     | FAISS, Chroma, Pinecone, etc. (you choose)     | Built-in or pluggable                      |
| Retrieval chain  | LCEL pipe<br>syntax (retriever   prompt   llm) | query_engine                               |
| Lines of code    | More (explicit control)                        | Fewer (more abstraction)                   |

| Aspect   | LangChain                                   | LlamaIndex                         |
|----------|---------------------------------------------|------------------------------------|
| Best for | Custom pipelines, agents, complex workflows | Quick RAG prototypes, document Q&A |

Key takeaway: Both frameworks solve the same problem (RAG), but LangChain gives you more explicit control over each step, while LlamaIndex abstracts more away. Choose based on your needs.

With the RAG pipeline working, let us now see what happens when we give the LLM more autonomy. Instead of a fixed retrieve-then-generate pipeline, what if the model could choose which tools to use based on the question? This is the jump from chains to agents – and it is where LangChain’s architecture really starts to pay off.

### 13.1.10 Beyond RAG: LangChain Agents with Tools

RAG is just one use case for LangChain. Its real power lies in **Agents** – LLMs that can decide which tools to call based on the user’s question. While a RAG chain always follows the same retrieve-then-generate path, an agent reasons about which action to take next.

In this section, we give the LLM three tools:

- Company Lookup – searches our vector store (reusing the RAG retriever from above)
- Calculator – performs arithmetic
- Current Date – returns today’s date

The LLM reads the question, decides which tool to call by outputting a JSON action, executes it, reads the result, and generates a final answer. This is the **ReAct** pattern (Reason + Act), implemented with a simple prompt – no `bind_tools` or special API features required. It works with any LLM provider, which is why we chose this approach over provider-specific tool-calling APIs.

As you read the code below, notice the contrast with the RAG chain: the RAG chain is a fixed pipeline, while the agent is a decision loop where the LLM chooses its own path through the available tools.

```
--- Tool definitions (plain functions) ---

def company_lookup(query):
 """Search the company knowledge base."""
 docs = retriever.invoke(query)
 return chr(10).join(doc.page_content[:500] for doc in docs)

def calculator(expression):
 """Evaluate a math expression like 42.3 * 1.34"""
 try:
 allowed = set('0123456789.+*/() ')
 if not all(c in allowed for c in expression):
 return 'Error: Only basic arithmetic allowed.'
 return str(round(eval(expression), 4))
 except Exception as e:
 return f'Error: {e}'

def current_date(dummy=''):
 """Returns today's date."""
 from datetime import date
 return date.today().strftime('%B %d, %Y')

TOOLS = {
 'company_lookup': company_lookup,
 'calculator': calculator,
 'current_date': current_date,
}
```

```
print('Defined tools:', list(TOOLS.keys()))
```

```
Defined tools: ['company_lookup', 'calculator', 'current_date']
```

### 13.1.11 Create and Run the Agent

We implement a **ReAct agent loop** using LCEL pipes, just like the RAG chain.

The key insight: an agent is a loop around a single step. We express ONE step of the ReAct loop as an LCEL chain:

```
agent_step = invoke_llm | parse_tool_call | execute_tool
```

Each step:

- `invoke_llm` — sends the current message history to the LLM and captures the reply
- `parse_tool_call` — checks if the reply contains a JSON tool call
- `execute_tool` — if there's a tool call, runs the tool and appends the result to the message history; otherwise returns the final answer

Then we call `agent_step.invoke(state)` in a Python `for` loop until the agent returns a final answer (or we hit a step limit). The Python loop is needed because LCEL chains are linear by design — they can't natively express “repeat until condition”. This is exactly the kind of situation where LangGraph shines (see the `LangGraph_demo` notebook), since it's built specifically for stateful graph-based workflows with branches and loops.

#### 13.1.11.1 Why do we need `RunnableLambda`?

LCEL's pipe operator `|` only works between **Runnables** — objects that have an `.invoke()` method. Plain Python functions don't have a `|` operator defined, so `my_func1 | my_func2` would fail with a `TypeError`.

`RunnableLambda(func)` wraps a plain function into a Runnable, giving it an `.invoke()` method and enabling the pipe operator. You only need to wrap ONE function to bootstrap the pipe — once the left side is a Runnable, LangChain auto-coerces any plain functions on the right into Runnables.

That's why our chain looks like:

```
agent_step = RunnableLambda(invoke_llm) | parse_tool_call | execute_tool
```

Only the first function needs explicit wrapping. The same thing happens in the earlier RAG chain: `retriever | format_docs` works without wrapping `format_docs`, because `retriever` is already a Runnable (LangChain auto-wraps `format_docs`).

```
import json as _json
import re as _re
from langchain_core.messages import HumanMessage, AIMessage, SystemMessage
from langchain_core.runnables import RunnableLambda

AGENT_SYSTEM = """You are a helpful business analyst. You have these tools:
company_lookup(query): Search knowledge base about Acme, Globex, or Nextera.
calculator(expression): Evaluate math like 42.3 * 1.34
current_date(): Get today's date.

To use a tool, reply with ONLY this JSON (nothing else):
{"tool": "tool_name", "input": "the input"}

After receiving a tool result, answer the original question.
If you can answer without a tool, just reply directly."""

---- Three small functions that become LCEL steps via RunnableLambda ----
```

```

def invoke_llm(state):
 """LCEL step 1: send messages to LLM, capture the reply."""
 response = llm.invoke(state["messages"])
 return {**state, "reply": response.content.strip()}

def parse_tool_call(state):
 """LCEL step 2: check if reply contains a tool call JSON."""
 tool_call = None
 m = re.search(r"\{(?:[^}]+\})*\}", state["reply"])
 if m:
 try:
 parsed = json.loads(m.group())
 if "tool" in parsed and "input" in parsed:
 tool_call = parsed
 except json.JSONDecodeError:
 pass
 return {**state, "tool_call": tool_call}

def execute_tool(state):
 """LCEL step 3: run the tool (or mark as final answer if none)."""
 if state["tool_call"] is None:
 # No tool call -- this is the final answer
 return {**state, "final": state["reply"]}

 # Execute the requested tool
 tool_name = state["tool_call"]["tool"]
 tool_input = state["tool_call"]["input"]
 tool_fn = TOOLS.get(tool_name)
 if tool_fn:
 result = tool_fn(tool_input)
 else:
 result = f"Unknown tool: {tool_name}"

 # Append assistant reply and tool result to the message history
 new_messages = state["messages"] + [
 AIMessage(content=state["reply"]),
 HumanMessage(content=f"Tool result: {result}"),
]
 return {
 **state,
 "messages": new_messages,
 "final": None,
 "last_tool_name": tool_name,
 "last_tool_input": tool_input,
 "last_tool_result": result,
 }

---- Build the LCEL agent-step chain: invoke_llm | parse_tool_call | execute_tool

LCEL's pipe (|) only works between Runnables. Plain Python functions don't
have a | operator, so we wrap the FIRST function as RunnableLambda to start
the pipe. Once the left side is a Runnable, LangChain auto-coerces any plain
functions on the right into Runnables. So only ONE RunnableLambda is needed.
agent_step = RunnableLambda(invoke_llm) | parse_tool_call | execute_tool

def run_agent(question, verbose=True):
 """Run the LCEL agent_step in a loop until it returns a final answer."""
 state = {

```

```

 "messages": [SystemMessage(content=AGENT_SYSTEM),
HumanMessage(content=question)],
 "final": None,
 }

 for step in range(5):
 state = agent_step.invoke(state)

 if state["final"] is not None:
 if verbose:
 print(f" [Step {step+1}] Final answer")
 return state["final"]

 if verbose:
 print(f" [Step {step+1}] Calling: {state['last_tool_name']}
({state['last_tool_input']})")
 print(f" Result: {str(state['last_tool_result'])[:150]}...")

 return "Agent exceeded maximum steps."

print("LCEL agent ready: agent_step = invoke_llm | parse_tool_call | execute_tool")

```

```
LCEL agent ready: agent_step = invoke_llm | parse_tool_call | execute_tool
```

```

agent_questions = [
 #'Who is the CTO of Globex Cybersecurity?',
 'Acme Analytics made $42.3 million in 2025. If they grow 34%, what would revenue
be?',
 #"What is today's date?",
 #'What awards did Nextera Green Solutions win in 2025?',
 #'Globex has 412 employees and $78.5M revenue. What is revenue per employee?',
]

if has_llm_provider():
 for i, q in enumerate(agent_questions, start=1):
 print('=' * 80)
 print(f'Agent Q{i}. {q}')
 print('=' * 80)
 try:
 result = run_agent(q)
 print(f'Final Answer: {result}')
 except Exception as e:
 if _is_retriable_error(e):
 print(' Rate limited -- skipping')
 else:
 print(f' Agent error: {e}')
 print()
 else:
 print('Set at least one API key first.')

```

```

=====
Agent Q1. Acme Analytics made $42.3 million in 2025. If they grow 34%, what would
revenue be?
=====

```

```

[Step 1] Calling: calculator(42.3 * 1.34)
Result: 56.682...

```

```
[Step 2] Final answer
```

```

Final Answer: If Acme Analytics grows 34% from $42.3 million, their revenue would be
$56.68 million (or $56.7 million rounded).

```

### 13.1.11.2 Key Takeaways: Agents vs. Chains

|                  | RAG Chain (LCEL)                   | Agent with Tools                             |
|------------------|------------------------------------|----------------------------------------------|
| Decision making  | Always retrieves, always generates | LLM chooses which tools to call              |
| Multi-step       | Single retrieve then generate      | Can chain multiple tool calls                |
| Math / logic     | LLM must do math in its head       | Delegates to calculator tool                 |
| Extensibility    | Fixed pipeline                     | Add any tool (APIs, databases, web search)   |
| Provider support | Any LLM                            | Any LLM (prompt-based, no bind_tools needed) |

When to use which:

- RAG Chain – simple document Q&A with a known pipeline
- Agent – complex questions requiring reasoning, multiple data sources, or computation

### 13.1.12 FAQ: Common Questions About LCEL and Agents

If the pipe syntax and `RunnableLambda` wrappers felt confusing on first read, you are not alone. These are the most common questions students ask after working through this notebook, and understanding the answers will make every future LangChain project easier to read and write.

#### 13.1.12.1 Why do we need `RunnableLambda` for the agent but not the RAG chain?

LCEL's pipe operator `|` only works between Runnables (objects with an `.invoke()` method).

In the RAG chain:

```
chain = retriever | format_docs | rag_prompt | llm | StrOutputParser()
```

`retriever` is already a Runnable (it inherits from LangChain's `BaseRetriever`), so the pipe starts with a Runnable. When LangChain sees `retriever | format_docs`, it auto-wraps `format_docs` (a plain Python function) into a `RunnableLambda` internally. Every subsequent `|` keeps the chain going.

In the agent:

```
agent_step = RunnableLambda(invoke_llm) | parse_tool_call | execute_tool
```

None of our three functions are Runnables – they're plain Python functions we defined. If we wrote `invoke_llm | parse_tool_call | execute_tool`, Python would try to call `function.__or__(function)`, which doesn't exist, and raise `TypeError`. That's why we wrap the first function explicitly. Once the left side is a Runnable, LangChain auto-coerces the rest.

**Rule:** Wrap ONE function as `RunnableLambda` to bootstrap the pipe. The rest are auto-coerced.

#### 13.1.12.2 Why three steps for the agent? Why not one big function?

Each LCEL step does ONE thing – the same “small composable pieces” philosophy as the RAG chain. Mapping the steps to what they do:

| Step                         | What it does                                                                  |
|------------------------------|-------------------------------------------------------------------------------|
| <code>invoke_llm</code>      | Send message history to LLM, get its reply                                    |
| <code>parse_tool_call</code> | Check if the reply is a tool call (JSON) or a final answer                    |
| <code>execute_tool</code>    | If tool call -> run the tool and update history. If final answer -> mark done |

You could write all three as ONE big function, but then:

- You lose LCEL's streaming/batching/tracing benefits
- Each step can't be unit-tested or swapped independently
- It's less readable than three small functions with clear names

### 13.1.13 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

[https://colab.research.google.com/github/KarAnalytics/code\\_demos/blob/main/LangChain\\_demo.ipynb](https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/LangChain_demo.ipynb)





## Chapter 14

# LangGraph

LangGraph extends LangChain with **stateful graphs** — pipelines where each step is a visible node connected by edges, and the state flows explicitly between them. This chapter demonstrates three increasingly complex graph patterns:

- **Linear RAG** — `retrieve → generate → END` (same as a LangChain chain, but with visible state)
- **Conditional routing** — a classifier node routes questions to specialized handlers via conditional edges
- **Retry loop** — the graph evaluates its own answer and loops back to retry if insufficient

Each pattern includes a graph visualization and a state trace showing how data flows through the nodes.

## 14.1 LangGraph RAG Demo: Graph-Based Retrieval Pipeline

**Estimated run time:** ~8 minutes (multiple graph executions + LLM calls)

This notebook demonstrates how **LangGraph** lets you build RAG pipelines as **explicit stateful graphs**. Unlike LangChain's chain abstraction where steps are implicit, LangGraph makes every step a visible **node** connected by **edges** — giving you full control over the flow.

### How it works:

- Load **synthetic company documents** from three separate directories.
- Build a FAISS vector store (same as the LangChain notebook).
- Define a **graph** with three nodes: `retrieve` `arrow.r` `generate` `arrow.r` `format_answer`.
- Run questions through the graph and compare **with RAG** vs. **without RAG**.

### Why LangGraph?

- Each step in your pipeline is a **node** — easy to debug, test, and modify independently
- **State** flows through the graph, so you can inspect what each node produced
- You can add **conditional edges** (e.g., route to different retrievers based on the question)
- Natural fit for **agentic workflows** where the LLM decides what to do next

**Provider setup:** This notebook uses the `llm_cascade` package, which auto-detects your API keys and falls back to the next provider if one is unavailable. Supported providers: OpenAI, Gemini, Ollama, Grok (xAI), Groq, HuggingFace, Cohere, OpenRouter.

LangGraph sits on top of the same LangChain primitives you saw in Chapter 13's `langchain_demo.ipynb` — the retriever, embeddings, and prompts are unchanged; what changes is that the linear LCEL pipe gets promoted into an explicit graph you can branch and loop over.

```
!pip install -q -U langgraph langchain langchain-community langchain-huggingface
langchain-text-splitters faiss-cpu google-genai openai langchain-google-genai
git+https://github.com/KarAnalytics/llm_cascade.git sentence-transformers
```

### 14.1.1 Imports and Provider Helpers (7-Vendor Cascade)

Same provider cascade used across all our RAG notebooks. By this point the setup should feel familiar – and that is intentional. The infrastructure layer (API keys, provider fallback, embeddings) stays constant across LlamaIndex, LangChain, and LangGraph notebooks so you can focus on what changes: the orchestration pattern. LangGraph's contribution is not a new way to call LLMs, but a new way to structure the flow between calls.

```
from pathlib import Path
from llm_cascade.providers import PROVIDERS, _load_env, _get_key,
_is_retriable_error

_load_env()

def get_available_providers():
 return [p for p in PROVIDERS if _get_key(p['key_env'])]

def has_llm_provider():
 return len(get_available_providers()) > 0

Print status
available = get_available_providers()
if available:
 print('Providers configured (in fallback order):')
 for p in available:
 print(f" + {p['name']:<16} model = {p['default_model']}")
```

```
else:
 print('WARNING: No API keys found.')
```

```
Providers configured (in fallback order):
+ Gemini model = gemini-2.5-flash
+ Ollama model = kimi-k2.5:cloud
+ Groq model = llama-3.3-70b-versatile
+ HuggingFace model = meta-llama/Llama-3.3-70B-Instruct
+ Cohere model = command-a-03-2025
+ OpenRouter model = meta-llama/llama-3.3-70b-instruct:free
+ OpenAI model = gpt-4o-mini
```

### 14.1.2 Configure LLM, Embeddings, and Build Vector Store

Same setup as the LangChain notebook: local HuggingFace embeddings + first available LLM provider. We also load documents and build the FAISS vector store here. This entire section is deliberately identical to what we did in the LangChain notebook – the vector store, the embeddings, the document loading are all the same. The only thing that changes in LangGraph is *how we orchestrate* the retrieval and generation steps, not the steps themselves.

### 14.1.3 Create Synthetic Company Documents

We generate **fictional documents for three companies** that the LLM has never seen in training. This ensures that correct answers can **only** come from RAG retrieval, not from parametric memory. Each company's documents live in a **separate directory**, simulating real-world data silos.

| Directory     | Company                                                     | Documents                                            |
|---------------|-------------------------------------------------------------|------------------------------------------------------|
| acme_docs/    | Acme Analytics Inc. (BI tools, Lawrence KS)                 | Overview, Products, Financials, Handbook, Case Study |
| globex_docs/  | Globex Cybersecurity Corp. (cybersecurity, Boston)          | Overview, Products, Incident Report                  |
| nextera_docs/ | Nextera Green Solutions (sustainability consulting, Denver) | Overview, Services, Impact Report                    |

```
DOCUMENTS = {
 "company_overview.txt": """ACME ANALYTICS INC. – COMPANY OVERVIEW

Acme Analytics Inc. was founded in 2019 by Dr. Sarah Chen and Marcus Rivera
in Lawrence, Kansas. The company specializes in AI-powered business intelligence
tools for mid-market companies (100–5,000 employees).

Headquarters: 1420 Jayhawk Boulevard, Lawrence, KS 66045
Employees: 287 (as of January 2026)
Annual Revenue (2025): $42.3 million
Funding: Series B ($18M raised in March 2023 from Midwest Ventures)

Mission: "To democratize data analytics so every business decision is informed
by evidence, not intuition."

The company operates three offices: Lawrence (HQ), Chicago (sales), and
Austin (engineering). CEO: Dr. Sarah Chen. CTO: Marcus Rivera. CFO: Linda Park.""",
 "products_and_services.txt": """ACME ANALYTICS – PRODUCTS AND SERVICES

1. InsightBoard Pro (Flagship Product)
- Real-time dashboard platform with natural language query interface
- Pricing: $49/user/month (Standard), $89/user/month (Enterprise)
- Supports PostgreSQL, MySQL, Snowflake, BigQuery, and Redshift
- 1,847 active enterprise customers as of Q4 2025
```

## 2. DataPipe ETL

- Automated data pipeline builder with 200+ pre-built connectors
- Pricing: starts at \$500/month for up to 10 million records/day
- Launched in September 2024

## 3. PredictIQ

- ML-powered forecasting add-on for InsightBoard Pro
- Pricing: \$29/user/month (requires InsightBoard Pro subscription)
- Uses proprietary time-series model trained on retail and logistics data
- Beta launched March 2025, GA release planned for June 2026

All products include 24/7 email support. Enterprise plans include dedicated account manager and 99.9% SLA."",

"q4\_2025\_financials.txt": ""ACME ANALYTICS – Q4 2025 FINANCIAL RESULTS (CONFIDENTIAL)

Period: October 1 – December 31, 2025

Revenue: \$12.8 million (Q4) / \$42.3 million (FY 2025)  
Gross Margin: 78.2%  
Operating Profit: \$1.9 million (Q4) / \$5.1 million (FY 2025)  
Net Income: \$1.4 million (Q4)  
Cash on Hand: \$14.7 million  
Burn Rate: Company is cash-flow positive since Q2 2025

### Key Metrics:

- ARR (Annual Recurring Revenue): \$48.6 million (up 34% YoY)
- Net Revenue Retention: 118%
- Customer Acquisition Cost (CAC): \$8,200
- Customer Lifetime Value (LTV): \$67,400
- LTV/CAC Ratio: 8.2x

Headcount grew from 241 to 287 employees during 2025.

R&D spending: 31% of revenue. Sales & Marketing: 28% of revenue."",

"employee\_handbook\_excerpt.txt": ""ACME ANALYTICS – EMPLOYEE HANDBOOK (EXCERPT)

### PAID TIME OFF (PTO):

- All full-time employees receive 22 days of PTO per year (accrued monthly).
- PTO increases to 27 days after 3 years of service.
- Unused PTO can be carried over (max 5 days) or paid out at year-end.
- Sick leave: 10 days per year (separate from PTO).

### REMOTE WORK POLICY:

- Engineering and Data Science teams: fully remote eligible.
- Sales and Customer Success: hybrid (minimum 2 days/week in office).
- All employees may work remotely up to 4 weeks/year from any US location.
- International remote work requires VP approval and tax review.

### PROFESSIONAL DEVELOPMENT:

- Annual learning budget: \$2,500 per employee.
- Conference attendance: up to 2 conferences per year with manager approval.
- Tuition reimbursement: up to \$5,250/year for degree programs."",

"customer\_case\_study.txt": ""CUSTOMER CASE STUDY: BIGRETAIL CORP

Company: BigRetail Corp (1,200 retail stores across 38 states)

Challenge: Siloed data across POS, inventory, and CRM systems made it impossible for regional managers to get timely insights.

Solution: Deployed InsightBoard Pro Enterprise + DataPipe ETL

- Connected 14 data sources in 3 weeks using DataPipe
- 340 regional managers now use InsightBoard daily
- Natural language queries replaced manual SQL report requests

Results (after 6 months):

- 62% reduction in time-to-insight (from 4 days to 1.5 days average)
- \$3.2 million saved in inventory carrying costs
- 23% increase in regional manager satisfaction scores
- SQL report request backlog eliminated entirely

Quote from BigRetail CIO Janet Torres:

"InsightBoard Pro transformed how our managers interact with data. They went from waiting days for a report to asking questions in plain English and getting answers in seconds."

```
Write documents to disk
DOC_DIR = Path("acme_docs")
DOC_DIR.mkdir(exist_ok=True)

for filename, content in DOCUMENTS.items():
 (DOC_DIR / filename).write_text(content, encoding="utf-8")

print(f"Created {len(DOCUMENTS)} documents in '{DOC_DIR}/':")
for f in sorted(DOC_DIR.iterdir()):
 print(f" {f.name} ({f.stat().st_size} bytes)")
```

```
Created 5 documents in 'acme_docs/':
 company_overview.txt (715 bytes)
 customer_case_study.txt (924 bytes)
 employee_handbook_excerpt.txt (828 bytes)
 products_and_services.txt (915 bytes)
 q4_2025_financials.txt (709 bytes)
```

# --- Company 2: Globex Cybersecurity Corp. → globex\_docs/ ---

```
GLOBEX_DOCUMENTS = {
 "globex_overview.txt": """GLOBEX CYBERSECURITY CORP. – COMPANY OVERVIEW
```

Globex Cybersecurity Corp. was founded in 2017 by former NSA analyst James Whitfield and AI researcher Dr. Priya Narayanan in Boston, Massachusetts. The company provides AI-driven cybersecurity solutions for financial services and healthcare organizations.

Headquarters: 88 Federal Street, Suite 1200, Boston, MA 02110  
 Employees: 412 (as of January 2026)  
 Annual Revenue (2025): \$78.5 million  
 Funding: Series C (\$45M raised in June 2024 from CyberVentures Capital and Goldman Sachs Growth Equity)

Mission: "To make enterprise-grade cybersecurity autonomous, adaptive, and accessible to every regulated industry."

The company operates four offices: Boston (HQ), Washington D.C. (government relations), Tel Aviv (R&D), and Singapore (APAC sales).  
 CEO: James Whitfield. CTO: Dr. Priya Narayanan. CISO: Robert Tanaka."

```
 "globex_products.txt": """GLOBEX CYBERSECURITY – PRODUCTS AND SERVICES
```

1. ShieldAI Platform (Flagship Product)
  - AI-powered threat detection and automated incident response

- Uses behavioral analysis to detect zero-day attacks in real time
- Pricing: \$120/endpoint/year (Standard), \$195/endpoint/year (Premium)
- 623 enterprise customers across 18 countries as of Q4 2025
- Average detection time: 2.3 minutes (industry average: 197 days)

## 2. VaultGuard

- Data encryption and access management platform for regulated industries
- HIPAA, SOX, and PCI-DSS compliant out of the box
- Pricing: \$35,000/year (up to 500 users), \$60,000/year (unlimited users)
- Launched January 2023

## 3. ThreatIntel Live

- Real-time threat intelligence feed with dark web monitoring
- Pricing: \$8,500/month (includes API access and analyst dashboard)
- Aggregates data from 2,400+ sources updated every 15 minutes
- Integrates with Splunk, CrowdStrike, Palo Alto, and Microsoft Sentinel

All products include a dedicated Security Operations Center (SOC) team for Premium customers. Globex holds ISO 27001 and SOC 2 Type II certifications.""",

"globex\_incident\_report.txt": ""GLOBEX CYBERSECURITY – 2025 THREAT LANDSCAPE REPORT (PUBLIC)

Published: February 2026

### KEY FINDINGS FROM GLOBEX'S CUSTOMER BASE (2025):

Total incidents analyzed: 14,237 across 623 customers

- Ransomware attacks: up 42% YoY (3,891 incidents)
- Phishing campaigns: up 28% YoY (5,102 incidents)
- Supply chain attacks: up 67% YoY (1,245 incidents – fastest-growing category)
- Insider threats: relatively stable at 1,834 incidents

### ShieldAI Platform Performance:

- Blocked 97.3% of attacks before any data exfiltration
- Mean time to detect (MTTD): 2.3 minutes
- Mean time to respond (MTTR): 8.7 minutes (automated response)
- False positive rate: 0.4% (down from 1.2% in 2024)

### NOTABLE CASE: MERIDIAN HEALTH SYSTEM

- 340-bed hospital network targeted by Medusa ransomware variant
- ShieldAI detected anomalous lateral movement within 47 seconds
- Automated containment isolated 3 affected servers in under 2 minutes
- Zero patient data exfiltrated; zero downtime for clinical systems
- Estimated savings: \$12M+ (average healthcare ransomware cost per Ponemon)

### INDUSTRY PREDICTIONS FOR 2026:

- AI-generated phishing emails will become indistinguishable from human-written
- Quantum-resistant encryption adoption will begin in financial services
- Average cost of a data breach expected to exceed \$5.2 million globally""",

```
GLOBEX_DIR = Path("globex_docs")
```

```
GLOBEX_DIR.mkdir(exist_ok=True)
```

```
for filename, content in GLOBEX_DOCUMENTS.items():
 (GLOBEX_DIR / filename).write_text(content, encoding="utf-8")
```

```
print(f"Created {len(GLOBEX_DOCUMENTS)} documents in '{GLOBEX_DIR}/':")
```

```
for f in sorted(GLOBEX_DIR.iterdir()):
 print(f" {f.name} ({f.stat().st_size} bytes)")
```

```
Created 3 documents in 'globex_docs/':
globex_incident_report.txt (1321 bytes)
globex_overview.txt (864 bytes)
globex_products.txt (1131 bytes)
```

```
--- Company 3: Nextera Green Solutions → nextera_docs/ ---
```

```
NEXTERA_DOCUMENTS = {
 "nextera_overview.txt": ""NEXTERA GREEN SOLUTIONS – COMPANY OVERVIEW

Nextera Green Solutions was founded in 2020 by climate scientist Dr. Elena
Vasquez and former McKinsey partner David Okafor in Denver, Colorado.
The company provides sustainability consulting and carbon accounting software
to help companies meet ESG (Environmental, Social, Governance) targets.

Headquarters: 1700 Lincoln Street, Suite 3400, Denver, CO 80203
Employees: 156 (as of January 2026)
Annual Revenue (2025): $23.8 million
Funding: Series A ($12M raised in November 2023 from Climate Capital Partners
and Patagonia Ventures)

Mission: "To give every company a clear, data-driven path to net zero –
turning climate commitments into measurable action."

The company operates two offices: Denver (HQ) and London (European operations).
CEO: Dr. Elena Vasquez. COO: David Okafor. VP Engineering: Kenji Tanaka.
Nextera is a certified B Corporation and a member of the UN Global Compact.""",

 "nextera_services.txt": ""NEXTERA GREEN SOLUTIONS – SERVICES AND PLATFORM

1. CarbonLens Platform (Flagship Product)
- Automated Scope 1, 2, and 3 carbon emissions tracking
- Integrates with ERP systems (SAP, Oracle, NetSuite) to pull real data
- AI-powered recommendations for emission reduction strategies
- Pricing: $2,500/month (Standard, up to $500M revenue companies)
 $6,500/month (Enterprise, unlimited + custom reporting)
- 312 active customers as of Q4 2025

2. ESG Reporting Suite
- Automated generation of sustainability reports (GRI, SASB, TCFD, CDP)
- Audit-ready documentation with full data lineage
- Pricing: $18,000/year (bundled with CarbonLens Enterprise)
- Standalone: $24,000/year
- Launched April 2024

3. Sustainability Strategy Consulting
- Science-Based Targets initiative (SBTi) alignment services
- Supply chain decarbonization roadmaps
- Pricing: Project-based ($75,000 – $250,000 per engagement)
- Team of 28 consultants with backgrounds in climate science and industry

4. Carbon Offset Marketplace
- Verified carbon credits from 45 projects across 12 countries
- Blockchain-verified chain of custody for each credit
- Transaction fee: 4.5% per credit purchase
- Beta launched August 2025

All platform customers receive quarterly sustainability benchmarking reports
comparing their performance against industry peers.""",

 "nextera_impact_report.txt": ""NEXTERA GREEN SOLUTIONS – 2025 IMPACT REPORT
```

Published: January 2026

#### COMPANY IMPACT METRICS:

- Total customer carbon emissions tracked: 47.3 million metric tons CO2e
- Verified emission reductions enabled: 6.8 million metric tons CO2e (14.4%)
- Customers who improved their CDP score after using Nextera: 89%
- Average time to produce ESG report reduced from 14 weeks to 3.5 weeks

#### NOTABLE CLIENT RESULTS:

1. SummitBrew Coffee (specialty coffee chain, 380 locations)
  - Reduced Scope 3 supply chain emissions by 31% in 18 months
  - Switched 62% of logistics to electric vehicles using Nextera roadmap
  - First coffee chain to achieve SBTi-validated near-term target
2. Pacific Northwest Lumber Co. (timber and building materials)
  - Used CarbonLens to discover 40% of emissions came from transportation
  - Optimized shipping routes: saved 8,200 metric tons CO2e and \$1.4M/year
  - Achieved carbon-neutral certification for 2025 operations
3. Meridian Financial Group (regional bank, \$8B assets)
  - Implemented financed emissions tracking (PCAF methodology)
  - Identified \$2.1B in loan portfolio exposed to high transition risk
  - Developed green lending program: \$340M in sustainability-linked loans

#### AWARDS AND RECOGNITION (2025):

- Fast Company "Most Innovative Companies" – Energy category (#7)
  - B Corp "Best for the World" – Environment category
  - Colorado Governor's Award for Clean Energy Innovation""
- }

```
NEXTERA_DIR = Path("nextera_docs")
```

```
NEXTERA_DIR.mkdir(exist_ok=True)
```

```
for filename, content in NEXTERA_DOCUMENTS.items():
 (NEXTERA_DIR / filename).write_text(content, encoding="utf-8")

print(f"Created {len(NEXTERA_DOCUMENTS)} documents in '{NEXTERA_DIR}/':")
for f in sorted(NEXTERA_DIR.iterdir()):
 print(f" {f.name} ({f.stat().st_size} bytes)")
```

```
Created 3 documents in 'nextera_docs/':
nextera_impact_report.txt (1434 bytes)
nextera_overview.txt (938 bytes)
nextera_services.txt (1394 bytes)
```

```
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.chat_models import ChatOpenAI

---- Embedding model (local, no API key needed) -----
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
print("Embedding model: sentence-transformers/all-MiniLM-L6-v2 (local)")

---- LLM with runtime fallback (cascades if quota is exhausted) -----
llm_candidates = []
for provider in get_available_providers():
 api_key = _get_key(provider["key_env"])
 try:
 if provider["style"] == "gemini":
 from langchain_google_genai import ChatGoogleGenerativeAI
```



```

 candidate = ChatGoogleGenerativeAI(
 google_api_key=api_key,
 model=provider["default_model"],
)
 else:
 candidate = ChatOpenAI(
 api_key=api_key,
 base_url=provider["base_url"],
 model=provider["default_model"],
)
 llm_candidates.append((provider, candidate))
 print(f" + {provider['name']:<16} ({provider['default_model']})")
except Exception as e:
 print(f" - {provider['name']:<16} skipped: {e}")

if llm_candidates:
 primary_provider, primary_llm = llm_candidates[0]
 fallback_llms = [c for _, c in llm_candidates[1:]]

 if fallback_llms:
 llm = primary_llm.with_fallbacks(fallback_llms)
 print(f"\nPrimary LLM: {primary_provider['name']} ({primary_provider['default_model']})")
 print(f"Fallbacks: {', '.join(p['name'] for p, _ in llm_candidates[1:])}")
 print("(If primary hits quota, automatically tries the next provider)")
 else:
 llm = primary_llm
 print(f"\nLLM provider: {primary_provider['name']} ({primary_provider['default_model']})")
 print("(No fallback providers configured)")

 llm_provider = primary_provider
else:
 llm = None
 primary_llm = None
 llm_provider = None
 print("ERROR: Could not configure any LLM provider. Check your API keys.")

---- Load documents from three company directories -----
from langchain_community.document_loaders import DirectoryLoader, TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import FAISS

DOC_DIRS = {
 "acme_docs": "Acme Analytics (BI tools)",
 "globex_docs": "Globex Cybersecurity (security)",
 "nextera_docs": "Nextera Green Solutions (sustainability)",
}

all_docs = []
for dir_name, label in DOC_DIRS.items():
 loader = DirectoryLoader(dir_name, glob="**/*.txt", loader_cls=TextLoader,
 loader_kwargs={"encoding": "utf-8"})
 docs = loader.load()
 print(f" Loaded: '{dir_name}' -> {len(docs)} docs [{label}]")
 all_docs.extend(docs)

---- Split and index -----
text_splitter = RecursiveCharacterTextSplitter(
 chunk_size=1000, chunk_overlap=200,
 separators=["\n--\n", "\n\n", "\n", " "],
)
chunks = text_splitter.split_documents(all_docs)

```

```
vectorstore = FAISS.from_documents(chunks, embeddings)
retriever = vectorstore.as_retriever(search_kwargs={"k": 3})
print(f"FAISS vector store: {len(chunks)} chunks indexed, retriever ready
(top_k=3)")
```

```
modules.json: 0%| | 0.00/349 [00:00<?, ?B/s]
```

```
config_sentence_transformers.json: 0%| | 0.00/116 [00:00<?, ?B/s]
```

```
README.md: 0.00B [00:00, ?B/s]
```

```
sentence_bert_config.json: 0%| | 0.00/53.0 [00:00<?, ?B/s]
```

```
config.json: 0%| | 0.00/612 [00:00<?, ?B/s]
```

```
model.safetensors: 0%| | 0.00/90.9M [00:00<?, ?B/s]
```

```
Loading weights: 0%| | 0/103 [00:00<?, ?it/s]
```

```
BertModel LOAD REPORT from: sentence-transformers/all-MiniLM-L6-v2
```

| Key                     | Status     |  |
|-------------------------|------------|--|
| embeddings.position_ids | UNEXPECTED |  |

Notes:

- UNEXPECTED :can be ignored when loading from different task/architecture; not ok if you expect identical arch.

```
tokenizer_config.json: 0%| | 0.00/350 [00:00<?, ?B/s]
```

```
vocab.txt: 0.00B [00:00, ?B/s]
```

```
tokenizer.json: 0.00B [00:00, ?B/s]
```

```
special_tokens_map.json: 0%| | 0.00/112 [00:00<?, ?B/s]
```

```
config.json: 0%| | 0.00/190 [00:00<?, ?B/s]
```

```
Embedding model: sentence-transformers/all-MiniLM-L6-v2 (local)
+ Gemini (gemini-2.5-flash)
```

```
/tmp/ipykernel_6060/2431139388.py:20: LangChainDeprecationWarning: The class
`ChatOpenAI` was deprecated in LangChain 0.0.10 and will be removed in 1.0. An
updated version of the class exists in the `langchain-openai` package and should be
used instead. To use it run `pip install -U `langchain-openai` and import as `from
`langchain_openai import ChatOpenAI``.
 candidate = ChatOpenAI(
```

```
+ Ollama (kimi-k2.5:cloud)
+ Groq (llama-3.3-70b-versatile)
+ HuggingFace (meta-llama/Llama-3.3-70B-Instruct)
```

```

+ Cohere (command-a-03-2025)
+ OpenRouter (meta-llama/llama-3.3-70b-instruct:free)
+ OpenAI (gpt-4o-mini)

Primary LLM: Gemini (gemini-2.5-flash)
Fallbacks: Ollama, Groq, HuggingFace, Cohere, OpenRouter, OpenAI
(If primary hits quota, automatically tries the next provider)
 Loaded: 'acme_docs/' -> 5 docs [Acme Analytics (BI tools)]
 Loaded: 'globex_docs/' -> 3 docs [Globex Cybersecurity (security)]
 Loaded: 'nextera_docs/' -> 3 docs [Nextera Green Solutions (sustainability)]
FAISS vector store: 15 chunks indexed, retriever ready (top_k=3)

```

#### 14.1.4 Define the RAG Graph

This is where LangGraph diverges from LangChain. Instead of composing steps with the `|` pipe operator, we build an **explicit graph** where each step is a **node** and the connections between them are **edges**. The state that flows through the graph is defined as a Python `TypedDict` – a plain dictionary with typed fields like `question`, `context`, `answer`, and `sources`. Every node is a function that reads from this state dict and writes back to it.

| Node     | What it does                                     |
|----------|--------------------------------------------------|
| retrieve | Searches the vector store for relevant chunks    |
| generate | Sends the question + retrieved chunks to the LLM |

The graph flows: **START** -> **retrieve** -> **generate** -> **END**

Why bother with a graph for what is essentially a two-step pipeline? Because this simple linear graph is a stepping stone. Once you have the graph infrastructure in place, adding conditional branches, retry loops, or parallel paths is just a matter of adding nodes and edges – no restructuring required. The linear graph teaches you the API; the conditional and looping graphs that follow show why it matters.

```
START --> retrieve --> generate --> END
```

```

from typing import TypedDict
from langgraph.graph import StateGraph, START, END

Step 1: Define the STATE that flows through the graph

class RAGState(TypedDict):
 question: str # user's question (input)
 context: str # retrieved chunks (set by retrieve node)
 sources: list[str] # source file names (set by retrieve node)
 answer: str # LLM's answer (set by generate node)

Step 2: Define the NODES (each is a plain function)

def retrieve(state: RAGState) -> dict:
 """Search the vector store for relevant chunks."""
 docs = retriever.invoke(state["question"])
 context = "\n\n---\n\n".join(doc.page_content for doc in docs)
 sources = list(set(doc.metadata.get("source", "unknown") for doc in docs))
 return {"context": context, "sources": sources}

def generate(state: RAGState) -> dict:
 """Send question + retrieved context to the LLM."""
 prompt = f"""Answer the question based ONLY on the following context.

```

If the context doesn't contain the answer, say "I don't have enough information."

Context:

```
{state['context']}
```

Question: {state['question']}

Answer:""

```
 response = llm.invoke(prompt)
 return {"answer": response.content}
```

```

Step 3: Build the GRAPH

workflow = StateGraph(RAGState)

Add nodes
workflow.add_node("retrieve", retrieve)
workflow.add_node("generate", generate)

Add edges: START → retrieve → generate → END
workflow.add_edge(START, "retrieve")
workflow.add_edge("retrieve", "generate")
workflow.add_edge("generate", END)

Compile the graph
graph = workflow.compile()
print("RAG graph compiled: START \u2192 retrieve \u2192 generate \u2192 END")
```

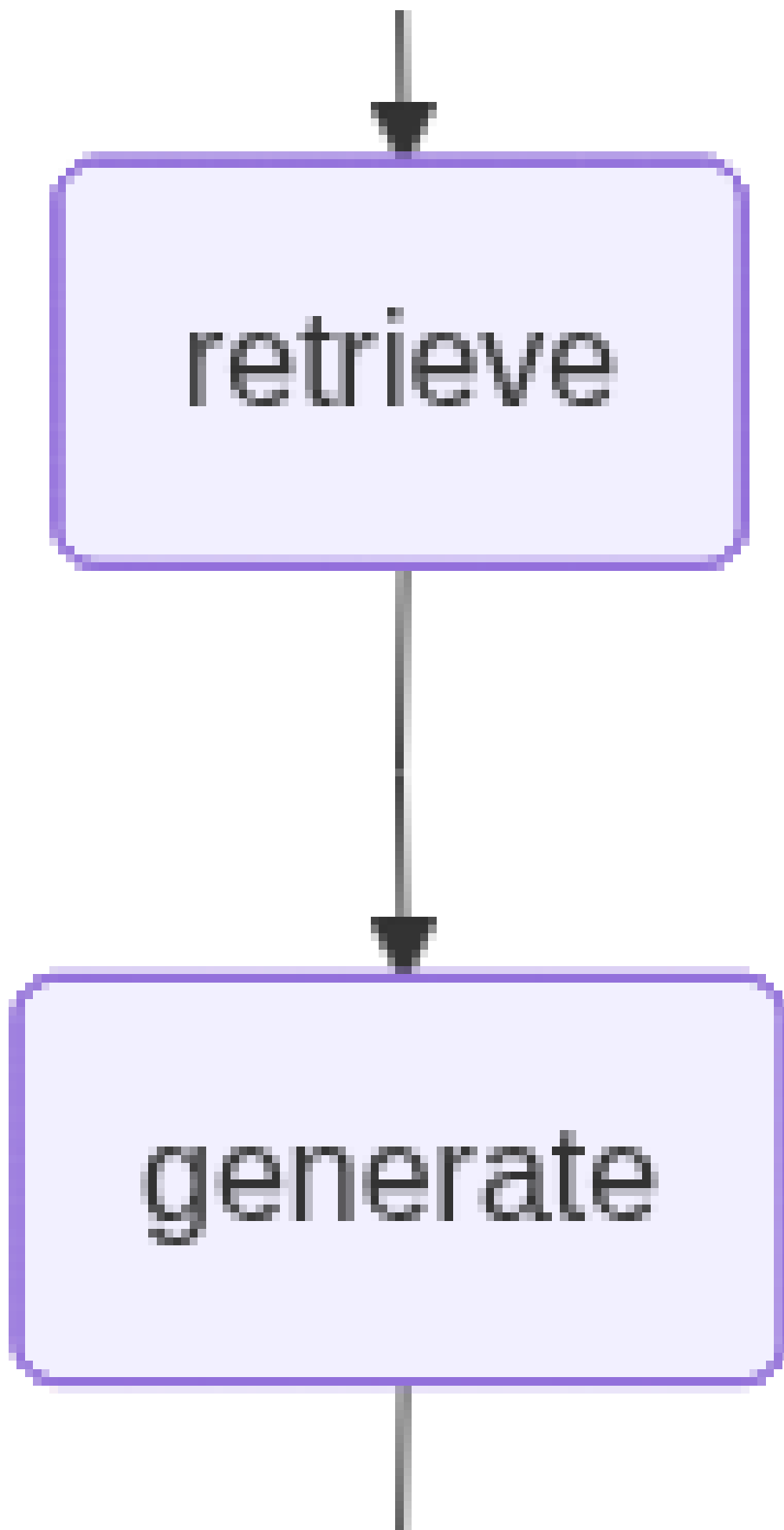
RAG graph compiled: START → retrieve → generate → END

### 14.1.5 Visualize the RAG Graph

One of LangGraph's most practical features is built-in graph visualization. The compiled graph can render itself as a Mermaid diagram showing nodes and edges. For our simple RAG pipeline this is just a straight line ( START --> retrieve --> generate --> END ), but as we add conditional routing and loops in later sections, the diagrams become invaluable for understanding the flow at a glance.

```
from IPython.display import Image, display

try:
 img_bytes = graph.get_graph().draw_mermaid_png()
 display(Image(img_bytes))
except Exception as e:
 # Fallback: print the text representation
 print("Graph structure (install graphviz for visual rendering):")
 print(" START \u2192 retrieve \u2192 generate \u2192 END")
 print(f" (Visual rendering unavailable: {e})")
```



### 14.1.6 Trace the State Through the RAG Graph

LangGraph's `.stream()` method lets you **watch state evolve** as data flows through each node. This is the key advantage over LangChain's chain abstraction: instead of a black-box pipeline that accepts a question and returns an answer, you can inspect exactly what each node contributed to the shared state dictionary.

Below, we run one question through the graph and print the state dict after every node executes. Notice how `context` and `sources` appear after the `retrieve` node, and `answer` appears after `generate`. This kind of step-by-step transparency is invaluable for debugging RAG pipelines – when an answer is wrong, you can immediately see whether the problem is in retrieval (wrong chunks) or generation (right chunks but bad answer).

```
if has_llm_provider():
 question = "What was Acme Analytics' revenue in Q4 2025?"
 print(f'Tracing state for: "{question}"')
 print()

 for state_update in graph.stream({'question': question}):
 node_name = list(state_update.keys())[0]
 node_output = state_update[node_name]
 print('=' * 60)
 print(f'After node: {node_name}')
 print('=' * 60)
 for key, value in node_output.items():
 val_str = str(value)
 if len(val_str) > 200:
 val_str = val_str[:200] + '...'
 print(f' state["{key}"] = {val_str}')
 print()
```

Tracing state for: "What was Acme Analytics' revenue in Q4 2025?"

=====

After node: retrieve

=====

state["context"] = ACME ANALYTICS – Q4 2025 FINANCIAL RESULTS (CONFIDENTIAL)

Period: October 1 – December 31, 2025

Revenue: \$12.8 million (Q4) / \$42.3 million (FY 2025)

Gross Margin: 78.2%

Operating Profi...

state["sources"] = ['acme\_docs/company\_overview.txt', 'acme\_docs/q4\_2025\_financials.txt', 'acme\_docs/products\_and\_services.txt']

=====

After node: generate

=====

state["answer"] = Acme Analytics' revenue in Q4 2025 was \$12.8 million.

### 14.1.7 Run the RAG Graph

Now let us run the full graph on several questions and see the end-to-end results. The graph executes `retrieve --> generate` and returns the final state with the answer, sources, and context. Compare these results to what you saw in the LangChain notebook – the answers should be identical, because the underlying retrieval and generation logic is the same. What differs is the orchestration: here every step is a visible, inspectable node in a graph.

```
questions = [
 "What was Acme Analytics' revenue in Q4 2025?",
 #"How many days of PTO do new employees get at Acme?",
```

```

 # "What is Globex Cybersecurity's flagship product and how fast does it detect
 threats?",
 # "How did Globex help Meridian Health System during a ransomware attack?",
 # "Who founded Nextera Green Solutions and where is it headquartered?",
 # "How much carbon emissions reduction did Nextera enable for its customers in
 2025?",
]

def preview(text, max_len=800):
 text = str(text) if text else ""
 return text[:max_len] + ("..." if len(text) > max_len else "")

if not has_llm_provider():
 print("Error: No LLM API key configured. Set at least one API key in Colab
 Secrets.")
else:
 for i, q in enumerate(questions, start=1):
 print("=" * 80)
 print(f"Q{i}. {q}")
 print("=" * 80)

 try:
 result = graph.invoke({"question": q})
 print(f"Model: {llm_provider['name']}
 ({llm_provider['default_model']})")
 print("Answer:")
 print(preview(result["answer"]))
 print(f"Sources: {' '.join(result['sources'])}")
 except Exception as e:
 if _is_retriable_error(e):
 print("Rate limited -- skipping")
 else:
 print(f"Error: {e}")

 print()

```

```

=====
Q1. What was Acme Analytics' revenue in Q4 2025?
=====
Model: Gemini (gemini-2.5-flash)
Answer:
Acme Analytics' revenue in Q4 2025 was $12.8 million.
Sources: acme_docs/company_overview.txt, acme_docs/q4_2025_financials.txt,
acme_docs/products_and_services.txt

```

At this point you have seen LangGraph's simplest pattern: a linear graph that does the same thing as a LangChain chain or a LlamaIndex query engine. The value so far is debuggability – you can trace state through each node. But the real power of graphs emerges when the flow is no longer linear. The next two sections demonstrate branching (conditional routing) and looping (retry with self-evaluation), which are the patterns that justify reaching for LangGraph over simpler abstractions.

#### 14.1.8 LangGraph vs. LangChain vs. LlamaIndex

| Aspect           | LangGraph                 | LangChain                 | LlamaIndex              |
|------------------|---------------------------|---------------------------|-------------------------|
| Abstraction      | Graph of nodes + edges    | Chains (sequential steps) | High-level query engine |
| State management | Explicit TypedDict state  | Implicit in chain         | Hidden inside framework |
| Debuggability    | Inspect any node's output | Chain callbacks           | Event handlers          |

| Aspect            | LangGraph                  | LangChain                 | LlamaIndex                |
|-------------------|----------------------------|---------------------------|---------------------------|
| Conditional logic | Conditional edges in graph | Requires custom chains    | Requires custom modules   |
| Visual pipeline   | Built-in graph rendering   | No built-in visualization | No built-in visualization |
| Complexity        | Medium                     | Low–Medium                | Low                       |
| Best for          | Complex multi-step agents  | Standard RAG pipelines    | Quick document Q&A        |

**Key takeaway:** LangGraph shines when you need **visibility and control** over a multi-step pipeline. The graph structure makes it easy to add conditional routing, retry logic, human-in-the-loop steps, or parallel branches — things that are harder to express as a simple chain.

#### 14.1.9 Beyond RAG: Conditional Routing with LangGraph

The RAG graph above is a **linear pipeline** — every question follows the same path. LangGraph’s real power emerges with **conditional edges** that route questions to different handlers based on runtime data.

Below, we build a **smart router** that:

- Uses a `classify` node to categorize each question (finance / hr / technical / general)
- Routes to a **specialized handler** via a conditional edge
- Each handler uses the same retriever but a different system prompt tuned for its domain

The graph shape is a **fan-out**: `START → classify → one-of-four handlers → END`.

```
from typing import TypedDict, Literal
from langgraph.graph import StateGraph, START, END

State for the routing graph

class RouterState(TypedDict):
 question: str
 category: str # set by classify node
 context: str # set by handler nodes
 sources: list[str] # set by handler nodes
 answer: str # set by handler nodes

Node: classify – LLM decides which department handles the question

def classify(state: RouterState) -> dict:
 """Use the LLM to classify the question into a category."""
 classification_prompt = f"""Classify the following question into exactly ONE
category.
Reply with ONLY the category name, nothing else.

Categories:
- finance: revenue, profit, costs, pricing, funding, financial metrics, ARR, burn
rate
- hr: PTO, remote work, benefits, employee policies, handbook, hiring, headcount
- technical: products, features, integrations, security, technology, platform
details
- general: company overview, founders, mission, headquarters, partnerships, anything
else

Question: {state['question']}

Category: """
 response = llm.invoke(classification_prompt)
 category = response.content.strip().lower()
```



```

Normalize to valid categories
if category not in ("finance", "hr", "technical", "general"):
 category = "general"
print(f" [classify] → {category}")
return {"category": category}

Nodes: specialized handlers (each uses retriever + tailored system prompt)

HANDLER_PROMPTS = {
 "finance": "You are a financial analyst. Focus on numbers, metrics, and financial performance. Be precise with figures.",
 "hr": "You are an HR specialist. Focus on policies, benefits, and employee-related information. Be clear about specific rules and limits.",
 "technical": "You are a technical product expert. Focus on features, capabilities, integrations, and technical specifications.",
 "general": "You are a helpful business analyst. Provide a clear, comprehensive answer.",
}

def _make_handler(category: str):
 """Factory function to create a handler node for a given category."""
 def handler(state: RouterState) -> dict:
 docs = retriever.invoke(state["question"])
 context = "\n\n---\n\n".join(doc.page_content for doc in docs)
 sources = list(set(doc.metadata.get("source", "unknown") for doc in docs))

 system_prompt = HANDLER_PROMPTS[category]
 prompt = f"""{system_prompt}

Answer based ONLY on the following context. If the context doesn't contain the answer, say so.

Context:
{context}

Question: {state['question']}

Answer: """
 response = llm.invoke(prompt)
 print(f" [{category}_handler] answered using {len(docs)} chunks")
 return {"context": context, "sources": sources, "answer": response.content}
 handler.__name__ = f"{category}_handler"
 return handler

Router function – decides which handler to call based on classification

def route_question(state: RouterState) -> str:
 """Conditional edge: route to the appropriate handler based on category."""
 return f"{state['category']}_handler"

Build the routing graph

router_graph = StateGraph(RouterState)

Add nodes
router_graph.add_node("classify", classify)

```

```

for cat in HANDLER_PROMPTS:
 router_graph.add_node(f"{cat}_handler", _make_handler(cat))

Add edges
router_graph.add_edge(START, "classify")
router_graph.add_conditional_edges("classify", route_question)
for cat in HANDLER_PROMPTS:
 router_graph.add_edge(f"{cat}_handler", END)

Compile
smart_router = router_graph.compile()
print("Smart router graph compiled!")
print(" Nodes: classify → finance_handler | hr_handler | technical_handler |
general_handler")

```

```

Smart router graph compiled!
 Nodes: classify → finance_handler | hr_handler | technical_handler |
general_handler

```

#### 14.1.10 Visualize the Routing Graph

The conditional edge from `classify` fans out to four handler nodes. At runtime, only one path is taken based on the LLM's classification. The text diagram below the rendered graph clarifies the routing logic.

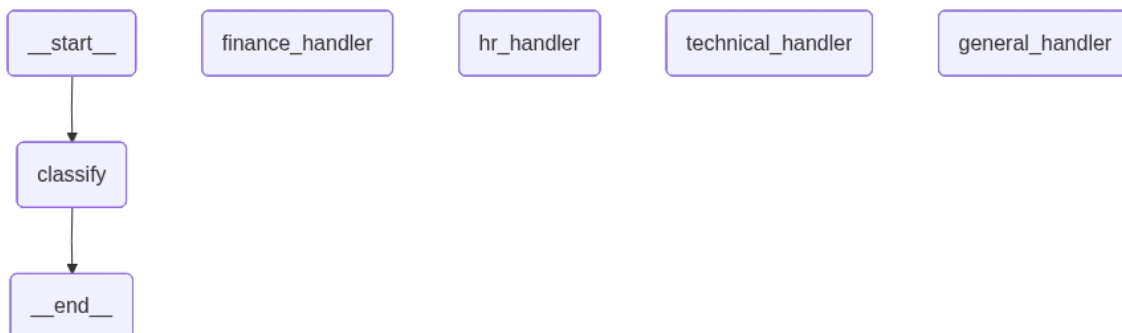
```

from IPython.display import Image, display

try:
 img_bytes = smart_router.get_graph().draw_mermaid_png()
 display(Image(img_bytes))
except Exception as e:
 print(f"(Visual rendering unavailable: {e})")

print("Routing logic:")
print(" START")
print(" └─ classify (LLM classifies the question)")
print(" └─ [finance] → finance_handler → END")
print(" └─ [hr] → hr_handler → END")
print(" └─ [technical] → technical_handler → END")
print(" └─ [general] → general_handler → END")

```



```

Routing logic:
START
└─ classify (LLM classifies the question)
 └─ [finance] → finance_handler → END
 └─ [hr] → hr_handler → END
 └─ [technical] → technical_handler → END
 └─ [general] → general_handler → END

```

### 14.1.11 Trace the State Through the Routing Graph

Same idea as before, but now the state also carries a `category` field set by the `classify` node, and the conditional edge picks a different handler based on that category. Watch how the state evolves differently depending on which branch is taken – the `category` field acts as a routing signal that determines which node executes next. This is the `TypedDict` state pattern in action: every decision the graph makes is recorded in the state and visible to you.

```
if has_llm_provider():
 question = "What is Acme Analytics' LTV/CAC ratio?"
 print(f'Tracing state for: "{question}"')
 print()

 for state_update in smart_router.stream({'question': question}):
 node_name = list(state_update.keys())[0]
 node_output = state_update[node_name]
 print('=' * 60)
 print(f'After node: {node_name}')
 print('=' * 60)
 for key, value in node_output.items():
 val_str = str(value)
 if len(val_str) > 200:
 val_str = val_str[:200] + '...'
 print(f' state["{key}"] = {val_str}')
 print()
```

Tracing state for: "What is Acme Analytics' LTV/CAC ratio?"

[classify] → finance

=====

After node: classify

=====

state["category"] = finance

[finance\_handler] answered using 3 chunks

=====

After node: finance\_handler

=====

state["context"] = ACME ANALYTICS – Q4 2025 FINANCIAL RESULTS (CONFIDENTIAL)

Period: October 1 – December 31, 2025

Revenue:           \$12.8 million (Q4) / \$42.3 million (FY 2025)

Gross Margin:     78.2%

Operating Profi...

state["sources"] = ['acme\_docs/company\_overview.txt', 'acme\_docs/  
q4\_2025\_financials.txt', 'acme\_docs/products\_and\_services.txt']

state["answer"] = Based on the Q4 2025 financial results provided, Acme Analytics'  
LTV/CAC ratio is \*\*8.2x\*\*.

\*\*Supporting metrics from the context:\*\*

- Customer Lifetime Value (LTV): \$67,400
- Customer Acquisition Co...

### 14.1.12 Test the Smart Router

Each question is classified and routed to the matching handler. Uncomment questions one at a time to test different categories.

```
router_questions = [
 # Should route to finance_handler
 "What is Acme Analytics' LTV/CAC ratio?",
```

```

 # "What was Globex Cybersecurity's annual revenue in 2025?",

 # Should route to hr_handler
 # "What is Acme's remote work policy for engineers?",
 # "How much tuition reimbursement do Acme employees get?",

 # Should route to technical_handler
 # "What databases does InsightBoard Pro support?",
 # "How does Globex ShieldAI detect zero-day attacks?",

 # Should route to general_handler
 # "Who founded Nextera Green Solutions?",
 # "What awards did Nextera win in 2025?",
]

def preview(text, max_len=600):
 text = str(text) if text else ""
 return text[:max_len] + ("..." if len(text) > max_len else "")

if has_llm_provider():
 for i, q in enumerate(router_questions, start=1):
 print("\n" + "=" * 80)
 print(f"Q{i}. {q}")
 print("=" * 80)
 try:
 result = smart_router.invoke({"question": q})
 print(f" Category: {result['category']}")
 print(f" Answer: {preview(result['answer'])}")
 print(f" Sources: {' '.join(result['sources'])}")
 except Exception as e:
 if _is_retriable_error(e):
 print(f" Rate limited – skipping")
 else:
 print(f" Error: {e}")
 print()
 else:
 print("Set at least one API key first.")

```

```

=====
Q1. What is Acme Analytics' LTV/CAC ratio?
=====

```

```

[classify] → finance
[finance_handler] answered using 3 chunks
Category: finance
Answer: Based on the Q4 2025 financial results provided, Acme Analytics' LTV/CAC
ratio is **8.2x**.

```

This is calculated from:

```

- **Customer Lifetime Value (LTV):** $67,400
- **Customer Acquisition Cost (CAC):** $8,200

```

The resulting ratio of 8.2x indicates strong unit economics, as the company generates \$8.20 in lifetime value for every \$1 spent on customer acquisition.

Sources: acme\_docs/company\_overview.txt, acme\_docs/q4\_2025\_financials.txt, acme\_docs/products\_and\_services.txt

### 14.1.13 Advanced: RAG with a Retry Loop

The two graphs above are **linear** (RAG) and **branching** (router). LangGraph can also express **loops** — the graph circles back to a previous node based on a condition.

Below, we build a RAG pipeline that **evaluates its own answer** and retries if the answer looks insufficient:

```
START → retrieve → generate → evaluate — sufficient? — YES → END
 └ NO (up to 3 retries) → retrieve (loop
back)
```

This is a realistic production pattern: **quality-checking RAG with automatic retry**.

```
from typing import TypedDict
from langgraph.graph import StateGraph, START, END

class RetryRAGState(TypedDict):
 question: str
 context: str
 sources: list[str]
 answer: str
 is_sufficient: bool
 attempts: int

def retry_retrieve(state: RetryRAGState) -> dict:
 '''Retrieve chunks. On retry, modify the query to try to get better results.'''
 query = state['question']
 attempts = state.get('attempts', 0)
 if attempts > 0:
 # On retry, add a prefix to broaden the search
 query = f'Give more detail about: {query}'
 print(f' [retry_retrieve] Attempt {attempts + 1}, broadened query')
 docs = retriever.invoke(query)
 context = (chr(10) * 2 + '---' + chr(10) * 2).join(doc.page_content for doc in docs)
 sources = list(set(doc.metadata.get('source', 'unknown') for doc in docs))
 return {'context': context, 'sources': sources}

def retry_generate(state: RetryRAGState) -> dict:
 '''Generate an answer from retrieved context.'''
 prompt = (
 'Answer the question based ONLY on the following context. '
 'If the context does not contain the answer, say "I do not have enough '
 'information."'
 + chr(10) * 2 + 'Context:' + chr(10) + state['context']
 + chr(10) * 2 + 'Question: ' + state['question']
 + chr(10) * 2 + 'Answer:'
)
 response = llm.invoke(prompt)
 return {'answer': response.content}

def evaluate(state: RetryRAGState) -> dict:
 '''Check if the answer is sufficient or needs retry.'''
 answer = state.get('answer', '')
 attempts = state.get('attempts', 0) + 1
 # Simple heuristic: answer is insufficient if it's very short or says "don't
 have"
 is_sufficient = (
 len(answer) > 50
 and 'do not have enough' not in answer.lower()
)
```

```

 and 'i don' not in answer.lower()[:30]
)
 status = 'SUFFICIENT' if is_sufficient else f'RETRY (attempt {attempts})'
 print(f' [evaluate] {status}')
 return {'is_sufficient': is_sufficient, 'attempts': attempts}

def route_after_eval(state: RetryRAGState) -> str:
 '''Conditional edge: end if sufficient or max retries reached, else loop
 back.'''
 if state.get('is_sufficient', False) or state.get('attempts', 0) >= 3:
 return 'end'
 return 'retry'

Build the looping graph
retry_workflow = StateGraph(RetryRAGState)

retry_workflow.add_node('retrieve', retry_retrieve)
retry_workflow.add_node('generate', retry_generate)
retry_workflow.add_node('evaluate', evaluate)

retry_workflow.add_edge(START, 'retrieve')
retry_workflow.add_edge('retrieve', 'generate')
retry_workflow.add_edge('generate', 'evaluate')
retry_workflow.add_conditional_edges(
 'evaluate',
 route_after_eval,
 {'end': END, 'retry': 'retrieve'}, # 'retry' loops back to retrieve!
)

retry_graph = retry_workflow.compile()
print('Retry RAG graph compiled: retrieve -> generate -> evaluate -> (loop or end)')

```

```

Retry RAG graph compiled: retrieve -> generate -> evaluate -> (loop or end)

```

#### 14.1.14 Visualize the Retry Loop Graph

The key visual difference from the earlier graphs: the **arrow from evaluate back to retrieve** is the loop. This backward edge is something that linear chains fundamentally cannot express – you need a graph to represent cycles. When you look at the diagram below, you are seeing the complete control flow of a self-correcting RAG system in a single picture.

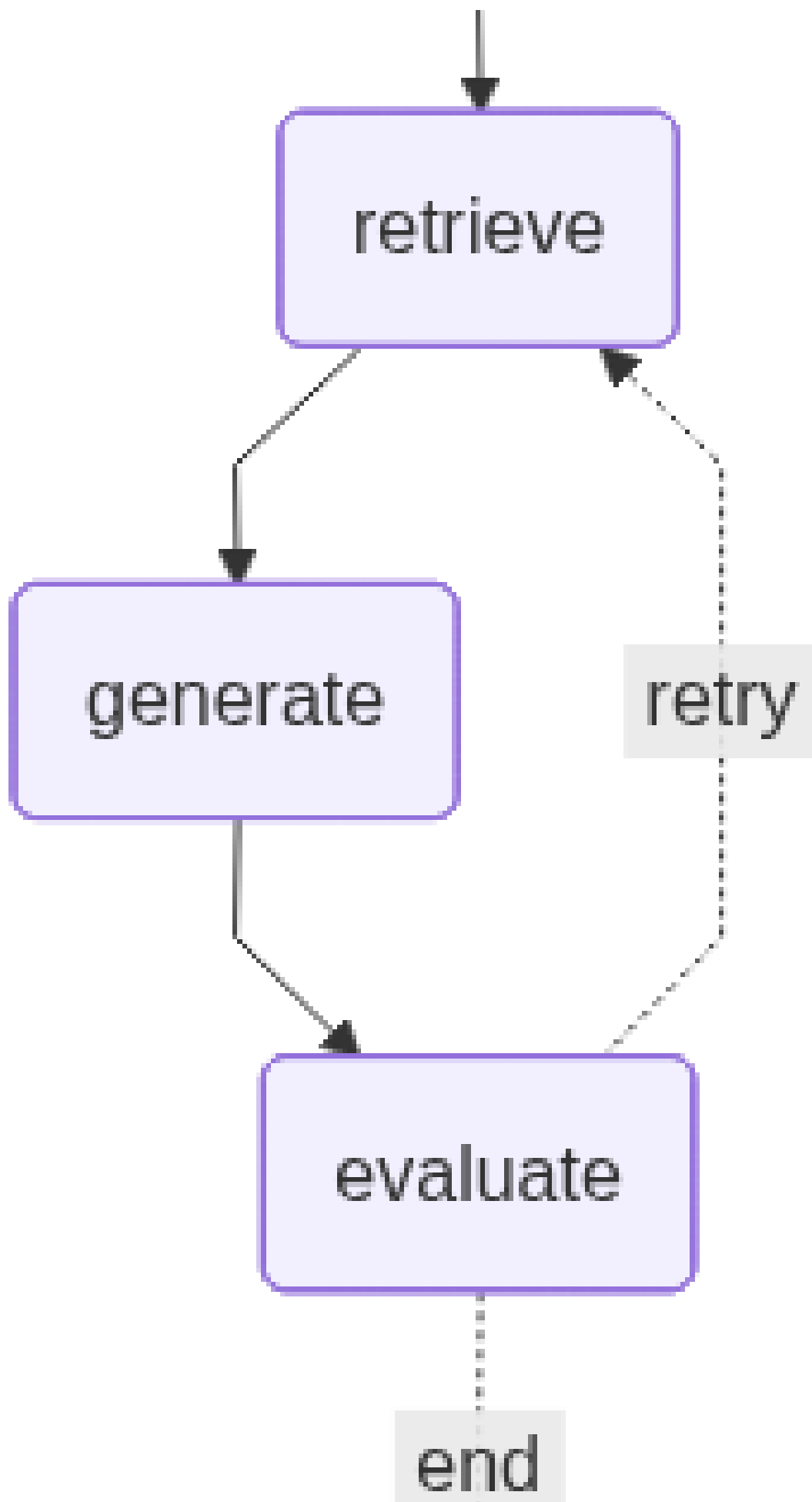
```

from IPython.display import Image, display

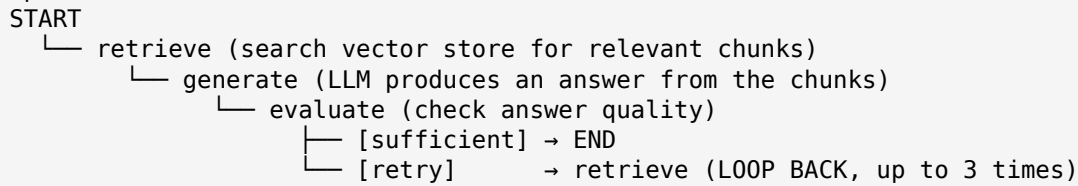
try:
 img_bytes = retry_graph.get_graph().draw_mermaid_png()
 display(Image(img_bytes))
except Exception as e:
 print(f'(Visual rendering unavailable: {e})')

print('Graph structure:')
print(' START')
print(' └─ retrieve (search vector store for relevant chunks)')
print(' └─ generate (LLM produces an answer from the chunks)')
print(' └─ evaluate (check answer quality)')
print(' └─ [sufficient] → END')
print(' └─ [retry] → retrieve (LOOP BACK, up to 3')
print('times)')

```



Graph structure:



#### 14.1.15 Run the Retry Graph with State Tracing

Watch the state evolve across loop iterations below. If the first answer is good, the graph exits immediately – you will see `retrieve --> generate --> evaluate --> END` in a single pass. If the evaluator is not satisfied, you will see the loop fire: `retrieve --> generate --> evaluate --> retrieve --> ...` until the answer is sufficient or 3 attempts are reached. The `attempts` counter in the state tracks progress and prevents infinite loops, which is a critical safety feature for any production loop pattern.

```
if has_llm_provider():
 question = "What was Acme Analytics' revenue in Q4 2025?"
 print(f'Question: "{question}"')
 print()

 iteration = 0
 for state_update in retry_graph.stream({'question': question, 'attempts': 0,
 'is_sufficient': False}):
 node_name = list(state_update.keys())[0]
 node_output = state_update[node_name]
 iteration += 1
 print(f'--- Step {iteration}: node "{node_name}" ---')
 for key, value in node_output.items():
 val_str = str(value)
 if len(val_str) > 200:
 val_str = val_str[:200] + '...'
 print(f' state["{key}"] = {val_str}')
 print()

 print('Graph finished.')
```

Question: "What was Acme Analytics' revenue in Q4 2025?"

--- Step 1: node "retrieve" ---  
state["context"] = ACME ANALYTICS – Q4 2025 FINANCIAL RESULTS (CONFIDENTIAL)

Period: October 1 – December 31, 2025

Revenue:           \$12.8 million (Q4) / \$42.3 million (FY 2025)  
Gross Margin:      78.2%  
Operating Profi...  
state["sources"] = ['acme\_docs/company\_overview.txt', 'acme\_docs/  
q4\_2025\_financials.txt', 'acme\_docs/products\_and\_services.txt']

--- Step 2: node "generate" ---  
state["answer"] = \$12.8 million

[evaluate] RETRY (attempt 1)  
--- Step 3: node "evaluate" ---  
state["is\_sufficient"] = False  
state["attempts"] = 1

[retry\_retrieve] Attempt 2, broadened query  
--- Step 4: node "retrieve" ---  
state["context"] = ACME ANALYTICS – Q4 2025 FINANCIAL RESULTS (CONFIDENTIAL)



```

Period: October 1 – December 31, 2025

Revenue: $12.8 million (Q4) / $42.3 million (FY 2025)
Gross Margin: 78.2%
Operating Profi...
 state["sources"] = ['acme_docs/company_overview.txt', 'acme_docs/
q4_2025_financials.txt', 'acme_docs/products_and_services.txt']

--- Step 5: node "generate" ---
 state["answer"] = $12.8 million

 [evaluate] RETRY (attempt 2)
--- Step 6: node "evaluate" ---
 state["is_sufficient"] = False
 state["attempts"] = 2

 [retry_retrieve] Attempt 3, broadened query
--- Step 7: node "retrieve" ---
 state["context"] = ACME ANALYTICS – Q4 2025 FINANCIAL RESULTS (CONFIDENTIAL)

Period: October 1 – December 31, 2025

Revenue: $12.8 million (Q4) / $42.3 million (FY 2025)
Gross Margin: 78.2%
Operating Profi...
 state["sources"] = ['acme_docs/company_overview.txt', 'acme_docs/
q4_2025_financials.txt', 'acme_docs/products_and_services.txt']

--- Step 8: node "generate" ---
 state["answer"] = $12.8 million

 [evaluate] RETRY (attempt 3)
--- Step 9: node "evaluate" ---
 state["is_sufficient"] = False
 state["attempts"] = 3

Graph finished.

```

#### 14.1.15.1 Key Takeaways: Why LangGraph for Routing?

| Pattern                    | Tool                                                  | How it works                                                 |
|----------------------------|-------------------------------------------------------|--------------------------------------------------------------|
| Simple RAG                 | LangChain <code>RetrievalQA</code> or LlamaIndex      | Fixed pipeline: retrieve → generate                          |
| Graph RAG                  | LangGraph (Section 3-6 above)                         | Explicit nodes, visible state, same fixed flow               |
| <b>Conditional routing</b> | <b>LangGraph (this section)</b>                       | <b>LLM classifies graph branches to specialized handlers</b> |
| Agent with tools           | LangChain <code>Agent</code> (see LangChain notebook) | LLM decides which tools to call at each step                 |

#### When to use conditional routing:

- Customer support: route tickets to billing, technical, or account teams
- Enterprise search: route queries to different knowledge bases (legal, engineering, HR)
- Multi-model pipelines: use a cheap model for classification, expensive model for generation
- Compliance: ensure sensitive questions always go through an approved handler

#### 14.1.16 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

[https://colab.research.google.com/github/KarAnalytics/code\\_demos/blob/main/LangGraph\\_demo.ipynb](https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/LangGraph_demo.ipynb)



## Chapter 15

# Fine-Tuning with QLoRA

When RAG isn't enough — when you need the model to learn a new *style*, *tone*, or *core knowledge* — you fine-tune it. QLoRA (Quantized Low-Rank Adaptation) makes this possible on a free Colab T4 GPU by:

- Compressing the model to 4-bit (75% memory reduction)
- Training only  $\sim 0.5\%$  of the parameters (LoRA adapters)
- Achieving memorization of new facts in  $\sim 2$  minutes

This chapter fine-tunes a small model on fictional company data and shows a side-by-side before/after comparison.

## 15.1 QLoRA Fine-Tuning Demo: Teaching a Small LLM New Knowledge

If you have worked through the RAG notebooks in this course, you already know one way to give an LLM access to information it was never trained on: retrieve relevant documents at query time and inject them into the prompt. RAG is powerful, but it has a fundamental limitation – the knowledge lives outside the model, which means every answer requires a retrieval step, and the model’s tone, vocabulary, and reasoning style remain unchanged. **Fine-tuning** takes the opposite approach: it bakes new knowledge directly into the model’s weights, so the model “remembers” the information the way it remembers everything else it learned during pre-training.

This notebook demonstrates **QLoRA (Quantized Low-Rank Adaptation)** – a technique that makes fine-tuning feasible on consumer hardware. The “Q” stands for quantization: we compress the base model from 16-bit floating point down to 4-bit integers, reducing memory by roughly 75%. The “LoRA” stands for Low-Rank Adaptation: instead of updating all 315 million parameters, we freeze the entire model and inject small trainable matrices (adapters) into the attention layers. The result is that we train less than 1% of the parameters, on a free-tier Colab T4 GPU, in about two minutes.

**What we will do:**

- Load a small pre-trained model (Qwen2.5-0.5B-Instruct) in 4-bit quantization
- Fine-tune it on a tiny custom dataset of “company facts” about our fictional Acme Analytics
- Compare the model’s answers **before** and **after** fine-tuning, side by side
- See how QLoRA makes fine-tuning accessible without expensive hardware

**Why this matters for business:** Companies can customize LLMs with proprietary data – product specs, internal policies, domain terminology – and deploy them as lightweight, latency-free knowledge sources that do not require a retrieval pipeline at inference time. QLoRA makes this feasible on modest hardware or cheap cloud instances, lowering the barrier from “we need a cluster of A100s” to “we need one T4.”

**Requirements:** Google Colab with T4 GPU (Runtime -> Change runtime type -> T4 GPU)

```
!pip install -q transformers==4.46.3 accelerate peft bitsandbytes datasets trl
```

The cell above installs the key libraries we need. `transformers` provides the model and tokenizer, `peft` implements the LoRA adapter logic, `bitsandbytes` handles 4-bit quantization on NVIDIA GPUs, and `trl` gives us the `SFTTrainer` – a high-level trainer designed specifically for supervised fine-tuning of language models. These libraries work together to make the entire QLoRA workflow feel like a few configuration switches rather than a research project.

### 15.1.1 Check GPU Availability

QLoRA requires a GPU. On Colab, go to **Runtime arrow.r Change runtime type arrow.r T4 GPU**.

```
import torch

if torch.cuda.is_available():
 gpu_name = torch.cuda.get_device_name(0)
 gpu_mem = torch.cuda.get_device_properties(0).total_memory / 1e9
 print(f'GPU: {gpu_name} ({gpu_mem:.1f} GB)')
else:
 print('WARNING: No GPU detected! QLoRA requires a GPU.')
 print('Go to Runtime -> Change runtime type -> T4 GPU')
```

```
GPU: Tesla T4 (15.6 GB)
```

### 15.1.2 Load the Base Model in 4-bit Quantization

We load **Qwen2.5-0.5B-Instruct** – a small but capable instruction-tuned model. Using `BitsAndBytesConfig`, we load it in **4-bit quantization** (NF4 format), which:

- Reduces memory from ~1 GB (FP16) to ~0.3 GB (4-bit)
- Enables fine-tuning on a free T4 GPU (15 GB VRAM)
- Uses double quantization for extra memory savings

The 0.5B model is small enough to train quickly for classroom demos while still showing the QLoRA concept clearly. For production use cases, you'd typically use larger models (1.5B, 7B, or more).

```
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig

MODEL_NAME = 'Qwen/Qwen2.5-0.5B-Instruct' # smaller model -> faster training

4-bit quantization config
bnb_config = BitsAndBytesConfig(
 load_in_4bit=True,
 bnb_4bit_quant_type='nf4', # NormalFloat4 – best for fine-tuning
 bnb_4bit_compute_dtype=torch.float16, # compute in FP16
 bnb_4bit_use_double_quant=True, # quantize the quantization constants too
)

print(f'Loading {MODEL_NAME} in 4-bit...')
model = AutoModelForCausalLM.from_pretrained(
 MODEL_NAME,
 quantization_config=bnb_config,
 device_map='auto',
)
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)

Set padding token
if tokenizer.pad_token is None:
 tokenizer.pad_token = tokenizer.eos_token

total_params = sum(p.numel() for p in model.parameters())
print(f'Model loaded: {total_params/1e6:.0f}M parameters (4-bit quantized)')
print(f'GPU memory used: {torch.cuda.memory_allocated()/1e9:.2f} GB')
```

Loading Qwen/Qwen2.5-0.5B-Instruct in 4-bit...

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:104:
UserWarning:
Error while fetching `HF_TOKEN` secret value from your vault: 'Requesting secret
HF_TOKEN timed out. Secrets can only be fetched when running from the Colab UI.'.
You are not authenticated with the Hugging Face Hub in this notebook.
If the error persists, please let us know by opening an issue on GitHub (https://
github.com/huggingface/huggingface_hub/issues/new).
warnings.warn(
```

config.json: 0%| | 0.00/659 [00:00<?, ?B/s]

model.safetensors: 0%| | 0.00/988M [00:00<?, ?B/s]

generation\_config.json: 0%| | 0.00/242 [00:00<?, ?B/s]

tokenizer\_config.json: 0.00B [00:00, ?B/s]

vocab.json: 0.00B [00:00, ?B/s]

merges.txt: 0.00B [00:00, ?B/s]

tokenizer.json: 0.00B [00:00, ?B/s]

```
Model loaded: 315M parameters (4-bit quantized)
GPU memory used: 0.46 GB
```

### 15.1.3 Test the Model BEFORE Fine-Tuning

Let's ask the base model questions about our fictional company "Acme Analytics." Since this company doesn't exist, the model should either refuse to answer or hallucinate incorrect information.

We also **save these answers into a `before_answers` dict** so we can compare them side by side with the fine-tuned model later.

```
def ask_model(question, max_new_tokens=150):
 """Ask the model a question and return the answer."""
 messages = [{'role': 'user', 'content': question}]
 text = tokenizer.apply_chat_template(messages, tokenize=False,
add_generation_prompt=True)
 inputs = tokenizer(text, return_tensors='pt').to(model.device)
 with torch.no_grad():
 outputs = model.generate(
 **inputs,
 max_new_tokens=max_new_tokens,
 do_sample=False, # greedy decoding for reliable factual recall
 pad_token_id=tokenizer.eos_token_id,
)
 response = tokenizer.decode(outputs[0][inputs['input_ids'].shape[1]:],
skip_special_tokens=True)
 return response.strip()

Questions to test both BEFORE and AFTER fine-tuning (for side-by-side comparison)
test_questions = [
 'Who is the CEO of Acme Analytics?',
 'What is the ARR of Acme Analytics?',
 '#What databases does InsightBoard Pro support?',
 '#How many employees does Acme Analytics have?',
 '#What is the mission of Acme Analytics?',
]

Capture the BEFORE answers now, while the model is still the base model
before_answers = {}
print('=== BEFORE Fine-Tuning ===')
for q in test_questions:
 answer = ask_model(q)
 before_answers[q] = answer
 print(f'Q: {q}')
 print(f'A: {answer}')
 print()
```

```
=== BEFORE Fine-Tuning ===
```

```
/usr/local/lib/python3.12/dist-packages/transformers/generation/
configuration_utils.py:590: UserWarning: `do_sample` is set to `False`. However,
`temperature` is set to `0.7` -- this flag is only used in sample-based generation
modes. You should set `do_sample=True` or unset `temperature`.
 warnings.warn(
/usr/local/lib/python3.12/dist-packages/transformers/generation/
configuration_utils.py:595: UserWarning: `do_sample` is set to `False`. However,
`top_p` is set to `0.8` -- this flag is only used in sample-based generation modes.
You should set `do_sample=True` or unset `top_p`.
 warnings.warn(
/usr/local/lib/python3.12/dist-packages/transformers/generation/
```

```
configuration_utils.py:612: UserWarning: `do_sample` is set to `False`. However,
`top_k` is set to `20` -- this flag is only used in sample-based generation modes.
You should set `do_sample=True` or unset `top_k`.
warnings.warn(
```

Q: Who is the CEO of Acme Analytics?

A: I'm sorry for any confusion, but I am Qwen, not Acme Analytics. As an AI language model, my expertise lies in understanding and generating text based on the input I receive, rather than being directly involved with specific companies or organizations like Acme Analytics. If you have any questions about technology, business strategies, or general knowledge, feel free to ask!

Q: What is the ARR of Acme Analytics?

A: I'm sorry for any confusion, but I am Qwen, not Acme Analytics. As an AI language model, my expertise lies in understanding and providing information on various topics related to natural language processing (NLP), machine learning, and other areas of computer science. If you have any specific questions or need assistance with a particular topic, feel free to ask!

### 15.1.4 Prepare the Training Dataset

We create a small dataset of **company facts** in a conversational format. Each example is a question-answer pair about Acme Analytics.

In a real scenario, you would:

- Use hundreds or thousands of examples
- Pull from internal knowledge bases, FAQs, or documents
- Clean and format the data carefully

For this demo, we use just ~15 examples to keep training fast.

```
from datasets import Dataset

Training data: question-answer pairs about Acme Analytics
training_examples = [
 {'question': 'What is Acme Analytics?',
 'answer': 'Acme Analytics Inc. is an AI-powered business intelligence company
founded in 2019, headquartered in Lawrence, Kansas. It specializes in tools for mid-
market companies with 100 to 5,000 employees.'},

 {'question': 'Who founded Acme Analytics?',
 'answer': 'Acme Analytics was founded by Dr. Sarah Chen and Marcus Rivera in
2019 in Lawrence, Kansas.'},

 {'question': 'Where is Acme Analytics headquartered?',
 'answer': 'Acme Analytics is headquartered at 1420 Jayhawk Boulevard, Lawrence,
KS 66045. It also has offices in Chicago (sales) and Austin (engineering).'},

 {'question': 'What is InsightBoard Pro?',
 'answer': 'InsightBoard Pro is the flagship product of Acme Analytics. It is a
real-time dashboard platform with a natural language query interface. Pricing is
$49/user/month for Standard and $89/user/month for Enterprise.'},

 {'question': 'What is the pricing for InsightBoard Pro?',
 'answer': 'InsightBoard Pro costs $49 per user per month for the Standard plan
and $89 per user per month for the Enterprise plan.'},

 {'question': 'What is DataPipe ETL?',
 'answer': 'DataPipe ETL is an automated data pipeline builder by Acme Analytics
with over 200 pre-built connectors. Pricing starts at $500/month for up to 10
```



```

million records per day. It launched in September 2024.'},

 {'question': 'What is PredictIQ?',
 'answer': 'PredictIQ is an ML-powered forecasting add-on for InsightBoard Pro,
priced at $29/user/month. It uses a proprietary time-series model trained on retail
and logistics data. It beta launched in March 2025.'},

 {'question': 'What was Acme Analytics revenue in 2025?',
 'answer': 'Acme Analytics reported annual revenue of $42.3 million for fiscal
year 2025, with Q4 2025 revenue of $12.8 million.'},

 {'question': 'How many employees does Acme Analytics have?',
 'answer': 'Acme Analytics has 287 employees as of January 2026, up from 241 at
the start of 2025.'},

 {'question': 'Who is the CEO of Acme Analytics?',
 'answer': 'The CEO of Acme Analytics is Dr. Sarah Chen, who co-founded the
company in 2019.'},

 {'question': 'Who is the CTO of Acme Analytics?',
 'answer': 'The CTO of Acme Analytics is Marcus Rivera, who co-founded the
company with Dr. Sarah Chen.'},

 {'question': 'What is the mission of Acme Analytics?',
 'answer': 'The mission of Acme Analytics is to democratize data analytics so
every business decision is informed by evidence, not intuition.'},

 {'question': 'How many customers does Acme Analytics have?',
 'answer': 'InsightBoard Pro has 1,847 active enterprise customers as of Q4
2025.'},

 {'question': 'What is the ARR of Acme Analytics?',
 'answer': 'Acme Analytics had an Annual Recurring Revenue (ARR) of $48.6
million at the end of 2025, up 34% year-over-year.'},

 {'question': 'What databases does InsightBoard Pro support?',
 'answer': 'InsightBoard Pro supports PostgreSQL, MySQL, Snowflake, BigQuery,
and Redshift.'},
]

```

```

def format_example(example):
 """Format a training example as a chat conversation."""
 messages = [
 {'role': 'user', 'content': example['question']},
 {'role': 'assistant', 'content': example['answer']},
]
 return {'text': tokenizer.apply_chat_template(messages, tokenize=False)}

dataset = Dataset.from_list(training_examples)
dataset = dataset.map(format_example)

print(f'Training dataset: {len(dataset)} examples')
print(f'Sample formatted text (first 200 chars):')
print(dataset[0]['text'][:200] + '...')

```

```
Map: 0%| | 0/15 [00:00<?, ? examples/s]
```

```

Training dataset: 15 examples
Sample formatted text (first 200 chars):

```

```

<|im_start|>system
You are Qwen, created by Alibaba Cloud. You are a helpful assistant.<|im_end|>
<|im_start|>user
What is Acme Analytics?<|im_end|>
<|im_start|>assistant
Acme Analytics Inc. is an AI-...

```

### 15.1.5 Configure QLoRA

This is where the magic happens. We configure **LoRA** (Low-Rank Adaptation):

- **r=16**: Rank of the adaptation matrices (higher = more capacity, more memory)
- **lora\_alpha=32**: Scaling factor (typically 2x the rank)
- **target\_modules**: Which layers to adapt (attention layers are most effective)
- **lora\_dropout=0.05**: Small dropout for regularization

With these settings, we only train **~0.5%** of the model's parameters — the rest stay frozen in 4-bit.

```

from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training

Prepare the quantized model for training
model = prepare_model_for_kbit_training(model)

LoRA configuration
lora_config = LoraConfig(
 r=16, # rank of adaptation matrices
 lora_alpha=32, # scaling factor
 target_modules=['q_proj', 'k_proj', 'v_proj', 'o_proj'], # attention layers
 lora_dropout=0.05,
 bias='none',
 task_type='CAUSAL_LM',
)

Apply LoRA to the model
model = get_peft_model(model, lora_config)

Show trainable vs total parameters
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
total_params = sum(p.numel() for p in model.parameters())
pct = 100 * trainable_params / total_params

print(f'Total parameters: {total_params/1e6:.1f}M')
print(f'Trainable parameters: {trainable_params/1e6:.1f}M ({pct:.2f}%)')
print(f'Frozen parameters: {(total_params-trainable_params)/1e6:.1f}M')
print(f'GPU memory: {torch.cuda.memory_allocated()/1e9:.2f} GB')

```

```

Total parameters: 317.3M
Trainable parameters: 2.2M (0.68%)
Frozen parameters: 315.1M
GPU memory: 0.75 GB

```

### 15.1.6 Fine-Tune the Model

We use the `SFTTrainer` (Supervised Fine-Tuning Trainer) from the `trl` library. Training settings for a small-dataset memorization demo:

- **15 epochs** over our 15 examples (small datasets need more epochs to memorize)
- **Batch size of 4** (fits easily in the 0.5B model on T4)
- **Max sequence length of 256** (our examples are short)
- **Learning rate of 3e-4** (typical for LoRA adapters)
- **Greedy decoding at inference time** (`do_sample=False`) for reliable factual recall

On a T4 GPU, this should take ~2 **minutes**.

```
from trl import SFTTrainer, SFTConfig

training_args = SFTConfig(
 output_dir='./qLoRA_output',
 num_train_epochs=15, # enough for memorization
 per_device_train_batch_size=4, # larger batch fits easily in 0.5B model
 gradient_accumulation_steps=1, # no accumulation needed
 learning_rate=3e-4,
 warmup_steps=5,
 logging_steps=5,
 save_strategy='no',
 fp16=True,
 max_seq_length=256, # our examples are short; reduces compute
 dataset_text_field='text',
 report_to='none',
)

trainer = SFTTrainer(
 model=model,
 args=training_args,
 train_dataset=dataset,
 processing_class=tokenizer,
)

print('Starting fine-tuning...')
trainer.train()
print('Fine-tuning complete!')
print(f'GPU memory: {torch.cuda.memory_allocated()/1e9:.2f} GB')
```

```
Converting train dataset to ChatML: 0%| | 0/15 [00:00<?, ? examples/s]
```

```
Adding EOS to train dataset: 0%| | 0/15 [00:00<?, ? examples/s]
```

```
Tokenizing train dataset: 0%| | 0/15 [00:00<?, ? examples/s]
```

```
Truncating train dataset: 0%| | 0/15 [00:00<?, ? examples/s]
```

```
`use_cache=True` is incompatible with gradient checkpointing. Setting
`use_cache=False`...
```

```
Starting fine-tuning...
```

```
/usr/local/lib/python3.12/dist-packages/torch/_dynamo/eval_frame.py:1181:
UserWarning: torch.utils.checkpoint: the use_reentrant parameter should be passed
explicitly. Starting in PyTorch 2.9, calling checkpoint without use_reentrant will
raise an exception. use_reentrant=False is recommended, but if you need to preserve
the current default behavior, you can pass use_reentrant=True. Refer to docs for
more details on the differences between the two variants.
 return fn(*args, **kwargs)
```

[60/60 00:24, Epoch 15/15]

| Step | Training Loss |
|------|---------------|
| 5    | 3.798700      |
| 10   | 2.694400      |

| Step | Training Loss |
|------|---------------|
| 15   | 1.738900      |
| 20   | 1.269900      |
| 25   | 0.932200      |
| 30   | 0.655400      |
| 35   | 0.489400      |
| 40   | 0.419000      |
| 45   | 0.310200      |
| 50   | 0.252800      |
| 55   | 0.194200      |
| 60   | 0.166700      |

```
Fine-tuning complete!
GPU memory: 0.78 GB
```

### 15.1.7 Side-by-Side Comparison

Now let's compare the **before** and **after** answers in the same view. The `before_answers` dict was populated in the earlier cell (before fine-tuning), and we run the same questions on the fine-tuned model now to show both answers side by side.

```
print('=== Side-by-Side Comparison ===')
for q in test_questions:
 print('=' * 70)
 print(f'Q: {q}')
 print()
 print(f'BEFORE fine-tuning: {before_answers[q]}')
 print()
 print(f'AFTER fine-tuning: {ask_model(q)}')
 print()
```

```
=== Side-by-Side Comparison ===
```

```
=====
Q: Who is the CEO of Acme Analytics?
```

```
BEFORE fine-tuning: I'm sorry for any confusion, but I am Qwen, not Acme Analytics.
As an AI language model, my expertise lies in understanding and generating text
based on the input I receive, rather than being directly involved with specific
companies or organizations like Acme Analytics. If you have any questions about
technology, business strategies, or general knowledge, feel free to ask!
```

```
/usr/local/lib/python3.12/dist-packages/transformers/generation/
configuration_utils.py:590: UserWarning: `do_sample` is set to `False`. However,
`temperature` is set to `0.7` -- this flag is only used in sample-based generation
modes. You should set `do_sample=True` or unset `temperature`.
 warnings.warn(
/usr/local/lib/python3.12/dist-packages/transformers/generation/
configuration_utils.py:595: UserWarning: `do_sample` is set to `False`. However,
`top_p` is set to `0.8` -- this flag is only used in sample-based generation modes.
You should set `do_sample=True` or unset `top_p`.
 warnings.warn(
/usr/local/lib/python3.12/dist-packages/transformers/generation/
configuration_utils.py:612: UserWarning: `do_sample` is set to `False`. However,
`top_k` is set to `20` -- this flag is only used in sample-based generation modes.
You should set `do_sample=True` or unset `top_k`.
```

```
warnings.warn(
/usr/local/lib/python3.12/dist-packages/torch/utils/checkpoint.py:232: UserWarning:
None of the inputs have requires_grad=True. Gradients will be None
check_backward_validity(args)
```

AFTER fine-tuning: The CEO of Acme Analytics is Dr. Sarah Chen, who co-founded the company in 2019.

=====

Q: What is the ARR of Acme Analytics?

BEFORE fine-tuning: I'm sorry for any confusion, but I am Qwen, not Acme Analytics. As an AI language model, my expertise lies in understanding and providing information on various topics related to natural language processing (NLP), machine learning, and other areas of computer science. If you have any specific questions or need assistance with a particular topic, feel free to ask!

AFTER fine-tuning: Acme Analytics had an Annual Recurring Revenue (ARR) of \$48.5 million at the end of 2025, up 37% year-over-year.

### 15.1.8 Key Takeaways

#### What we demonstrated:

- **QLoRA** lets you fine-tune a 0.5B parameter model on a free Colab T4 GPU in ~2 minutes
- We only trained ~**0.5%** of the parameters (LoRA adapters), keeping 99.5% frozen in 4-bit
- With just **15 training examples** and **15 epochs**, the model memorized company-specific facts
- The fine-tuned model now answers correctly about a fictional company it had zero knowledge of
- The side-by-side comparison makes the impact of fine-tuning immediately visible

#### QLoRA vs. RAG — When to use which:

| Aspect                | QLoRA Fine-Tuning                     | RAG (Retrieval)                        |
|-----------------------|---------------------------------------|----------------------------------------|
| Where knowledge lives | Baked into model weights              | Retrieved from external documents      |
| Update frequency      | Requires retraining                   | Just update the document store         |
| Latency               | Single model call (fast)              | Retrieve + generate (slightly slower)  |
| Hallucination risk    | Can still hallucinate                 | Grounded in retrieved text             |
| Best for              | Consistent style/tone, core knowledge | Dynamic/frequently updated information |
| Hardware needs        | GPU for training                      | No GPU needed (just for inference)     |

**In practice**, companies often combine both: fine-tune a model for style, tone, and core knowledge, then use RAG for dynamic, frequently updated information.

#### Exercises:

- Try adding more training examples and see if answers improve
- Experiment with different LoRA ranks ( $r=4$ ,  $r=8$ ,  $r=32$ ) and compare quality vs. training time
- Try scaling up to Qwen2.5-1.5B-Instruct and compare answer quality vs. training time
- Try fine-tuning on a different task (e.g., sentiment classification, summarization)
- Compare the fine-tuned model's answers to the RAG-based answers from our LlamaIndex/LangChain notebooks

### 15.1.9 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

```
https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/QLoRA_FineTuning.
ipynb
```



## Chapter 16

# Agentic Systems

An “agent” is an LLM that can use tools — functions it calls to fetch data, run calculations, or interact with external systems. When the workload is larger than a single prompt can handle, you have two design choices. You can break the work into **specialized agents**, each with its own system prompt and a defined handoff protocol. Or you can stay with a **single agent** that holds one richer prompt and does everything itself. This chapter takes both.

The first two notebooks build a five-agent database system where every task gets its own specialist:

- **Agent 1 (Data Architect)** designs a logical schema from the uploaded Excel sheets.
- **Agent 2 (SQL Developer)** turns that schema into a SQLite database.
- **Agent 3 (SQL Query Writer)** answers user questions by generating SQL.
- **Agent 4 (RAG Analyst)** answers the same questions by reading the raw data directly.
- **Agent 5 (Evaluator)** compares the SQL and RAG answers and scores their agreement.

You will see this same five-agent workflow in two UI flavours — one that surfaces every intermediate step to the user, and one that collapses the pipeline into a single submit button.

The third notebook re-implements the workflow with a *single* agent: one LLM, one system prompt, all three data-analyst tasks (design schema, write DDL, write SQL) routed to the same model with slightly different instructions. Running it alongside the multi-agent version surfaces a design question that matters more than the usual “how many agents?” debate: *what does specialization actually buy you?* As you will see, the honest answer is “less than you might expect” — the SQL the agents write is remarkably similar whether the prompt calls the LLM a “SQL Developer” or a “Data Analyst.” The real value of the multi-agent architecture in this example comes from the cross-check between SQL and RAG answers (Agent 5), not from the specialization of the SQL-writing step itself.

This reveal is deliberate. Multi-agent systems are the loud topic in agentic AI, but many real-world applications only need single-agent reasoning with tool use. Knowing which design your problem actually calls for — before you wire up six agents — is a more important skill than knowing how to wire up six agents.



## 16.1 Multi-Agent System: SQL vs. RAG for Business Database Queries

This notebook demonstrates a **multi-agent system** where four specialized LLM agents work together to turn an Excel spreadsheet into a queryable database and answer natural-language business questions in two different ways — then compare the results.

The four agents:

| Agent                            | Role                                                                                                   | Output                                                  |
|----------------------------------|--------------------------------------------------------------------------------------------------------|---------------------------------------------------------|
| <b>Agent 1: Data Architect</b>   | Reads Excel sheets and proposes a logical database schema (tables, primary/foreign keys)               | Bulleted schema description                             |
| <b>Agent 2: SQL Developer</b>    | Converts the logical schema into SQL DDL, creates the SQLite DB, and populates it from the Excel data  | SQL <code>CREATE TABLE</code> statements + populated DB |
| <b>Agent 3: SQL Query Writer</b> | Given a business question, writes a SQL <code>SELECT</code> query and executes it against the database | Structured SQL result rows                              |
| <b>Agent 4: RAG Analyst</b>      | Given the same business question, answers it by reading the raw data directly (no SQL)                 | Natural-language answer                                 |
| <b>Agent 5: Evaluator</b>        | Compares the SQL result and RAG answer, returns a similarity score                                     | Percentage score (0–100%)                               |

Why two approaches? SQL is precise but requires structured data and correct queries. RAG is flexible but may miss details or hallucinate. Comparing them on the same question shows the tradeoffs clearly.

Workflow:

- Phase 1 (one-time setup): Upload an Excel file `arrow.r`. Agents 1 and 2 build the database
- Phase 2 (repeat as needed): Submit natural-language queries `arrow.r`. Agents 3, 4, and 5 run on each query

You can run many queries in Phase 2 without redoing the Phase 1 setup.

```
1. INSTALL DEPENDENCIES
!pip install -q gradio openai pandas chromadb openpyxl langchain-community
langchain-openai git+https://github.com/KarAnalytics/llm_cascade.git sentence-
transformers
```

The cell above installs the full toolkit for the multi-agent system. Beyond the basics ( `pandas` , `sqlite3` ), we add `chromadb` for vector storage (used by the RAG agent), `langchain-community` and `langchain-openai` for embedding utilities, `sentence-transformers` for local embedding models, and `llm_cascade` for provider-agnostic LLM calls. The multi-agent version requires more libraries than the single-agent notebook because it runs two parallel answer pathways – SQL and RAG – plus an evaluator to compare them.

### 16.1.1 Multi-Agent System: SQL vs. RAG for Business Database Queries

Why would you use five agents when one could do the job? The answer lies in a principle borrowed from software engineering: **separation of concerns**. Each agent in this notebook has a narrow, well-defined role and a system prompt tailored to that role. The Data Architect thinks about table relationships and primary keys. The SQL Developer thinks about data types and `CREATE TABLE` syntax. The SQL Query Writer thinks about `JOIN`s and `WHERE` clauses. The RAG Analyst ignores SQL entirely and reads the raw data. And the Evaluator acts as an impartial judge, comparing the two answers without knowing which approach produced which.

This specialization offers two concrete benefits. First, each agent’s system prompt can be short, focused, and easy to debug – if the SQL is wrong, you know exactly which agent and which prompt to inspect. Second, by running two independent answer pathways (SQL and RAG) and comparing them, the system

catches mistakes that either approach alone would miss. When the SQL answer says “total revenue is USD 1.2M” and the RAG answer says “USD 1.2M,” you have high confidence the number is correct. When they disagree, that disagreement itself is valuable information.

The five agents:

| Agent                     | Role                                                          | What it specializes in                             |
|---------------------------|---------------------------------------------------------------|----------------------------------------------------|
| Agent 1: Data Architect   | Reads Excel sheets, proposes a logical schema                 | Understanding data structure, primary/foreign keys |
| Agent 2: SQL Developer    | Converts the logical schema into DDL and creates the database | SQL syntax, data types, table creation             |
| Agent 3: SQL Query Writer | Writes and executes SQL SELECT queries for business questions | Query optimization, JOINS, aggregations            |
| Agent 4: RAG Analyst      | Answers the same question by reading raw data (no SQL)        | Natural language reasoning over tabular data       |
| Agent 5: Evaluator        | Compares SQL and RAG answers, returns a similarity score      | LLM-as-a-judge pattern, cross-validation           |

Workflow:

- Phase 1 (one-time setup): Upload an Excel file -> Agents 1 and 2 build the database
- Phase 2 (repeat as needed): Submit natural-language queries -> Agents 3, 4, and 5 run on each query

#### 16.1.1.1 Imports and Setup

We use `pandas` to read Excel, `sqlite3` for the database, `chromadb` (for future vector RAG), and `llm_cascade` for automatic LLM provider fallback across 8 free-tier providers.

```
import os
import sqlite3
import pandas as pd
import chromadb
from llm_cascade import get_cascade

llm = get_cascade()

Global database config
DB_NAME = 'agent_database.db'
CHROMA_CLIENT = chromadb.Client()
```

```
LLM Cascade - available providers:
+ Gemini model=gemini-2.5-flash
+ Ollama model=kimi-k2.5:cloud
+ Groq model=llama-3.3-70b-versatile
+ HuggingFace model=meta-llama/llama-3.3-70B-Instruct
+ Cohere model=command-a-03-2025
+ OpenRouter model=meta-llama/llama-3.3-70b-instruct:free
+ OpenAI model=gpt-4o-mini
Not configured (skipped):
- Grok (xAI) (set XAI_API_KEY)
```

#### 16.1.1.2 Agent 1: Data Architect

Role: Read each Excel sheet, look at the column names and data types, and propose a logical database schema in plain English.

How it works:

- Reads each sheet with `pandas`
- Extracts column names and dtypes

- Sends this metadata to the LLM with a prompt asking for a bulleted schema description (primary keys, foreign keys, relationships)
- The LLM returns a human-readable schema – no SQL yet

The logical schema serves as the shared plan that Agent 2 (SQL Developer) will implement.

```
def agent_1_ingest(file_path):
 '''Agent 1 -- Data Architect: reads Excel, proposes logical schema.'''
 xl = pd.ExcelFile(file_path)
 sheet_data = ''
 data_map = {}

 for sheet in xl.sheet_names:
 df = pd.read_excel(file_path, sheet_name=sheet)
 data_map[sheet] = df
 cols = ', '.join([f'{col} ({dtype})' for col, dtype in zip(df.columns,
df.dtypes)])
 sheet_data += f'Sheet Name: {sheet}' + chr(10) + f'Columns detected: {cols}'
 + chr(10) + chr(10)

 prompt = (
 'Analyze these sheets and provide a logical database schema. '
 'Use ONLY bullet points. Identify suggested Primary Keys and Foreign Keys:'
 + chr(10) + sheet_data
)
 system_prompt = 'You are a Data Architect. Output a logical schema in bulleted
text. Do NOT write SQL code.'
 response = llm.generate(prompt, system_prompt=system_prompt)
 return response.text, data_map

print('Agent 1 (Data Architect) ready.')
```

Agent 1 (Data Architect) ready.

### 16.1.1.3 Agent 2: SQL Developer

Role: Take the logical schema from Agent 1 and turn it into an actual SQLite database populated with the Excel data.

How it works:

- Prompts the LLM to generate SQL CREATE TABLE DDL statements from the logical schema
- Executes the DDL against a fresh SQLite database
- Uses pandas `to_sql()` to populate each table from the corresponding Excel sheet
- Returns the DDL text and a status message

After this agent runs, we have a queryable SQLite database ready for business questions.

```
def agent_2_sql_developer(logical_schema, data_map):
 '''Agent 2 -- SQL Developer: generates DDL, creates DB, populates tables.'''
 prompt = f'Based on this logical schema, write SQLite CREATE TABLE statements:'
 + chr(10) + logical_schema
 system_prompt = 'Return ONLY raw SQL CREATE TABLE statements. No explanation.'
 response = llm.generate(prompt, system_prompt=system_prompt)
 ddl_resp = response.text.replace('```sql', '').replace('```', '')

 # Execute DDL and insert data
 if os.path.exists(DB_NAME):
 os.remove(DB_NAME)
 conn = sqlite3.connect(DB_NAME)
 cursor = conn.cursor()
 try:
```

```

 cursor.executescript(ddl_resp)
 for table, df in data_map.items():
 clean_table_name = table.replace(' ', '_')
 df.to_sql(clean_table_name, conn, if_exists='replace', index=False)
 conn.commit()
 status = 'Database Ready: Tables created and data inserted.'
 except Exception as e:
 status = f'Database Error: {str(e)}'
 finally:
 conn.close()

 return ddl_resp, status

print('Agent 2 (SQL Developer) ready.')

```

Agent 2 (SQL Developer) ready.

#### 16.1.1.4 Agent 3: SQL Query Writer

Role: Given a natural-language business question, write a SQL `SELECT` query and execute it against the database.

How it works:

- Sends the schema + the user's question to the LLM
- Asks the LLM to return ONLY the SQL query (no explanation)
- Strips any markdown code fences
- Runs the query with pandas and returns the result as a formatted string

Strengths: Precise, verifiable, handles aggregations and joins well. Weaknesses: Fails if the LLM writes bad SQL or misunderstands the schema.

```

def agent_3_sql_query(question, logical_schema, conn):
 '''Agent 3 -- SQL Query Writer: writes and executes a SQL SELECT for the
 question.'''
 sql_gen_prompt = (
 'Schema:' + chr(10) + logical_schema + chr(10)
 + 'Query: ' + question + chr(10)
 + 'Generate ONLY the SQL SELECT statement. Table names might have
underscores.'
)
 response = llm.generate(sql_gen_prompt)
 sql_query = response.text.replace('```sql', '').replace('```', '').strip()

 try:
 result_df = pd.read_sql_query(sql_query, conn)
 return sql_query, result_df.to_string()
 except Exception as e:
 return sql_query, f'SQL Execution Error: {str(e)}'

print('Agent 3 (SQL Query Writer) ready.')

```

Agent 3 (SQL Query Writer) ready.

#### 16.1.1.5 Agent 4: RAG Analyst

Role: Answer the same business question using **retrieval-augmented generation** — no SQL, just raw data passed into the LLM prompt as context.

How it works:

- Builds a context string from the first 20 rows of each table (as plain text)
- Sends that context + the user's question to the LLM
- The LLM reads the raw data and answers in natural language

Strengths: Works without valid SQL, handles fuzzy or open-ended questions. Weaknesses: Limited by context window (only sees first 20 rows here), may hallucinate, less precise for numeric aggregations.

```
def agent_4_rag_analyst(question, data_map):
 '''Agent 4 -- RAG Analyst: answers from raw data passed as context.'''
 context = chr(10).join(
 ['Table ' + k + ':' + chr(10) + v.head(20).to_string() for k, v in
data_map.items()]
)
 rag_prompt = (
 'Data Context:' + chr(10) + context + chr(10)
 + 'User Question: ' + question + chr(10)
 + 'Answer the question based on the data provided.'
)
 response = llm.generate(rag_prompt)
 return response.text

print('Agent 4 (RAG Analyst) ready.')
```

Agent 4 (RAG Analyst) ready.

#### 16.1.1.6 Agent 5: Evaluator

Role: Compare the SQL answer (Agent 3) and the RAG answer (Agent 4) for the same question, and return a similarity score.

How it works:

- Sends both answers to the LLM
- Asks it to return ONLY a percentage score (0–100%)
- High score = both approaches agree; low score = they diverge (worth investigating why)

This is a simple “LLM-as-a-judge” pattern — using an LLM to evaluate other LLM outputs.

```
def agent_5_evaluator(sql_result, rag_result):
 '''Agent 5 -- Evaluator: compares SQL and RAG answers, returns a similarity
score.'''
 eval_prompt = (
 'Compare these two answers and provide ONLY a Similarity Score (0-100%).' +
chr(10)
 + 'Answer A: ' + sql_result + chr(10)
 + 'Answer B: ' + rag_result
)
 system_prompt = 'You are a calculator. Output only a percentage score.'
 response = llm.generate(eval_prompt, system_prompt=system_prompt)
 return response.text

print('Agent 5 (Evaluator) ready.')
```

Agent 5 (Evaluator) ready.

#### 16.1.1.7 Interactive App: Upload Excel and Ask Questions

The cell below launches a Gradio app that gives you an interactive UI for the multi-agent system:

- Step 1: Upload your Excel file and click Build Database. Agents 1 and 2 run: the Data Architect designs a logical schema, and the SQL Developer writes the DDL and creates a SQLite database.
- Step 2: Type a business question and click Submit Query (or press Enter). Agents 3, 4, and 5 run: the SQL Query Writer generates SQL, the RAG Analyst answers from raw data, and the Evaluator compares the two. Each Q&A is added to the chat history.
- Repeat Step 2 as many times as you want — no need to re-upload the file.

If you do not have an Excel file on hand, download this sample dataset and upload it: [SUPPLIER\\_PARTS.xlsx](#) (click **Download raw file** on the GitHub page).

This is the classroom-friendly way to compare SQL vs. RAG live. Students can submit their own questions and watch the agents disagree (or agree) in real time.

```
import gradio as gr

Module-level state so the Gradio UI can persist the schema across queries
_app_state = {'logical_schema': None, 'data_map': None, 'db_ready': False}

def ui_build_database(file):
 '''Phase 1: run Agents 1 and 2 on the uploaded file.'''
 if file is None:
 return '', '', 'Please upload an Excel file first.'

 try:
 logical_schema, data_map = agent_1_ingest(file.name)
 ddl, status = agent_2_sql_developer(logical_schema, data_map)

 _app_state['logical_schema'] = logical_schema
 _app_state['data_map'] = data_map
 _app_state['db_ready'] = 'Ready' in status
 return logical_schema, ddl, status
 except Exception as e:
 return '', '', f'Error building database: {e}'

def ui_ask_question(question, history):
 '''Phase 2: run Agents 3, 4, 5 on a single question.'''
 if not _app_state['db_ready']:
 bot_msg = 'Please upload an Excel file and click Build Database first.'
 return history + [[question, bot_msg]], ''

 if not question or not question.strip():
 return history, question

 try:
 conn = sqlite3.connect(DB_NAME)
 try:
 sql_query, sql_result = agent_3_sql_query(question,
 _app_state['logical_schema'], conn)
 finally:
 conn.close()

 rag_result = agent_4_rag_analyst(question, _app_state['data_map'])
 similarity = agent_5_evaluator(sql_result, rag_result)

 response = (
 '**Agent 3 (SQL):**' + chr(10) + '```sql' + chr(10) + sql_query +
 chr(10) + '```' + chr(10) + chr(10)
 + '**SQL Result:**' + chr(10) + '```' + chr(10) + sql_result + chr(10) +
 '```' + chr(10) + chr(10)
 + '**Agent 4 (RAG):**' + chr(10) + rag_result + chr(10) + chr(10)
 + '**Agent 5 (Similarity):**' + chr(10) + similarity
)
 except Exception as e:
 return history, question, f'Error running query: {e}'
 except Exception as e:
 return history, question, f'Error asking question: {e}'
```

```

)
 except Exception as e:
 response = f'Error: {e}'

 return history + [[question, response]], ''

with gr.Blocks(title='Multi-Agent DB') as demo:
 gr.Markdown('# Multi-Agent: Excel to SQLite to SQL vs. RAG Comparison')

 gr.Markdown('### Step 1: Upload Excel file and build the database (Agents 1 & 2)')
 with gr.Row():
 file_input = gr.File(label='Excel File', file_types=['.xlsx', '.xls'])
 build_btn = gr.Button('Build Database', variant='primary')

 with gr.Row():
 with gr.Column():
 out_schema = gr.Markdown(label='Logical Schema (Agent 1)')
 with gr.Column():
 out_ddl = gr.Code(label='SQL DDL (Agent 2)', language='sql')
 out_status = gr.Textbox(label='Status', interactive=False)

 build_btn.click(
 ui_build_database,
 inputs=[file_input],
 outputs=[out_schema, out_ddl, out_status],
)

 gr.Markdown('### Step 2: Ask business questions (Agents 3, 4 & 5 run per query)')
 chatbot = gr.Chatbot(label='Q&A History', height=400)
 query_input = gr.Textbox(
 label='Your question',
 placeholder='e.g., How many total sales? Who is the top employee?',
)
 submit_btn = gr.Button('Submit Query', variant='primary')

 submit_btn.click(
 ui_ask_question,
 inputs=[query_input, chatbot],
 outputs=[chatbot, query_input],
)
 query_input.submit(
 ui_ask_question,
 inputs=[query_input, chatbot],
 outputs=[chatbot, query_input],
)

demo.launch(debug=False)

```

/tmp/ipykernel\_1601/2530079974.py:77: UserWarning: You have not specified a value for the `type` parameter. Defaulting to the 'tuples' format for chatbot messages, but this is deprecated and will be removed in a future version of Gradio. Please set type='messages' instead, which uses openai-style dictionaries with 'role' and 'content' keys.

chatbot = gr.Chatbot(label='Q&A History', height=400)

/tmp/ipykernel\_1601/2530079974.py:77: DeprecationWarning: The default value of 'allow\_tags' in gr.Chatbot will be changed from False to True in Gradio 6.0. You will need to explicitly set allow\_tags=False if you want to disable tags in your chatbot.

chatbot = gr.Chatbot(label='Q&A History', height=400)

```
It looks like you are running Gradio on a hosted Jupyter notebook, which requires
`share=True`. Automatically setting `share=True` (you can turn this off by setting
`share=False` in `launch()` explicitly).
```

```
Colab notebook detected. To show errors in colab notebook, set debug=True in
launch()
```

```
* Running on public URL: https://a196d879b4ffc4d788.gradio.live
```

```
This share link expires in 1 week. For free permanent hosting and GPU upgrades, run
`gradio deploy` from the terminal in the working directory to deploy to Hugging Face
Spaces (https://huggingface.co/spaces)
```

#### 16.1.1.8 Key Takeaways

What we demonstrated:

- A multi-agent system where each agent has a narrow, well-defined role
- Agents pass outputs to each other (logical schema → DDL → populated DB → queries)
- The same business question answered in two ways (SQL vs. RAG) with automatic comparison
- Separation of setup phase (expensive, run once) from query phase (cheap, repeat many times)

When to use SQL vs. RAG on structured data:

|                            | SQL (Agent 3) | RAG (Agent 4) |
|----------------------------|---------------|---------------|
| Numeric aggregations       | Excellent     | Often wrong   |
| Fuzzy/open-ended questions | Limited       | Better        |
| Joins across tables        | Excellent     | Struggles     |
| Context size limit         | None          | Yes           |
| Requires valid schema      | Yes           | No            |
| Precision                  | High          | Variable      |

In practice, systems often use both: SQL for precise numeric queries, RAG for explanatory or fuzzy questions – and an LLM-as-a-judge (Agent 5) to flag when they disagree.

Exercises:

- Add a new agent that generates a chart from the SQL result
- Extend Agent 4 to use vector embeddings (ChromaDB) instead of raw context
- Add an Agent 6 that explains WHY the two answers disagreed (when similarity is low)
- Replace `llm_cascade` with a specific provider and see how different LLMs perform at each agent role

#### 16.1.1.9 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

```
https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/MultiAgent_DB_multi_
human_input.ipynb
```



## 16.2 Multi-Agent System: SQL vs. RAG Comparison (Single-Input Version)

This notebook implements a **multi-agent system** where specialized AI agents collaborate to analyze an uploaded Excel file. It is the original Gradio-based version of the multi-agent demo, designed around a **single-button workflow**: you upload your data, type your questions, click one button, and all four agents run in sequence.

**How the agents work together:**

- **Agent 1 (Data Architect)** examines the uploaded Excel file and generates a logical database schema
- **Agent 2 (SQL Developer)** converts that schema into SQLite tables and loads the data
- **Agent 3 (RAG Analyst)** answers your questions using retrieval-augmented generation over the raw data
- **Agent 4 (Evaluator)** compares the SQL-based and RAG-based answers and scores their similarity

This contrasts with the multi-input version (`multi_agent_multi_human_db.ipynb`), which separates the workflow into distinct steps with separate buttons for each phase. The single-input design is simpler to use but gives less visibility into intermediate results. Both versions demonstrate the same core concept: different AI agents can specialize in different tasks and hand off work to each other.

```
1. INSTALL DEPENDENCIES
!pip install -q gradio openai pandas chromadb openpyxl langchain-community
langchain-openai git+https://github.com/KarAnalytics/llm_cascade.git sentence-
transformers
```

**IGNORE THE ABOVE ERRORS AND PROCEED, the code will still execute. These errors are specific to google colab.**

```
import os
import sqlite3
import pandas as pd
import gradio as gr
import chromadb
```

### 16.2.0.1 Setting Up the LLM and Database

We initialize the LLM cascade (which automatically finds whichever API key you have configured) and set up SQLite for structured queries and ChromaDB for vector-based retrieval. This dual-database approach is what makes the SQL-vs-RAG comparison possible: the same data lives in both a relational database and a vector store, and we can compare the answers each approach produces.

```
from llm_cascade import get_cascade

llm = get_cascade()

--- GLOBAL STATE & DB CONFIG ---
DB_NAME = 'agent_database.db'
CHROMA_CLIENT = chromadb.Client()
```

```
LLM Cascade - available providers:
+ Gemini model=gemini-2.5-flash
+ Ollama model=kimi-k2.5:cloud
+ Groq model=llama-3.3-70b-versatile
+ HuggingFace model=meta-llama/llama-3.3-70B-Instruct
+ Cohere model=command-a-03-2025
+ OpenRouter model=meta-llama/llama-3.3-70b-instruct:free
+ OpenAI model=gpt-4o-mini
Not configured (skipped):
- Grok (xAI) (set XAI_API_KEY)
```

```
--- AGENT LOGIC ---
```

```

def agent_1_ingest(file_path):
 '''Data Ingestion & Logical Schema Identification'''
 xl = pd.ExcelFile(file_path)
 sheet_data = ''
 data_map = {}

 for sheet in xl.sheet_names:
 df = pd.read_excel(file_path, sheet_name=sheet)
 data_map[sheet] = df
 cols = ', '.join([f'{col} ({dtype})' for col, dtype in zip(df.columns,
df.dtypes)])
 sheet_data += f'Sheet Name: {sheet}' + chr(10) + f'Columns detected: {cols}'
+ chr(10) + chr(10)

 # Prompting for Bulleted Logical Schema ONLY
 prompt = f'Analyze these sheets and provide a logical database schema. Use ONLY
bullet points. Identify suggested Primary Keys and Foreign Keys:' + chr(10) +
sheet_data
 system_prompt = 'You are a Data Architect. Output a logical schema in bulleted
text. Do NOT write SQL code.'
 response = llm.generate(prompt, system_prompt=system_prompt)
 return response.text, data_map

def agent_2_sql_developer(logical_schema, data_map):
 '''SQL Developer Agent: Create & Populate'''
 # 1. Generate DDL
 prompt = f'Based on this logical schema, write SQLite CREATE TABLE statements:'
+ chr(10) + logical_schema
 system_prompt = 'Return ONLY raw SQL CREATE TABLE statements. No explanation.'
 response = llm.generate(prompt, system_prompt=system_prompt)
 ddl_resp = response.text.replace('```sql', '').replace('```', '')

 # 2. Execute DDL and Insert Data immediately
 if os.path.exists(DB_NAME): os.remove(DB_NAME)
 conn = sqlite3.connect(DB_NAME)
 cursor = conn.cursor()
 try:
 cursor.executescript(ddl_resp)
 # Match dataframes to newly created tables
 for table, df in data_map.items():
 # Basic cleanup: remove spaces from sheet names for SQL safety
 clean_table_name = table.replace(' ', '_')
 df.to_sql(clean_table_name, conn, if_exists='replace', index=False)
 conn.commit()
 status = 'Database Ready: Tables created and data inserted.'
 except Exception as e:
 status = f'Database Error: {str(e)}'
 finally:
 conn.close()

 return ddl_resp, status

def process_queries(queries_text, logical_schema, data_map):
 '''Handles multiple queries and runs Agent 2, 3, and 4 for each.'''
 queries = [q.strip() for q in queries_text.split(chr(10)) if q.strip()]
 results = []

 conn = sqlite3.connect(DB_NAME)

 for q in queries:
 # Agent 2: SQL Approach
 sql_gen_prompt = 'Schema:' + chr(10) + logical_schema + chr(10) + 'Query: '
+ q + chr(10) + 'Generate ONLY the SQL SELECT statement. Table names might have

```

```

underscores.'
 response = llm.generate(sql_gen_prompt)
 sql_query = response.text.replace('```sql', '').replace('```', '')

 try:
 sql_data = pd.read_sql_query(sql_query, conn).to_string()
 except Exception as e:
 sql_data = f'SQL Execution Error: {str(e)}'

 # Agent 3: RAG Approach
 context = chr(10).join(['Table ' + k + ':' + chr(10) +
v.head(20).to_string() for k, v in data_map.items()])
 rag_prompt = 'Data Context:' + chr(10) + context + chr(10) + 'User Question:
' + q + chr(10) + 'Answer the question based on the data provided.'
 response = llm.generate(rag_prompt)
 rag_res = response.text

 # Agent 4: Similarity Scorer Only
 eval_prompt = 'Compare these two answers and provide ONLY a Similarity Score
(0-100%).' + chr(10) + 'Answer A: ' + sql_data + chr(10) + 'Answer B: ' + rag_res
 system_prompt = 'You are a calculator. Output only a percentage score.'
 response = llm.generate(eval_prompt, system_prompt=system_prompt)
 score = response.text

 results.append('QUERY: ' + q + chr(10) + 'SQL Result: ' + sql_data + chr(10)
+ 'RAG Result: ' + rag_res + chr(10) + 'SIMILARITY SCORE: ' + score + chr(10) + '-'
* 40)

 conn.close()
 return chr(10).join(results)

--- WORKFLOW CONTROLLER ---
def run_app(file, queries):
 logical_schema, data_map = agent_1_ingest(file.name)
 ddl, db_status = agent_2_sql_developer(logical_schema, data_map)
 comparison_results = process_queries(queries, logical_schema, data_map)
 return logical_schema, ddl, db_status, comparison_results

```

### 16.2.0.2 The Agent Functions

The code above defines the core logic for all four agents. Agent 1 reads the Excel file and asks the LLM to propose a database schema. Agent 2 takes that schema and generates DDL (CREATE TABLE statements), then loads the actual data. For each user query, Agent 2 also generates a SQL SELECT statement, Agent 3 answers the same question using RAG over the raw data, and Agent 4 scores how similar the two answers are. This pipeline runs end-to-end with a single function call – the orchestration is handled by the `run_app` wrapper.

```

--- GRADIO UI ---
with gr.Blocks(theme=gr.themes.Monochrome()) as demo:
 gr.Markdown("# 🚀 Agentic SQL vs. RAG Comparison")

 with gr.Row():
 file_input = gr.File(label="Upload Excel File")
 query_input = gr.Textbox(label="Business Queries (One per line)", lines=5,
 placeholder="How many sales in North?\nWho is the
top employee?")

 run_btn = gr.Button("🚀 Process All Agents", variant="primary")

 with gr.Tabs():
 with gr.TabItem("Data Architecture (Agent 1 & 2)"):
 out_schema = gr.Markdown(label="Logical Schema (Agent 1)")

```

```

 out_ddl = gr.Code(label="Generated DDL (Agent 2)", language="sql")
 out_status = gr.Textbox(label="DB Population Status")

 with gr.TabItem("Comparative Results (Agent 2, 3 & 4)":
 out_comparison = gr.Textbox(label="The Showdown Results", lines=20)

 run_btn.click(
 run_app,
 inputs=[file_input, query_input],
 outputs=[out_schema, out_ddl, out_status, out_comparison]
)

demo.launch()

```

```

/tmp/ipykernel_6238/297498658.py:2: DeprecationWarning: The 'theme' parameter in the
Blocks constructor will be removed in Gradio 6.0. You will need to pass 'theme' to
Blocks.launch() instead.
 with gr.Blocks(theme=gr.themes.Monochrome()) as demo:

```

It looks like you are running Gradio on a hosted Jupyter notebook, which requires `share=True`. Automatically setting `share=True` (you can turn this off by setting `share=False` in `launch()` explicitly).

Colab notebook detected. To show errors in colab notebook, set debug=True in launch()

\* Running on public URL: <https://3fc4aa289497fecaa4.gradio.live>

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the working directory to deploy to Hugging Face Spaces (<https://huggingface.co/spaces>)

### 16.2.0.3 The Gradio Interface

The Gradio UI provides a simple two-input interface: upload an Excel file on the left, type your business questions (one per line) on the right, and click the Process button. Results appear in two tabs – one showing the schema and DDL from Agents 1 and 2, the other showing the side-by-side SQL vs. RAG comparison from Agents 3 and 4. If you do not have an Excel file on hand, download this sample dataset and upload it: [SUPPLIER\\_PARTS.xlsx](#) (click **Download raw file** on the GitHub page). When running in Google Colab, Gradio automatically creates a public share link so you can access the interface from any browser.

### 16.2.1 Key takeaways

- **Specialized agents** with distinct system prompts (Architect, SQL Developer, RAG Analyst, Evaluator) can be chained into a single end-to-end pipeline.
- **Dual-path answering** – the same question is answered via SQL and via RAG – surfaces disagreements that a single approach would silently hide.
- **An evaluator agent** that scores similarity between the two answers is a cheap, LLM-based form of cross-validation you can add to any pipeline.
- **Single-input UX** trades visibility for simplicity: one click runs every agent, but you see less of what happens between them than in a stepwise interface.
- **SQLite plus ChromaDB** side by side gives a clean substrate for comparing structured retrieval against semantic retrieval over the same dataset.

### 16.2.2 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. (If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)

```

https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/MultiAgent_DB_single_human_input.ipynb

```

## 16.3 Single-Agent System: One Agent Handles Everything

This is the **contrast notebook** to `MultiAgent_DB_multi_human_input.ipynb`. It follows the **same workflow** – upload an Excel file, infer a schema, build a SQLite database, then answer business questions with SQL – but with one critical difference:

**One agent persona does all the work.** There is no “Data Architect” agent, “SQL Developer” agent, “SQL Query Writer” agent, etc. Instead, a single generalist “Data Analyst” uses the **same system prompt** for every task: designing the schema, writing DDL, and writing SQL queries.

**What we keep from the multi-agent notebook:**

- Excel file input arrow.r sheets inferred as SQL tables
- SQLite database is actually created and populated
- Phase 1 (one-time setup) arrow.r Phase 2 (repeatable business queries)
- SQL generation and execution for each query

**What we remove:**

- No separate specialized agents – just one agent wearing multiple “hats”
- No RAG approach (Agent 4 in the multi-agent version)
- No evaluator / similarity scorer (Agent 5 in the multi-agent version)
- One answer per query (from SQL), not two

**Why build this contrast?** To see whether the multi-agent specialization actually helps, or whether one well-prompted agent can do the same job.

Every LLM call here goes through the `llm_cascade` wrapper introduced in Chapter 5, so the agent transparently falls back across providers as quotas or keys change.

```
1. INSTALL DEPENDENCIES
!pip install -q gradio pandas openpyxl git+https://github.com/KarAnalytics/llm_cascade.git
```

The cell above installs the necessary libraries. The `llm_cascade` package is a lightweight wrapper that automatically detects which LLM API keys you have configured and falls back through providers in priority order – Gemini first, then Groq, HuggingFace, Cohere, and others. This means the notebook works regardless of which free-tier provider you have access to, and it gracefully handles rate limits by trying the next provider in line.

### 16.3.1 Single-Agent System: One Agent Handles Everything

In the multi-agent notebook, we split the work across five specialized personas: a Data Architect to design the schema, a SQL Developer to write DDL, a SQL Query Writer to answer questions, a RAG Analyst for a second opinion, and an Evaluator to compare the two. It is an elegant architecture, but it raises a natural question: **is all that specialization actually necessary?** Could one well-prompted generalist agent do the same job, faster and cheaper?

This notebook answers that question by building the **exact same workflow** – upload an Excel file, infer a schema, build a SQLite database, then answer business questions with SQL – but with a single “Data Analyst” agent that wears all the hats. There is one system prompt shared across every task. The agent does not know it is playing multiple roles; it simply follows whatever user-prompt instructions it receives.

**The key design insight is that the system prompt defines the agent’s identity, while the user prompt defines the task.** By keeping one system prompt and varying the user prompt, we get the simplicity of a single agent without sacrificing the ability to perform diverse tasks. The tradeoff is transparency: with five agents, you can inspect each one’s output independently; with one agent, you see only the final result.

**What we keep from the multi-agent notebook:**

- Excel file input with sheets inferred as SQL tables
- SQLite database is actually created and populated
- Phase 1 (one-time setup) followed by Phase 2 (repeatable business queries)

## What we remove:

- No separate specialized agents – just one agent wearing multiple “hats”
- No RAG approach (Agent 4 in the multi-agent version)
- No evaluator / similarity scorer (Agent 5 in the multi-agent version)
- One answer per query (from SQL), not two

### 16.3.1.1 Imports and Setup

Same libraries as the multi-agent notebook, minus `chromadb` (we’re not building a RAG system). We use `pandas` to read Excel, `sqlite3` for the database, and `llm_cascade` for automatic LLM provider fallback.

```
import os
import sqlite3
import pandas as pd
from llm_cascade import get_cascade

llm = get_cascade()

DB_NAME = 'single_agent_database.db'
```

### 16.3.1.2 The Single Agent: Data Analyst

**Role:** A generalist Data Analyst that can design schemas, write DDL, and write SQL queries — all tasks previously split across four different specialized agents in the multi-agent notebook.

**Key design decision:** All three tasks below use the **same** `ANALYST_SYSTEM_PROMPT`. The system prompt describes the agent’s overall role; only the user prompt changes per task. This is what makes it a “single agent” — there’s one persona, not five.

#### Tasks the single agent performs:

| Task                           | Function                               | Replaces                   |
|--------------------------------|----------------------------------------|----------------------------|
| Design logical schema          | <code>analyst_design_schema()</code>   | Agent 1 (Data Architect)   |
| Write SQL DDL                  | <code>analyst_write_ddl()</code>       | Agent 2 (SQL Developer)    |
| Write SQL query for a question | <code>analyst_write_sql_query()</code> | Agent 3 (SQL Query Writer) |

Compare the brevity of this section to the five separate agent cells in the multi-agent notebook.

```
One system prompt shared across all analyst tasks
ANALYST_SYSTEM_PROMPT = (
 'You are a skilled Data Analyst. You can design logical database schemas, '
 'write SQL DDL statements, and write SQL queries to answer business questions. '
 'Follow the user instructions precisely and output only what is requested.'
)

def read_excel_to_map(file_path):
 '''Read an Excel file into a dict of {sheet_name: DataFrame}. Not an LLM
 call.'''
 xl = pd.ExcelFile(file_path)
 data_map = {}
 for sheet in xl.sheet_names:
 data_map[sheet] = pd.read_excel(file_path, sheet_name=sheet)
 return data_map

def analyst_design_schema(data_map):
 '''Task 1: ask the agent to propose a logical schema from the sheets.'''
 sheet_summary = ''
 for sheet, df in data_map.items():
 cols = ', '.join([f'{c} ({dtype})' for c, dtype in zip(df.columns,
```

```

df.dtypes)])
 sheet_summary += f'Sheet Name: {sheet}' + chr(10) + f'Columns detected: {cols}' + chr(10) + chr(10)

 prompt = (
 'Analyze these sheets and provide a logical database schema. '
 'Use ONLY bullet points. Identify suggested Primary Keys and Foreign Keys:'
 + chr(10)
 + sheet_summary
)
 response = llm.generate(prompt, system_prompt=ANALYST_SYSTEM_PROMPT)
 return response.text

def analyst_write_ddl(logical_schema):
 '''Task 2: ask the agent to turn the logical schema into SQL CREATE TABLE statements.'''
 prompt = (
 'Based on this logical schema, write SQLite CREATE TABLE statements. '
 'Return ONLY raw SQL. No explanation, no markdown fences.' + chr(10)
 + logical_schema
)
 response = llm.generate(prompt, system_prompt=ANALYST_SYSTEM_PROMPT)
 return response.text.replace('```sql', '').replace('```', '')

def analyst_write_sql_query(question, logical_schema):
 '''Task 3: ask the agent to write a SQL SELECT for a business question.'''
 prompt = (
 'Schema:' + chr(10) + logical_schema + chr(10)
 + 'Question: ' + question + chr(10)
 + 'Generate ONLY the SQL SELECT statement. Table names might have underscores. '
 + 'No explanation, no markdown fences.'
)
 response = llm.generate(prompt, system_prompt=ANALYST_SYSTEM_PROMPT)
 return response.text.replace('```sql', '').replace('```', '').strip()

print('Single agent (Data Analyst) ready. One persona, three tasks.')

```

### 16.3.1.3 Interactive App: Upload Excel and Ask Questions

The cell below launches a **Gradio app** that gives you an interactive UI for the single agent:

- **Step 1:** Upload your Excel file and click **Build Database**. The single agent reads the sheets, designs a logical schema, writes the DDL, and creates a SQLite database.
- **Step 2:** Type a business question and click **Submit Query** (or press Enter). The same agent writes a SQL query, executes it, and shows the result. Your question and the answer are added to the chat history.
- **Repeat Step 2** as many times as you want — no need to re-upload the file.

If you do not have an Excel file on hand, download this sample dataset and upload it: [SUPPLIER\\_PARTS.xlsx](#) (click **Download raw file** on the GitHub page).

**Compare with the multi-agent notebook:** that notebook runs 3 LLM calls per query (SQL + RAG + Evaluator). This one runs just 1 LLM call per query. You should notice it feels faster, and the generated SQL is often very similar.

```

import gradio as gr

Module-level state so the Gradio UI can persist the schema across queries

```



```

_app_state = {'logical_schema': None, 'data_map': None, 'db_ready': False}

def ui_build_database(file):
 '''Phase 1: run the single agent on the uploaded file to build the DB.'''
 if file is None:
 return '', '', 'Please upload an Excel file first.'

 try:
 data_map = read_excel_to_map(file.name)
 logical_schema = analyst_design_schema(data_map)
 ddl = analyst_write_ddl(logical_schema)

 if os.path.exists(DB_NAME):
 os.remove(DB_NAME)
 conn = sqlite3.connect(DB_NAME)
 try:
 conn.executescript(ddl)
 for table, df in data_map.items():
 clean = table.replace(' ', '_')
 df.to_sql(clean, conn, if_exists='replace', index=False)
 conn.commit()
 finally:
 conn.close()

 _app_state['logical_schema'] = logical_schema
 _app_state['data_map'] = data_map
 _app_state['db_ready'] = True
 return logical_schema, ddl, 'Database ready. You can now ask questions below.'
 except Exception as e:
 return '', '', f'Error building database: {e}'

def ui_ask_question(question, history):
 '''Phase 2: run the single agent on a single question.'''
 if not _app_state['db_ready']:
 bot_msg = 'Please upload an Excel file and click Build Database first.'
 return history + [[question, bot_msg]], ''

 if not question or not question.strip():
 return history, question

 try:
 sql_query = analyst_write_sql_query(question, _app_state['logical_schema'])
 conn = sqlite3.connect(DB_NAME)
 try:
 result_df = pd.read_sql_query(sql_query, conn)
 sql_result = result_df.to_string()
 except Exception as e:
 sql_result = f'SQL Execution Error: {e}'
 finally:
 conn.close()

 response = '**Generated SQL:**' + chr(10) + '```sql' + chr(10) + sql_query + chr(10) + '```' + chr(10) + chr(10) + '**Result:**' + chr(10) + '```' + chr(10) + sql_result + chr(10) + '```'
 except Exception as e:
 response = f'Error: {e}'

 return history + [[question, response]], ''

```



```

with gr.Blocks(title='Single Agent DB') as demo:
 gr.Markdown('# Single Agent: Excel to SQLite to Natural-Language Queries')

 gr.Markdown('### Step 1: Upload Excel file and build the database')
 with gr.Row():
 file_input = gr.File(label='Excel File', file_types=['.xlsx', '.xls'])
 build_btn = gr.Button('Build Database', variant='primary')

 with gr.Row():
 with gr.Column():
 out_schema = gr.Markdown(label='Logical Schema (agent output)')
 with gr.Column():
 out_ddl = gr.Code(label='SQL DDL (agent output)', language='sql')
 out_status = gr.Textbox(label='Status', interactive=False)

 build_btn.click(
 ui_build_database,
 inputs=[file_input],
 outputs=[out_schema, out_ddl, out_status],
)

 gr.Markdown('### Step 2: Ask business questions (repeat as needed)')
 chatbot = gr.Chatbot(label='Q&A History', height=400)
 query_input = gr.Textbox(
 label='Your question',
 placeholder='e.g., How many total sales? Who is the top employee?',
)
 submit_btn = gr.Button('Submit Query', variant='primary')

 submit_btn.click(
 ui_ask_question,
 inputs=[query_input, chatbot],
 outputs=[chatbot, query_input],
)
 query_input.submit(
 ui_ask_question,
 inputs=[query_input, chatbot],
 outputs=[chatbot, query_input],
)

demo.launch(debug=False)

```

#### 16.3.1.4 Single-Agent vs. Multi-Agent: The Tradeoffs

This notebook solves the same problem as `MultiAgent_DB_multi_human_input.ipynb` but with one agent persona instead of five specialized ones. Here's the comparison:

| Aspect                       | Single-Agent (this notebook)          | Multi-Agent                         |
|------------------------------|---------------------------------------|-------------------------------------|
| Number of agent personas     | 1 (one system prompt)                 | 5 (five specialized system prompts) |
| LLM calls per Phase 1 setup  | 2 (schema + DDL)                      | 2 (schema + DDL)                    |
| LLM calls per business query | 1 (SQL only)                          | 3 (SQL + RAG + Evaluator)           |
| SQLite database created?     | Yes                                   | Yes                                 |
| RAG approach included?       | No                                    | Yes (Agent 4)                       |
| Cross-check / evaluator?     | No                                    | Yes (Agent 5)                       |
| Code complexity              | Lower (one system prompt to maintain) | Higher (five system prompts)        |

| Aspect                | Single-Agent (this notebook)             | Multi-Agent                        |
|-----------------------|------------------------------------------|------------------------------------|
| Verifiability         | Same as multi-agent (SQL is inspectable) | Same                               |
| Speed per query       | ~3x faster (1 call vs. 3)                | Slower                             |
| Cost per query        | ~3x cheaper (1 call vs. 3)               | Higher                             |
| Catches LLM mistakes? | No (only one answer)                     | Yes (Agent 5 compares SQL vs. RAG) |

#### 16.3.1.4.1 When is the specialization worth it?

##### Single-agent wins when:

- You trust the SQL output (simple queries, good LLM)
- Speed and cost matter more than cross-validation
- You just want answers, not multiple perspectives

##### Multi-agent wins when:

- You need a sanity check (the evaluator agent catches when SQL and RAG disagree)
- Specialization genuinely produces better outputs (e.g., a dedicated “SQL expert” prompt may write better queries than a generalist)
- The workflow is complex enough that debugging individual steps is valuable
- You want to audit where each answer came from

#### 16.3.1.4.2 The honest lesson

For this particular problem, the single-agent approach is surprisingly competitive. **The multi-agent architecture adds value mainly through cross-checking (Agent 5), not through specialization of the SQL-writing step.** In practice, the same LLM with the same prompt writes similar SQL whether you call it “SQL Developer Agent” or “Data Analyst Agent.”

**A common pattern:** start with one agent; add more agents only when you discover a specific failure mode that specialization (or cross-checking) would fix. Don’t build five agents upfront just because frameworks encourage it.

#### 16.3.1.4.3 Exercises

- Run the **same question** through both notebooks and compare the generated SQL. Does the specialization help?
- Try a question where the LLM writes buggy SQL. Does the multi-agent evaluator catch it?
- Time both approaches. What’s the latency difference per query?
- Add a second answer path to this notebook (e.g., a RAG approach) so you get the evaluator benefit without splitting into multiple agent personas.

#### 16.3.1.5 Key takeaways

- **One generalist agent** with a single system prompt can handle schema design, DDL generation, and SQL query writing that the multi-agent version split across five personas.
- **System prompt defines identity, user prompt defines task** – varying the user prompt alone is often enough to switch between data-architect, DBA, and analyst behaviors.
- **Fewer LLM calls** (one per business query instead of three) makes the single-agent version roughly 3x faster and cheaper per question.
- **What you lose** is cross-checking: with only one answer path, there is no evaluator agent to catch buggy SQL or flag disagreements between approaches.
- **Start simple** – add specialized agents only when you observe a concrete failure mode that specialization or cross-validation would fix.

#### 16.3.1.6 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

**Estimated run time:** ~3 minutes (requires Excel file upload)

[https://colab.research.google.com/github/KarAnalytics/code\\_demos/blob/main/SingleAgent\\_DB.ipynb](https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/SingleAgent_DB.ipynb)



## Chapter 17

# Autonomous Agents

Previous chapters used agents with predefined workflows — humans decided the pipeline, agents just filled the roles. An **autonomous agent** takes a high-level goal and figures out the steps itself.

This chapter builds a Business Idea Validator that autonomously:

- **Plans** its own research questions
- **Executes** each question with accumulated context
- **Synthesizes** findings into a recommendation
- **Self-critiques** with a confidence score

No human tells it what questions to ask — the LLM decides the entire workflow from a one-sentence goal.

## 17.1 Autonomous Agent: Business Idea Validator

In every agent notebook so far, a human has decided the workflow. The multi-agent system runs Agents 1 through 5 in a fixed order. The single-agent system runs schema design, then DDL, then queries. Even the tool-calling parking assistant follows a human-designed ReAct loop. The agent may choose *which* tool to call at each step, but the overall structure – “answer one question using tools” – is predefined.

This notebook breaks that pattern. Here, the agent receives only a **one-sentence goal** (“Validate this business idea”) and figures out everything else on its own: what questions to research, how to answer each one, how to synthesize the findings into a recommendation, and whether that recommendation is actually any good. No human intervenes between phases. No pipeline is hardcoded. The agent plans, executes, synthesizes, and self-critiques – autonomously.

**What makes this “autonomous” vs. the previous agents?**

|                             | Tool-calling agent                    | Multi-agent system                  | <b>Autonomous workflow (this)</b>                       |
|-----------------------------|---------------------------------------|-------------------------------------|---------------------------------------------------------|
| Who decides the task steps? | Agent picks tools at each step        | Humans pre-define the pipeline      | <b>Agent plans the whole workflow</b>                   |
| Fixed or dynamic pipeline?  | Dynamic (tool-by-tool)                | Fixed (Agent 1 -> 2 -> 3...)        | <b>Dynamic (plan changes per input)</b>                 |
| Self-critique?              | No                                    | Only if an Evaluator agent is added | <b>Yes, as a built-in phase</b>                         |
| Typical use case            | Answer questions using external tools | Well-defined workflows (ETL, BI)    | <b>Open-ended goals (research, writing, validation)</b> |

**Why self-critique matters:** Without it, the agent would produce a recommendation and stop, with no indication of whether the recommendation is strong or weak. The Reflect phase acts as a built-in skeptic – it identifies gaps in the reasoning, flags unsupported assumptions, and assigns a confidence score. This gives the human consumer a calibrated signal: a confidence of 8/10 means “probably act on this”; a confidence of 4/10 means “do more research before deciding.” In production systems, you could feed a low-confidence critique back into the planner for another iteration, creating a self-improving loop.

The contrast is direct: in Chapter 16’s multi-agent and single-agent notebooks a human fixes the pipeline, while here the LLM plans it from a single-sentence goal.

```
1. INSTALL DEPENDENCIES
!pip install -q git+https://github.com/KarAnalytics/llm_cascade.git
```

**IGNORE THE ABOVE ERRORS AND PROCEED** — they are specific to Google Colab’s preinstalled packages and the code will still execute correctly.

### 17.1.1 Imports and Setup

Minimal dependencies: just `llm_cascade` for automatic LLM provider fallback. No tools, no databases, no vector stores — this is a pure LLM reasoning workflow.

```
from llm_cascade import get_cascade

llm = get_cascade()
```

### 17.1.2 The Four Phases of the Autonomous Workflow

Each phase is a single function that makes one LLM call with a phase-specific system prompt. Together, the four phases form the **Plan-Execute-Synthesize-Reflect** loop – a pattern that appears in many real-world autonomous agent frameworks (AutoGPT, BabyAGI, LangGraph research agents).

Notice that **each phase has a different system prompt**, but this is not the same as the multi-agent architecture. In the multi-agent notebook, each agent has a different *role* (Data Architect, SQL Developer).

Here, each phase has a different *cognitive task*: the Planner needs to think strategically about what questions matter; the Executor needs to think concretely about answering one question at a time; the Synthesizer needs to weigh tradeoffs and make a judgment call; and the Critic needs to be genuinely skeptical, looking for holes in the reasoning. These are four modes of thinking, not four job titles.

1. **Plan** – Given the goal, generate 4-5 specific questions that need to be answered
2. **Execute** – Loop through the plan and answer each question, passing prior answers as accumulating context
3. **Synthesize** – Combine all answers into a final recommendation with a clear go/no-go/conditional verdict
4. **Reflect** – Critique the recommendation, identify weaknesses, and assign a confidence score from 1 to 10

```
Phase 1: Planner -- decides WHAT needs to be researched
PLANNER_SYSTEM = (
 'You are a strategic business planner. Given a high-level goal, break it down '
 'into the 4-5 most critical questions that must be answered to achieve it. '
 'Output ONLY a numbered list of questions. No preamble, no conclusion.'
)

def plan(goal):
 '''Phase 1: generate a research plan as a list of questions.'''
 prompt = f'Goal: {goal}' + chr(10) + chr(10) + 'List the 4-5 most critical '
 questions to answer.'
 response = llm.generate(prompt, system_prompt=PLANNER_SYSTEM)

 # Parse the numbered list into a Python list of strings
 steps = []
 for line in response.text.split(chr(10)):
 line = line.strip()
 if not line:
 continue
 # Strip leading '1.', '1)', '-', '*', etc.
 cleaned = line.lstrip('0123456789.- *').strip()
 if cleaned and len(cleaned) > 5:
 steps.append(cleaned)
 return steps

print('Phase 1 (Planner) ready.')
```

The Planner's system prompt instructs it to think like a strategic consultant: given a vague goal, identify the critical questions that will determine success or failure. The output is a numbered list of 4-5 questions – no preamble, no analysis, just the questions. This constrained output format makes it easy to parse programmatically and feed into the next phase. Try changing the business idea later and notice how the plan changes completely – an AI housing app generates questions about data sources and student demographics, while a meal kit service generates questions about supply chains and unit economics.

The Executor answers one research question at a time, but with an important twist: it receives all prior answers as accumulated context. This means each successive answer can build on the previous ones, creating a chain of reasoning rather than a collection of isolated answers. The second question's answer might reference a data source mentioned in the first answer; the fourth might synthesize patterns from the first three. This context accumulation is what makes the execution phase feel coherent rather than fragmented.

The Synthesizer plays the role of a senior executive who must make a decision based on the research. Its system prompt asks for a clear verdict – go, no-go, or conditional – followed by three bullet points of key reasoning. This forces the model to commit to a position rather than hedging endlessly, which is a common failure mode when LLMs are asked for recommendations without explicit structural constraints.

The Critic is deliberately adversarial. Its system prompt instructs it to be “skeptical” and to identify what is “weak or missing.” This is by design: the Synthesizer is optimistic (it just made a recommendation and wants it to be right), so the Critic provides the necessary counterbalance. In the example output you will see below, the Critic often assigns a much lower confidence score than you might expect from reading the recommendation alone, because it focuses on risks, missing evidence, and competitive threats that the Synthesizer glossed over.

```
Phase 2: Executor -- answers ONE research question at a time
EXECUTOR_SYSTEM = (
 'You are a business analyst. Answer the question concisely (2-4 sentences) '
 'with concrete reasoning. Use the prior context if helpful, but stay focused '
 'on the current question.'
)

def execute_step(question, prior_context):
 '''Phase 2: answer one research question with access to prior answers.'''
 prompt = 'Prior context:' + chr(10) + prior_context + chr(10) + chr(10) +
 'Question: ' + question
 response = llm.generate(prompt, system_prompt=EXECUTOR_SYSTEM)
 return response.text

print('Phase 2 (Executor) ready.')
```

```
Phase 3: Synthesizer -- combines findings into a recommendation
SYNTHESIZER_SYSTEM = (
 'You are a senior executive. Given research findings, synthesize them into a '
 'clear, decisive recommendation. State the recommendation (go / no-go / '
 'conditional), '
 'followed by 3 bullet points of key reasoning.'
)

def synthesize(goal, research_results):
 '''Phase 3: combine all research answers into a final recommendation.'''
 findings = chr(10).join([f'Q: {q}' + chr(10) + f'A: {a}' for q, a in
research_results])
 prompt = (
 'Goal: ' + goal + chr(10) + chr(10)
 + 'Research findings:' + chr(10) + findings + chr(10) + chr(10)
 + 'Synthesize into a final recommendation.'
)
 response = llm.generate(prompt, system_prompt=SYNTHESIZER_SYSTEM)
 return response.text

print('Phase 3 (Synthesizer) ready.')
```

```
Phase 4: Critic -- self-reflects on the recommendation
CRITIC_SYSTEM = (
 'You are a skeptical business critic. Given a recommendation, identify '
 'what is weak or missing, what is strong, and assign a confidence level '
 'from 1 (very weak) to 10 (very strong). Be concise.'
)

def reflect(recommendation):
 '''Phase 4: critique the recommendation and assign confidence.'''
 prompt = 'Recommendation to critique:' + chr(10) + recommendation
 response = llm.generate(prompt, system_prompt=CRITIC_SYSTEM)
 return response.text
```



```
print('Phase 4 (Critic) ready.')
```

### 17.1.3 Inspect the Full Trace

The `result` dictionary holds everything the agent produced across all four phases, so you can examine each phase individually. This is useful for debugging (did the planner ask the right questions? did the executor miss something?) and for teaching (trace how the agent’s reasoning flowed from plan to recommendation to critique).

The trace output reveals something important about autonomous workflows: the quality of the final recommendation depends heavily on the quality of the plan. If the planner asks shallow questions, the executor produces shallow answers, and the synthesizer builds on a weak foundation. This is why the planner’s system prompt matters so much – it sets the ceiling for everything that follows.

### 17.1.4 Run the Agent

Edit the `business_idea` string below and run the cell. The agent will autonomously plan its research, execute each step, synthesize a recommendation, and self-critique – with no further input from you. The verbose output prints each phase as it completes, so you can watch the agent “think out loud” and trace how the plan’s questions get answered, how those answers feed into the recommendation, and how the critic pokes holes in the final output.

Try different ideas to see how the plan changes. The agent generates a completely different research plan for a fintech startup vs. a coffee shop vs. a SaaS tool – which is exactly what an autonomous agent should do. The plan is not hardcoded; it emerges from the LLM’s reasoning about what matters for each specific goal.

```
def run_autonomous_workflow(goal, verbose=True):
 '''The full autonomous workflow: Plan -> Execute -> Synthesize -> Reflect.'''
 if verbose:
 print('=' * 70)
 print('GOAL:', goal)
 print('=' * 70)

 # --- Phase 1: Plan ---
 if verbose:
 print(chr(10) + '[PHASE 1 - PLANNING]')
 steps = plan(goal)
 if verbose:
 print(f'Agent decided to research {len(steps)} questions:')
 for i, s in enumerate(steps, 1):
 print(f' {i}. {s}')

 # --- Phase 2: Execute each step ---
 if verbose:
 print(chr(10) + '[PHASE 2 - EXECUTION]')
 results = []
 prior_context = ''
 for i, step in enumerate(steps, 1):
 if verbose:
 print(chr(10) + f'Step {i}: {step}')
 answer = execute_step(step, prior_context)
 results.append((step, answer))
 prior_context += chr(10) + f'Q: {step}' + chr(10) + f'A: {answer}'
 if verbose:
 print(f' -> {answer}')

 # --- Phase 3: Synthesize ---
 if verbose:
 print(chr(10) + '[PHASE 3 - SYNTHESIS]')
 recommendation = synthesize(goal, results)
```

```

 if verbose:
 print(recommendation)

 # --- Phase 4: Reflect ---
 if verbose:
 print(chr(10) + '[PHASE 4 - SELF-CRITIQUE]')
 critique = reflect(recommendation)
 if verbose:
 print(critique)

 return {
 'goal': goal,
 'plan': steps,
 'research': results,
 'recommendation': recommendation,
 'critique': critique,
 }

print('Autonomous workflow function ready.')

```

```

business_idea = 'An AI-powered app that helps college students find affordable off-
campus housing by predicting rent trends.'

business_idea = 'A subscription meal kit service for busy professionals that uses
local ingredients.'
business_idea = 'A mobile game that teaches kids basic accounting through story-
based puzzles.'

result = run_autonomous_workflow(business_idea)

```

### 17.1.5 Inspect the Full Trace

The `result` dict holds everything the agent did, so you can examine each phase individually. This is useful for debugging or for teaching how the agent’s reasoning flowed from plan to final recommendation.

### 17.1.6 Key Takeaways

**What makes this workflow “autonomous”:**

- You provide only a **high-level goal**, not a list of steps
- The agent generates its own **research plan** based on the goal
- Each step is executed with access to the **prior context** (results accumulate)
- The agent self-critiques its output, giving you a **confidence signal** for free

**What’s NOT happening (and why this example is simple):**

- No external tools (no web search, no database queries)
- No iteration: the agent doesn’t re-plan based on the critique
- No parallelism: execution is strictly sequential
- No human-in-the-loop: the workflow runs end-to-end without interruption

**Scaling this up in the real world:**

- Add **tools** so execution steps can fetch real data (web search, database, APIs)
- Add an **iteration loop**: if the critic’s confidence is low, feed the critique back to the planner to generate a better plan
- Use **LangGraph** to express this as a stateful graph with conditional edges (see `LangGraph_demo.ipynb` for the foundation)
- Add **memory** so the agent can remember prior runs and improve over time

**Comparison with our other notebooks:**

| Notebook                                    | Who decides the steps?                                         |
|---------------------------------------------|----------------------------------------------------------------|
| LlamaIndex_RAG , LangChain_demo (RAG chain) | Humans — the pipeline is fixed                                 |
| LangChain_demo (agent with tools)           | LLM picks tools at each turn, but the goal is one question     |
| MultiAgent_DB                               | Humans — 5 specialized agents run in a fixed order             |
| SingleAgent_DB                              | Humans — one agent runs a fixed workflow                       |
| <b>This notebook</b>                        | <b>LLM decides the whole workflow from a one-sentence goal</b> |

### Exercises:

- Try a business idea the LLM is unlikely to have strong prior knowledge about (e.g., a niche B2B tool). Does the plan still look reasonable?
- Modify `run_autonomous_workflow` to loop: after the critic phase, if confidence < 7, re-run the plan with the critique appended to the goal.
- Add a fifth phase that outputs a one-page marketing pitch based on the recommendation.
- Replace the `PLANNER_SYSTEM` prompt with a different persona (e.g., “venture capitalist”) and see how the plan changes.

#### 17.1.7 Run the code

To run this notebook, copy the URL below into your browser’s address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

**Estimated run time:** ~3 minutes (7-8 LLM calls total)

[https://colab.research.google.com/github/KarAnalytics/code\\_demos/blob/main/AutonomousAgent\\_BusinessValidator.ipynb](https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/AutonomousAgent_BusinessValidator.ipynb)

## 17.2 Autonomous Agent: Business Idea Validator (Multi-Vendor Edition)

This notebook extends the base autonomous agent workflow by assigning a **different LLM vendor to each phase** of the reasoning process. Instead of routing all calls through a single shared cascade, each cognitive phase – planning, execution, synthesis, and critique – gets its own dedicated LLM provider.

The multi-vendor approach demonstrates an important principle: different LLMs have different strengths. A model that excels at strategic decomposition (planning) may not be the best at fast factual retrieval (execution) or nuanced summarization (synthesis). By pinning each phase to a specific vendor, you can optimize the overall workflow quality while also reducing dependence on any single provider's availability or rate limits.

### 17.2.0.1 Default phase-to-vendor assignments

| Phase             | Role              | Default vendor | Rationale                          |
|-------------------|-------------------|----------------|------------------------------------|
| <b>Plan</b>       | Strategic planner | Gemini         | Strong at structured decomposition |
| <b>Execute</b>    | Business analyst  | Groq           | Fast, factual, low latency         |
| <b>Synthesize</b> | Senior executive  | Cohere         | Strong summarisation / reasoning   |
| <b>Reflect</b>    | Skeptical critic  | HuggingFace    | Independent perspective            |

#### 17.2.0.1.1 You're free to change these

Each phase is wired in the **Imports and Setup** cell below with a single line: `pin('VendorName')`. Three things you can do:

- **Swap any vendor for another.** Change `pin('Gemini')` to `pin('OpenAI')`, `pin('Groq')`, or any other provider. The available vendor names print at startup so you can see what's configured.
- **Use the same vendor for multiple (or all) phases.** Repeating vendors is fine – e.g. pin every phase to the one provider whose API key you have set. The four-phase architecture still works; you've just chosen to run all of them through the same backend.
- **Add fallback per phase.** Pass multiple names: `pin('Gemini', 'Groq')` tries Gemini first and falls back to Groq if Gemini is unavailable. This is the production pattern – a preferred vendor with a reliable backup.

**Important:** by default, `pin('SingleVendor')` has **no fallback**. If that one vendor is rate-limited or unavailable, the whole workflow stops with `RuntimeError: All providers exhausted`. The fix is to add a backup vendor to the `pin()` call.

If you have not yet seen the single-vendor version (`autonomous_agent.ipynb`), review that first – the workflow logic is identical, and this notebook focuses on the multi-vendor routing layer.

```
1. INSTALL DEPENDENCIES
!pip install -q git+https://github.com/KarAnalytics/llm_cascade.git
```

**IGNORE THE ABOVE ERRORS AND PROCEED** – they are specific to Google Colab's preinstalled packages and the code will still execute correctly.

### 17.2.1 Autonomous Agent: Business Idea Validator (Multi-Vendor)

This notebook extends the base autonomous workflow by assigning a **different free-tier LLM vendor to each phase**. Instead of one shared cascade instance, each agent phase has its own `LLMCascade` instance pinned to a specific provider – demonstrating that autonomous workflows can mix and match models from different vendors within the same run.

**Phase arrow.r Vendor mapping (defaults):**

| Phase      | Role              | Default vendor | Why                                |
|------------|-------------------|----------------|------------------------------------|
| Plan       | Strategic planner | Gemini         | Strong at structured decomposition |
| Execute    | Business analyst  | Groq           | Fast, factual, low latency         |
| Synthesize | Senior executive  | Cohere         | Strong summarisation / reasoning   |

| Phase   | Role             | Default vendor | Why                     |
|---------|------------------|----------------|-------------------------|
| Reflect | Skeptical critic | HuggingFace    | Independent perspective |

Each vendor is loaded from your existing API keys — no extra setup required beyond what `llm_cascade` already supports.

#### 17.2.1.0.1 How is this different from `AutonomousAgent_BusinessValidator.ipynb` ?

|                    | Single-vendor autonomous         | Multi-vendor autonomous (this)                                    |
|--------------------|----------------------------------|-------------------------------------------------------------------|
| LLM instances      | 1 shared cascade                 | <b>4 separate per-phase instances</b>                             |
| Provider per phase | Whichever responds first         | <b>Pinned to a specific vendor</b>                                |
| Fallback           | Yes — across all configured keys | <b>Optional — can keep cascade fallback per phase</b>             |
| Teaching point     | Autonomous workflow basics       | <b>Different LLMs can specialise in different cognitive tasks</b> |

#### 17.2.1.1 Imports and Setup

Each phase gets its own `LLMCascade` instance pinned to a specific provider. The `pin()` helper accepts one or more provider names; if you pass multiple names the phase still gets cascade-style fallback but only across those named vendors.

Edit the vendor names passed to `pin()` to swap models — all available names are listed by `get_cascade()` at startup.

```
from llm_cascade import get_cascade
from llm_cascade.providers import LLMCascade, PROVIDERS

Show all configured providers so students know what's available
print('All configured providers:')
_ = get_cascade()

def pin(*provider_names):
 """Return an LLMCascade restricted to the named provider(s).
 Falls back across names in the order given, just like the full cascade.
 If a named provider has no API key it is silently skipped."""
 selected = [p for p in PROVIDERS if p['name'] in provider_names]
 if not selected:
 raise ValueError(f'None of {provider_names} found in PROVIDERS. '
 f'Available: {[p["name"] for p in PROVIDERS]}')
 return LLMCascade(providers=selected, verbose=True)

--- Assign one vendor per phase ---
Change any of these to any provider name shown above.
You can also pass multiple names for per-phase fallback, e.g. pin('Gemini',
'Groq')
planner_llm = pin('Gemini') # strategic decomposition
executor_llm = pin('Groq') # fast, factual analyst
synthesizer_llm = pin('Cohere') # summarisation & recommendation
reflector_llm = pin('HuggingFace') # independent skeptic

print()
print('Phase → vendor assignments:')
for label, instance in [('Planner', planner_llm), ('Executor', executor_llm),
 ('Synthesizer', synthesizer_llm), ('Reflector',
reflector_llm)]:
 names = ', '.join(p['name'] for p in instance.available) or 'NONE (key
```

```
missing!)'
 print(f' {label:<12} -> {names}')
```

### 17.2.1.2 The Four Phases of the Autonomous Workflow

The logic is identical to the single-vendor version. The only change is that each function calls its own dedicated LLM instance instead of the shared `llm` object.

1. **Plan** — `planner_llm` (Gemini) generates 4–5 research questions
2. **Execute** — `executor_llm` (Groq) answers each question, accumulating context
3. **Synthesize** — `synthesizer_llm` (Cohere) combines findings into a recommendation
4. **Reflect** — `reflector_llm` (HuggingFace) critiques the recommendation and scores it

```
Phase 1: Planner -- decides WHAT needs to be researched
PLANNER_SYSTEM = (
 'You are a strategic business planner. Given a high-level goal, break it down '
 'into the 4-5 most critical questions that must be answered to achieve it. '
 'Output ONLY a numbered list of questions. No preamble, no conclusion.'
)

def plan(goal):
 '''Phase 1: generate a research plan as a list of questions (uses
 planner_llm).'''
 prompt = f'Goal: {goal}' + chr(10) + chr(10) + 'List the 4-5 most critical
 questions to answer.'
 response = planner_llm.generate(prompt, system_prompt=PLANNER_SYSTEM)

 # Parse the numbered list into a Python list of strings
 steps = []
 for line in response.text.split(chr(10)):
 line = line.strip()
 if not line:
 continue
 # Strip leading '1.', '1)', '-', '*', etc.
 cleaned = line.lstrip('0123456789.)- *').strip()
 if cleaned and len(cleaned) > 5:
 steps.append(cleaned)
 return steps

print('Phase 1 (Planner) ready.')
```

```
Phase 2: Executor -- answers ONE research question at a time
EXECUTOR_SYSTEM = (
 'You are a business analyst. Answer the question concisely (2-4 sentences) '
 'with concrete reasoning. Use the prior context if helpful, but stay focused '
 'on the current question.'
)

def execute_step(question, prior_context):
 '''Phase 2: answer one research question with access to prior answers (uses
 executor_llm).'''
 prompt = 'Prior context:' + chr(10) + prior_context + chr(10) + chr(10) +
 'Question: ' + question
 response = executor_llm.generate(prompt, system_prompt=EXECUTOR_SYSTEM)
 return response.text

print('Phase 2 (Executor) ready.')
```

```

Phase 3: Synthesizer -- combines findings into a recommendation
SYNTHESIZER_SYSTEM = (
 'You are a senior executive. Given research findings, synthesize them into a '
 'clear, decisive recommendation. State the recommendation (go / no-go / '
 'conditional), '
 'followed by 3 bullet points of key reasoning.'
)

def synthesize(goal, research_results):
 '''Phase 3: combine all research answers into a final recommendation (uses
 synthesizer_llm).'''
 findings = chr(10).join([f'Q: {q}' + chr(10) + f'A: {a}' for q, a in
 research_results])
 prompt = (
 'Goal: ' + goal + chr(10) + chr(10)
 + 'Research findings:' + chr(10) + findings + chr(10) + chr(10)
 + 'Synthesize into a final recommendation.'
)
 response = synthesizer_llm.generate(prompt, system_prompt=SYNTHESIZER_SYSTEM)
 return response.text

print('Phase 3 (Synthesizer) ready.')

```

```

Phase 4: Critic -- self-reflects on the recommendation
CRITIC_SYSTEM = (
 'You are a skeptical business critic. Given a recommendation, identify '
 'what is weak or missing, what is strong, and assign a confidence level '
 'from 1 (very weak) to 10 (very strong). Be concise.'
)

def reflect(recommendation):
 '''Phase 4: critique the recommendation and assign confidence (uses
 reflector_llm).'''
 prompt = 'Recommendation to critique:' + chr(10) + recommendation
 response = reflector_llm.generate(prompt, system_prompt=CRITIC_SYSTEM)
 return response.text

print('Phase 4 (Critic) ready.')

```

### 17.2.1.3 The Autonomous Loop

Same logic as the single-vendor notebook, but now each phase label also shows **which vendor responded**. Watch the [Response from ...] lines to see the four different LLMs handing off to each other.

Total LLM calls per run: **1 (plan) + 4–5 (execute) + 1 (synthesize) + 1 (reflect) = 7–8 calls**, each routed to a different provider.

```

def run_autonomous_workflow(goal, verbose=True):
 '''The full autonomous workflow: Plan -> Execute -> Synthesize -> Reflect.
 Each phase uses a different LLM vendor.'''
 if verbose:
 print('=' * 70)
 print('GOAL:', goal)
 print('=' * 70)

 # --- Phase 1: Plan (planner_llm) ---
 if verbose:
 print(chr(10) + '[PHASE 1 - PLANNING | vendor: ' +

```

```

 ', '.join(p['name'] for p in planner_llm.available) + ''])
steps = plan(goal)
if verbose:
 print(f'Agent decided to research {len(steps)} questions:')
 for i, s in enumerate(steps, 1):
 print(f' {i}. {s}')

--- Phase 2: Execute each step (executor_llm) ---
if verbose:
 print(chr(10) + '[PHASE 2 - EXECUTION | vendor: ' +
 ', '.join(p['name'] for p in executor_llm.available) + ''])
results = []
prior_context = ''
for i, step in enumerate(steps, 1):
 if verbose:
 print(chr(10) + f'Step {i}: {step}')
 answer = execute_step(step, prior_context)
 results.append((step, answer))
 prior_context += chr(10) + f'Q: {step}' + chr(10) + f'A: {answer}'
 if verbose:
 print(f' -> {answer}')

--- Phase 3: Synthesize (synthesizer_llm) ---
if verbose:
 print(chr(10) + '[PHASE 3 - SYNTHESIS | vendor: ' +
 ', '.join(p['name'] for p in synthesizer_llm.available) + ''])
recommendation = synthesize(goal, results)
if verbose:
 print(recommendation)

--- Phase 4: Reflect (reflector_llm) ---
if verbose:
 print(chr(10) + '[PHASE 4 - SELF-CRITIQUE | vendor: ' +
 ', '.join(p['name'] for p in reflector_llm.available) + ''])
critique = reflect(recommendation)
if verbose:
 print(critique)

return {
 'goal': goal,
 'plan': steps,
 'research': results,
 'recommendation': recommendation,
 'critique': critique,
}

print('Autonomous multi-vendor workflow function ready.')

```

#### 17.2.1.4 Run the Agent

Edit `business_idea` and run the cell. Watch the vendor tag on each phase header to confirm that different LLMs are handling different stages of the reasoning chain.

```

business_idea = 'An AI-powered app that helps college students find affordable off-
campus housing by predicting rent trends.'

business_idea = 'A subscription meal kit service for busy professionals that uses
local ingredients.'
business_idea = 'A mobile game that teaches kids basic accounting through story-
based puzzles.'

result = run_autonomous_workflow(business_idea)

```



### 17.2.1.5 Inspect the Full Trace

The `result` dict holds everything the agent produced. Each answer in `result['research']` was generated by `executor_llm`; the recommendation by `synthesizer_llm`; and the critique by `reflector_llm` — three different LLMs contributing to one coherent report.

```
print('=' * 70)
print('FULL TRACE')
print('=' * 70)
print(chr(10) + 'GOAL:', result['goal'])
print(chr(10) + 'PLAN (what the agent decided to research):')
for i, step in enumerate(result['plan'], 1):
 print(f' {i}. {step}')
print(chr(10) + 'FINAL RECOMMENDATION:')
print(result['recommendation'])
print(chr(10) + 'SELF-CRITIQUE:')
print(result['critique'])
```

### 17.2.1.6 Key Takeaways

What's new vs. the single-vendor version:

- Each phase is served by a **different LLM from a different vendor** — they communicate by passing text, not by sharing state
- The `pin()` helper lets you **assign any provider from PROVIDERS** to any phase in one line
- You can pass multiple names to `pin()` for per-phase fallback: `pin('Gemini', 'Groq')` tries Gemini first, then Groq — useful if one key is missing
- The vendor name appears in each phase header so you can verify the routing in the output

Why use different vendors per phase?

- Some models excel at **decomposition** (planning), others at **fast factual retrieval** (execution), others at **synthesis** or **critical evaluation**
- It reduces dependence on a single provider's quota or availability
- It's the same principle behind multi-agent frameworks like CrewAI or AutoGen, but without the orchestration overhead

Exercises:

- Swap the vendor assignments and compare the quality of the output — does a different planner generate better research questions?
- Pass two vendors to `pin()` (e.g., `pin('Cohere', 'Groq')`) and pull the API key for the first one to test per-phase fallback
- Add a fifth phase (e.g., a marketing writer) using whichever vendor has remaining quota
- Modify `run_autonomous_workflow` so that if the critic's confidence score is below 7, the critique is fed back to `planner_llm` for a revised plan

---

### 17.2.1.7 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually — the viewer may have truncated the link at a line break.)*

**Estimated run time:** ~3 minutes

```
https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/AutonomousAgent_multivendor.ipynb
```



## Chapter 18

# Tool-Using Agents and MCP

This chapter brings together tools, agents, and the **Model Context Protocol (MCP)** — the standard for exposing tools to LLMs as reusable services.

We build a KU Parking Assistant that:

- Uses **real deterministic tools** (Haversine distance, fuzzy building matching, Google Maps URLs)
- The LLM decides *when* to call which tool based on the question
- The same tools are deployed to a **Google Cloud Run** container as an MCP server
- Both a Python notebook and a Dify workflow can call the same tools

Two notebooks contrast:

- **Inline tools** — tools defined as Python functions in the notebook
- **MCP tools** — tools accessed over HTTPS from the Cloud Run endpoint

## 18.1 KU Parking Assistant: A Simple Tool-Using Agent

Every agent notebook you have seen so far in this course relies on the LLM to do all the heavy lifting – reasoning, planning, and generating answers from its own knowledge. But there are tasks where pure LLM reasoning is the wrong tool for the job. Calculating the distance between two GPS coordinates, for example, requires precise trigonometry. An LLM might approximate the answer, but it will sometimes get it wrong in ways that matter (ranking a lot as “closest” when it is actually third-closest). A Python function using the **Haversine formula** will get it right every time, in microseconds, with no hallucination risk.

This notebook builds a small agent that combines the best of both worlds. The LLM handles what it is good at – understanding natural language (“Where can I park near the business school?”), fuzzy matching building names, and formatting a helpful response. Python functions handle what they are good at – looking up coordinates in a dictionary, computing great-circle distances, sorting results, and constructing Google Maps URLs. The LLM decides when to call which tool, but it never does the math itself.

What makes this different from the other agent notebooks:

- We use real deterministic tools (distance math, data lookups) instead of a pure text-in/text-out workflow
- The LLM decides when to call which tool, but it does not do the math itself – it delegates to Python
- This is a pure workflow, not a chatbot: one input produces one structured output

**Teaching goal:** Show the difference between fuzzy LLM reasoning (good for language understanding: “business school” maps to “Capitol Federal Hall”) and precise tool execution (good for deterministic tasks: distance math, sorting, data lookup). A well-designed agent lets each side do what it is best at.

Note on data: The real KU parking map is at <https://parking.ku.edu/sites/parking/files/documents/parkingmap.pdf>. For this demo we use a small illustrative dataset with approximate coordinates so the notebook runs standalone.

This notebook picks up where the agents in Chapters 16 and 17 left off – the single-agent, multi-agent, and autonomous notebooks all reasoned over text alone, while here the LLM delegates the precise work to real Python tools it chooses to call.

```
!pip install -q git+https://github.com/KarAnalytics/llm_cascade.git
```

### 18.1.1 Imports and LLM Setup

We use `llm_cascade` for automatic LLM provider fallback, and Python’s built-in `math` module for the distance calculations. No external APIs needed.

```
import math
import json as _json
import re as _re

from llm_cascade import get_cascade

llm = get_cascade()
```

### 18.1.2 Data: KU Buildings and Parking Lots

We hardcode two small datasets that represent the physical world the agent needs to reason about. In a real deployment, these would come from a database, a GIS system, or a parsed PDF. For this classroom demo, approximate coordinates are sufficient to demonstrate the architecture.

KU Buildings are the destinations students might ask about. The dictionary maps human-readable names (with common aliases in parentheses) to latitude/longitude tuples. This naming convention is deliberate: when a student asks about “the business school,” our fuzzy matcher will find “Capitol Federal Hall (Business School)” because the alias appears in the dictionary key.

KU Parking Lots include a color code indicating who can park there. Understanding these colors is essential context for the agent’s final answer – it is not helpful to recommend a Yellow (faculty-only) lot to a student with a Blue permit. The colors are:

- Blue – student commuter permit
- Yellow – faculty / staff permit
- Gold – premium permit (close to central campus)
- Red – restricted / reserved
- Visitor – open to visitors, usually metered or short-term
- Park & Ride – free remote lot with shuttle service

```
--- KU Buildings (approximate coordinates for illustration) ---
KU_BUILDINGS = {
 'Capitol Federal Hall (Business School)': (38.953505, -95.249740),
 'Kansas Union': (38.959200, -95.243500),
 'Allen Fieldhouse': (38.954306, -95.252394),
 'Strong Hall': (38.958542, -95.247614),
 'Watson Library': (38.956621, -95.244787),
 'Anschutz Library': (38.957323, -95.249742),
 'Dyche Hall': (38.958610, -95.243890),
 'Fraser Hall': (38.957160, -95.243590),
 'Lied Center': (38.954940, -95.262890),
 'Memorial Stadium': (38.963330, -95.246390),
 'Wescoe Hall': (38.957430, -95.247830),
 'Spencer Museum of Art': (38.959634, -95.244569),
 'Ambler Student Recreation Fitness Center': (38.952512, -95.247929),
}

--- Informal names, nicknames, and school/department associations people
actually type. Any of these should resolve to the canonical building name. ---
BUILDING_ALIASES = {
 'Capitol Federal Hall (Business School)': [
 'business school', 'b-school', 'bschool', 'biz school',
 'school of business', 'ku business school', 'ku business',
 'capitol federal', 'capitol federal hall', 'capfed', 'cap fed',
],
 'Kansas Union': [
 'union', 'student union', 'ku union', 'the union', 'ku student union',
 'memorial union', 'bookstore',
],
 'Allen Fieldhouse': [
 'allen', 'fieldhouse', 'the phog', 'phog', 'phog allen',
 'basketball stadium', 'phog stadium', 'basketball arena',
 'ku basketball', 'basketball', 'jayhawk basketball',
],
 'Strong Hall': [
 'strong', 'admin building', 'administration', 'admin',
 'chancellor', 'chancellor's office',
],
 'Watson Library': [
 'watson', 'watson lib', 'main library', 'research library',
 'ku library',
],
 'Anschutz Library': [
 'anschultz', 'anschultz lib', 'science library', 'stem library',
],
 'Dyche Hall': [
 'dyche', 'natural history museum', 'nhm',
 'biodiversity institute', 'museum of natural history',
],
 'Fraser Hall': [
 'fraser', 'psychology building', 'psych building',
],
 'Lied Center': [
 'lied', 'performing arts', 'performing arts center',
 'concert hall', 'theater',
],
}
```

```

'Memorial Stadium': [
 'stadium', 'football stadium', 'memorial', 'kivisto field',
 'ku football', 'football arena', 'jayhawk football',
],
'Wescoe Hall': [
 'wescoe', 'wescoe beach', 'liberal arts building',
 'college of liberal arts', 'humanities building',
],
'Spencer Museum of Art': [
 'spencer', 'spencer museum', 'art museum', 'museum of art',
],
'Ambler Student Recreation Fitness Center': [
 'ambler', 'ambler gym', 'ambler fitness', 'ambler fitness center',
 'ambler rec', 'ambler rec center', 'ambler recreation',
 'ambler srfc', 'srfc',
 'student gym', 'student rec', 'student rec center',
 'recreation center', 'fitness center', 'rec center',
 'campus gym', 'campus rec', 'ku gym', 'the rec', 'gym',
],
}

--- KU Parking Lots (approximate coordinates for illustration) ---
KU_PARKING_LOTS = [
 {'lot': 'Lot 3' , 'color': 'Blue' , 'lat': 38.958942, 'lng':
-95.247614},
 {'lot': 'Lot 5' , 'color': 'Yellow' , 'lat': 38.95861, 'lng':
-95.24441},
 {'lot': 'Lot 10' , 'color': 'Blue' , 'lat': 38.956221, 'lng':
-95.244787},
 {'lot': 'Lot 14' , 'color': 'Yellow' , 'lat': 38.95716, 'lng':
-95.24307},
 {'lot': 'Lot 16' , 'color': 'Visitor' , 'lat': 38.9592, 'lng':
-95.24298},
 {'lot': 'Lot 18' , 'color': 'Yellow' , 'lat': 38.95703, 'lng':
-95.24783},
 {'lot': 'Lot 60' , 'color': 'Blue' , 'lat': 38.96333, 'lng':
-95.24691},
 {'lot': 'Lot 70' , 'color': 'Blue' , 'lat': 38.953906, 'lng':
-95.252394},
 {'lot': 'Lot 71' , 'color': 'Yellow' , 'lat': 38.953666, 'lng':
-95.252394},
 {'lot': 'Lot 90' , 'color': 'Gold' , 'lat': 38.952512, 'lng':
-95.247929},
 {'lot': 'Lot 91' , 'color': 'Gold' , 'lat': 38.959634, 'lng':
-95.244569},
 {'lot': 'Lot 118' , 'color': 'Gold' , 'lat': 38.953505, 'lng':
-95.24974},
 {'lot': 'Lot 300A-D' , 'color': 'Gold' , 'lat': 38.95494, 'lng':
-95.26237},
 {'lot': 'Lot 300E-G' , 'color': 'Gold' , 'lat': 38.95494, 'lng':
-95.26289},
 {'lot': 'Allen Fieldhouse Garage', 'color': 'Gold' , 'lat': 38.954306, 'lng':
-95.252394},
]

print(f'Loaded {len(KU_BUILDINGS)} buildings, '
 f'{sum(len(v) for v in BUILDING_ALIASES.values())} aliases, '
 f'and {len(KU_PARKING_LOTS)} parking lots.')

```

### 18.1.3 The Tools (plain Python functions)

Three simple tools the agent can call. Notice there is no LLM inside these functions – they are pure deterministic Python. The LLM will decide *when* to call them based on the user’s question, but every computation inside them produces an exact, repeatable result.

| Tool                                                 | What it does                                                | Why it exists                                                    |
|------------------------------------------------------|-------------------------------------------------------------|------------------------------------------------------------------|
| <code>list_ku_buildings()</code>                     | Returns the list of available buildings                     | So the agent can show options when the user’s query is ambiguous |
| <code>find_parking_near_building(name, top_k)</code> | Finds the top-K nearest parking lots within a 2-mile radius | The core function – distance math, sorting, and URL generation   |
| <code>get_parking_colors_legend()</code>             | Explains what each parking color means                      | So the agent can annotate its answer with permit-type context    |

The distance calculation uses the **Haversine formula**, which computes the great-circle distance between two latitude/longitude points on a sphere. For the short distances on a university campus (typically under 1 mile), the Haversine result is accurate to within a few feet – far better than asking an LLM to eyeball coordinate differences. The 2-mile radius filter is a practical constraint that keeps the results useful: a parking lot 5 miles away is technically “near” the campus, but no student would walk there.

#### 18.1.3.1 Approximate building matching

Real users rarely type the full canonical building name. They write “Ambler gym”, “student gym”, “the phog”, “basketball stadium”, or “school of business”. Our `_match_building` helper handles this by combining three strategies:

1. Alias table. Each canonical building name has a list of common nicknames and school/department associations (e.g., Allen Fieldhouse gets `phog`, `phog stadium`, `basketball arena`, `ku basketball`; Ambler SRFC gets `ambler gym`, `student gym`, `fitness center`, `gym`).
2. Substring matching in both directions, against the canonical name and every alias.
3. Token-overlap scoring (F1 over tokens, with a small stopword list) for phrases that don’t substring-match but share enough words with the building or an alias.

The matcher tries exact arrow.r substring arrow.r token-overlap in that order and requires a token overlap of at least 0.5 before it commits to an approximate match. That threshold prevents random words from latching onto a building. The alias table lives right next to the coordinates, so adding a new nickname is a one-line edit. The Google Maps URLs are constructed by embedding the latitude and longitude into a standard `maps?q=LAT,LNG` template, which produces a clickable pin link that works on any device.

```
def _haversine_miles(lat1, lng1, lat2, lng2):
 '''Great-circle distance between two lat/lng points, in miles.'''
 R = 3958.8 # Earth radius in miles
 phi1, phi2 = math.radians(lat1), math.radians(lat2)
 dphi = math.radians(lat2 - lat1)
 dlam = math.radians(lng2 - lng1)
 a = math.sin(dphi / 2) ** 2 + math.cos(phi1) * math.cos(phi2) * math.sin(dlam /
2) ** 2
 return 2 * R * math.asin(math.sqrt(a))

def _google_maps_pin(lat, lng):
 '''Return a clickable Google Maps pin URL for a coordinate.'''
 return f'https://www.google.com/maps?q={lat},{lng}'

---- Approximate building matcher ----
Small stopword list -- filler words that should not drive a token match.
_STOPWORDS = {
```

```

 'the', 'a', 'an', 'of', 'at', 'on', 'in', 'and', 'or', 'to',
 'ku', 'kansas', 'university', 'hall', 'building', 'bldg',
}

def _tokenize(s):
 return [t for t in _re.split(r'^a-z0-9+', s.lower()) if t and t not in
_STOPWORDS]

def _token_score(query, candidate):
 '''F1-style token overlap score between two strings (0 to 1).'''
 q_tokens = set(_tokenize(query))
 c_tokens = set(_tokenize(candidate))
 if not q_tokens or not c_tokens:
 return 0.0
 overlap = len(q_tokens & c_tokens)
 if overlap == 0:
 return 0.0
 precision = overlap / len(q_tokens)
 recall = overlap / len(c_tokens)
 return (2 * precision * recall) / (precision + recall)

def _match_building(query):
 '''Approximate building match. Accepts informal names like "Ambler gym",
 "student gym", "the phog", "business school" and resolves them to the
 canonical building name. Order of preference:
 1. Exact match on canonical name or alias
 2. Substring match (query in name, or query in alias, or alias in query)
 3. Token-overlap scoring across canonical names and aliases (>= 0.5)
 ...
 q = query.lower().strip()
 if not q:
 return None

 # 1a. Exact canonical name
 for name in KU_BUILDINGS:
 if name.lower() == q:
 return name
 # 1b. Exact alias
 for name, aliases in BUILDING_ALIASES.items():
 for alias in aliases:
 if alias.lower() == q:
 return name

 # 2a. Substring against canonical name (either direction)
 for name in KU_BUILDINGS:
 nlow = name.lower()
 if q in nlow or nlow in q:
 return name
 # 2b. Reverse substring on the non-parenthetical short form
 for name in KU_BUILDINGS:
 short_key = name.split('(')[0].strip().lower()
 if short_key and (short_key in q or q in short_key):
 return name
 # 2c. Substring either direction against any alias
 for name, aliases in BUILDING_ALIASES.items():
 for alias in aliases:
 al = alias.lower()
 if al in q or q in al:
 return name

```



```

3. Token-overlap scoring across canonical name + all aliases
best_name, best_score = None, 0.0
for name in KU_BUILDINGS:
 for cand in [name] + BUILDING_ALIASES.get(name, []):
 score = _token_score(q, cand)
 if score > best_score:
 best_score = score
 best_name = name
Require meaningful overlap so random words do not latch onto a building.
return best_name if best_score >= 0.5 else None

---- Tools the agent can call ----

def list_ku_buildings(_=''):
 '''Return the list of KU buildings available for parking lookup.'''
 return chr(10).join(f'- {name}' for name in KU_BUILDINGS.keys())

def find_parking_near_building(building_name, top_k=5, max_distance_miles=2.0):
 '''Find nearest parking lots within max_distance_miles of the building, top-K
 results.'''
 matched = _match_building(building_name)
 if matched is None:
 return (
 f'No KU building matches "{building_name}". '
 f'Use list_ku_buildings to see available options.'
)

 target_lat, target_lng = KU_BUILDINGS[matched]

 # Compute distance to every lot and sort
 ranked = []
 for lot in KU_PARKING_LOTS:
 dist = _haversine_miles(target_lat, target_lng, lot['lat'], lot['lng'])
 ranked.append({
 'lot': lot['lot'],
 'color': lot['color'],
 'distance_miles': round(dist, 3),
 'walk_minutes': round(dist * 20), # ~3 mph walking speed
 'google_maps': _google_maps_pin(lot['lat'], lot['lng']),
 })

 ranked.sort(key=lambda r: r['distance_miles'])

 # Filter to lots within max_distance_miles of the destination
 within_range = [r for r in ranked if r['distance_miles'] <= max_distance_miles]

 if not within_range:
 return (
 f'No parking lots within {max_distance_miles} miles of {matched}. '
 f'Try a different building or increase the radius.'
)

 header = (
 f'Parking lots within {max_distance_miles} miles of {matched} '
 f'(top {top_k}, ranked by distance):' + chr(10)
)
 lines = [header]
 for i, r in enumerate(within_range[:top_k], start=1):
 lines.append(
 f" {i}. {r['lot']} ({r['color']}) - "
 f"{r['distance_miles']} mi ({r['walk_minutes']} min walk) "

```

```

 f"-> {r['google_maps']}"
)
 return chr(10).join(lines)

def get_parking_colors_legend(_=''):
 '''Explain what each parking color means at KU.'''
 return (
 'KU parking color legend:' + chr(10)
 + '- Blue: student commuter permit' + chr(10)
 + '- Yellow: faculty/staff permit' + chr(10)
 + '- Gold: premium permit (close to central campus)' + chr(10)
 + '- Red: restricted/reserved' + chr(10)
 + '- Visitor: open to visitors (often metered or short-term)' + chr(10)
 + '- Park & Ride: free remote lot with shuttle service'
)

Register tools for the agent
TOOLS = {
 'list_ku_buildings': list_ku_buildings,
 'find_parking_near_building': find_parking_near_building,
 'get_parking_colors_legend': get_parking_colors_legend,
}

Quick sanity check that the matcher handles informal names:
for _q in ['ambler gym', 'student gym', 'the phog', 'basketball stadium',
 'school of business', 'the union', 'fitness center']:
 print(f' {_q!r:35s} -> {_match_building(_q)}')

print()
print('Tools registered:', list(TOOLS.keys()))

```

#### 18.1.4 Test the Tools (no LLM yet)

Before wiring up the agent, let's call the tools directly to make sure they work. This is a critical debugging practice: if the raw tool output is wrong, no amount of prompt engineering on the agent side will fix it. By testing the tools in isolation, we can verify that the fuzzy matcher resolves "business school" correctly, that the distance calculations are sensible, and that the Google Maps URLs point to the right locations. Only after we trust the tools should we hand them to an LLM.

```

Call one tool directly
print(find_parking_near_building('business school', top_k=5))
print()
print(find_parking_near_building('Kansas Union', top_k=3))

```

#### 18.1.5 The Agent (LLM that calls the tools)

We use a prompt-based **ReAct** loop – no `bind_tools`, no special API features, works with any LLM. The system prompt describes the three available tools and tells the LLM to output a JSON action like `{"tool": "find_parking_near_building", "input": "business school"}` when it wants to call one. Our loop parses the JSON, executes the tool, feeds the result back into the conversation, and repeats until the LLM produces a natural-language final answer instead of a tool call.

This is a deliberately simple architecture. Framework-based agents (LangChain, CrewAI) add convenience methods and error handling, but the core pattern is exactly what you see here: prompt the LLM, check if it wants to call a tool, execute the tool if so, feed the result back, repeat. Understanding this loop demystifies what frameworks do under the hood and gives you the confidence to debug agent behavior when something goes wrong.

```

AGENT_SYSTEM = (
 'You are a helpful KU parking assistant. You have these tools:' + chr(10)
 + '- list_ku_buildings(): returns the list of KU buildings available.' + chr(10)
 + '- find_parking_near_building(name): returns the top 5 nearest parking lots to
a building, '
 + 'with color, distance, walk time, and Google Maps pin.' + chr(10)
 + '- get_parking_colors_legend(): explains the KU parking color codes.' +
chr(10) + chr(10)
 + 'To use a tool, reply with ONLY this JSON (nothing else):' + chr(10)
 + '{"tool": "tool_name", "input": "the input string"}' + chr(10) + chr(10)
 + 'After receiving a tool result, write a helpful final answer for the user that
includes '
 + 'the ranked parking list, color meanings, and clickable Google Maps links.' +
chr(10) + chr(10)
 + 'IMPORTANT -- find_parking_near_building does APPROXIMATE matching on the
building name, '
 + 'so pass the user\'s phrase through as-is instead of trying to guess the
canonical name. '
 + 'Examples that all resolve correctly:' + chr(10)
 + ' - "business school" or "school of business" -> Capitol Federal Hall
(Business School)' + chr(10)
 + ' - "ambler", "ambler fitness center", "ambler gym", "student gym", "fitness
center", '
 + '"the rec", "ku gym" -> Ambler Student Recreation Fitness Center' + chr(10)
 + ' - "the union", "student union", "bookstore" -> Kansas Union' + chr(10)
 + ' - "the phog", "basketball stadium", "phog stadium", "ku basketball" ->
Allen Fieldhouse' + chr(10)
 + ' - "football stadium", "ku football" -> Memorial Stadium' + chr(10)
 + ' - "natural history museum" -> Dyche Hall' + chr(10)
 + ' - "art museum" -> Spencer Museum of Art' + chr(10)
 + 'Only call list_ku_buildings if the tool explicitly reports that no building
matches.'
)

def run_parking_agent(question, verbose=True):
 '''Simple ReAct-style loop: LLM decides which tool to call, we execute it, feed
result back.'''
 messages = [
 {'role': 'system', 'content': AGENT_SYSTEM},
 {'role': 'user', 'content': question},
]

 # Flatten into a single prompt with conversation history
 def flatten(msgs):
 parts = []
 for m in msgs:
 parts.append(f"{m['role'].upper()}: {m['content']}")
 return chr(10).join(parts)

 for step in range(5): # max 5 tool rounds
 prompt = flatten(messages[1:]) # send everything except system (passed
separately)
 response = llm.generate(prompt, system_prompt=AGENT_SYSTEM)
 reply = response.text.strip()

 # Try to parse a tool call
 tool_call = None
 m = re.search(r'\{[^\}]+\}', reply)
 if m:
 try:
 parsed = _json.loads(m.group())
 if 'tool' in parsed and 'input' in parsed:

```

```

 tool_call = parsed
 except _json.JSONDecodeError:
 pass

 if tool_call is None:
 # No tool call -- this is the final answer
 if verbose:
 print(f' [Step {step+1}] Final answer' + chr(10))
 return reply

 # Execute the requested tool
 tool_name = tool_call['tool']
 tool_input = tool_call['input']
 tool_fn = TOOLS.get(tool_name)
 if tool_fn:
 if verbose:
 print(f' [Step {step+1}] Calling: {tool_name}({tool_input})')
 result = tool_fn(tool_input)
 if verbose:
 preview = str(result).replace(chr(10), ' | ')[:150]
 print(f' Result: {preview}...')
 else:
 result = f'Unknown tool: {tool_name}'

 messages.append({'role': 'assistant', 'content': reply})
 messages.append({'role': 'user', 'content': f'Tool result: {result}'})

 return 'Agent exceeded maximum steps.'

print('Parking agent ready.')

```

### 18.1.6 Ask the Agent Some Questions

Now we let the agent run end-to-end. For each question, the agent should: (1) figure out which tool to call based on the user's natural language, (2) call `find_parking_near_building` with the user's phrase (the fuzzy matcher handles informal names), and (3) format a helpful response with the ranked list, permit colors, and clickable Google Maps links. Try uncommenting the other questions to see how the agent handles visitor-specific queries, requests for the color legend, and different building names.

```

from IPython.display import display, Markdown

questions = [
 'Where can I park near the business school?',
 # Approximate/informal names all resolve to the right canonical building:
 # 'Where can I park near the Ambler fitness center?',
 # 'I want to go to the student gym, where should I park?',
 # 'parking near Ambler gym',
 # 'I need parking near the student union. What are my options?',
 # 'parking close to the phog',
 # 'basketball stadium parking',
 # 'football stadium parking',
 # 'What colors of parking are available at KU and what do they mean?',
 # 'I am visiting KU and want to park near Allen Fieldhouse. Where should I go?',
]

for q in questions:
 display(Markdown('---'))
 display(Markdown(f'***Q:** {q}'))
 answer = run_parking_agent(q)
 # Render as Markdown so Google Maps URLs become clickable links.

```

```
Colab auto-linkifies any URL inside a Markdown-rendered cell.
display(Markdown(answer))
```

### 18.1.7 How to Build the Same Thing in Dify (multi-node workflow with real tools)

A proper **Dify** workflow uses alternating tool and LLM nodes, matching the architecture you see in typical agentic workflow diagrams. Each tool node fetches data, each LLM node reasons about what to do next with that data.

#### 18.1.7.1 The workflow architecture

```
Start (user types a KU building name)
|
v
[TOOL #1] Knowledge Retrieval - KU_Buildings KB
| (acts as the "access Google Maps" step: returns the building's lat/lng)
v
[LLM #1] Parse the building coordinates
| (extracts a clean {lat, lng} from the retrieved text)
v
[TOOL #2] Knowledge Retrieval - KU_Parking_Lots KB
| (acts as the "access KU map" step: returns all parking lot data)
v
[LLM #2] Compute distances, filter to 2 miles, rank top 5, format output
| (does the math approximately using the coordinates)
v
End (returns formatted parking list with Google Maps pin URLs)
```

Five processing nodes: 2 Knowledge Retrieval (tool) nodes + 2 LLM nodes + Start/End. No Code nodes, no custom APIs, no Python.

#### 18.1.7.2 Why Knowledge Retrieval instead of HTTP Request?

In Dify’s workflow builder, a “tool” is any node that fetches external data. Knowledge Retrieval nodes are the easiest kind of tool because:

- You upload the data once (buildings + parking lots as markdown files)
- Dify chunks, embeds, and indexes it automatically
- Retrieval is just a vector search — fast and free
- No API keys, no rate limits, no billing

An alternative would be an HTTP Request node calling Nominatim (OpenStreetMap’s free geocoding API), but Nominatim is rate-limited and doesn’t know anything about specific KU building nicknames like “business school” or “the phog”. Our pre-uploaded Knowledge Base is more reliable for a classroom demo.

#### 18.1.7.3 Approximate building matching in Dify

The Colab agent above uses an explicit alias table plus token-overlap scoring. Dify’s Knowledge Retrieval node does not have that — instead, approximate matching comes from vector embeddings: the user’s phrase is embedded and compared against embedded chunks from `ku_buildings.md`. For informal queries like “Ambler gym” or “the phog” to surface the correct chunk, the aliases must appear in the markdown itself.

That is why `Agentic/dify_data/ku_buildings.md` embeds each building’s common aliases directly in its `Description` line (nicknames, school/department affiliations, department functions). For example, the Allen Fieldhouse entry mentions “basketball stadium”, “the Phog”, and “Phog Allen” explicitly so a vector search against “phog stadium” still retrieves that chunk. If you add a new nickname that users type, edit the markdown, re-upload (or sync) the Knowledge Base, and re-run the workflow.

If you ever find vector retrieval missing a phrase that the Colab agent handles correctly, either:

- Add that phrase to the building’s `Description` line in `ku_buildings.md`, or
- Switch the Dify workflow to call the Supabase MCP endpoint from the sibling `ku_parking_mcp.ipynb` notebook — that endpoint uses the same explicit alias table as the Colab code cell above.

#### 18.1.7.4 Step-by-step setup

##### 18.1.7.4.1 Part A: upload the two data files as Knowledge Bases

The notebook directory includes two markdown files at `Agentic/dify_data/`:

- `ku_buildings.md` - one section per building with coordinates and alias-rich descriptions
- `ku_parking_lots.md` - one section per parking lot with color and coordinates

In Dify:

1. Go to [cloud.dify.ai](https://cloud.dify.ai) -> Knowledge -> Create Knowledge
2. Upload `ku_buildings.md`. Name the KB `KU_Buildings`.
3. On the chunking screen, pick Custom chunking:
  - Chunk delimiter: `\n#` (hash + space — this splits on every heading)
  - Max chunk length: 500 tokens
  - This ensures each building becomes its own retrievable chunk, and the alias-rich description stays within that chunk.
4. Indexing: High Quality (vector embeddings).
5. Click Save & Process. Wait ~30 seconds.
6. Repeat: create a second Knowledge Base called `KU_Parking_Lots` and upload `ku_parking_lots.md` with the same chunking settings.

##### 18.1.7.4.2 Part B: build the workflow

1. **Studio -> Create App -> Workflow** (not Chatflow).
2. Name it `KU Parking Assistant`.
3. On the canvas, click the Start node:
  - Add input variable: name = `destination`, type = Text, label = "Which KU building are you visiting? (informal names like 'the phog', 'Ambler gym', 'business school' all work)"
4. **Add TOOL #1: Knowledge Retrieval (Buildings)**
  - Click + after Start -> Knowledge Retrieval
  - Query variable: `Start.destination`
  - Knowledge: select `KU_Buildings`
  - Rename this node to `Retrieve Building`
5. **Add LLM #1: Parse Building Coordinates**
  - Click + after Retrieve Building -> LLM
  - Model: your preferred (gpt-4o-mini, gemini-2.5-flash, etc.)
  - System prompt:

You are a data extractor. You will receive:

1. A user query that may use a nickname or informal name (e.g., "the phog", "Ambler gym", "student union", "business school").
2. One or more retrieved KU building chunks, each containing a canonical building name, its coordinates, and a description that lists common aliases and school/department associations.

Pick the single chunk whose canonical name or description best matches the user's query, and extract its canonical name and coordinates. If the query uses a nickname, use the alias list in the description to choose the correct building (e.g., "the phog" -> Allen Fieldhouse; "Ambler gym" or "student gym" -> Ambler Student Recreation Fitness Center; "school of business" -> Capitol Federal Hall).

Reply with ONLY this JSON format (nothing else):

```
{"building": "full canonical name", "lat": 38.9573, "lng": -95.2519}
```

- User prompt:

```
Retrieved building information:
{{#Retrieve_Building.result#}}

User query: {{#Start.destination#}}
```

- Rename this node to `Parse Coords`

#### 1. Add TOOL #2: Knowledge Retrieval (Parking Lots)

- Click + after Parse Coords -> Knowledge Retrieval
- Query variable: `Start.destination` (or a fixed query like “parking lots”)
- Knowledge: select `KU_Parking_Lots`
- In the node settings, set Top K to 15 (to retrieve all lots, since our KB is small)
- Rename this node to `Retrieve Parking`

#### 2. Add LLM #2: Distance + Filter + Format

- Click + after Retrieve Parking -> LLM
- System prompt:

You are a parking assistant. You will receive:

1. A destination building with its latitude and longitude (as JSON).
2. A list of KU parking lots with their coordinates and colors.

Your job:

- Compute the approximate great-circle distance in miles from the destination to each parking lot. For Lawrence KS, 0.001 degrees of latitude is about 0.069 miles and 0.001 of longitude is about 0.054 miles.
- Filter to lots within 2 miles of the destination.
- Rank the remaining lots from nearest to farthest.
- Return the top 5 as a numbered list with these fields for each:
  - \* Rank, lot name, permit color
  - \* Distance in miles (2 decimals) and approximate walk time (3 mph walking speed)
  - \* Google Maps pin URL in this exact format: <https://www.google.com/maps?q=LAT,LNG>
- End with a short note explaining what the permit color of the closest lot means.
- If no lots are within 2 miles, say so and suggest the Park & Ride shuttle.

- User prompt:

```
Destination: {{#Parse_Coords.text#}}
```

```
Available parking lots:
{{#Retrieve_Parking.result#}}
```

Give me the top 5 parking lots within 2 miles.

- Rename this node to `Rank and Format`

#### 1. Configure the End node

- Output variable: `Rank_and_Format.text`

#### 2. Publish and click Run to test the approximate matcher end-to-end:

- `business school / school of business` -> Capitol Federal Hall
- `the phog / basketball stadium / phog stadium` -> Allen Fieldhouse
- `Ambler gym / student gym / fitness center` -> Ambler SRFC
- `the union / student union` -> Kansas Union
- `football stadium / ku football` -> Memorial Stadium

#### 18.1.7.5 What each node does and why

| Node  | Type  | Job                                            | Deterministic or LLM? |
|-------|-------|------------------------------------------------|-----------------------|
| Start | Input | Collect user destination (any informal phrase) | •                     |

| Node              | Type                       | Job                                                                                  | Deterministic or LLM?               |
|-------------------|----------------------------|--------------------------------------------------------------------------------------|-------------------------------------|
| Retrieve Building | Tool (Knowledge Retrieval) | Vector search against buildings KB — finds the right chunk from an alias match       | Deterministic (vector search)       |
| Parse Coords      | LLM                        | Picks the right canonical building from the retrieved chunks and extracts {lat, lng} | LLM (disambiguation + extraction)   |
| Retrieve Parking  | Tool (Knowledge Retrieval) | Look up parking lots from the Parking KB                                             | Deterministic (vector search)       |
| Rank and Format   | LLM                        | Compute distances, filter 2 miles, rank, format                                      | LLM (approximate math + formatting) |
| End               | Output                     | Return the formatted result                                                          | •                                   |

Notice the alternation: Tool -> LLM -> Tool -> LLM. Each tool fetches fresh data, each LLM reasons about what to do with it. This is the standard pattern in Dify workflows and maps directly to the architecture diagram you described.

#### 18.1.7.6 Colab (real tools) vs. Dify (workflow tools) - the teaching contrast

|                                 | Colab notebook                               | Dify workflow                                                       |
|---------------------------------|----------------------------------------------|---------------------------------------------------------------------|
| How are tools defined?          | Python functions                             | Dify nodes (Knowledge Retrieval, LLM, HTTP Request)                 |
| Who decides which tool to call? | The LLM, at each step of the agent loop      | Fixed by the workflow diagram (same tools in the same order)        |
| Approximate building matching   | Explicit alias table + token-overlap scoring | Vector embeddings over alias-rich descriptions + LLM disambiguation |
| Distance math                   | Exact (Haversine formula)                    | Approximate (LLM reasoning over coordinates)                        |
| Data source                     | Python dicts                                 | Dify Knowledge Bases with vector retrieval                          |
| Flexibility                     | High — agent can call tools in any order     | Low — flow is fixed, but predictable                                |
| Best for                        | Complex questions with branching reasoning   | Simple, predictable pipelines with the same steps every time        |

#### 18.1.7.7 When to pick each approach

- **Colab** agent (this notebook): when the agent needs to decide dynamically whether to call `list_buildings`, `find_parking_near_building`, or `get_parking_colors_legend` based on the question. Good for conversational or exploratory use cases.
- Dify workflow (above): when every query follows the same steps (lookup destination -> lookup parking -> rank). Good for a production app where you want consistent, debuggable behavior without per-query reasoning overhead.

#### 18.1.7.8 Limitations of the Dify version

- The LLM's distance math is approximate. For 15 lots in a small area it usually gets the top 3 right, but lots at very similar distances may be mis-ranked by 1-2 positions.
- Approximate building matching depends on the aliases being present in the markdown. A nickname that isn't written into `ku_buildings.md` will only match if it's semantically close enough for the



embedding model to score highly — and that is less reliable than the explicit alias table in the Colab agent.

- If your Knowledge Base grows beyond a few hundred entries, the vector retrieval step may miss relevant items. At that scale you'd switch to HTTP Request calls to a real database, or to the MCP endpoint in `ku_parking_mcp.ipynb`.
- The 2-mile filter is enforced in the LLM prompt, not in code. A rogue LLM response could include a lot outside 2 miles. In production you'd add a Template Transform or Parameter Extractor node to enforce the filter programmatically.

#### 18.1.8 Exercises

- Add a new building (e.g., 'Budig Hall': (lat, lng) ) and a new parking lot, then rerun — notice you don't need to change any tool code.
- Add a fourth tool: `find_parking_by_color(color, near_building)` that filters to only lots of a specific permit color.
- Replace the hardcoded data with a real parse of the [KU parking map PDF](#) — this is where real data engineering begins.
- In Dify, try removing the coordinates from the system prompt and just listing lot names. How does answer quality change? This shows why giving the LLM structured numeric data matters.
- Compare the Colab and Dify outputs for the same query. How often do they agree on the top-3 ranking?

#### 18.1.9 Key takeaways

- **Tool-using agents** combine LLM strengths (fuzzy language understanding) with Python strengths (deterministic math and lookups), letting each side do what it does best.
- **The Haversine formula** gives exact great-circle distances in microseconds, which is far more reliable than asking the LLM to eyeball coordinate differences.
- **Alias tables plus token-overlap scoring** resolve informal phrases like “the phog” or “student gym” to canonical building names before any tool runs.
- **A prompt-based ReAct loop** – emit JSON action, run the tool, feed the result back – is all you need to build a working agent without a framework.
- **Dify workflows** are the fixed-flow counterpart: alternating Knowledge Retrieval and LLM nodes give predictable behavior, while the Python agent offers dynamic tool selection.

---

#### 18.1.10 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

**Estimated run time:** ~2 minutes (1-3 LLM calls per question)

```
https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/KU_Parking_Assistant.ipynb
```

## 18.2 KU Parking Assistant via Cloud Run MCP Endpoint

In the previous notebook ( `KU_Parking_Assistant.ipynb` ), we built a parking assistant where every tool – distance math, building lookup, color legend – lived as a plain Python function inside the notebook. That works beautifully for a single-notebook demo, but it creates a problem the moment you want a second client to use the same tools. A Dify workflow, a mobile app, and a Cursor plugin would each need their own copy of the parking data and distance logic, and any update to the data (a new parking lot opens, coordinates get corrected) would require updating every client independently.

**Model Context Protocol (MCP)** solves this by moving the tools to a server. The server exposes a standard JSON-RPC 2.0 interface: any client can call `tools/list` to discover what tools are available, then `tools/call` to execute one. The protocol is transport-agnostic – it works over HTTP, WebSockets, or stdio – and it is the same standard that Claude Desktop, Cursor, and Dify use to integrate external tools. In this notebook, our MCP server is a **Google Cloud Run** container written in TypeScript, deployed to the cloud, and accessible at a single HTTPS URL.

What changes from the inline-tools version:

- No hardcoded `KU_BUILDINGS` or `KU_PARKING_LOTS` dictionaries – the data lives on the server
- No local `find_parking_near_building` function – we call it remotely over HTTPS
- The same tool interface and the same agent ReAct loop – only the transport changes
- Any MCP-compatible client can call the same endpoint and get identical results

**What is JSON-RPC?** It is a lightweight remote procedure call protocol that uses JSON for encoding. Each request contains a `method` (like `tools/call`), a `params` object (the tool name and arguments), and an `id` so the client can match responses to requests. The server returns a JSON object with either a `result` or an `error` field. MCP builds on top of JSON-RPC by standardizing the method names ( `tools/list`, `tools/call` ) and the shape of tool descriptions, so any client and any server that speak MCP can interoperate without custom integration code.

|                                 | <code>KU_Parking_Assistant.ipynb</code> | <code>KU_Parking_mcp.ipynb</code> (this one)       |
|---------------------------------|-----------------------------------------|----------------------------------------------------|
| Where do tools live?            | Inline Python functions                 | Cloud Run container (TypeScript)                   |
| Where does data live?           | Hardcoded in the notebook               | Hardcoded in the container                         |
| How does the agent call a tool? | Direct Python call                      | HTTP POST with JSON-RPC body                       |
| Runs where?                     | 100% in Colab                           | Logic runs on Cloud Run, notebook is just a client |
| Can other apps use the tools?   | No                                      | Yes – Dify, Cursor, web apps, etc.                 |

**Estimated run time:** ~2 minutes (requires Cloud Run MCP endpoint)

```
!pip install -q git+https://github.com/KarAnalytics/llm_cascade.git
```

### 18.2.1 Imports and LLM Setup

We use `llm_cascade` for automatic LLM provider fallback and `requests` to speak JSON-RPC over HTTPS with the Cloud Run MCP endpoint. No local parking data, no distance math library – all of that lives on the server now.

```
import json as _json
import re as _re
import requests

from llm_cascade import get_cascade

llm = get_cascade()
```

```
The deployed Cloud Run MCP endpoint URL
MCP_URL = 'https://ku-parking-mcp-1098133188018.us-central1.run.app'
```

```
LLM Cascade - available providers:
+ Gemini model=gemini-2.5-flash
+ Ollama model=kimi-k2.5:cloud
+ Groq model=llama-3.3-70b-versatile
+ HuggingFace model=meta-llama/Llama-3.3-70B-Instruct
+ Cohere model=command-a-03-2025
+ OpenRouter model=meta-llama/llama-3.3-70b-instruct:free
+ OpenAI model=gpt-4o-mini
Not configured (skipped):
- Grok (xAI) (set XAI_API_KEY)
```

### 18.2.2 The MCP Client (a thin wrapper over `requests`)

One of the most liberating things about MCP is how simple the client side is. There is no SDK to install, no proto files to compile, no complex authentication dance. MCP communication is just JSON-RPC 2.0 over HTTP POST – you send a JSON object with a `method`, `params`, and `id`, and the server replies with a JSON object containing either a `result` or an `error`. The two methods we care about are `tools/list` (discover what tools the server offers) and `tools/call` (execute a specific tool with arguments).

The `mcp_call` function below is the entire MCP client: it builds a JSON-RPC payload, POSTs it to the Cloud Run URL, and returns the parsed response. The `mcp_tool` convenience wrapper adds one more step – it extracts the text content from the tool’s response, so callers get a clean string instead of the full JSON-RPC envelope. Compare this to the inline-tools notebook, where the “client” was a direct Python function call. The interface to the agent is identical; only the plumbing underneath has changed.

```
def mcp_call(method, params=None, request_id=1):
 '''Send a JSON-RPC 2.0 request to the MCP server.'''
 payload = {
 'jsonrpc': '2.0',
 'id': request_id,
 'method': method,
 'params': params or {},
 }
 response = requests.post(MCP_URL, json=payload, timeout=30)
 response.raise_for_status()
 return response.json()

def mcp_tool(name, **arguments):
 '''Call a named MCP tool and return its text content.'''
 result = mcp_call('tools/call', {
 'name': name,
 'arguments': arguments,
 })
 if 'error' in result:
 return f"MCP error: {result['error']['message']}"
 return result['result']['content'][0]['text']

print('MCP client ready.')
```

```
MCP client ready.
```

### 18.2.3 Discover the Tools (no LLM yet)

Before we build the agent, let’s ask the MCP server what tools it exposes. This demonstrates the **tool discovery** step of MCP – a critical feature that sets it apart from ad-hoc HTTP APIs. In a traditional integration, the client must hardcode the endpoint URLs, parameter names, and response shapes. With

MCP, the client calls `tools/list` once, receives a machine-readable schema for every available tool (name, description, parameter types), and uses that schema to construct valid calls dynamically.

This means that if someone adds a new tool to the server (say, `find_parking_by_color`), existing clients discover it automatically on their next `tools/list` call – no code changes, no redeployment, no coordination between teams. It is the same principle that made REST APIs with self-describing schemas (like OpenAPI/Swagger) so successful, applied specifically to the problem of giving LLM agents access to external tools.

```
tools_response = mcp_call('tools/list')
tools = tools_response['result']['tools']

print(f'Discovered {len(tools)} tools on the MCP server:' + chr(10))
for t in tools:
 print(f" - {t['name']}")
 desc = t.get('description', '').strip().split(chr(10))[0]
 print(f" {desc}")
```

Discovered 3 tools on the MCP server:

- `list_ku_buildings`  
List all KU buildings available for parking lookup. Returns a newline-separated list of building names.
- `find_parking_near_building`  
Find the nearest parking lots within a given radius of a KU building. Approximate matching is supported -- informal names, abbreviations, and partial phrases all resolve to the right building. Examples: 'business school' -> Capitol Federal Hall (Business School); 'ambler', 'ambler gym', 'student gym', or 'fitness center' -> Ambler Student Recreation Fitness Center; 'the union' or 'student union' -> Kansas Union; 'the phog' -> Allen Fieldhouse. Returns a ranked list where each lot includes all applicable permit colors (slash-separated for multi-zone lots), a location hint, distance in miles, walk time, and a Google Maps pin URL.
- `get_parking_colors_legend`  
Return the KU parking color legend, grouped by faculty/staff tiered permits, student housing permits, and everyone-else options (Visitor, Park & Ride, Garage). Sourced from [parking.ku.edu](https://parking.ku.edu).

### 18.2.4 Test One Tool Directly

Before wiring up the LLM agent, let's call `find_parking_near_building` manually to verify the server works and returns the expected format. This is the same debugging practice we used in the inline-tools notebook – always test your tools in isolation before letting an LLM loose on them. If the raw tool output looks wrong, no amount of prompt engineering will produce a correct final answer. Notice that the response format (lot name, color, distance, walk time, Google Maps URL) is identical to what the local Python function returned in the previous notebook, because the MCP server implements the same logic in TypeScript on the server side.

```
result = mcp_tool(
 'find_parking_near_building',
 building_name='business school',
 max_distance_miles=2.0,
 top_k=5,
)
print(result)
```

Parking lots within 2 miles of Capitol Federal Hall (Business School) (top 5, ranked by distance):

1. Lot 118 (Blue) near E. Capitol Federal Hall - 0 mi (0 min walk) -> <https://www.google.com/maps?q=38.953505,-95.24974>
2. Lot 90 (Blue/Red/Yellow) near Rec Center (Ambler SRFC) - 0.119 mi (2 min walk)

```
-> https://www.google.com/maps?q=38.952512,-95.247929
3. Lot 71 (Yellow) near S. Allen Fieldhouse - 0.143 mi (3 min walk) -> https://
www.google.com/maps?q=38.953666,-95.252394
4. Lot 70 (Red) near S. Allen Fieldhouse - 0.145 mi (3 min walk) -> https://www.
google.com/maps?q=38.953906,-95.252394
5. Lot 117 (Blue/Red/Yellow) near E. Watkins Health Center - 0.151 mi (3 min walk)
-> https://www.google.com/maps?q=38.954,-95.247
```

### 18.2.5 The Agent (LLM that calls MCP tools)

Here is where the architectural elegance of MCP becomes visible. The ReAct-style agent loop below is structurally identical to the one in `KU_Parking_Assistant.ipynb` – the LLM reads the question, decides which tool to call, outputs a JSON action, and we parse and execute it. The only difference is that instead of calling `TOOLS[tool_name](tool_input)` (a local Python function), we call `mcp_tool(tool_name, building_name=tool_input)` (an HTTPS request to Cloud Run). The LLM does not know or care where the tools live – it sees the same tool descriptions in its system prompt and produces the same JSON actions. That separation of concerns is the whole point of a protocol: it decouples the agent's reasoning from the tool's implementation.

```
tool_block = chr(10).join(f"- {t['name']}: {t['description']}" for t in tools)

AGENT_SYSTEM = (
 'You are a helpful KU parking assistant. Tools available via MCP:' + chr(10)
 + tool_block + chr(10) + chr(10)
 + 'To use a tool, reply with ONLY this JSON (nothing else):' + chr(10)
 + '{"tool": "tool_name", "input": "the input string"}' + chr(10) + chr(10)
 + 'After a tool result, write a helpful final answer for the user that includes'
 ,
 + 'the ranked parking list, color meanings, and clickable Google Maps links. '
 + 'Pass the user\'s phrase through as-is to find_parking_near_building – the'
 server '
 + 'handles approximate matching. Only call list_ku_buildings if the tool reports'
 ,
 + 'that no building matched.'
)

def run_parking_agent(question, verbose=True):
 '''ReAct loop: LLM picks a tool -> MCP executes it -> LLM formats the final
 answer.'''
 messages = [
 {'role': 'system', 'content': AGENT_SYSTEM},
 {'role': 'user', 'content': question},
]

 def flatten(msgs):
 return chr(10).join(f"{m['role'].upper()}: {m['content']}" for m in msgs)

 for step in range(5):
 prompt = flatten(messages[1:]) # everything except system
 response = llm.generate(prompt, system_prompt=AGENT_SYSTEM)
 reply = response.text.strip()

 # Try to parse a tool call JSON from the reply
 tool_call = None
 m = _re.search(r'\{[^\}]+\}', reply)
 if m:
 try:
 parsed = _json.loads(m.group())
 if 'tool' in parsed and 'input' in parsed:
 tool_call = parsed
 except _json.JSONDecodeError:
```

```

 pass

 if tool_call is None:
 # No tool call -- this is the final answer
 if verbose:
 print(f' [Step {step+1}] Final answer' + chr(10))
 return reply

 # Dispatch the tool call to the MCP server
 tool_name = tool_call['tool']
 tool_input = tool_call['input']
 if verbose:
 print(f' [Step {step+1}] MCP call: {tool_name}({tool_input!r})')

 # Each tool takes a different argument name -- map them
 if tool_name == 'find_parking_near_building':
 result = mcp_tool(tool_name, building_name=tool_input)
 else:
 result = mcp_tool(tool_name)

 if verbose:
 preview = str(result).replace(chr(10), ' | ')[:150]
 print(f' Result: {preview}...')

 messages.append({'role': 'assistant', 'content': reply})
 messages.append({'role': 'user', 'content': f'Tool result: {result}'})

 return 'Agent exceeded maximum steps.'

print('Parking agent ready (tools backed by Cloud Run MCP endpoint).')

```

```
Parking agent ready (tools backed by Cloud Run MCP endpoint).
```

### 18.2.6 Ask the Agent Some Questions

Each question runs through the agent loop. Notice the output shows **MCP call: ...** at each step – that is the agent dispatching tool calls to the Cloud Run endpoint over HTTPS. The rest of the notebook never touches the parking data directly. If you compare the agent's final answer here with the one from the inline-tools notebook, they should be nearly identical, because both notebooks call the same logical tools with the same data. The difference is entirely in the plumbing: local function calls versus remote JSON-RPC calls. Try uncommenting the additional questions to see how the agent handles different phrasing, visitor-specific queries, and requests for the color legend.

```

from IPython.display import display, Markdown

questions = [
 # 'Where can I park near the business school?',
 # Approximate/informal names all resolve to the right canonical building:
 # 'Where can I park near the Ambler fitness center?',
 # 'I want to go to the student gym, where should I park?',
 # 'parking near Ambler gym',
 # 'parking close to the phog',
 # 'basketball stadium parking',
 # 'football stadium parking',
 'I need parking near the student union. What are my options?',
 # 'What colors of parking are available at KU and what do they mean?',
 # 'I am visiting KU and want to park near Allen Fieldhouse. Where should I go?',
]

for q in questions:

```

```
display(Markdown('---'))
display(Markdown(f'***Q:** {q}'))
answer = run_parking_agent(q)
Render as Markdown so the Google Maps URLs become clickable links.
display(Markdown(answer))
```

<IPython.core.display.Markdown object>

<IPython.core.display.Markdown object>

```
[Response from Gemini / gemini-2.5-flash]
[Step 1] MCP call: find_parking_near_building('student union')
 Result: Parking lots within 2 miles of Kansas Union (top 5, ranked by
distance): | | 1. Lot 16 (Gold) near E. Kansas Union - 0.028 mi (1 min walk) ->
https...
[Response from Gemini / gemini-2.5-flash]
[Step 2] Final answer
```

<IPython.core.display.Markdown object>

### 18.2.7 Using the Same MCP Server from Dify

The whole point of exposing tools via MCP is that other apps can use them too. This section walks you through connecting the same Cloud Run MCP endpoint (the one we just used from this notebook) to a **Dify** Workflow. You'll build a no-code version of the parking assistant that calls the same `find_parking_near_building` tool.

Approximate building names work here too. The MCP server does fuzzy matching server-side, so whatever a Dify end user types – “business school”, “Ambler gym”, “student rec center”, “the phog”, “basketball stadium” – gets resolved to the correct canonical building before distances are computed. Your Dify workflow does not need to normalize or spell-check the user's input.

#### 18.2.7.1 What you need

- A Dify account — sign up at [cloud.dify.ai](https://cloud.dify.ai) (free tier works)
- At least one LLM provider configured in Dify (Settings arrow.r Model Provider)
- The Cloud Run MCP URL: `https://ku-parking-mcp-1098133188018.us-central1.run.app`

Dify offers two ways to connect to the MCP server. Option A works on any Dify version; Option B is cleaner if your instance has the MCP plugin installed.

#### 18.2.7.2 Option A — HTTP Request node (always works, no plugin needed)

This is the guaranteed path: Dify's built-in HTTP Request node posts JSON to the Cloud Run URL and we parse the response.

##### 18.2.7.2.1 Step 1: Create a Workflow app

1. In Dify, click **Studio arrow.r Create App arrow.r Workflow** (not Chatflow, not Agent)
2. Name it `KU Parking via MCP`

##### 18.2.7.2.2 Step 2: Configure the Start node

1. Click the **Start** node on the canvas
2. Add an input variable:
  - Variable name: `destination`
  - Type: Text



- Label: Which KU building are you visiting? (informal names like "Ambler gym", "student union", "the phog", or "business school" all work)
- Required: Yes

#### 18.2.7.2.3 Step 3: Add an HTTP Request node

1. Click + after Start arrow.r **HTTP Request**
2. Configure it:
  - Method: `POST`
  - URL: `https://ku-parking-mcp-1098133188018.us-central1.run.app`
  - Headers: `Content-Type: application/json`
  - Body type: `JSON`
  - Body:

```
{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "tools/call",
 "params": {
 "name": "find_parking_near_building",
 "arguments": {
 "building_name": "{{#Start.destination#}}",
 "max_distance_miles": 2.0,
 "top_k": 5
 }
 }
}
```

- Note: pass `{{#Start.destination#}}` through as-is. The MCP server's approximate matcher handles informal names, abbreviations, and partial phrases, so you do not need to normalize the input on the Dify side.

1. Rename the node to `Call MCP`

#### 18.2.7.2.4 Step 4: Add a Code node to extract the answer

The HTTP Request returns the full JSON-RPC response. We need to pull out `result.content[0].text`.

1. Click + after the HTTP Request arrow.r **Code**
2. Language: `Python 3`
3. Input variable: `response_body` = `Call_MCP.body`
4. Code:

```
import json

def main(response_body: str) -> dict:
 data = json.loads(response_body)
 if "result" in data and "content" in data["result"]:
 return {"parking_info": data["result"]["content"][0]["text"]}
 return {"parking_info": "MCP error: " + str(data.get("error", "unknown"))}
```

1. Output variable: `parking_info` (String)
2. Rename the node to `Extract Result`

#### 18.2.7.2.5 Step 5: Add an LLM node to format the answer

1. Click + after the Code node arrow.r **LLM**
2. Model: pick whatever you have configured (gpt-4o-mini, gemini-2.5-flash, etc.)
3. System prompt:

You are a helpful KU parking assistant. The user asked about parking near a KU building. Below is the raw parking information returned from the KU Parking



MCP server.

The server does APPROXIMATE matching on building names, so whatever phrase the user typed (e.g., "business school", "Ambler gym", "student gym", "the phog", "basketball stadium", "student union") will have already been resolved to the correct canonical building -- do not second-guess the match. The first line of the parking information tells you which canonical building was resolved; echo that name back to the user so they can confirm the match.

Present the result in a friendly, well-formatted response that includes the ranked parking list with permit color, distance, walk time, and clickable Google Maps links.

IMPORTANT guardrails:

- Do NOT mention "list\_ku\_buildings" or any tool/command name in your reply. Those tools are internal to this workflow and the end user has no way to invoke them directly.
- If the server reports that no building matched, apologize briefly and ask the user to rephrase or try a nearby landmark (e.g., "try 'student union' or 'Ambler gym'"). Give a few example phrasings but do NOT suggest they run a command.
- The MCP server already accepts informal names like "the phog", "basketball stadium", or "student gym". If the user appears to have typed the correct canonical name yet you still got a no-match error, suggest they try again and mention that approximate names work too.

1. User prompt:

User question: Where can I park near {{#Start.destination#}}?

Parking information from the MCP server:

{{#Extract\_Result.parking\_info#}}

Please format a helpful response for the user. Remember: never mention list\_ku\_buildings or any other tool name; the user cannot invoke tools here.

1. Rename the node to `Format Answer`

#### 18.2.7.2.6 Step 6: Configure the End node

1. Click the **End** node
2. Output variable: `Format_Answer.text`

#### 18.2.7.2.7 Step 7: Publish and test

1. Click **Publish** (top right) arrow.r **Update Publishing**
2. Click **Run** and try a few destinations that exercise the approximate matcher:
  - `business school` / `school of business` arrow.r Capitol Federal Hall (Business School)
  - `Ambler gym`, `student gym`, `fitness center`, `the rec` arrow.r Ambler Student Recreation Fitness Center
  - `the union`, `student union` arrow.r Kansas Union
  - `the phog`, `basketball stadium`, `phog stadium` arrow.r Allen Fieldhouse
  - `football stadium`, `ku football` arrow.r Memorial Stadium

You should see the same ranked parking list you got in this notebook for each of these.

---

#### 18.2.7.3 Option B — MCP plugin (cleaner if available)

If your Dify instance has the MCP SSE or MCP HTTP plugin installed (check under **Tools** arrow.r **Install Tool** and search for "mcp"):

1. Install the MCP plugin
2. Go to **Tools** and click the MCP plugin

3. Add a new MCP server:

- Name: `ku-parking`
- URL: `https://ku-parking-mcp-1098133188018.us-central1.run.app`
- Save

4. Dify will call `tools/list` on the server and automatically discover the three tools

5. In any Workflow, the three MCP tools now appear as native Dify tool nodes you can drag onto the canvas. You skip the HTTP Request + Code node dance entirely.

With Option B, your workflow becomes:

```
Start -> [MCP Tool: find_parking_near_building] -> LLM (format) -> End
```

Just three processing nodes, no manual JSON parsing. The MCP tool node accepts the raw user phrase as `building_name` and the server handles approximate matching – same behavior as Option A, just a cleaner wiring. The same “do not mention tool names” guardrail still applies to the Format Answer LLM node.

---

#### 18.2.7.4 Which option should you try?

- Start with Option A if you’re not sure whether your Dify instance has the MCP plugin. The HTTP Request + Code node approach always works on cloud Dify and self-hosted Dify alike.
- Upgrade to Option B once you confirm the plugin is available. It’s cleaner and shows off MCP’s real value – tool discovery happens automatically.

Either way, the same Cloud Run endpoint (the one we called from this notebook) powers the Dify app. Update the data in the container – or add a new alias to `BUILDING_ALIASES` so that “b-school” or “the rec” also resolves correctly – redeploy, and both this notebook and the Dify app see the new behavior immediately.

---

### 18.2.8 Key Takeaways

Same agent, different transport. The ReAct agent loop in this notebook is identical to the one in `KU_Parking_Assistant.ipynb`. The only change is that tool calls now dispatch via `mcp_tool()` – a tiny HTTPS wrapper – instead of calling local Python functions.

Write the tools once, use them everywhere. The Cloud Run container is the single source of truth for parking data. This notebook, Dify, Claude Desktop, a web app, and a mobile app can all call the same `find_parking_near_building` tool and get identical results.

Update data in one place. Edit `KU_PARKING_LOTS` in `infra/ku-parking-mcp/cloudrun/index.ts`, redeploy from that folder with `gcloud run deploy ku-parking-mcp --source . --region us-central1 --allow-unauthenticated`, and every client sees the new data on its next request. No client code changes, no notebook reruns needed.

Tool discovery is automatic. Clients don’t hardcode tool names. They call `tools/list` first, receive the server’s schema, and use that to build their prompts. If you add a new tool to the container, existing clients discover it without code changes.

#### 18.2.8.1 Exercises

- Add a new tool to the container (e.g., `find_parking_by_color` that filters to one permit color). Redeploy and see the new tool appear automatically in `tools/list`.
- Update a building coordinate in the container and watch this notebook pick up the new value without any local changes.
- Build the Dify version following Option A above and compare its output with this notebook’s agent. Do they agree?
- Add authentication. Drop the `--allow-unauthenticated` flag when redeploying, then update `mcp_call` to send an `Authorization: Bearer <id-token>` header (get the token via `gcloud auth print-identity-token`). Now the endpoint is no longer public.
- Add caching. In `mcp_call`, cache the response of `tools/list` so the client only discovers tools once per session instead of on every run.

### 18.2.9 Run the code

To run this notebook, copy the URL below into your browser's address bar. The link opens the notebook directly in Google Colab. *(If your PDF viewer makes the URL clickable and lands on a broken page, copy the full text manually – the viewer may have truncated the link at a line break.)*

[https://colab.research.google.com/github/KarAnalytics/code\\_demos/blob/main/KU\\_Parking\\_mcp.ipynb](https://colab.research.google.com/github/KarAnalytics/code_demos/blob/main/KU_Parking_mcp.ipynb)



## Chapter 19

# Deployment and No-Code

The final chapter takes AI from notebooks to production:

- **MCP on Cloud Run** — deploy tools as a container on Google Cloud Run that any MCP client can call over HTTPS
- **Dify Workflows** — build the same AI applications visually with no code, using Dify's workflow builder

These sections show that the concepts from earlier chapters (RAG, tools, agents) translate directly to production platforms — the same patterns, different delivery mechanisms.

## 19.1 MCP Servers on Google Cloud Run

Deploying tools as a cloud-hosted MCP server gives every client (notebooks, web apps, Dify, Claude Desktop) access to the same tools via a standard HTTPS interface.

This section walks through deploying the KU Parking MCP server to Google Cloud Run and calling it from Python and Dify.

The deployment guide and full source live in this repo under `infra/ku-parking-mcp/`. The TypeScript server is at `infra/ku-parking-mcp/cloudrun/index.ts`, and the container build is defined in the sibling `Dockerfile`. To deploy:

```
cd infra/ku-parking-mcp/cloudrun
gcloud run deploy ku-parking-mcp --source . --region us-central1 --allow-unauthenticated --min-instances 0 --max-instances 10
```

Cloud Build packages the folder into a container, pushes it to Artifact Registry, and Cloud Run starts serving it at a public HTTPS URL. No local Docker required — `--source .` hands the build off to Google's infrastructure.

## 19.2 No-Code AI with Dify

Dify is a visual LLM application builder. This section shows how to replicate two of our notebook-based applications in Dify without writing any code:

**Document RAG** — upload company documents to a Dify Knowledge Base, add a Knowledge Retrieval node + LLM node, and get a working Q&A app in 15 minutes.

**KU Parking via MCP** — connect Dify to the Cloud Run MCP endpoint using an HTTP Request node, parse the JSON-RPC response, and format the answer with an LLM node.

Step-by-step instructions are included in the respective notebook markdown sections.





# Appendix

## 20.1 Learning Path and Chapter Dependencies

This page maps the dependencies between chapters. Use it to plan your reading order if you want to skip ahead to a specific topic.

### 20.1.1 The Recommended Sequential Path

```
Ch 1 (AI at a Glance)
↓
Ch 2 (LLM Foundations)
↓
Ch 3 (LLM Providers) → Ch 4 (Local LLMs)
↓ ↓
Ch 5 (Vibe Coding) ←-----
↓
Ch 6 (RAG First Principles)
↓
Ch 7 (Vector DBs) → Ch 8 (Structured Data) → Ch 9 (Multimedia)
↓
Ch 10-12 (LlamaIndex → LangChain → LangGraph)
↓
Ch 13 (Fine-Tuning)
↓
Ch 14-17 (Single Agent → Multi-Agent → Autonomous → Tool Agents/MCP)
↓
Ch 18 (Deployment)
```

### 20.1.2 What You Can Skip To

If you already have experience with a topic, here is what each chapter actually requires:

| Chapter                        | Prerequisites | Can skip if you know...                                 |
|--------------------------------|---------------|---------------------------------------------------------|
| Ch 1: AI at a Glance           | None          | How LLMs are trained (pre-training, SFT, RLHF)          |
| Ch 2: LLM Foundations          | None          | Tokenization, embeddings, attention, transformers       |
| Ch 3: LLM Providers            | Ch 2          | How to call OpenAI/Gemini APIs                          |
| Ch 4: Local LLMs               | Ch 3          | How to run LM Studio or Ollama                          |
| Ch 5: Vibe Coding              | Ch 3          | How to use Cursor, Copilot, or Claude Code              |
| Ch 6: RAG First Principles     | Ch 2, Ch 3    | Chunking, embedding, cosine similarity, prompt stuffing |
| Ch 7: RAG with Vector DBs      | Ch 6          | ChromaDB, Pinecone, FAISS                               |
| Ch 8: RAG with Structured Data | Ch 6          | Text-to-SQL, Cypher, schema-as-context                  |
| Ch 9: RAG with Multimedia      | Ch 6          | CLIP, image/video embeddings                            |
| Ch 10: LlamaIndex              | Ch 6          | LlamaIndex SimpleDirectoryReader, VectorStoreIndex      |
| Ch 11: LangChain               | Ch 6          | LCEL pipe syntax, LangChain chains and agents           |
| Ch 12: LangGraph               | Ch 11         | StateGraph, conditional edges, graph loops              |
| Ch 13: Fine-Tuning             | Ch 2, Ch 3    | LoRA, QLoRA, SFTTrainer                                 |
| Ch 14: Single Agent            | Ch 3          | LLM + tools pattern, system prompts                     |
| Ch 15: Multi-Agent             | Ch 14         | Agent handoff, LLM-as-a-judge                           |

| Chapter                    | Prerequisites | Can skip if you know...                      |
|----------------------------|---------------|----------------------------------------------|
| Ch 16: Autonomous Agents   | Ch 14         | Plan-Execute-Synthesize-Reflect pattern      |
| Ch 17: Tool Agents and MCP | Ch 14         | MCP protocol, JSON-RPC, Cloud Run containers |
| Ch 18: Deployment          | Ch 17         | Cloud Run deployment, Dify workflows         |

### 20.1.3 Three Fast Tracks

**“I just want to build a document Q&A app”:** Ch 1 [arrow.r](#) Ch 2 [arrow.r](#) Ch 3 [arrow.r](#) Ch 6 [arrow.r](#) Ch 10 (LlamaIndex)

**“I want to understand agents”:** Ch 1 [arrow.r](#) Ch 2 [arrow.r](#) Ch 3 [arrow.r](#) Ch 14 [arrow.r](#) Ch 15 [arrow.r](#) Ch 16 [arrow.r](#) Ch 17

**“I want to fine-tune a model”:** Ch 1 [arrow.r](#) Ch 2 [arrow.r](#) Ch 3 [arrow.r](#) Ch 13

## 20.2 Glossary

Key terms used throughout this book, listed alphabetically.

**Agent** — An LLM that can use tools (functions, APIs, databases) to take actions beyond text generation. Agents decide which tools to call based on the user's question.

**Attention** — The mechanism that allows a transformer model to weigh the relevance of different parts of the input when producing each output token. Self-attention computes relationships between all pairs of tokens in a sequence.

**Base Model** — An LLM after pre-training but before any fine-tuning or instruction tuning. Base models are good at text completion but not at following instructions.

**BPE (Byte Pair Encoding)** — A subword tokenization algorithm that iteratively merges the most frequent pairs of characters or tokens. Used by GPT models.

**Cascade (llm\_cascade)** — The automatic provider fallback system used in this book. When one LLM provider hits its quota, the cascade automatically tries the next available provider.

**ChromaDB** — An open-source vector database used for storing and searching embeddings. Used in several RAG chapters of this book.

**CLIP (Contrastive Language-Image Pre-training)** — A model by OpenAI that embeds both text and images into the same vector space, enabling cross-modal retrieval. Used in the Image RAG and Video RAG chapters.

**Cosine Similarity** — A measure of how similar two vectors are, computed as the cosine of the angle between them. Ranges from -1 (opposite) to 1 (identical). Used in RAG to find the most relevant document chunks for a query.

**Cypher** — A query language for graph databases, analogous to SQL for relational databases. Used with Kùzu in the Graph RAG chapter.

**Diffy** — A visual LLM application builder that supports workflows, chatflows, knowledge bases, and tool integration. Used in the Deployment chapter.

**Embedding** — A dense vector representation of a token, word, sentence, or document. Embeddings capture semantic meaning: similar concepts have similar vectors.

**FAISS (Facebook AI Similarity Search)** — A library for efficient similarity search of dense vectors. Used in the LangChain chapter.

**Fine-Tuning** — Updating a pre-trained model's weights on a specific dataset to teach it new knowledge or behavior. Contrasts with RAG, which retrieves knowledge at query time.

**GGUF** — A file format for quantized LLM weights optimized for CPU inference. Used by LM Studio and Ollama.

**GloVe (Global Vectors)** — A pre-trained word embedding model that captures semantic relationships from co-occurrence statistics. Used in the Embeddings notebook.

**Google Cloud Run** — A managed container platform on Google Cloud that autoscales from zero to many instances. Used in this book to host the MCP server as a public HTTPS endpoint. Each deploy ships a Docker container built automatically from the source folder.

**Hallucination** — When an LLM generates confident but factually incorrect information. RAG and tool use are designed to reduce hallucination by grounding answers in retrieved evidence.

**Haversine Formula** — A formula for computing the great-circle distance between two points on a sphere given their latitudes and longitudes. Used in the KU Parking Assistant.

**JSON-RPC** — A remote procedure call protocol encoded in JSON. The MCP protocol uses JSON-RPC 2.0 for communication between clients and tool servers.

**LCEL (LangChain Expression Language)** — LangChain’s pipe syntax for composing chains. Uses the `|` operator: `retriever | format_docs | prompt | llm | parser`.

**LlamaHub** — A registry of 300+ data loaders for LlamaIndex, supporting sources like Google Drive, Notion, GitHub, databases, and local files.

**LLM (Large Language Model)** — A neural network trained on vast text data to predict the next token. Examples: GPT-4, Gemini, Claude, Llama.

**LM Studio** — A desktop application for downloading, managing, and running open-weight LLMs locally with a visual interface.

**LoRA (Low-Rank Adaptation)** — A parameter-efficient fine-tuning technique that injects small trainable matrices into model layers, training only ~0.5% of parameters while keeping the rest frozen.

**MCP (Model Context Protocol)** — A standard protocol for exposing tools to LLMs. MCP servers provide tool discovery ( `tools/list` ) and execution ( `tools/call` ) over a standard interface.

**Multimodal** — Models or systems that process multiple types of input (text, images, audio, video) using the same architecture.

**NF4 (NormalFloat4)** — A 4-bit quantization format optimized for fine-tuning. Used in QLoRA to compress model weights to 25% of their original size.

**Ollama** — A command-line tool for running open-weight LLMs locally. Exposes an OpenAI-compatible API server.

**Prompt Engineering** — The practice of crafting input prompts to elicit desired behavior from an LLM. Includes techniques like few-shot examples, chain-of-thought, and system prompts.

**QLoRA (Quantized Low-Rank Adaptation)** — Combines 4-bit quantization with LoRA to enable fine-tuning of large models on consumer GPUs. Covered in the Fine-Tuning chapter.

**Quantization** — Compressing model weights from higher precision (FP16/FP32) to lower precision (INT4/INT8) to reduce memory usage and increase inference speed.

**RAG (Retrieval-Augmented Generation)** — A technique that retrieves relevant documents from a knowledge base and includes them in the LLM prompt, grounding the model’s answer in evidence rather than parametric memory.

**ReAct (Reasoning + Acting)** — An agent pattern where the LLM reasons about which tool to call, calls it, observes the result, and repeats until it has enough information to answer.

**RLHF (Reinforcement Learning from Human Feedback)** — A post-training technique where human evaluators rank model outputs, and a reward model trains the LLM to produce higher-ranked responses.

**RunnableLambda** — A LangChain wrapper that converts a plain Python function into a Runnable, enabling it to participate in LCEL pipe chains.

**SFT (Supervised Fine-Tuning)** — Training a base model on curated prompt-response pairs to teach it instruction following. The first step of post-training.

**StateGraph** — LangGraph’s core abstraction: a directed graph where nodes are processing functions and edges define the flow of a typed state dictionary.

**System Prompt** — Instructions given to an LLM that define its role, behavior, and constraints. Separate from the user’s message.

**TF-IDF (Term Frequency-Inverse Document Frequency)** — A classical text representation that assigns weights to words based on their frequency in a document relative to the corpus. High weight = important to that specific document.

**Token** — The fundamental unit of text that an LLM processes. A token may be a word, subword, or character, depending on the tokenizer.

**Transformer** — The neural network architecture behind all modern LLMs. Introduced in the 2017 paper “Attention Is All You Need.” Uses self-attention to process sequences in parallel.

**TypedDict** — A Python typing construct that defines a dictionary with specific key names and value types. Used in LangGraph to define the state that flows through a graph.

**Vector Database** — A database optimized for storing and searching high-dimensional vectors (embeddings). Examples: ChromaDB, Pinecone, FAISS, Weaviate, Milvus, LanceDB.

**Vibe Coding** — Building software by describing intent in natural language and letting an AI agent write the code. Coined by Andrej Karpathy in 2025.

**Word2Vec** — An algorithm that learns word embeddings by predicting context words. Demonstrates that semantic relationships are captured as vector arithmetic (e.g., king - man + woman  $\approx$  queen).

## 20.3 Common Errors and Fixes

This appendix collects the most frequently encountered errors when running the book's notebooks on Google Colab, along with their solutions.

### 20.3.1 `ModuleNotFoundError: No module named 'X'`

**Cause:** A required package is not installed, or the pip install cell was not run.

**Fix:** Run the pip install cell at the top of the notebook. If the error persists after installation, restart the runtime (Runtime arrow → Restart runtime) so the new package is loaded into memory.

### 20.3.2 `numpy` version conflicts ( `cannot import name '_center' from 'numpy._core.umath'` )

**Cause:** The `-U` (upgrade) flag in pip install upgraded numpy to version 2.x, which is incompatible with older versions of `transformers` or `sentence-transformers`.

**Fix:** Remove the `-U` flag from the pip install command so Colab's pre-installed numpy stays at a compatible version. If already broken, run:

```
!pip install numpy==1.26.4
```

Then restart the runtime.

### 20.3.3 `transformers` / CLIP errors ( `ModuleNotFoundError: Could not import module 'CLIPProcessor'` )

**Cause:** Version conflict between `transformers` and `sentence-transformers`, or a too-new version of `transformers` that changed the CLIP import paths.

**Fix:** Pin the transformers version:

```
!pip install transformers==4.46.3
```

Restart the runtime after installing.

### 20.3.4 Gemini 429 / quota exhausted ( `resource_exhausted` , `429` , `rate limit` )

**Cause:** You exceeded the free-tier rate limit for your LLM provider (e.g., Gemini's 500 requests/day or Groq's 30 requests/minute).

**Fix:** The `llm_cascade` package automatically falls back to the next available provider. If all providers are exhausted, wait a few minutes and try again. To avoid this during demos, uncomment only one question at a time rather than running all questions in a single cell.

### 20.3.5 `401 Unauthorized` / `expired_api_key`

**Cause:** An API key stored in Colab Secrets is expired, revoked, or invalid.

**Fix:** Go to Colab Secrets (🔑 icon in the left sidebar), update or remove the expired key. The `llm_cascade` package treats auth errors as retrievable and will automatically try the next provider.

### 20.3.6 No LLM provider configured. Set at least one API key.

**Cause:** No API keys are detected. Either no keys are set in Colab Secrets, or the "Notebook access" toggle is off.

**Fix:**

1. Click the 🔑 icon in Colab's left sidebar
2. Add at least one key (e.g., `GEMINI_API_KEY`)
3. Toggle "Notebook access" ON for that key (easy to miss)
4. Re-run the `llm = get_cascade()` cell

### 20.3.7 Gemini returns empty or truncated answers

**Cause:** Gemini 2.5 Flash has "thinking mode" enabled by default, which consumes tokens on internal reasoning before producing visible output. Long prompts may exhaust the token budget during thinking, leaving nothing for the actual answer.

**Fix:** The `llm_cascade` package (v0.1.0+) disables thinking mode automatically via `ThinkingConfig(thinking_budget=0)`. If you are using an older version, update:

```
!pip install --no-cache-dir git+https://github.com/KarAnalytics/llm_cascade.git
```

### 20.3.8 `NotImplementedError` on `llm.bind_tools(tools)`

**Cause:** The LangChain `ChatOpenAI` wrapper for some providers (especially Gemini via the OpenAI-compatible endpoint) does not support native tool binding.

**Fix:** Use the prompt-based tool-calling pattern instead of `bind_tools()`. The LangChain notebook demonstrates this approach: the LLM outputs a JSON action string, and a Python loop parses and executes it.

### 20.3.9 `SyntaxError: unterminated string literal in notebook cells`

**Cause:** A `\n` inside a Python string was stored as a literal newline in the notebook's JSON source, breaking the string across lines.

**Fix:** Replace `\n` inside strings with `chr(10)` or use triple-quoted strings. If you encounter this in a downloaded notebook, re-download the latest version from the [code\\_demos repository](#).

### 20.3.10 `ChromaDB ValueError: An embedding function already exists`

**Cause:** A previous notebook run created a ChromaDB collection with the default embedding function, and a subsequent run tries to open it with an explicitly specified embedding function.

**Fix:** Delete the existing collection before creating a new one:

```
try:
 chroma_client.delete_collection(name=collection_name)
except Exception:
 pass
collection = chroma_client.create_collection(name=collection_name,
embedding_function=embedding_fn)
```

### 20.3.11 `FileNotFoundError` for data files

**Cause:** The notebook expects a data file (Excel, CSV, SQL) that hasn't been downloaded yet.

**Fix:** Run the download cell (usually a `!wget` or `urllib.request.urlretrieve` call) before the cell that reads the file. Some notebooks download from GitHub automatically; make sure those cells run successfully.

### 20.3.12 `torch` / CUDA errors on Colab

**Cause:** Colab's pre-installed PyTorch version was overwritten by a pip install that pulled in a CPU-only version.

**Fix:** Do not include `torch` in pip install commands on Colab — Colab already has PyTorch pre-installed with GPU support. Remove `torch` from the install line and restart the runtime.

### 20.3.13 Colab disconnects during long training (QLoRA notebook)

**Cause:** Colab's free tier disconnects after ~90 minutes of idle time or if the browser tab is inactive.

**Fix:** Keep the browser tab active during training. The QLoRA notebook with the 0.5B model and 15 epochs takes only ~2 minutes on a T4 GPU, so this is rarely an issue with the default settings.

### 20.3.14 `pip install` dependency resolver warnings

**Cause:** Colab's pre-installed packages have version conflicts with newly installed packages. The warnings look like:

```
ERROR: pip's dependency resolver does not currently take into account all the
packages that are installed.
```

**Fix:** These are warnings, not errors — the notebook will still run. They occur because Colab's environment has pre-installed packages (tensorflow, torch, etc.) with pinned versions that conflict with newer installations. As long as the notebook executes correctly, these warnings can be ignored.



## 20.4 What to Build Next

After working through the chapters, here are project ideas organized by difficulty and the chapters they build on. Each project combines concepts from multiple chapters, giving you practice integrating what you have learned.

### 20.4.1 Beginner Projects

**Company FAQ Bot** (Chapters 8, 12) Collect your company’s FAQ page or internal documentation as text files. Build a RAG pipeline that answers employee questions grounded in the actual documents. Start with the RAG First Principles approach, then upgrade to LlamaIndex for a production-ready version.

**Meeting Notes Summarizer** (Chapters 2, 5, 7) Take meeting transcripts (or record your own with a transcription tool) and build a system that summarizes key decisions, action items, and owners. Use `llm_cascade` for the LLM call and experiment with different prompt structures to get the best summaries.

**Product Review Analyzer** (Chapters 2, 8) Scrape product reviews from a public dataset, embed them, and build a system that answers questions like “What do customers complain about most?” and “What features do people love?” This is RAG applied to customer feedback.

### 20.4.2 Intermediate Projects

**SQL Business Intelligence Assistant** (Chapters 10, 16) Connect to a real database (your company’s data warehouse, a public dataset, or a Kaggle dataset loaded into SQLite) and build a text-to-SQL system. Users ask business questions in plain English; the system generates SQL, executes it, and explains the results.

**Internal Knowledge Base with Access Control** (Chapters 9, 12, 19) Build a RAG system over internal company documents with role-based access. HR documents are only retrievable by HR staff; engineering docs by engineers. Deploy as a Dify workflow with authentication. This teaches RAG at a production level.

**Competitive Intelligence Monitor** (Chapters 8, 9, 17) Scrape competitor websites and press releases periodically, chunk and embed them, and build a RAG system that answers questions like “What did Competitor X announce this quarter?” Use a local LLM for privacy if the competitive data is sensitive.

**Multi-Format Document Q&A** (Chapters 8, 9, 10, 11) Build a system that ingests PDFs, Word documents, spreadsheets, and images into a single searchable index. Use LlamaIndex’s data loaders for document ingestion, ChromaDB for storage, and a multimodal LLM for image-containing documents.

### 20.4.3 Advanced Projects

**Autonomous Research Agent** (Chapters 17, 18) Extend the Business Idea Validator from Chapter 17 with real tools: web search, database queries, and document retrieval. The agent should plan its own research, execute each step with actual data sources, synthesize findings, and self-critique. Add an iteration loop so it can refine its plan based on its own critique.

**Customer Support Multi-Agent System** (Chapters 16, 18, 19) Build a multi-agent customer support system where:

- Agent 1 classifies the ticket (billing, technical, account)
- Agent 2 retrieves relevant knowledge base articles (RAG)
- Agent 3 drafts a response
- Agent 4 checks the response for accuracy and tone Deploy as an MCP server on Google Cloud Run so it can be called from a web app or Dify.

**Fine-Tuned Domain Expert** (Chapters 8, 15, 16) Fine-tune a small model (Qwen 0.5B or 1.5B) on domain-specific Q&A pairs from your industry. Compare its answers to RAG-based answers from the same knowledge base. When does fine-tuning win? When does RAG win? Build a hybrid system that uses both.

**Campus Services App with MCP** (Chapters 18, 19) Extend the KU Parking Assistant pattern to other campus services: dining hall menus, library hours, bus schedules, event calendars. Each service is a separate MCP tool. Build a single agent that can answer any campus question by calling the appropriate tool. Deploy the tools on Google Cloud Run and build a Dify frontend.

**AI-Powered Business Dashboard** (Chapters 7, 10, 13, 16) Connect to a business database, build text-to-SQL for ad-hoc queries, add RAG over company reports for qualitative insights, and create a Gradio or

Streamlit dashboard that combines both. Users can ask “What were Q3 sales by region?” (SQL) and “What does our annual report say about expansion plans?” (RAG) in the same interface.

#### 20.4.4 Tips for Project Success

**Start small, then expand.** Get a minimal working version first (one data source, one question type, one agent). Then add complexity: more data, more agents, better error handling, a nicer UI.

**Use vibe coding.** Feed any notebook from this book to Claude Code, Cursor, or Antigravity and say “Adapt this for [your use case].” The AI will restructure the code for your specific data and requirements.

**Document your prompts.** The most important artifact in any LLM project is not the code — it is the system prompts. Keep a log of which prompts worked and which did not, and why.

**Measure, do not assume.** Always compare your AI system’s output against a known-correct answer for at least 10-20 test cases. Accuracy on your test set is the only metric that matters — not how impressive the demo looks.