Chapter 14: Retrieval, Agents, and Multimodal Models

Learning Objectives

After reading this chapter, the reader should be able to:

1. Design and implement a Retrieval-Augmented Generation (RAG) pipeline – including document chunking, embedding-based retrieval, and grounded generation – and evaluate it against a closed-book baseline on a knowledge-intensive QA task.
2. Explain the ReAct agent framework (Reason + Act), implement a multi-step agent loop that interleaves reasoning traces with tool calls, and identify the challenges of planning, error recovery, and termination in agentic systems.
3. Describe how vision-language models (CLIP, LLaVA, GPT-4V) bridge visual and textual modalities through contrastive pre-training and visual instruction tuning, and explain their connection to the text-prediction paradigm.
4. Apply parameter-efficient fine-tuning (LoRA) and inference-time optimization (quantization, KV-cache, speculative decoding) to deploy large language models under real-world resource constraints.

In Chapter 13, we discovered that a frozen language model can perform new tasks purely through prompting – no gradient updates, no fine-tuning, just the right words in the right order placed before the test input. That chapter closed with a decision framework: prompt when flexibility matters, fine-tune when consistency matters, and retrieve when knowledge matters. But we left the third option unexplored. We showed that prompting has limits – the model can only condition on what fits in its context window, can only reason over knowledge baked into its parameters, and can only produce text. What happens when the user asks about a document published yesterday? What if solving the problem requires a calculator, a database query, and three web searches executed in sequence? What if the input is a photograph, not a sentence? The model, remember, still just predicts the next token. It has no persistent memory, no ability to take actions in the world, no eyes. Everything else – the retrieval, the tool calls, the image understanding – is scaffolding we build around that prediction engine. This chapter is about the scaffolding.

We proceed in five sections that trace a natural arc from knowledge to action to perception to efficiency. Section 14.1 introduces Retrieval-Augmented Generation (RAG), which grounds the model's predictions in external documents it has never seen during training – the open-book exam for language models. Section 14.2 covers agents and planning, where the model becomes a component in a loop that reasons, acts, observes, and iterates. Section 14.3 extends the prediction paradigm to vision, showing how models like CLIP and LLaVA condition next-token prediction on images alongside text. Section 14.4 addresses the context-length bottleneck, surveying the techniques that push context windows from thousands to millions of tokens. Section 14.5 covers the engineering that makes all of this deployable: quantization, LoRA, KV-cache optimization, and speculative decoding. Throughout, we maintain the thread that has run through this entire book since Chapter 1: the fundamental operation never changes. The model predicts the next token. What changes, chapter by chapter, is what it conditions on.

14.1 Retrieval-Augmented Generation

How can a model answer questions about documents it has never seen?

Every language model is, in a sense, a compressed snapshot of its training data. The parameters of a 70-billion-parameter model encode statistical regularities extracted from trillions of tokens of text, and when prompted with a question, the model generates an answer by sampling from the distribution learned during training. This arrangement has a fundamental flaw that no amount of scaling will fix: the snapshot is frozen. A model trained in January 2024 knows nothing about events in February 2024. A model trained on English Wikipedia does not know the contents of a company's internal knowledge base. A model trained on the entire internet still cannot distinguish between facts it has reliably encoded and plausible-sounding fabrications it is generating on the fly – a phenomenon the field calls *hallucination*. The cross-entropy training objective, as we discussed in Chapter 2, does not teach the model to say "the answer is unknown." It teaches the model to assign high probability to likely continuations, and a confidently wrong continuation often looks, statistically, exactly like a confidently right one. Retrieval-Augmented Generation (RAG), introduced by Lewis et al. [Lewis et al., 2020], addresses all three limitations by conditioning generation on dynamically retrieved documents: the knowledge is current (from an updatable index), verifiable (the source document can be cited), and specific (the exact passage is provided to the model at inference time).

14.1.1 Why Retrieval? Grounding and Hallucination Reduction

The hallucination problem is not a bug in any specific model; it is a structural consequence of how language models are trained and deployed. Consider asking a state-of-the-art model: "What was the closing price of Apple stock yesterday?" The model has no mechanism to access real-time financial data. Its training data has a cutoff date, and even within that cutoff, stock prices are the kind of rapidly changing, high-precision information that language models encode poorly. Yet the model will almost certainly generate a specific number – fluently, confidently, and incorrectly. The root cause is that the model's training objective, next-token prediction via cross-entropy minimization, rewards fluency and statistical plausibility but provides no signal for factual accuracy. A training example that reads "Apple closed at $187.32" and one that reads "Apple closed at $192.47" look structurally identical to the loss function; both are fluent English sentences with plausible numerical values. The model has no way to know which price corresponds to which date, and at inference time, it has no way to check. This is the fundamental gap that retrieval fills.

RAG reframes the generation problem as an open-book exam rather than a closed-book exam. In a closed-book exam, the student must answer from memory alone – and if their memory is incomplete, outdated, or simply wrong, they confabulate. Sound familiar? In an open-book exam, the student has access to reference materials and can look up specific facts before composing their answer. This dramatically reduces errors from memory failures, though it does not prevent misreading the reference materials or ignoring them entirely. The analogy maps directly to language models. A closed-book language model generates answers purely from its parametric knowledge – the information encoded in its weights during training. A RAG-augmented model first retrieves relevant documents from an external knowledge base and then generates its answer conditioned on both the query and the retrieved context. If the answer is in the retrieved documents, the model can ground its response in specific evidence. If the retrieval returns nothing relevant, the model can, in principle, indicate uncertainty – though in practice, faithfulness to retrieved context must be explicitly encouraged through prompt design or fine-tuning. The key point is that RAG decouples the model's knowledge from its parameters. The model provides the reasoning and

language generation capabilities; the knowledge base provides the facts. When the facts change – a new paper is published, a product catalog is updated, a policy is revised – we update the knowledge base without retraining the model. This separation of concerns has made RAG the dominant architecture for knowledge-intensive enterprise applications since 2023.

14.1.2 The RAG Pipeline: Index, Retrieve, Generate

The RAG architecture comprises three stages that execute in sequence for every query: indexing (an offline preprocessing step that builds a searchable knowledge base), retrieval (an online step that finds relevant documents for a given query), and generation (the final step where the language model produces an answer conditioned on the query and the retrieved context). We examine each stage in detail, because the quality of a RAG system depends critically on decisions made at every stage, and failure at any stage propagates to the final output.

The indexing stage transforms a raw document collection into a searchable vector index. The process begins with *chunking*: splitting documents into passages of 256 to 512 tokens each, typically with 10–20% overlap between consecutive chunks to preserve context at chunk boundaries. The chunk size is a consequential design choice — and the tokenization method from Chapter 10 directly affects it, since chunk boundaries are measured in tokens, not characters. Chunks that are too small lose surrounding context – a sentence about "the company's Q3 results" means nothing without knowing which company is being discussed. Chunks that are too large dilute the signal – a 2,000-token chunk may contain one relevant sentence buried in twenty irrelevant ones, and when this chunk is retrieved, the language model must locate and extract the relevant information from a sea of noise. After chunking, each passage is mapped to a dense vector representation using a bi-encoder model – a neural network, typically a fine-tuned Transformer, that maps variable-length text to a fixed-dimensional embedding. Popular choices include the sentence-transformers family of models (e.g., all-MiniLM-L6-v2) that produce 384-dimensional embeddings, and larger models like E5 or BGE that produce 768- or 1024-dimensional embeddings. The resulting vectors are stored in a vector index that supports fast Approximate Nearest Neighbor (ANN) search – systems like FAISS, Pinecone, Weaviate, or Qdrant that can search millions of vectors in milliseconds.

The retrieval stage operates at query time. Given a user query x, we embed it using the same encoder that was used for document indexing (this symmetry is essential – the query and document embeddings must inhabit the same vector space) and compute the cosine similarity between the query embedding and every document embedding in the index:

$$\text{sim}(x, d) = \frac{\mathbf{e}_q(x)^\top \mathbf{e}_d(d)}{\|\mathbf{e}_q(x)\| \, \|\mathbf{e}_d(d)\|} \tag{14.1}$$

where $\mathbf{e}_q$ is the query encoder and $\mathbf{e}_d$ is the document encoder. In a bi-encoder architecture, these are often the same model applied to different inputs, though in some systems they are separate models fine-tuned for their respective roles. The retrieval step returns the top-k documents (typically $k = 3$ to 10) ranked by similarity score. In practice, the brute-force computation of similarity against every indexed document is replaced by ANN search algorithms (HNSW, IVF, or product quantization) that trade a small amount of recall for orders-of-magnitude speedups, enabling retrieval over corpora of millions of documents in under 100 milliseconds.

The generation stage takes the retrieved documents and constructs an augmented prompt for the language model. The standard approach concatenates the retrieved passages with the user query in

a template like: "Context: [retrieved documents]. Question: [user query]. Answer based on the context above." The language model then generates an answer conditioned on this augmented input. Formally, the RAG generation probability marginalizes over the retrieved documents:

$$P(y \mid x) = \sum_{d \in \mathcal{D}_k} P(y \mid x, d; \theta)\, P(d \mid x; \phi) \tag{14.2}$$

where $\mathcal{D}_k$ is the set of top-k retrieved documents, $P(d \mid x; \phi)$ is the retrieval probability (a softmax over the similarity scores from Equation 14.1), and $P(y \mid x, d; \theta)$ is the generator's probability of producing output y conditioned on both the query x and a retrieved document d. A common misconception is that RAG requires training the retriever and generator jointly end-to-end. In practice, most production RAG systems use off-the-shelf components and approximate the marginalization by simply concatenating all top-k documents into the prompt and generating once, rather than generating separately for each document and combining – the computational savings are substantial and the quality difference is negligible for most applications.

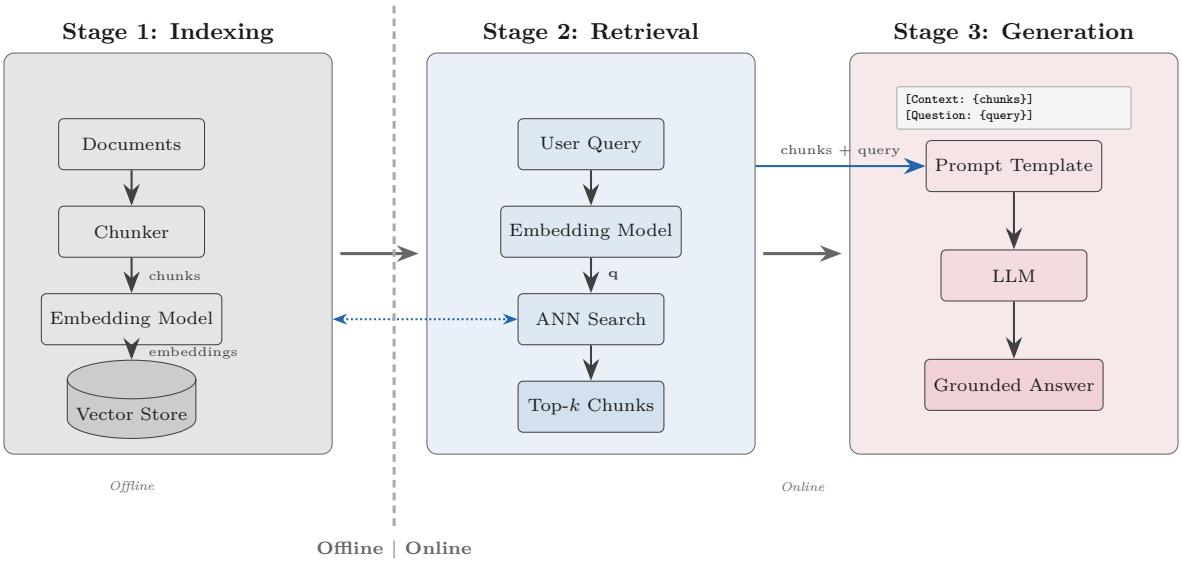


Figure 1: Figure 14.1 – RAG Pipeline Architecture

The following code implements a minimal RAG pipeline that demonstrates the index-retrieve-generate pattern. The embeddings are simulated as random vectors for self-containment, but in a production system, they would be produced by a sentence-transformer model.

```
import numpy as np

class SimpleRAG:
    """Minimal RAG: embed documents, retrieve by cosine sim, generate."""
    def __init__(self, docs, embed_fn, generate_fn, k=3):
        self.docs = docs
        self.generate_fn = generate_fn
        self.k = k
```

```
        # Index: embed all documents
        self.doc_embeddings = np.array([embed_fn(d) for d in docs])

    def retrieve(self, query_emb):
        """Find top-k documents by cosine similarity."""
        sims = self.doc_embeddings @ query_emb
        norms = np.linalg.norm(self.doc_embeddings, axis=1)
        sims /= (norms * np.linalg.norm(query_emb) + 1e-8)
        top_k = np.argsort(sims)[-self.k:][::-1]
        return [(self.docs[i], float(sims[i])) for i in top_k]

    def query(self, question, embed_fn):
        q_emb = embed_fn(question)
        retrieved = self.retrieve(q_emb)
        context = "\n".join([f"[Doc]: {doc}" for doc, _ in retrieved])
        prompt = f"Context:\n{context}\n\nQuestion: {question}\nAnswer:"
        return self.generate_fn(prompt), retrieved

# Demo with random embeddings (replace with real model in practice)
np.random.seed(42)
fake_embed = lambda text: np.random.randn(64)
fake_generate = lambda prompt: "Generated answer based on context."
docs = ["Paris is the capital of France.", "Berlin is in Germany.",
        "The Eiffel Tower is 330m tall.", "Python was created in 1991."]
rag = SimpleRAG(docs, fake_embed, fake_generate, k=2)
answer, sources = rag.query("What is the capital of France?", fake_embed)
print(f"Answer: {answer}")
print(f"Sources: {[doc for doc, score in sources]}")
```

14.1.3 Evaluation and Advanced Retrieval Strategies

Evaluating a RAG system requires measuring three dimensions independently, because a system can succeed on one dimension while failing on another. The first dimension is *retrieval quality*: did the retriever find the right documents? This is measured by standard information retrieval metrics – Recall@k (is the gold-standard document among the top-k retrieved documents?), Mean Reciprocal Rank or MRR (what is the average rank of the first relevant document?), and precision@k (what fraction of the top-k documents are relevant?). The second dimension is *faithfulness*: is the generated answer consistent with the retrieved documents? A system with perfect retrieval can still fail if the generator ignores the retrieved context and hallucinates an answer from its parametric knowledge. Faithfulness can be measured by natural language inference (NLI) models that assess whether the answer is entailed by the retrieved context, or by human evaluation. The third dimension is *answer correctness*: is the answer factually right? This is the bottom-line metric, measured by exact match or token-level F1 against gold-standard answers. The important insight is that these dimensions are partially independent: a system can retrieve the right documents but generate an unfaithful answer (the generator ignored the evidence), or it can retrieve irrelevant documents but still generate a correct answer (the generator relied on parametric knowledge). Comprehensive RAG evaluation requires measuring all three dimensions to diagnose where failures occur and where improvements should be directed.

Advanced retrieval strategies address the limitations of the basic retrieve-and-generate pipeline. *Re-ranking* is the most common enhancement: after the bi-encoder retrieves the top-k candidates (where k might be 20 or 50), a more powerful cross-encoder re-scores each query-document pair by processing them jointly through a single Transformer. The cross-encoder is more accurate because it can model fine-grained interactions between query and document tokens, but it is too slow for the initial retrieval stage (it would need to process every document in the corpus). The two-stage retrieve-then-rerank pipeline combines the speed of the bi-encoder with the accuracy of the cross-encoder. *Iterative retrieval* addresses the problem of complex queries that cannot be answered by a single retrieval step: the system generates a partial answer, uses it to formulate a new query, retrieves additional documents, and refines the answer – essentially applying the ReAct pattern (Section 14.2) to the retrieval process. *Query decomposition* breaks complex queries into sub-queries that are each easier to retrieve for: "Compare the economic policies of France and Germany in 2023" becomes "economic policy France 2023" and "economic policy Germany 2023," with separate retrieval for each sub-query and a synthesis step that combines the results. These advanced strategies improve RAG quality significantly but add latency and complexity, and the right combination depends on the application's requirements.

Sidebar: The RAG vs. Fine-Tuning Decision

When organizations first adopt large language models, they face a recurring question: should we fine-tune the model on our proprietary data, or should we retrieve and inject that data at inference time via RAG? The answer, as of 2024, is "it depends – and often both." Fine-tune when: (1) the task requires a specific output format or style that the base model does not produce reliably through prompting alone; (2) latency is critical and the overhead of retrieval is unacceptable; or (3) the knowledge is static and well-defined, such as a medical coding system or legal taxonomy. Use RAG when: (1) the knowledge base changes frequently – product catalogs, news feeds, internal documentation; (2) the user needs citations and source attribution; (3) the corpus is too large to encode in model weights, even with aggressive fine-tuning; or (4) the organization cannot justify the cost of fine-tuning infrastructure. The best enterprise systems in practice combine both: a fine-tuned model handles format, tone, and domain-specific reasoning, while RAG supplies up-to-date factual content with verifiable sources. The question is never "RAG or fine-tuning" but rather "how much of each."

14.2 Agents and Planning

What happens when the model needs to act in the world, not just generate text?

Everything we have discussed so far – pre-training, fine-tuning, prompting, retrieval – operates within a single inference cycle. The user provides an input, the model (possibly augmented with retrieved context) generates an output, and the interaction ends. But many real-world tasks cannot be completed in a single step. "What is the population of the capital of France?" requires two lookups: first the capital, then its population. "Book the cheapest flight from New York to London next Tuesday" requires searching multiple airlines, comparing prices, selecting an option, and completing a purchase. "Debug this Python script" might require reading the code, running it, examining the error message, formulating a hypothesis, editing the code, and running it again. These are multi-step tasks that demand reasoning, action, observation, and iteration – capabilities that a single-turn language model does not possess on its own. The emerging solution is to place

the language model inside a loop: the model reasons about what to do next, generates an action (a tool call, a search query, a code execution), receives the result, and decides whether to continue or stop. This is the *agent* paradigm, and it represents the most ambitious extension of the prediction framework we have seen in this book.

14.2.1 The ReAct Framework: Thought-Action-Observation

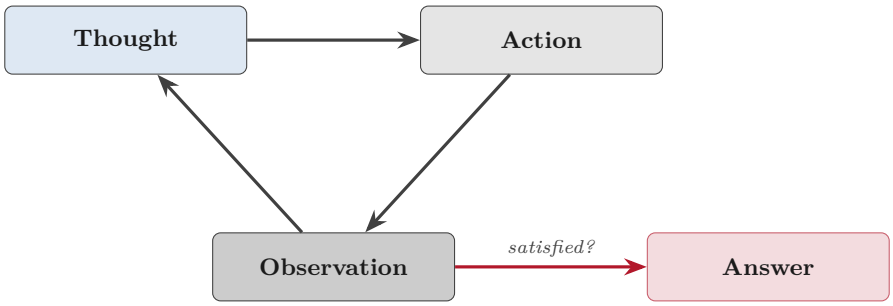
The ReAct framework, introduced by Yao et al. [Yao et al., 2023], structures the agent loop as an interleaving sequence of three components: *Thought* (a verbal reasoning trace where the model plans its next step), *Action* (a tool call or information request executed by an external orchestrator), and *Observation* (the result returned by the environment). The name "ReAct" is a portmanteau of "Reasoning" and "Acting," and the framework's core claim is that combining both outperforms either alone. Chain-of-thought prompting (Chapter 13) provides reasoning without action: the model thinks step by step but cannot access external tools or verify its intermediate conclusions. Tool-use prompting provides action without reasoning: the model calls tools but does not articulate why. ReAct interleaves both, producing a trace that is interpretable (we can read the model's reasoning), verifiable (we can check the tool outputs), and self-correcting (the model can adjust its plan based on what it observes).

A concrete example makes the framework tangible. Suppose the user asks: "What is the population of the capital of France?" A ReAct agent processes this as follows. *Thought 1:* "The task requires finding the capital of France first, then looking up its population." *Action 1:* search("capital of France"). *Observation 1:* "Paris is the capital of France." *Thought 2:* "The capital is Paris. The next step is to find the population of Paris." *Action 2:* search("population of Paris"). *Observation 2:* "The population of Paris is approximately 2.1 million in the city proper." *Thought 3:* "All necessary information has been gathered." *Answer:* "The population of the capital of France (Paris) is approximately 2.1 million." Each thought, action, and observation is generated or received in sequence. The thoughts and actions are produced by the language model; the observations are returned by external tools (in this case, a search function). The orchestrator – the software that manages the loop – is responsible for parsing the model's output into thoughts and actions, routing actions to the appropriate tools, collecting observations, and feeding them back to the model as additional context. The model decides when to stop by generating a final answer instead of another action.

The following code implements a minimal ReAct agent loop. The language model and tools are represented as callable functions that would, in practice, be API calls to an LLM service and external tool endpoints.

```
import re

def react_agent(question, tools, llm, max_steps=5):
    """Simple ReAct loop: Thought -> Action -> Observation."""
    context = f"Question: {question}\n"
    for step in range(max_steps):
        # LLM generates Thought + (Action or Answer)
        response = llm(context + "Thought:")
        context += f"Thought: {response}\n"
        # Check for final answer
        answer_match = re.search(r"Answer:\s*(.+)", response)
```



Worked Example:

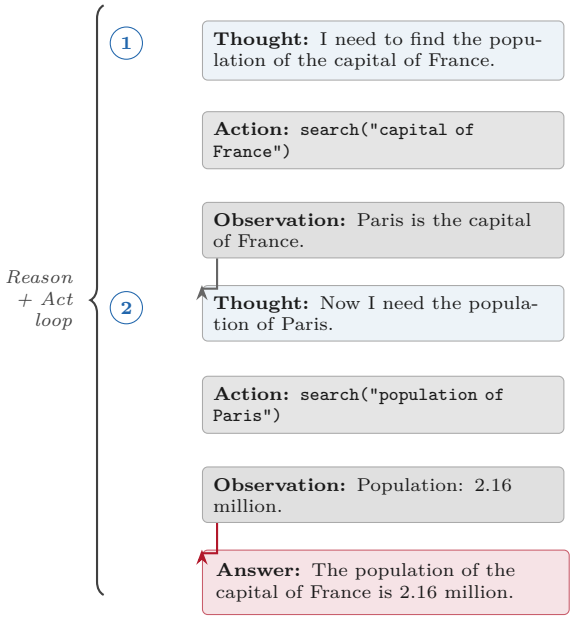


Figure 2: Figure 14.2 – The ReAct Agent Loop

```
        if answer_match:
            return answer_match.group(1).strip()
        # Parse action: tool_name[tool_input]
        action_match = re.search(r"Action:\s*(\w+)\[(.+?)\]", response)
        if not action_match:
            return response  # fallback: return raw response
        tool_name, tool_input = action_match.groups()
        if tool_name in tools:
            observation = tools[tool_name](tool_input)
        else:
            observation = f"Error: unknown tool '{tool_name}'"
        context += f"Action: {tool_name}[{tool_input}]\n"
        context += f"Observation: {observation}\n"
    return "Max steps reached without final answer."

# Usage: tools = {"search": search_fn, "calc": calculator_fn}
# answer = react_agent("What is 2 * the population of Paris?", tools, llm)
# print(f"Final answer: {answer}")
```

14.2.2 Planning and Task Decomposition

The ReAct loop handles multi-step tasks where each step is relatively independent: look up a fact, compute a result, look up another fact. But many real-world tasks have a richer structure that requires *planning*: decomposing a high-level goal into ordered sub-tasks, some of which depend on the results of others, some of which can be executed in parallel, and some of which must be abandoned and replanned if they fail. Consider the request "Book a flight from New York to London and a hotel near the conference venue." This decomposes into at least six sub-tasks: search for flights, compare prices and schedules, book the selected flight, identify the conference venue address, search for nearby hotels, and book the selected hotel. Some of these can proceed in parallel (flight search and hotel search), while others are strictly sequential (you cannot book a hotel before knowing the venue address). The planning challenge is to generate this decomposition, order it correctly, and adapt when things go wrong – if the preferred flight is sold out, the agent must replan, perhaps adjusting the hotel dates as well.

Current LLM-based planning approaches typically prompt the model to produce a numbered plan as a structured list, then execute each step using the ReAct loop. More sophisticated systems use tree search over possible plans, evaluating each candidate step for feasibility before committing to it. The honest assessment, however, is that LLM planning remains one of the weakest capabilities of current models. Models frequently produce plans that read well but are logistically infeasible – they skip necessary steps, assume information that has not been obtained, or sequence steps in an impossible order. A model might plan to "confirm the hotel reservation" before it has searched for hotels, or "compare the two options" when only one option was retrieved. These failures are especially insidious because the plans *sound* reasonable to a human reader skimming them; the errors become apparent only during execution. For this reason, high-stakes planning tasks – travel booking, financial transactions, medical decision support – currently require human oversight at every critical decision point. Fully autonomous planning by language models is a goal, not a reality.

14.2.3 Memory, Error Recovery, and Safety

An agent that persists across multiple interactions needs memory: a mechanism to store and retrieve information beyond what fits in the current context window. Memory in agentic systems is typically organized into three tiers. *Short-term memory* is the conversation context itself – the sequence of thoughts, actions, and observations accumulated during the current task. This is bounded by the model's context window and is discarded when the conversation ends. *Medium-term memory* consists of conversation summaries or compressed representations that are appended to the context to extend the effective memory horizon – for example, summarizing the first 50 turns of a conversation into a paragraph that is prepended to the context for turn 51. *Long-term memory* uses an external store – a vector database, a key-value store, or a structured knowledge graph – where the agent can save facts, decisions, and lessons learned across sessions and retrieve them when relevant. Long-term memory is essentially RAG (Section 14.1) applied to the agent's own history rather than to an external document collection. The architecture of memory systems reveals a deeper truth about current agents: they have no built-in ability to learn from experience in the way humans do. Every memory mechanism is an external scaffold, a system we build around the model to compensate for the fact that its parameters do not update during deployment.

Error recovery is the second critical challenge for agentic systems. In a single-turn language model interaction, errors are contained: the model generates a wrong answer, the user notices, and the conversation moves on. In an agent loop, errors compound. If the agent misinterprets an observation at step 3, all subsequent reasoning is built on a false premise. If a tool call returns an error and the agent does not detect it, the agent may proceed as if the call succeeded, generating increasingly disconnected plans. Robust error recovery requires the agent to detect when something has gone wrong (the observation is empty, the tool returned an error message, the retrieved information contradicts the agent's assumptions), diagnose the cause (wrong tool, wrong query, wrong interpretation), and adapt (reformulate the query, try a different tool, backtrack to an earlier state). Current models handle simple error cases – retrying a failed search with different keywords – but struggle with deeper diagnostic reasoning, especially when the failure is subtle rather than explicit.

Safety is the third and most consequential concern. An agent with access to tools – email, web browsing, code execution, file management, API calls – can cause real-world harm if it takes incorrect or unauthorized actions. The harms range from minor (sending a poorly worded email) to severe (executing destructive code, making unauthorized financial transactions, or leaking sensitive information through tool calls). The emerging consensus in the agent safety community follows a principle borrowed from computer security: *least privilege*. An agent should have access to only the tools it needs for the current task, should require human approval for any irreversible action (sending, deleting, purchasing, posting), and should operate within a sandboxed environment where its actions can be monitored and rolled back. The kill switch – a mechanism for a human operator to immediately terminate an agent's execution – is not a luxury but a requirement. We return to the broader safety implications of autonomous systems in Chapter 15.

> **Sidebar: From Assistants to Autonomous Agents – How Far Can We Go?**
>
> The trajectory from chatbots to autonomous agents has been the most ambitious and most controversial direction in LLM research. In 2023, demonstrations like AutoGPT and BabyAGI captivated the public with agents that could, in principle, independently accomplish complex tasks: "build me a website," "plan my vacation," "write and debug this software." The reality was more sobering. Early agents spent most of their time in

error loops, hallucinated action plans that led nowhere, and produced worse results than a skilled human using a simple chat interface. By 2025, the field had matured into a more nuanced understanding: agents work well for structured, bounded tasks with clear success criteria – filling out forms, structured data extraction, multi-step search – but fail on open-ended, creative, or high-stakes tasks that require genuine judgment. The safety concerns are real and unresolved. An agent operating with minimal human oversight that has access to email, web, and code execution can cause harm that a passive text generator cannot. The responsible path forward is not to avoid agents but to constrain them: minimal authority, human approval for consequential actions, well-defined task boundaries, and robust termination conditions.

14.3 Vision-Language Models

What if the input is a photograph, not a sentence?

Up to this point in the book, every model we have discussed operates exclusively on text. The inputs are sequences of tokens, the outputs are sequences of tokens, and the internal representations are sequences of vectors derived from tokens. But the world is not made of text. Humans communicate through images, diagrams, charts, gestures, sounds, and combinations of all of these. A student asking for help with a math problem might photograph a handwritten equation. A doctor reviewing a medical case might upload a CT scan alongside the patient's history. A designer requesting feedback might share a screenshot of a user interface. If language models are to be truly useful as general-purpose reasoning systems, they must process visual information alongside text – and they must do so within the same prediction framework that has served us throughout this book. The insight that makes this possible, and that we explore in this section, is surprisingly simple: we do not need to redesign the language model. We need to give it eyes. That is, we need a way to convert visual information into the token embedding space that the language model already understands, so that the model can condition its next-token predictions on both textual and visual context. The model still predicts the next token; the input vocabulary simply grows to include visual features.

14.3.1 CLIP: Contrastive Image-Text Pre-training

Contrastive Language-Image Pre-training (CLIP), introduced by Radford et al. [Radford et al., 2021], established the foundation for modern vision-language models by learning a shared embedding space for images and text. The architecture is a dual encoder: one encoder (typically a Vision Transformer, or ViT) processes images into fixed-dimensional vectors, and another encoder (a standard Transformer) processes text into vectors of the same dimensionality. The two encoders are trained jointly on 400 million image-caption pairs scraped from the internet, with a contrastive objective that pushes matching image-caption pairs together and non-matching pairs apart in the shared embedding space. CLIP is not, strictly speaking, a language model – it does not predict the next token. But it solves a prerequisite problem that all subsequent vision-language models depend on: it creates a representation where images and text are commensurable, inhabiting the same geometric space and subject to the same similarity computations. Without CLIP or a system like it, there is no bridge between pixels and tokens.

The training objective is an instance of the InfoNCE contrastive loss applied symmetrically across a batch of N image-text pairs. For each image embedding $\mathbf{z}_i^I$ in the batch, the matching text

embedding $\mathbf{z}_i^T$ is the positive pair, and all other text embeddings $\mathbf{z}_j^T$ (where $j \neq i$) are negative pairs. The loss for the image-to-text direction is:

$$\mathcal{L}_{\text{CLIP}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{\exp(\text{sim}(\mathbf{z}_i^I, \mathbf{z}_i^T)/\tau)}{\sum_{j=1}^{N}\exp(\text{sim}(\mathbf{z}_i^I, \mathbf{z}_j^T)/\tau)} \tag{14.3}$$

where $\text{sim}(\mathbf{z}_i^I, \mathbf{z}_j^T) = \frac{(\mathbf{z}_i^I)^\top \mathbf{z}_j^T}{\|\mathbf{z}_i^I\|\|\mathbf{z}_j^T\|}$ is the cosine similarity, and τ is a learned temperature parameter that controls the sharpness of the softmax distribution. An identical loss is computed in the text-to-image direction (treating each text embedding as the anchor and the images as candidates), and the total loss is the average of both. This symmetric formulation ensures that images are close to their matching captions and captions are close to their matching images. The temperature τ is critical: a low temperature makes the softmax peaky, concentrating the learning signal on hard negatives (the most similar non-matching pairs), while a high temperature distributes the signal more uniformly. CLIP learns τ as a parameter alongside the encoder weights, letting the model automatically calibrate the difficulty of the contrastive task during training.

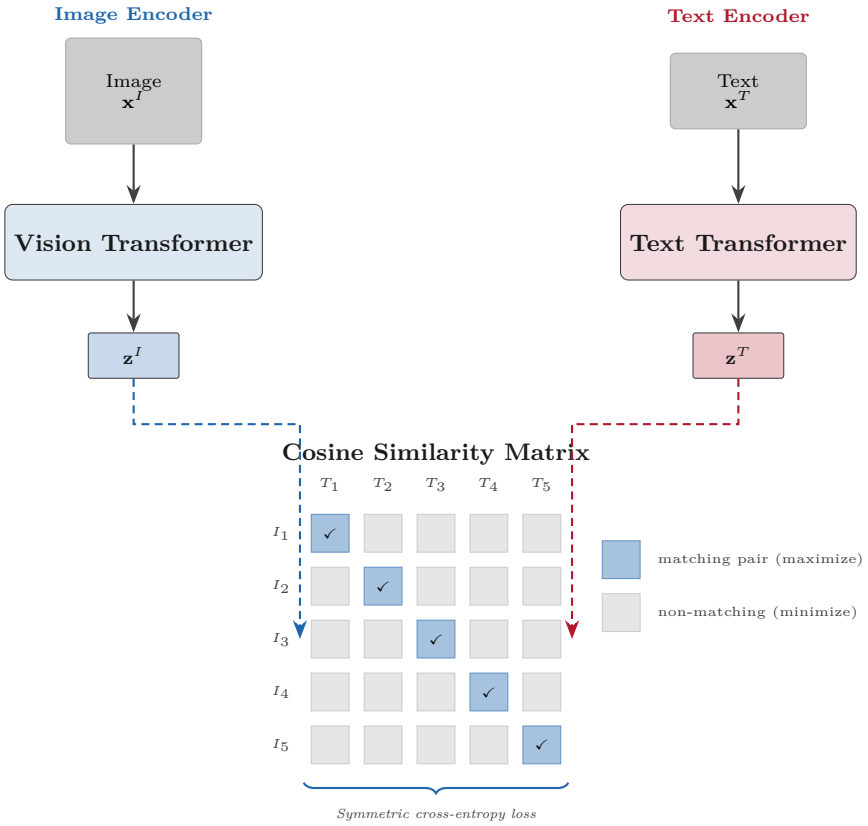


Figure 3: Figure 14.3 – CLIP Dual-Encoder Architecture

The practical consequence of CLIP's training is remarkable: zero-shot image classification without any task-specific training data. To classify an image into one of K categories, we construct K

text prompts of the form "a photo of a [class name]" (e.g., "a photo of a dog," "a photo of a cat," "a photo of a car"), embed each prompt with the text encoder, embed the image with the image encoder, and select the class whose text embedding has the highest cosine similarity to the image embedding. On ImageNet – the canonical image classification benchmark with 1,000 classes – CLIP achieved 76.2% top-1 accuracy *without seeing a single ImageNet training image*. This zero-shot transfer ability, learned entirely from the statistical association between web images and their captions, demonstrated that contrastive pre-training on noisy, web-scale data can produce visual representations that rival decades of supervised learning on curated datasets. For the purposes of this chapter, CLIP's primary role is as a component: it provides the visual encoder that subsequent models use to bridge the gap between images and language models.

14.3.2 Visual Instruction Tuning – LLaVA

CLIP gives us a shared embedding space for images and text, but it does not give us a model that can hold a conversation about an image. CLIP can tell you which caption best matches an image (or vice versa), but it cannot generate a detailed description, answer follow-up questions, or reason about the visual content in the way that a language model reasons about text. The gap between CLIP's capability (matching) and what we want (generation conditioned on vision) was bridged by LLaVA (Large Language and Vision Assistant), introduced by Liu et al. [Liu et al., 2023]. LLaVA's architecture is disarmingly simple: take a pre-trained CLIP visual encoder, take a pre-trained language model, and connect them with a trainable projection layer that maps the visual encoder's output features into the language model's input embedding space. The image features, once projected, are interleaved with the text token embeddings as if they were additional tokens. The language model then performs autoregressive generation conditioned on both the visual and textual tokens – predicting the next token in exactly the same way it would for a text-only input, but now with visual context embedded in the sequence. Nothing about the prediction mechanism changes. The model still predicts the next token given everything that came before; "everything that came before" simply now includes projected image features alongside text embeddings.

The training procedure has two phases. In the first phase, the visual encoder and the language model are both frozen, and only the projection layer is trained on image-caption pairs. This phase teaches the projection to produce visual tokens that the language model can interpret – essentially learning a translation from visual feature space to language embedding space. In the second phase, the projection layer and the language model are fine-tuned together on visual instruction data: question-answer pairs about images generated by GPT-4 (e.g., "What is happening in this image?" followed by a detailed description, or "How many people are in this photo?" followed by an answer). This phase teaches the model to follow visual instructions – to answer questions, describe scenes, read text in images, and interpret charts and diagrams. The two-phase approach is computationally efficient because the expensive visual encoder (CLIP ViT-L, with 300 million parameters) is never fine-tuned, and the language model is only fine-tuned in the second phase with parameter-efficient methods. The key insight for the narrative of this book is architectural: LLaVA demonstrates that a standard autoregressive language model can become multimodal with nothing more than a projection layer and instruction tuning. We do not need a fundamentally new architecture; we need a way to translate visual information into the representation space the model already operates in.

14.3.3 Frontier Multimodal Models

LLaVA demonstrated the viability of connecting visual encoders to language models. The frontier multimodal models – GPT-4V (OpenAI, 2023), Gemini (Google DeepMind, 2023), and Claude

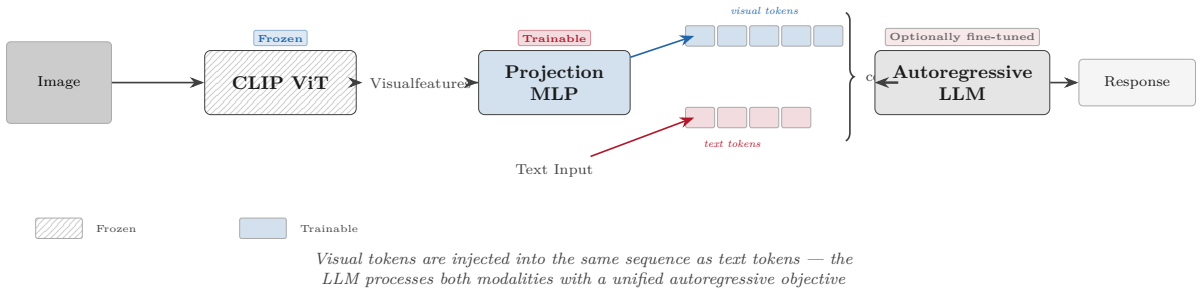


Figure 4: Figure 14.4 – LLaVA Architecture: Visual Instruction Tuning

3 (Anthropic, 2024) – have scaled this approach to production quality, processing interleaved image and text inputs natively, handling complex visual reasoning tasks, and generating text that reflects detailed understanding of visual content. These models represent the most general form of the prediction paradigm we have encountered: predict the next token given any combination of modalities in the context. The exact architectures are proprietary and have not been fully disclosed, but the general pattern follows the LLaVA template: a powerful visual encoder feeds features into a large language model, with training on massive multimodal datasets that include image-text pairs, visual question-answering data, chart and diagram understanding tasks, and interleaved image-text documents.

The capabilities of frontier multimodal models are genuinely impressive and, for anyone who remembers the state of computer vision five years ago, somewhat startling. These models can describe the contents of photographs in rich detail, answer questions about charts and graphs by reading axes and data points, perform optical character recognition (OCR) on handwritten and printed text, interpret diagrams and flowcharts, identify objects and their spatial relationships, and engage in multi-turn conversations about visual content. For a BSc student encountering these capabilities for the first time, it is worth pausing to appreciate what is happening under the hood: the model is still predicting the next token. When it "reads" a chart, it is conditioning on projected visual features and predicting text tokens that describe what those features represent. When it "counts" objects, it is translating spatial patterns in its visual representations into numerical text tokens. The prediction paradigm is unchanged; the input modality has expanded. That said, the limitations are real and worth cataloging honestly: frontier multimodal models still hallucinate visual content (describing objects that are not present in the image), struggle to count objects accurately (especially beyond five or six), make errors in spatial reasoning (confusing left and right, above and below), and fail on tasks that require fine-grained visual discrimination (distinguishing nearly identical objects or reading very small text). These models are remarkably capable, but they do not "see" the way humans see; they process visual features statistically, and their failures reveal the gap between statistical pattern matching and genuine visual understanding.

14.4 Long-Context Models

If we could give the model a million tokens of context, would RAG become obsolete?

The context window is the fundamental bottleneck of autoregressive language models. Every

technique we have discussed in this chapter – RAG, agents, multimodal inputs – generates additional context that must fit within the model's maximum sequence length. A RAG system that retrieves ten documents of 500 tokens each consumes 5,000 tokens of context before the model has even begun to generate. An agent executing ten steps accumulates thousands of tokens of thoughts, actions, and observations. A multimodal model processing a high-resolution image might consume hundreds of visual tokens. When the context window is 4,096 tokens (the original GPT-3 limit), these demands quickly become unmanageable. The push toward longer context windows – 32K, 128K, and ultimately 1 million tokens or more – is therefore not merely an engineering convenience but a qualitative expansion of what language models can do. However, longer context introduces its own challenges: position encodings that were trained for short sequences degrade at long distances, the quadratic cost of standard attention becomes prohibitive, and models exhibit troubling patterns in how they use (or fail to use) information across very long contexts.

14.4.1 Position Encoding Extrapolation

Transformer models trained with a fixed context length embed each token's position using positional encodings – learned or sinusoidal vectors that inform the attention mechanism about token order (Chapter 8). A model trained on sequences of 4,096 tokens has positional encodings defined for positions 0 through 4,095. At position 5,000, the encoding is undefined or out-of-distribution, and the model's behavior degrades. The challenge of extending context length beyond the training length is fundamentally a challenge of positional encoding extrapolation: making the model's notion of "where am I in the sequence?" work for positions it has never seen during training. Two families of approaches have emerged, corresponding to two philosophies about how positions should be represented.

Rotary Position Embeddings (RoPE), used by the LLaMA and Mistral model families, encode positions by rotating the query and key vectors in the attention mechanism by angles that are functions of position. The rotation frequencies are determined by a base parameter (originally 10,000). To extrapolate to longer sequences, RoPE scaling modifies this base parameter – effectively "compressing" the position IDs so that a 16K-length sequence maps to the same range of rotational angles that a 4K sequence would occupy in the original encoding. NTK-aware interpolation and YaRN (Yet Another RoPE Extension) refine this approach by applying different scaling factors to different frequency components: high-frequency components (which encode local position information) are preserved, while low-frequency components (which encode global position) are compressed. After scaling, the model requires a brief fine-tuning phase – typically 100 to 1,000 steps on long-context data – to adapt to the new positional scheme. The second approach, ALiBi (Attention with Linear Biases), used by the BLOOM and MPT model families, avoids learned positional encodings entirely. Instead, ALiBi adds a negative linear bias to the attention scores that is proportional to the distance between the query and key tokens: $\text{bias}_{i,j} = -m \cdot |i - j|$, where m is a per-head slope parameter. This bias naturally decays attention to distant tokens without any explicit positional encoding, and because the bias function is defined for any distance, ALiBi extrapolates to arbitrary sequence lengths without modification. The trade-off is that ALiBi's linear distance assumption may not capture complex positional relationships as flexibly as RoPE's rotational scheme, but its extrapolation properties are superior.

14.4.2 Sub-Quadratic Attention Alternatives

Standard self-attention computes a full $n \times n$ attention matrix, where n is the sequence length, resulting in $\mathcal{O}(n^2 d)$ time and $\mathcal{O}(n^2)$ memory complexity (Chapter 8). For $n = 4{,}096$, this is

manageable; for $n = 1{,}000{,}000$, it is catastrophic – the attention matrix alone would require terabytes of memory. Several families of sub-quadratic attention alternatives have been proposed, trading varying degrees of expressiveness for computational efficiency.

Sliding-window attention restricts each token to attend only to the nearest w tokens (a local window), plus a small number of global tokens that attend to and are attended by all positions. Mistral uses a sliding window of $w = 4{,}096$ tokens, which provides rich local context while allowing global tokens to propagate information across the full sequence. The complexity is $\mathcal{O}(nw)$ – linear in sequence length for a fixed window size. *Sparse attention* patterns generalize the sliding window by combining local attention with strided attention (attending to every k-th token at longer distances), as in Longformer and BigBird. These patterns capture both local coherence and long-range dependencies at sub-quadratic cost. *Linear attention* approximations replace the softmax in standard attention with kernel functions: instead of computing $\text{softmax}(\mathbf{QK}^\top)\mathbf{V}$, they compute $\phi(\mathbf{Q})(\phi(\mathbf{K})^\top\mathbf{V})$, where ϕ is a feature map. By rearranging the matrix multiplications (exploiting the associative property), this reduces complexity from $\mathcal{O}(n^2 d)$ to $\mathcal{O}(nd^2)$ – linear in n when $d \ll n$. Finally, *state-space models* like Mamba and S4 replace attention entirely with a recurrence that processes each token in constant time, achieving $\mathcal{O}(n)$ complexity with constant memory per token. These models are fundamentally different from Transformers and represent an active area of research that may reshape the architecture landscape in coming years. In practice, full attention with engineering optimizations like FlashAttention remains competitive up to 32K–128K tokens because modern GPU hardware is optimized for the dense matrix operations that attention requires, and the constant factors in sub-quadratic alternatives often outweigh their asymptotic advantages at moderate sequence lengths.

14.4.3 Practical Applications of Long Context

Long context windows of 128K to 1 million tokens enable qualitatively new applications that are impossible with shorter context. *Codebase understanding* allows a developer to upload an entire repository – tens of thousands of lines across hundreds of files – and ask questions about architectural patterns, potential bugs, or refactoring opportunities. *Multi-document synthesis* enables a researcher to provide ten or twenty papers simultaneously and ask for a summary of the consensus, the disagreements, and the open questions. *Book-length summarization* allows processing an entire novel or technical manual in a single inference pass, producing structured summaries that account for themes and arguments developed across hundreds of pages. *Extended conversations* enable agents to maintain full context of interactions spanning days or weeks without the lossy compression of conversation summaries.

However, longer context does not uniformly improve performance. Liu et al. [Liu et al., 2024] documented the "lost in the middle" phenomenon: when relevant information is placed in the middle of a long context (rather than at the beginning or end), models' accuracy drops by 15–30% compared to placement at the extremes. This finding suggests that long-context models develop attention patterns that prioritize the beginning and end of the input, with a blind spot in the middle – an artifact of training data distributions and the causal attention mask. The practical implication is significant: simply stuffing more context into the window does not guarantee the model will use it effectively. Careful placement of relevant information (at the beginning or end of the context) remains important even with million-token windows. A deeper implication is that long context does not replace RAG. Long context provides the *container* – the ability to hold more text in the model's working memory. RAG provides the *selection* – the ability to identify which text belongs in that container. Even with a million-token window, you cannot fit all of Wikipedia, all of a company's internal documentation, or all of a legal corpus. RAG selects the relevant passages; long context

holds more of them. The two techniques are complementary, not competing.

14.5 Efficient Inference

How do we run a 70-billion-parameter model on hardware that should not be able to hold it?

The models we have discussed throughout this chapter – the RAG generators, the agent reasoners, the multimodal processors – are large. A 70-billion-parameter model stored in 16-bit floating point (FP16) requires 140 gigabytes of memory just for the weights, before accounting for activations, attention caches, or batch processing. This exceeds the memory of any single consumer GPU (24 GB for an NVIDIA RTX 4090, 80 GB for an A100). Running inference requires either multiple expensive GPUs in parallel or, alternatively, engineering techniques that reduce the memory footprint and computational cost of the model itself. This section covers the three most important families of efficiency techniques: quantization (reducing the precision of model weights), parameter-efficient fine-tuning (adapting models without updating all parameters), and inference optimization (KV-cache management, speculative decoding, and distillation). Together, these techniques have democratized access to large language models, making it possible to run capable models on consumer hardware and to serve millions of concurrent users at acceptable cost.

14.5.1 Quantization: INT8, INT4, and Beyond

Quantization reduces model weight precision from floating-point representations (typically FP16, using 16 bits per parameter) to lower-precision integer representations (INT8 using 8 bits, or INT4 using 4 bits), shrinking model size by 2–4x and increasing inference speed by 1.5–3x with surprisingly small quality loss. The idea is straightforward: if a weight originally stored as a 16-bit float can be approximated as an 8-bit integer without catastrophically altering the layer's output, then we can cut the memory footprint in half. The challenge is that naive quantization – rounding each weight to the nearest integer value – introduces errors that accumulate across layers and can significantly degrade output quality. Modern quantization methods are far more sophisticated.

GPTQ (Generalized Post-Training Quantization), introduced by Frantar et al. [Frantar et al., 2023], quantizes each weight column while accounting for the interaction between weights in the same layer. The method uses the Hessian of the reconstruction error to determine which weights are most sensitive to quantization error and allocates more precision to those weights. AWQ (Activation-Aware Quantization) takes a different approach: it weights the quantization error by activation magnitude, reasoning that weights that multiply large activations have a disproportionate impact on the layer's output and should be quantized more carefully. Both methods operate post-training – they quantize a pre-trained model in a single pass without any additional training – making them practical for models that cost millions of dollars to train. GGUF is the file format optimized for CPU+GPU hybrid inference used by llama.cpp and its derivatives, enabling users to run 7B–13B parameter models on consumer laptops with no GPU at all, albeit at reduced speed.

The practical impact of quantization is transformative. A Llama-2-70B model in FP16 requires two A100 GPUs (160 GB total memory). In INT4 quantization, the same model fits on a single A100 (35 GB). A Llama-2-7B model in FP16 requires 14 GB; in INT4, it requires 3.5 GB, fitting comfortably on a consumer GPU or even running (slowly) on a modern laptop's CPU. The quality trade-offs are remarkably favorable: INT8 quantization typically loses less than 1% on standard benchmarks compared to FP16, and INT4 loses 1–3% depending on the model and task. For most applications –

Figure 14.6: Quantization Comparison

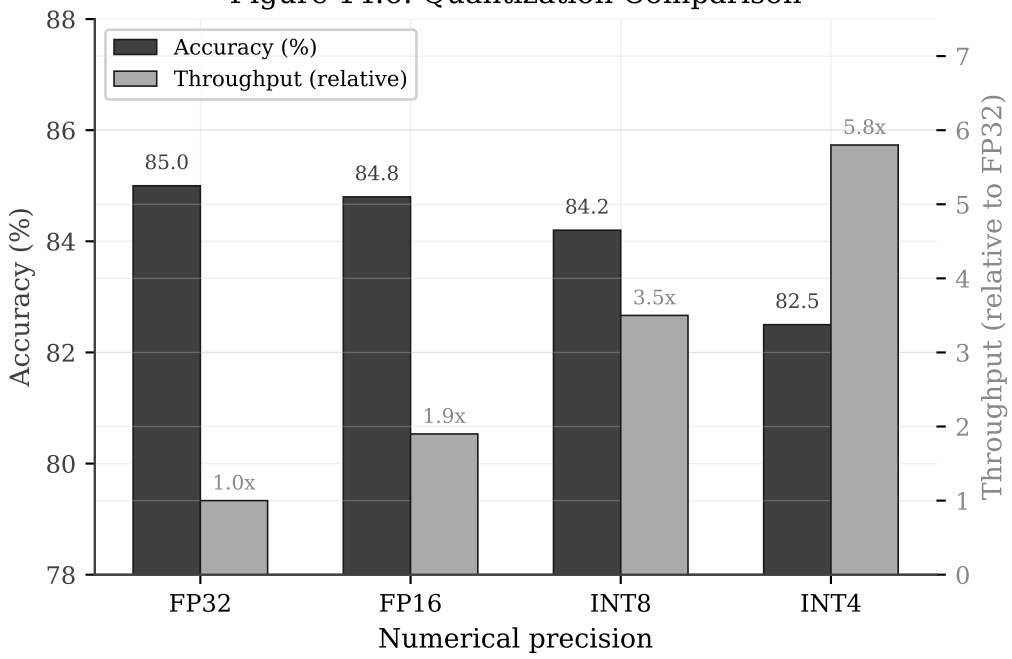


Figure 5: Figure 14.6 – Quantization Comparison

chatbots, summarization, code generation – INT4 quality is indistinguishable from FP16 in human evaluation, even when perplexity measurements show a small difference. The analogy to image compression is apt: quantization is like converting a high-resolution image to JPEG. The file shrinks dramatically, the perceptible quality loss is minimal for most uses, but zooming in reveals small artifacts. For language models, those artifacts manifest as slightly higher perplexity and occasional degradation on tasks requiring precise numerical reasoning or fine-grained knowledge retrieval.

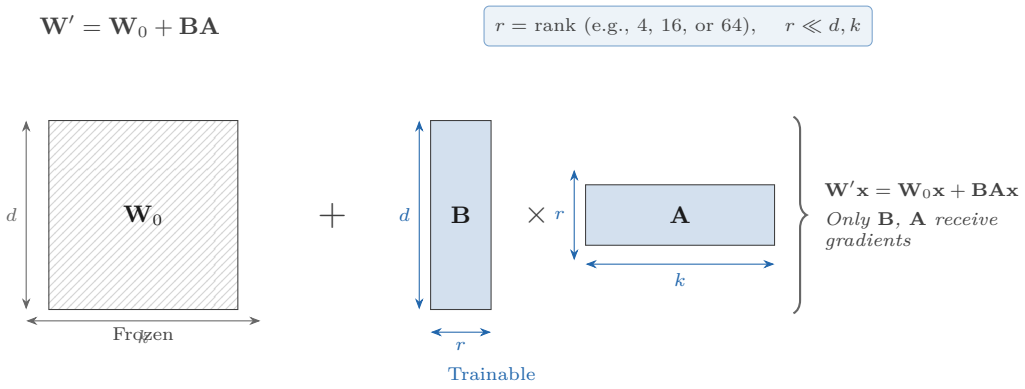
14.5.2 LoRA and Parameter-Efficient Fine-Tuning

Low-Rank Adaptation (LoRA), introduced by Hu et al. [Hu et al., 2021], is the dominant approach to parameter-efficient fine-tuning (PEFT): adapting a large pre-trained model to a specific task while training only a tiny fraction of its parameters. The key observation underlying LoRA is that weight updates during fine-tuning are empirically low-rank. When we fine-tune a model and examine the difference between the original and fine-tuned weight matrices, $\Delta\mathbf{W} = \mathbf{W}_{\text{fine-tuned}} - \mathbf{W}_0$, we find that most of the singular values of $\Delta\mathbf{W}$ are near zero – the update has low effective rank. LoRA exploits this by explicitly parameterizing the weight update as a low-rank product:

$$\mathbf{W}' = \mathbf{W}_0 + \mathbf{B}\mathbf{A}, \quad \mathbf{B} \in \mathbb{R}^{d\times r},\ \mathbf{A} \in \mathbb{R}^{r\times k} \tag{14.4}$$

where $\mathbf{W}_0 \in \mathbb{R}^{d\times k}$ is the frozen pre-trained weight matrix, $r \ll \min(d, k)$ is the LoRA rank (typically 4 to 64), and only $\mathbf{B}$ and $\mathbf{A}$ are trainable. The number of trainable parameters per adapted layer drops from dk (for full fine-tuning) to $r(d + k)$ – a reduction factor of $\frac{dk}{r(d+k)} = \frac{d}{r} \cdot \frac{k}{d+k}$. For a Transformer with $d = k = 4{,}096$ and $r = 16$, each LoRA pair has $16 \times (4{,}096 + 4{,}096) = 131{,}072$ trainable parameters compared to the original $4{,}096 \times 4{,}096 = 16{,}777{,}216$ – a 128x reduction.

Applied across all attention matrices (Q, K, V, O) in all 32 layers of a 7B model, LoRA adds approximately 20 million trainable parameters, which is 0.3% of the total parameter count. Training these 20 million parameters requires dramatically less GPU memory than full fine-tuning because we do not need to store optimizer states (momentum, variance) for the 7 billion frozen parameters.



Parameter Efficiency Comparison

Method	Trainable Para	Fraction of Full
Full fine-tuning	$d \times k$	100%
LoRA ($r = 4$)	$2 \times 4 \times d$	$\approx 0.1\%$
LoRA ($r = 16$)	$2 \times 16 \times d$	$\approx 0.4\%$
LoRA ($r = 64$)	$2 \times 64 \times d$	$\approx 1.6\%$

Figure 6: Figure 14.5 – LoRA Decomposition

A crucial property of LoRA is that the low-rank update can be *merged* into the base weights at inference time: once training is complete, we compute $\mathbf{W}' = \mathbf{W}_0 + \mathbf{BA}$ and use $\mathbf{W}'$ directly. The merged model has exactly the same architecture and inference cost as the original – there is no overhead from the LoRA decomposition during serving. This also enables a powerful deployment pattern: maintain a single base model and multiple task-specific LoRA adapters (one for medical QA, another for legal summarization, another for code generation), loading the appropriate adapter at inference time by simply adding the relevant $\mathbf{BA}$ product to the base weights. Switching tasks requires loading two small matrices, not an entirely new model. The analogy is apt: LoRA is like adding sticky notes to a textbook. The original text (frozen weights) is unchanged, but the notes (low-rank updates) customize the book for different courses. Different students (tasks) use different sets of notes, all applied to the same underlying text.

The following code implements a LoRA-augmented linear layer and demonstrates the parameter savings.

```
import torch
import torch.nn as nn

class LoRALinear(nn.Module):
    """Linear layer with LoRA: W' = W_0 + B @ A (W_0 frozen)."""
    def __init__(self, in_features, out_features, rank=4):
        super().__init__()
```

```
        self.W0 = nn.Linear(in_features, out_features, bias=False)
        self.W0.weight.requires_grad = False  # freeze pretrained
        # LoRA matrices: B (out x rank), A (rank x in)
        self.A = nn.Parameter(torch.randn(rank, in_features) * 0.01)
        self.B = nn.Parameter(torch.zeros(out_features, rank))

    def forward(self, x):
        base = self.W0(x)                # frozen forward pass
        lora = x @ self.A.T @ self.B.T   # low-rank update
        return base + lora

    def merge_weights(self):
        """Merge LoRA into base weights (zero overhead at inference)."""
        self.W0.weight.data += self.B @ self.A

# Compare parameter counts for d=4096, rank=16
d_in, d_out, rank = 4096, 4096, 16
layer = LoRALinear(d_in, d_out, rank)
total = sum(p.numel() for p in layer.parameters())
trainable = sum(p.numel() for p in layer.parameters() if p.requires_grad)
print(f"Total params: {total:,}")
print(f"Trainable:    {trainable:,} ({trainable/total:.2%})")
# Total params: 16,908,288
# Trainable:    131,072 (0.78%)
```

14.5.3 KV-Cache, Speculative Decoding, and Distillation

Three additional techniques address different bottlenecks in the inference pipeline, and together with quantization and LoRA, they form the toolkit that makes large-scale language model deployment practical.

The *KV-cache* addresses the computational redundancy of autoregressive generation. When generating token t, the attention mechanism computes queries, keys, and values for the new token and attends over all previous tokens. Without caching, the keys and values for tokens 1 through $t-1$ are recomputed from scratch at every generation step – a waste that grows quadratically with sequence length. The KV-cache stores the key and value vectors for all previously generated tokens, so that each new generation step only computes the key and value for the single new token and reuses the cached vectors for all prior tokens. This reduces the per-step cost from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, which is the difference between "painfully slow" and "usable" for long sequences. The cost is memory: the KV-cache for a 70B model generating a 4K-token sequence consumes approximately 40 GB – comparable to the model weights themselves. *Paged attention*, introduced by vLLM [Kwon et al., 2023], manages KV-cache memory like a virtual memory system: instead of pre-allocating contiguous memory blocks for each request's cache, it allocates memory in fixed-size pages that can be placed anywhere in physical memory and mapped via a page table. This eliminates memory fragmentation and enables efficient batching of many concurrent requests with different sequence lengths, achieving 2–4x throughput improvement over naive KV-cache management.

Speculative decoding [Leviathan et al., 2023] addresses a different bottleneck: the fact that autoregressive generation is inherently sequential – each token depends on all previous tokens, so we

cannot parallelize across the sequence dimension. Speculative decoding introduces parallelism by using a small, fast *draft model* to propose γ candidate tokens in a single pass, then verifying all γ proposals simultaneously with the large *target model* in a single forward pass. If the draft model's proposals align with what the target model would have generated, we accept all γ tokens at the cost of just one target-model call (plus the cheap draft-model call). If a proposal is rejected, we resample from the target model's distribution at that position and discard the remaining proposals. The critical theoretical property of speculative decoding is that it provably preserves the target model's output distribution exactly: any token the draft model proposes that the target model would not have generated is rejected through a carefully designed acceptance criterion. The model's output distribution is unchanged; only the speed of generating from it improves. With typical acceptance rates of 70–90% and draft lengths of $\gamma = 5$, speculative decoding yields 2–3x speedups with no quality degradation whatsoever. The analogy is vivid: speculative decoding is like a junior colleague drafting a report that the senior partner reviews. If the draft is good, the senior partner merely skims and approves. If not, they revise only the problematic sections, and the rest stands.

Distillation is conceptually the simplest of the three techniques: train a smaller student model to mimic the behavior of a larger teacher model. The student is trained not on the original data but on the teacher's outputs – specifically, on the teacher's output logits (the pre-softmax scores over the vocabulary), which contain richer information than the hard labels the teacher would produce after argmax. A temperature parameter softens the teacher's output distribution during distillation, exposing the relative probabilities of non-top tokens and giving the student a more nuanced signal to learn from. Distillation can produce student models that achieve 90–95% of the teacher's quality at 5–10x smaller size, making them practical for edge deployment, mobile applications, and high-throughput batch processing where the teacher's quality is desirable but its computational cost is not. The practical deployment landscape in 2025 typically combines several of these techniques: a LoRA-fine-tuned model, quantized to INT4, served with paged attention and speculative decoding, can handle thousands of concurrent users on hardware that would have been insufficient for a single inference call three years earlier. The engineering of efficient inference has expanded access to large language models from a handful of well-funded research labs to individual developers running models on their laptops.

We have now traced the arc from a model that merely predicts the next word to a system that retrieves documents, takes actions in the world, processes images, handles million-token contexts, and runs efficiently on consumer hardware. The remarkable thread through all of this is that the fundamental operation – next-token prediction – never changed. RAG enriches the context with retrieved documents. Agents wrap the predictor in a reasoning-and-action loop. Multimodal models project visual features into the token embedding space. Quantization and LoRA reduce the computational cost without altering the prediction paradigm. Speculative decoding generates from the same distribution, faster. In every case, the scaffolding around the model has grown more sophisticated, but the core mechanism remains what it was in Chapter 1: given a context, predict what comes next.

This expansion of capability brings an expansion of responsibility. A model that can retrieve documents can retrieve the wrong documents, or privileged documents, or documents that contain harmful content. An agent that can take actions can take harmful actions. A multimodal model that can interpret images can be used for surveillance. An efficient model that runs on a laptop can be deployed by anyone, including bad actors. In Chapter 15, we step back from the engineering to examine these ethical, social, and legal dimensions – the responsibilities that accompany the

capabilities we have built across this book.

The expanding capabilities of language model systems raise urgent questions that provide a natural transition to Chapter 15, where we examine the ethical, societal, and governance challenges posed by these technologies.

Exercises

Exercise 14.1 (Theory – Basic). Derive the LoRA parameter savings for a Transformer model with $L = 32$ layers, where each layer contains four weight matrices (Q, K, V, O) of dimension $d \times d$ with $d = 4{,}096$. If LoRA with rank $r = 16$ is applied to all four matrices in all layers, compute: (a) the total number of original parameters in these matrices, (b) the total number of LoRA trainable parameters, and (c) the ratio of trainable to total parameters. Verify that the ratio simplifies to $\frac{2r}{d}$ and compute its numerical value. Discuss: for what value of r would LoRA use 1% of the parameters? What about 5%?

Exercise 14.2 (Theory – Intermediate). Analyze the trade-off between retrieval precision and generation quality in RAG. Suppose the retriever returns k documents with precision p (the fraction that are actually relevant to the query). (a) Derive the expected number of relevant documents as a function of k and p. (b) Argue that increasing k has two competing effects: more chances to include the gold document, but also more irrelevant context for the generator to sift through. (c) If the generator's accuracy drops linearly with the fraction of irrelevant context, derive the optimal k as a function of p. (d) What does this analysis suggest about the relative importance of retrieval precision versus recall for RAG systems?

Exercise 14.3 (Theory – Intermediate). Explain why speculative decoding preserves the output distribution of the target model exactly. (a) Describe the acceptance criterion: for a draft token w proposed by the draft model with probability $q(w)$, the target model accepts with probability $\min(1, p(w)/q(w))$, where $p(w)$ is the target model's probability. Show that the marginal distribution of accepted tokens equals $p(w)$. (b) If the draft model achieves an acceptance rate of $\alpha = 0.8$ with draft length $\gamma = 5$, estimate the expected number of tokens generated per target-model forward pass. (c) Why does the acceptance rate depend on how well the draft model approximates the target model?

Exercise 14.4 (Programming – Basic). Build a RAG pipeline using sentence-transformers for embedding and a pre-trained language model for generation. Index 50 paragraphs from a Wikipedia article of your choice. For 10 factual questions about the article, compare: (a) closed-book generation (no retrieval), (b) RAG with $k = 1$, and (c) RAG with $k = 5$. Measure answer correctness using token-level F1 against manually written gold answers. Report the mean F1 for each condition and discuss the effect of k.

Exercise 14.5 (Programming – Intermediate). Implement a ReAct agent with two tools: a `search` function (that retrieves from your RAG index in Exercise 14.4) and a `calculator` function (that evaluates arithmetic expressions). Test the agent on 15 multi-step questions that require combining search and calculation (e.g., "If the Eiffel Tower is X meters tall, how many stories is that assuming 3 meters per story?"). Report: (a) the average number of reasoning steps, (b) tool usage frequency, and (c) final answer accuracy compared to direct prompting without tools.

Exercise 14.6 (Programming – Intermediate). Apply LoRA fine-tuning to a 7B-parameter model

(e.g., Mistral-7B) on 1,000 instruction-following examples using the HuggingFace PEFT library with rank $r = 16$, $\alpha = 32$, and `target_modules=["q_proj", "v_proj"]`. Measure and report: (a) total trainable parameters as a fraction of the model, (b) peak GPU memory during training, (c) training wall-clock time, and (d) qualitative output quality on five held-out prompts. Compare with the base model's zero-shot performance on the same five prompts.

Exercise 14.7 (Programming – Intermediate). Quantize a 7B-parameter model to INT8 and INT4 using the `bitsandbytes` library. For each precision level (FP16, INT8, INT4), measure: (a) model size on disk (in GB), (b) inference speed (tokens per second) averaged over 50 generation calls of 100 tokens each, and (c) perplexity on a 2,000-token sample from WikiText-103. Plot all three metrics as a grouped bar chart and discuss the quality-efficiency trade-off.

Exercise 14.8 (Programming – Advanced). Build a simplified vision-language pipeline. Use the CLIP model (`openai/clip-vit-base-patch32`) to encode 20 images from the COCO validation set. Train a small MLP (two layers, hidden size 512) to project the 512-dimensional CLIP image embeddings to the input embedding dimension of a small language model (e.g., GPT-2). Fine-tune on image-caption pairs and evaluate the generated captions on the 20 held-out images using BLEU-4. Report the BLEU score and discuss three failure modes you observe.

References

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. (2023). GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. *Proceedings of ICLR*.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. (2021). LoRA: Low-Rank Adaptation of Large Language Models. *Proceedings of ICLR*.

Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W. (2020). Dense Passage Retrieval for Open-Domain Question Answering. *Proceedings of EMNLP*.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. (2023). Efficient Memory Management for Large Language Model Serving with PagedAttention. *Proceedings of SOSP*.

Leviathan, Y., Kalman, M., and Matias, Y. (2023). Fast Inference from Transformers via Speculative Decoding. *Proceedings of ICML*.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Kuttler, H., Lewis, M., Yih, W., Rocktaschel, T., Riedel, S., and Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in NeurIPS*.

Liu, H., Li, C., Wu, Q., and Lee, Y. J. (2023). Visual Instruction Tuning. *Advances in NeurIPS*.

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. (2024). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the ACL*.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., and Sutskever, I. (2021). Learning Transferable Visual Models from Natural Language Supervision. *Proceedings of ICML*.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *Proceedings of ICLR*.