LLM Agent Patterns: From Single Agents to Orchestrations

Build React, RAG, Self Consistent and Many More Pattern Agents

Author: Sourena Khanzadeh

Date: January 4, 2026

Version: 0.0.0

Bio: I am an enthusiastic TMU PhD candidate in the
department of computer science, who loves the world of agents and LLMs.

Agents are now everywhre, might as well be ready to learn some patterns.

Acknowledgement

Preface

Large Language Models (LLMs) have transformed the way we interact with software and digital systems. What began as relatively simple text generation models has rapidly evolved into complex systems capable of powering intelligent agents that can reason, plan, retrieve knowledge, call tools, and collaborate with other agents to solve complex tasks.

However, building reliable LLM-powered systems requires moving beyond simple prompting techniques. While prompt engineering can achieve impressive results, developers and researchers quickly discover that effective systems require structure. They require patterns.

Over the past few years, a new design space has emerged around *LLM agent patterns*: reusable architectural approaches that enable models to reason, act, retrieve information, coordinate workflows, and self-correct. Patterns such as ReAct, Retrieval-Augmented Generation (RAG), self-consistency reasoning, multi-agent collaboration, and orchestration frameworks are rapidly becoming the foundation of modern AI systems.

Who is this for This book is intended for readers who want to learn about agentic design patterns and practical ways of building LLM-powered systems. Throughout the book, I primarily use LangGraph and briefly introduce LangChain when relevant. The goal is to keep the material technical and practical while maintaining an accessible and informal tone.

What are the prerequisites Readers are expected to have an intermediate understanding of Python. Familiarity with basic machine learning or LLM concepts may be helpful, but it is not strictly required.

What makes this book unique This book aims to provide a comprehensive and practical exploration of LLM agent patterns. Instead of focusing purely on theory or purely on frameworks, it focuses on architectural patterns that can be applied across tools and platforms.

What is not in this book This book is not tied to a specific framework. While examples may use LangChain or LangGraph to demonstrate implementation details, the underlying concepts and patterns are framework-agnostic.

Rather than treating agents as magical black boxes, we will study them as systems. Each pattern presented in this book focuses on answering questions such as:

- When should this agent pattern be used?
- What problem does it solve?
- How is it implemented in practice?
- What are the trade-offs and limitations?

The goal is not only to understand these patterns conceptually, but also to build them. Throughout the chapters, we will progressively move from simple single-agent systems to more advanced architectures involving orchestration, planning, memory, and collaboration between multiple agents.

This book is written for developers, researchers, and students who want to move beyond basic prompting and begin designing robust LLM-powered systems.

I wrote this book both as a a guide and as a learning companion. The field is evolving rapidly, and the patterns discussed here represent a snapshot of an exciting moment in the development of AI systems.

Agents are quickly becoming a fundamental building block of modern software. Understanding how they work—and how to design them well—will become an increasingly valuable skill.

I hope this book helps you build your own agents, experiment with new ideas, and explore the fascinating design space of LLM-powered systems.

Contents

I Foundations of LLM Agents **1**

Chapter 1 Introduction to LLM Agents **2**
1.1 Different Language Models . . . 2
1.2 From Prompting to Agentic Systems . . . 3

Chapter 2 Foundation of LLMs **5**
2.1 Neural Networks . . . 5
2.2 Representation Learning . . . 7
2.3 Language Modeling as a Learning Objective . . . 7
2.4 From Recurrent Models to Attention . . . 8
2.5 Attention Mechanism . . . 9
2.6 The Transformer Architecture . . . 10
2.7 Pretraining at Scale . . . 11
2.8 Fine-Tuning and Instruction Tuning . . . 11
2.9 Reinforcement Learning and Alignment . . . 12
2.10 From Language Models to LLM Systems . . . 12
2.11 Summary . . . 12

Chapter 3 Transformer Architecture **14**
3.1 Why Transformers Changed Language Modeling . . . 14
3.2 Representing Text as Model Input . . . 14
3.3 Self-Attention in Detail . . . 16
3.4 Multi-Head Attention . . . 17
3.5 The Transformer Block . . . 18
3.6 Causal Masking and Autoregressive Generation . . . 18
3.7 Encoder-Only, Decoder-Only, and Encoder–Decoder Models . . . 19
3.8 Depth, Width, and Model Capacity . . . 20
3.9 Why Transformer Architecture Matters for Agents . . . 21
3.10 Limitations of the Transformer . . . 21
3.11 Summary . . . 21

II Core Agent Patterns **23**

Chapter 4 Study of Single Agents **24**
4.1 Our First Graph in LangGraph . . . 25
4.2 Graphs with Multiple Processes . . . 26
4.3 Conditional Graphs and Branching . . . 28
4.4 Loops and Iterative Graphs . . . 30

4.5 Graphs with Tools and LLMs . . . 32

Chapter 5 State, Messages, and Memory in LangGraph 35
5.1 The Concept of State . . . 35
5.2 State Transformation Across Nodes . . . 36
5.3 State Reducers . . . 38
5.4 Messages as Structured Communication . . . 40
5.5 Message-Based State in LangGraph . . . 41
5.6 Conditional Routing with State . . . 43
5.7 Memory and Persistent Context . . . 44
5.8 Putting It All Together: A Stateful Agent . . . 47
5.9 Summary . . . 48

Chapter 6 Tool-Augmented Agents in LangGraph 50
6.1 Why Tools Matter . . . 50
6.2 Defining Tools in LangChain . . . 50
6.3 Binding Tools to Language Models . . . 52
6.4 The ToolNode Utility . . . 54
6.5 The ReAct Pattern . . . 55
6.6 Parallel and Sequential Tool Calls . . . 58
6.7 Error Handling in Tool Execution . . . 59
6.8 The Prebuilt ReAct Agent . . . 61
6.9 Design Considerations . . . 61
6.10 Summary . . . 62

III Planning and Reasoning Patterns 64

Chapter 7 Planning and Reasoning Strategies for Agents 65
7.1 The Limits of Reactive Agents . . . 65
7.2 Chain-of-Thought Reasoning . . . 65
7.3 Plan-and-Execute Architecture . . . 68
7.4 Adaptive Replanning . . . 72
7.5 Reflection and Self-Critique . . . 73
7.6 Tree of Thoughts . . . 77
7.7 Comparing Reasoning Strategies . . . 80
7.8 Summary . . . 81

Chapter 8 Verification and Reliable Reasoning for Agents 82
8.1 Why Reasoning Needs Verification . . . 82
8.2 Self-Consistency . . . 82
8.3 Verifier-Guided Reasoning . . . 86
8.4 Tool-Based Verification . . . 89
8.5 Confidence, Abstention, and Escalation . . . 91
8.6 Best-of-n and Reranking . . . 93

8.7 Verification for Tool-Using Agents . . . 94
8.8 Comparison of Verification Strategies . . . 94
8.9 Design Guidelines . . . 94
8.10 Summary . . . 96

IV Multi-Agent Systems **97**

Chapter 9 Multi-Agent Architectures **98**
9.1 Why Multiple Agents? . . . 98
9.2 Communication Topologies . . . 99
9.3 Sequential Pipeline . . . 99
9.4 Parallel Fan-Out and Aggregation . . . 101
9.5 Supervisor–Worker Architecture . . . 104
9.6 Collaborative Debate . . . 109
9.7 Hierarchical Multi-Agent Systems . . . 112
9.8 Design Patterns and Best Practices . . . 113
9.9 Summary . . . 114

V Agent Orchestration **116**

Chapter 10 Agent Orchestration and Human-in-the-Loop Control **117**
10.1 The Case for Human Oversight . . . 117
10.2 Breakpoints . . . 118
10.3 Human-in-the-Loop Patterns . . . 119
10.4 Time Travel . . . 122
10.5 Streaming . . . 123
10.6 Fault Tolerance and Recovery . . . 125
10.7 Dynamic Breakpoints with NodeInterrupt . . . 127
10.8 Orchestration in Production . . . 128
10.9 Summary . . . 128

VI Memory and Knowledge **130**

Chapter 11 Memory Systems and Knowledge Management **131**
11.1 A Taxonomy of Agent Memory . . . 131
11.2 Conversation Buffer Memory . . . 131
11.3 Conversation Window and Trimming . . . 133
11.4 Conversation Summary Memory . . . 134
11.5 Vector Stores and Semantic Search . . . 136
11.6 Retrieval-Augmented Generation (RAG) . . . 138
11.7 The LangGraph Store for Long-Term Memory . . . 141
11.8 Knowledge Graphs . . . 143

11.9 Memory Architecture Design 143
11.10 Summary 144

VII The Future of Agent Systems **146**

Chapter 12 The Road Ahead **147**
12.1 Where We Stand 147
12.2 The Three Frontiers 147
12.3 The Spectrum of Agent Autonomy 150
12.4 The Economics of Agent Systems 151
12.5 Governance and Responsible Deployment 152
12.6 Toward General-Purpose Agents 153
12.7 Closing Thoughts 154

Part I

Foundations of LLM Agents

Chapter 1 Introduction to LLM Agents

Generative language models can generally be categorized into three groups based on their scale and capabilities: **Small Language Models (SLMs)**, **Large Language Models (LLMs)**, and **Frontier Models (FMs)**. Each category represents a different trade-off between computational cost, efficiency, and reasoning capability.

1.1 Different Language Models

Here is a deep dive into the differences of these model as mentioned in Table 1.1.

Model Type	Parameters	Strengths	Typical Use Cases
SLMs	<10B	Efficient, low cost	Classification, routing
LLMs	10B–70B	General reasoning	Chatbots, summarization
Frontier Models	>100B	Advanced reasoning	Agents, complex workflows

Table 1.1: Comparison of language model categories

1.1.1 Small Language Models (SLMs)

Small Language Models typically contain fewer than 10 billion parameters and are designed for efficiency and precision within narrowly defined tasks. Due to their smaller size, SLMs offer lower latency, reduced inference costs, and can often be deployed on-premise, which makes them attractive for environments requiring strict data governance.

SLMs are particularly effective for structured or repetitive workflows where the task objective is clearly defined and variability in input is limited. Common use cases include document classification, routing, information extraction, summarization, and code routing.

Example 1.1 Examples of small language models include:

1. Microsoft Phi-2 / Phi-3: Highly capable models optimized for reasoning and coding tasks.
2. Google Gemma: A family of lightweight open-weights models.
3. TinyLlama: A compact 1.1B parameter model designed for efficiency.
4. Salesforce XLAM 1B: A model designed specifically for function calling.
5. DistilBERT: A smaller, faster, and cheaper distilled version of BERT.

1.1.2 Large Language Models (LLMs)

Large Language Models typically contain tens of billions of parameters and are trained on broad and diverse datasets spanning multiple domains. Their larger scale enables them to generalize better across unpredictable inputs and perform tasks requiring contextual understanding and reasoning.

LLMs are well suited for conversational systems, knowledge synthesis, and tasks that require interpreting complex or ambiguous instructions. They can combine information from multiple sources and generate coherent, context-aware responses.

Examples include models in the 30–70 billion parameter range such as Generative Pretrained Transformers (GPT), Text-to-Text Transfer Transformers (T5), and Bidirectional Encoder Representations from Transformers (BERT). Other notable systems include Claude (Anthropic), Gemini (Google), and Llama (Meta).

1.1.3 Frontier Models

Frontier Models represent the most advanced AI systems currently available and often exceed hundreds of billions of parameters. These models push the limits of current AI capabilities and are designed to handle complex reasoning, planning, and autonomous decision-making tasks.

Frontier models are particularly useful in dynamic environments where systems must reason through multiple steps, interact with tools and external APIs, evaluate outcomes, and continuously adapt their behavior. They form the foundation for modern agentic systems and advanced AI workflows.

Examples include proprietary models such as OpenAI's GPT series (e.g., GPT-4 and later versions), Anthropic's Claude models, and Google's Gemini models. Open-source frontier-scale models are also emerging from organizations such as Meta, Mistral, and Alibaba, offering greater customization and cost flexibility.

1.2 From Prompting to Agentic Systems

Early applications of language models primarily relied on simple prompting. In this paradigm, a user provides an input prompt and the model generates a response based solely on the information contained in that prompt and the knowledge encoded during training. While this approach proved powerful for tasks such as summarization, translation, and question answering, it remains fundamentally limited. The model performs a single forward pass and cannot dynamically interact with external systems, retrieve new information, or refine its reasoning.

As applications became more complex, researchers and engineers began extending this paradigm toward *agentic systems*. Instead of treating language models as standalone text generators, agentic architectures treat them as reasoning engines capable of planning, decision making, and tool usage.

An **LLM agent** is a system in which a language model is given the ability to:

- Reason about a task and break it into smaller steps
- Use external tools such as APIs, databases, or search engines
- Maintain memory of previous interactions
- Evaluate intermediate results and adjust its strategy

This shift transforms language models from passive responders into active problem solvers. Rather than producing a single response, an agent may execute a sequence of actions in order to complete a goal.

1.2.1 Core Components of an LLM Agent

Most LLM agent architectures consist of several key components that work together to enable autonomous behavior.

1.2.1.0.1 1. The Language Model The language model acts as the reasoning engine of the agent. It interprets user instructions, plans actions, and generates responses. Depending on the complexity of the task, this model may be a small language model for efficiency or a frontier model for advanced reasoning.

1.2.1.0.2 2. Tools Tools allow the agent to interact with the external world. Examples include:

- Web search engines
- Code execution environments
- Databases or vector stores

- External APIs

By invoking tools, the agent can retrieve up-to-date information, perform calculations, or manipulate external systems.

1.2.1.0.3 3. Memory Memory allows agents to retain context across multiple interactions. This can include:

- Short-term conversational memory
- Long-term knowledge storage
- Vector databases for semantic retrieval

Memory enables agents to maintain continuity in conversations and learn from previous interactions.

1.2.1.0.4 4. Planning and Control Agent frameworks often include a control loop that determines how the agent decides what to do next. The loop typically follows a reasoning pattern:

1. Observe the current state
2. Reason about the next action
3. Execute a tool or generate output
4. Evaluate the result
5. Repeat until the goal is achieved

This iterative structure allows agents to solve problems that require multiple steps.

1.2.2 Example: An Agent Execution Loop

A simplified agent loop can be described as follows:

Example 1.2 Agent reasoning cycle:

1. User asks: "Find recent research papers on LLM agents."
2. The agent decides to use a search tool.
3. The tool retrieves relevant papers.
4. The agent summarizes the results.
5. The agent returns the final answer to the user.

This pattern illustrates how agents combine reasoning with external actions.

1.2.3 Why Agents Matter

Agentic systems represent an important shift in how language models are used. Instead of generating isolated responses, agents can operate within complex workflows and interact with real-world systems.

This capability enables a wide range of applications, including:

- Autonomous research assistants
- Intelligent coding assistants
- Workflow automation systems
- Multi-step data analysis pipelines

As language models continue to improve, agent architectures are becoming increasingly central to the development of advanced AI systems.

In the following chapters, we will explore how LLM agents are designed, how they interact with tools and memory systems, and how they can be implemented using modern agent frameworks.

Chapter 2 Foundation of LLMs

Chapter 1 introduced the role of Large Language Models (LLMs) in modern AI systems and motivated why they matter for agentic workflows. However, to understand how such systems are built, it is necessary to first examine the technical foundations on which they rely. LLMs do not emerge as isolated systems; they are the result of several decades of progress in neural networks, representation learning, sequence modeling, optimization, and large-scale training.

This chapter presents the core ideas that form the basis of modern LLMs. We begin with neural networks, which provide the computational structure used to model complex patterns in data. We then discuss how text is represented numerically, how models learn relationships between words and tokens, and why attention mechanisms became a major breakthrough in natural language processing. Finally, we introduce the transformer architecture, large-scale pretraining, and alignment techniques such as reinforcement learning from human feedback. Together, these concepts form the conceptual and technical foundation of LLMs.

2.1 Neural Networks

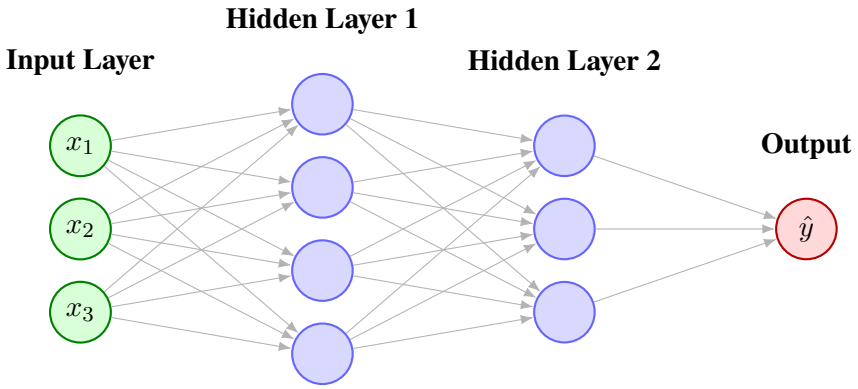


Figure 2.1: A simple feed-forward neural network. Each neuron applies a weighted transformation followed by a non-linear activation function. By stacking multiple hidden layers, the network can learn increasingly abstract representations from input data.

Neural Networks (NNs) are one of the fundamental building blocks of modern machine learning and are central to the development of LLMs. Influential works in deep learning demonstrate how neural networks can learn complex, hierarchical representations directly from data [2, 4]. In simple terms, a neural network is a parameterized function that maps an input to an output through a sequence of learned transformations. Figure 2.1 illustrates the layered structure of a feed-forward neural network.

2.1.1 Artificial Neurons

The basic unit of a neural network is the *artificial neuron*. Inspired loosely by biological neurons, an artificial neuron takes a set of input values, multiplies each input by a learned weight, adds a bias term, and applies a non-linear activation function. This process can be written as:

$$z = \mathbf{w}^\top \mathbf{x} + b \tag{2.1}$$

$$a = \sigma(z) \tag{2.2}$$

where $\mathbf{x}$ is the input vector, $\mathbf{w}$ is the weight vector, b is the bias, and $\sigma(\cdot)$ is an activation function such as ReLU, sigmoid, or GELU.

A single neuron is limited in expressive power. However, when many neurons are arranged into layers, they can approximate highly complex functions. This is the central idea behind deep learning.

2.1.2 Layers and Network Depth

Neural networks are usually organized into three types of layers:

- **Input layer:** receives the raw features
- **Hidden layers:** perform intermediate transformations
- **Output layer:** produces the final prediction

A network with many hidden layers is referred to as a *deep neural network*. Depth is important because each layer can learn progressively more abstract features. For example, in language tasks, earlier layers may learn local word relationships, while deeper layers may capture syntax, semantics, and discourse-level patterns.

This ability to learn hierarchical structure is one of the reasons neural networks became so successful across domains such as computer vision, speech recognition, and natural language processing.

2.1.3 Training Neural Networks

Neural networks learn by adjusting their parameters so that their predictions become closer to the desired outputs. This is done by defining a *loss function*, which measures the difference between the model's prediction and the ground truth.

Training typically involves three steps:

1. Perform a forward pass to compute the prediction
2. Compute the loss
3. Update the parameters using gradient-based optimization

The most common optimization method is gradient descent and its variants such as stochastic gradient descent (SGD) and Adam. Gradients are computed efficiently using the *backpropagation* algorithm [6], which applies the chain rule to determine how each parameter contributes to the overall error.

In practice, training large neural networks requires vast datasets, specialized hardware such as GPUs or TPUs, and carefully tuned optimization procedures.

2.1.4 Why Neural Networks Matter for LLMs

LLMs are built from neural networks because language is highly structured, ambiguous, and context dependent. Traditional rule-based systems struggle to capture this complexity at scale. Neural networks, by contrast, can learn patterns directly from large text corpora. They are able to represent relationships between words, phrases, and documents in a distributed form, making them suitable for modeling the statistical structure of language.

2.2 Representation Learning

A major strength of neural networks is their ability to perform *representation learning*. Instead of requiring handcrafted features, the model learns useful internal representations from raw input data.

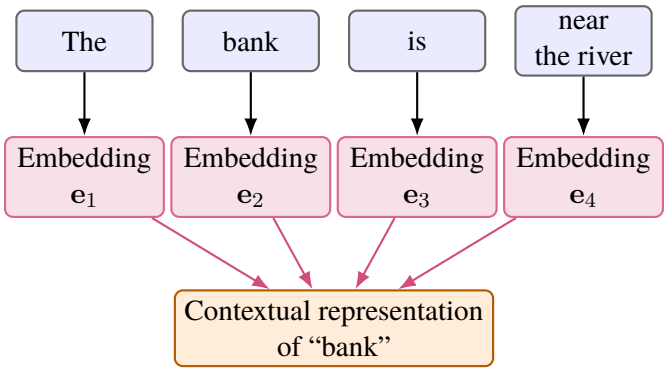
In language processing, representation learning is essential because text is inherently symbolic. Words and sentences must be converted into numerical vectors before they can be processed by a neural network.

2.2.1 From Words to Vectors

A simple representation of a word is a one-hot vector, where only one element is active and all others are zero. While straightforward, one-hot vectors do not capture any notion of semantic similarity. For example, the words *cat* and *dog* are treated as completely unrelated, even though they are semantically close.

To address this limitation, neural language models use *embeddings*, which map words or tokens into dense, continuous vector spaces [1]. In such spaces, semantically related words tend to be located near one another. This enables models to capture meaningful relationships such as similarity, analogy, and contextual usage.

2.2.2 Contextual Representations



Meaning depends on surrounding context.

Figure 2.2: From token embeddings to contextual representations. Modern language models do not assign a fixed meaning to a token in isolation; instead, the representation of a token such as *bank* is shaped by the surrounding words.

Early embedding methods assigned a single vector to each word regardless of context. However, natural language is highly context sensitive. The meaning of a word like *bank* differs in the sentences “I deposited money in the bank” and “we sat by the river bank.” As shown in Figure 2.2, the meaning of a token is shaped by the words surrounding it.

Modern LLMs address this by generating *contextual embeddings*, where the representation of a token depends on the surrounding sequence. This shift from static to contextual representations was one of the key developments that enabled strong performance in modern NLP systems.

2.3 Language Modeling as a Learning Objective

At the heart of an LLM is the task of *language modeling*. A language model estimates the probability of a sequence of tokens. Given a sequence $x_1, x_2, \ldots, x_T$, the model learns:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t}) \tag{2.3}$$

This factorization expresses the sequence probability as a product of conditional next-token probabilities.

2.3.1 Next-Token Prediction

Most generative LLMs are trained using the *next-token prediction* objective. The model is shown a prefix of text and learns to predict the next token. Repeating this process across very large corpora enables the model to internalize patterns of grammar, facts, style, reasoning traces, and even some task structure.

The training loss is commonly the cross-entropy loss:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(x_t \mid x_{<t}) \tag{2.4}$$

Although simple in formulation, this objective is remarkably powerful. When scaled with sufficient data, parameters, and compute, next-token prediction gives rise to models capable of translation, summarization, coding, reasoning, and dialogue.

2.3.2 Tokenization

Before text can be modeled, it must be broken into smaller units called *tokens*. Depending on the tokenizer, a token may correspond to a word, subword, punctuation mark, or character fragment. Modern LLMs typically use subword tokenization because it balances vocabulary size with flexibility. Rare words can be represented as combinations of smaller units, while common words may appear as single tokens.

Tokenization has important practical implications. It affects model efficiency, sequence length, multilingual performance, and cost during inference. In most real-world systems, the number of tokens is also directly related to latency and billing.

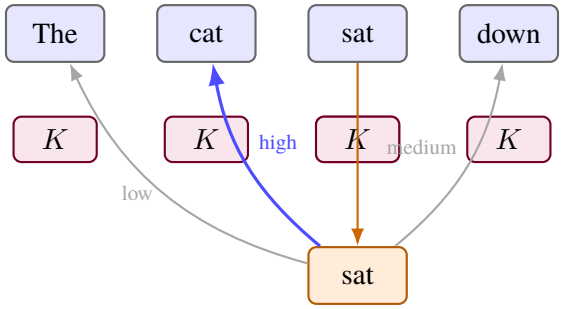
2.4 From Recurrent Models to Attention

Before transformers became dominant, language models were often built using recurrent neural networks (RNNs) and their variants such as Long Short-Term Memory (LSTM) networks [3, 7]. These models process text sequentially, updating a hidden state one token at a time.

RNN-based architectures made important contributions to sequence modeling, but they have several limitations:

- They are difficult to parallelize efficiently
- They struggle with very long-range dependencies
- Their hidden state can become a bottleneck for retaining information

These limitations motivated the search for alternative mechanisms that could model relationships between tokens more directly. This led to the development of the *attention mechanism*.



The token attends to other tokens in the sequence
to gather relevant contextual information.

Figure 2.3: Illustration of self-attention. When processing a token, the model computes attention scores over the other tokens in the sequence and combines their information with different weights.

2.5 Attention Mechanism

Attention allows a model to focus selectively on the most relevant parts of an input when processing a token. Rather than compressing all prior information into a single hidden state, attention gives the model direct access to other positions in the sequence. Figure 2.3 provides an intuitive view of how self-attention allows a token to gather information from the rest of the sequence.

2.5.1 Basic Idea of Attention

For each token, the model computes three vectors:

- **Query (Q):** what the token is looking for
- **Key (K):** what information each token offers
- **Value (V):** the content associated with each token

Attention scores are computed by comparing queries and keys. These scores are normalized and used to form a weighted combination of values. In scaled dot-product attention, this is written as [8]:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{2.5}$$

where d_k is the dimensionality of the key vectors.

This formulation enables each token to dynamically attend to other tokens in the sequence, regardless of their distance.

2.5.2 Self-Attention

In *self-attention*, the queries, keys, and values all come from the same input sequence. This allows the model to relate every token to every other token in the sequence. For example, when processing the pronoun "it," the model can attend to earlier nouns in the sentence to infer the correct reference.

Self-attention is particularly powerful for natural language because meaning often depends on long-distance relationships. Syntactic agreement, co-reference, discourse structure, and semantic constraints may all involve words far apart from one another.

2.5.3 Multi-Head Attention

Modern transformer models use *multi-head attention*, where several attention operations are performed in parallel. Each head can learn to focus on different aspects of the sequence, such as syntax, local context, long-range dependencies, or semantic roles.

This improves expressiveness and gives the model multiple perspectives on the same input.

2.6 The Transformer Architecture

The transformer architecture, introduced in "Attention Is All You Need" [8], marked a major turning point in NLP. It replaced recurrence with attention and enabled much more efficient large-scale training.

2.6.1 Core Building Blocks

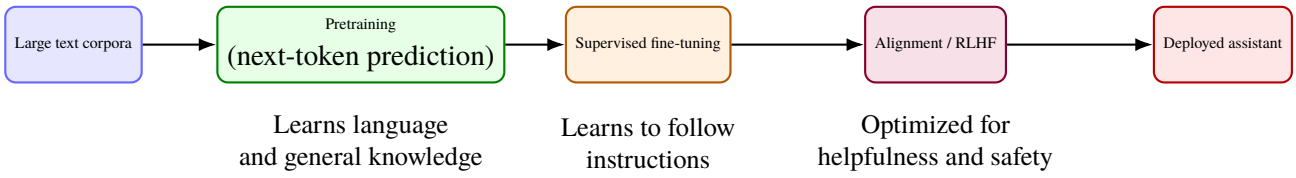


Figure 2.4: A simplified pipeline for building modern LLM systems. Models are first pretrained on large corpora, then adapted through supervised fine-tuning and alignment techniques such as reinforcement learning from human feedback.

A transformer layer typically includes the following components:

- Multi-head self-attention
- Feed-forward neural network
- Residual connections
- Layer normalization

Residual connections help stabilize training by allowing information to flow more easily across layers, while layer normalization improves optimization behavior. The overall development pipeline of modern LLMs is shown in Figure 2.4.

2.6.2 Positional Information

Because self-attention alone does not encode token order, transformers must incorporate positional information explicitly. This is usually done using positional encodings or learned positional embeddings. Without such information, the model would know which tokens are present but not the order in which they appear.

2.6.3 Encoder, Decoder, and Decoder-Only Models

Transformer architectures are often grouped into three categories:

- **Encoder-only models:** designed for understanding tasks such as classification and retrieval
- **Decoder-only models:** designed for autoregressive generation and commonly used in chat-based LLMs
- **Encoder-decoder models:** designed for sequence-to-sequence tasks such as translation and summarization

Many modern generative LLMs, including systems used for dialogue and coding, are decoder-only transformers. Their architecture is especially well suited to next-token prediction and open-ended text generation.

2.7 Pretraining at Scale

A transformer becomes a large language model when it is trained on massive datasets using large computational budgets. This stage is known as *pretraining*.

2.7.1 Large-Scale Data

Pretraining data may include books, articles, websites, code repositories, reference material, and other text sources. The objective is not to teach the model one specific task, but to expose it to broad patterns of language and knowledge.

The quality, diversity, and filtering of the training corpus have a major impact on model behavior. Poorly curated data can introduce noise, bias, duplication, and factual unreliability.

2.7.2 Scaling Laws

Empirical research has shown that model performance tends to improve predictably as parameters, data, and compute increase, provided the scaling is balanced. This observation is often described through *scaling laws*. As models become larger and better trained, they often exhibit new capabilities that are weak or absent at smaller scales.

This is one reason why LLM development is strongly tied to infrastructure: advances in algorithms alone are not sufficient without the computational resources needed to train at scale.

2.7.3 Base Models

The result of pretraining is often called a *base model*. A base model has learned the statistical structure of language, but it is not necessarily optimized for helpful dialogue, instruction following, or safe interaction. It can complete text, but it may not behave in ways aligned with user expectations.

For this reason, additional post-training stages are often applied.

2.8 Fine-Tuning and Instruction Tuning

After pretraining, a model can be adapted to more specific tasks or behaviors through *fine-tuning*. In supervised fine-tuning, the model is trained on curated input-output examples so that it learns to respond in a desired format.

A particularly important form of fine-tuning is *instruction tuning*, where models are trained to follow natural language instructions. This shifts the model from merely continuing text to acting more like an assistant.

Instruction tuning improves usability significantly. Instead of requiring carefully engineered prompts, users can give direct commands such as:

- Summarize this article
- Explain this code
- Compare these two methods

The model then learns to respond in a way that is more aligned with human intent.

2.9 Reinforcement Learning and Alignment

Even after instruction tuning, a model may produce outputs that are unhelpful, unsafe, or poorly aligned with user preferences. To address this, many LLM pipelines include an alignment stage based on human feedback.

2.9.1 Reinforcement Learning from Human Feedback

One influential approach is *Reinforcement Learning from Human Feedback* (RLHF) [5]. The process generally consists of three stages:

1. Train a base model through pretraining
2. Collect human preference data and train a reward model
3. Optimize the language model to produce outputs that score highly under the reward model

In practice, the reward model is trained on comparisons between alternative responses. Humans indicate which response is better, and the model learns to approximate these preferences. Reinforcement learning is then used to adjust the policy so that future generations are more helpful, truthful, and safe.

2.9.2 Why Alignment Matters

Alignment is important because raw predictive capability is not the same as reliable assistant behavior. A model may be fluent yet misleading, knowledgeable yet unsafe, or capable yet uncooperative. Alignment methods attempt to shape the model's behavior so that it better reflects human goals and preferences.

This is especially important for agentic systems. Once a model is allowed to use tools, access memory, or perform multiple-step workflows, errors in judgment or instruction following can have larger consequences. Therefore, strong alignment is not merely a usability enhancement; it is a practical requirement for deploying LLMs in real systems.

2.10 From Language Models to LLM Systems

An LLM by itself is a trained neural network that predicts tokens. However, in practice, real applications rarely use the model in isolation. Instead, LLMs are embedded within larger systems that provide prompting strategies, memory, tool interfaces, safety filters, and orchestration logic.

This systems perspective is crucial. The capabilities users observe are often not produced by the model alone, but by the interaction between the model and the surrounding software stack. For example, retrieval-augmented generation adds access to external knowledge, tool calling enables interaction with APIs, and memory components maintain context across turns.

As a result, understanding modern LLMs requires understanding both the underlying model and the broader architecture in which it operates.

2.11 Summary

This chapter introduced the foundational ideas behind LLMs. Neural networks provide the computational substrate, representation learning enables language to be encoded as dense vectors, and language modeling defines the training objective used to predict text. The attention mechanism solved important limitations of

earlier sequence models, and the transformer architecture made it possible to train highly capable models at scale. Large-scale pretraining produces base models, while fine-tuning and alignment techniques adapt these models for practical and safe use.

These foundations are essential for understanding the rest of this book. In the following chapters, we build on this material to examine the architecture of transformers in greater detail, the training pipeline of LLMs, and how these models are integrated into agentic systems with tools, memory, and control loops.

References

[1] Yoshua Bengio et al. "A neural probabilistic language model". In: *Journal of machine learning research* 3.Feb (2003), pp. 1137–1155.

[2] Ian Goodfellow et al. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[3] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[4] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[5] Long Ouyang et al. "Training language models to follow instructions with human feedback". In: *Advances in neural information processing systems* 35 (2022), pp. 27730–27744.

[6] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[7] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems* 27 (2014).

[8] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

Chapter 3 Transformer Architecture

In Chapter 2, we introduced the main ingredients that make Large Language Models (LLMs) possible, including neural networks, representation learning, language modeling, and attention. While these ideas provide the conceptual foundation, modern LLMs are ultimately implemented through a specific architectural design: the *transformer*. This architecture has become the dominant paradigm in natural language processing and forms the computational backbone of most contemporary generative models.

The transformer is important because it solves several limitations of earlier sequence models. Instead of processing text one token at a time through a recurrent hidden state, transformers allow all tokens in a sequence to interact through attention. This enables efficient parallel computation during training, stronger modeling of long-range dependencies, and more flexible contextual reasoning. These properties made it possible to train models on massive corpora and scale them into the systems now referred to as LLMs.

This chapter examines the transformer architecture in greater detail. We begin by motivating why the transformer replaced recurrent models, then explain how textual input is represented, how self-attention works in practice, how transformer layers are stacked, and why decoder-only transformers became the standard architecture for many generative systems. We conclude by connecting these architectural ideas to the broader behavior of modern LLMs.

3.1 Why Transformers Changed Language Modeling

Before transformers, many sequence modeling systems relied on Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks. These models process tokens in order, updating an internal hidden state after each step. Although this design was influential and enabled many early advances in machine translation and language modeling, it introduced several practical and conceptual limitations.

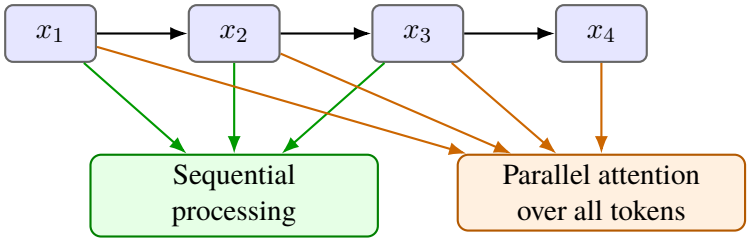
First, recurrent computation is inherently sequential. Because each hidden state depends on the previous one, training cannot fully exploit parallel hardware. This makes large-scale training slow and expensive. Second, recurrent models often struggle to preserve information across long spans of text. Even with mechanisms such as gating, distant tokens may become difficult to retrieve reliably. Third, recurrent hidden states compress the history of the sequence into a limited representation, which can act as a bottleneck when the input is long or structurally complex.

The transformer addressed these limitations by replacing recurrence with attention [1]. Instead of forcing information to pass through a chain of hidden states, the transformer allows each token to directly access other tokens in the sequence. This makes the flow of information more flexible and substantially improves scalability.

Figure 3.1 illustrates this conceptual shift. The difference is not merely architectural; it changed what could be trained in practice. Once attention-based models could scale efficiently, language modeling performance improved rapidly.

3.2 Representing Text as Model Input

Before a transformer can process language, text must be converted into a numerical form. This process involves several stages: tokenization, embedding, and the addition of positional information.



Earlier recurrent models propagate information step by step.Transformers compare tokens directly through attention.

Figure 3.1: A conceptual contrast between sequential recurrent processing and transformer-style attention. Transformers avoid a strictly step-by-step dependency during training and allow richer interactions across the sequence.

3.2.1 Tokenization

Transformers do not operate directly on words or sentences. Instead, text is segmented into units called *tokens*. In modern LLMs, tokens are usually subword units rather than full words. This choice balances vocabulary size and flexibility. Common words may appear as single tokens, while rare words can be decomposed into smaller pieces.

For example, the sentence

"Transformers are powerful models."

may be split into a sequence such as:

```
[Transform, ers, are, powerful, models, .]
```

The exact segmentation depends on the tokenizer, but the principle remains the same: text is mapped into a sequence of discrete symbols that can be indexed numerically.

3.2.2 Token Embeddings

Once tokenized, each token is mapped to a dense vector called an *embedding*. If the vocabulary size is V and the hidden dimension is d, the embedding matrix is typically a learned parameter matrix of size $V \times d$. Each token index selects one row from this matrix.

These embeddings allow the model to represent tokens in a continuous vector space, where geometric relationships can capture useful statistical regularities. Unlike one-hot vectors, embeddings are compact and learnable.

3.2.3 Positional Information

A transformer can compare tokens to one another, but self-attention alone does not inherently encode order. Without additional information, the model would know which tokens are present but not whether one token appears before or after another. To address this, the input representation includes *positional encodings* or *positional embeddings*.

If $\mathbf{e}_t$ is the token embedding at position t and $\mathbf{p}_t$ is the positional representation, the transformer input is often written as:

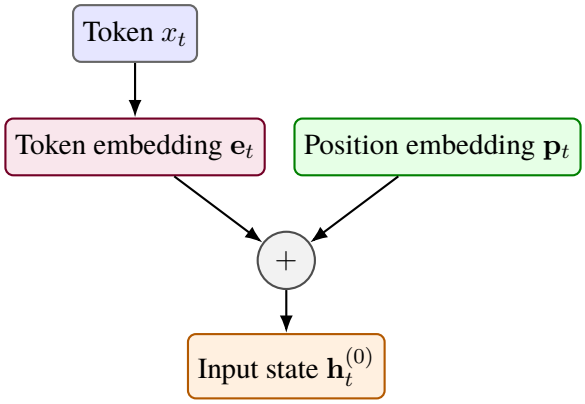


Figure 3.2: Each input position combines a token embedding with positional information before entering the transformer stack.

$$\mathbf{h}_t^{(0)} = \mathbf{e}_t + \mathbf{p}_t \tag{3.1}$$

This combined representation gives the model both lexical and positional information.

Figure 3.2 summarizes this process. Every position in the sequence becomes a vector that combines what the token is with where it appears.

3.3 Self-Attention in Detail

Self-attention is the defining operation of the transformer. It allows each token to dynamically gather information from other tokens in the sequence.

3.3.1 Queries, Keys, and Values

For each input representation, the model computes three derived vectors:

- a **query** vector, which represents what the current token is looking for,
- a **key** vector, which represents what each token offers,
- a **value** vector, which represents the information content of each token.

These are obtained through learned linear projections:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V \tag{3.2}$$

where X is the matrix of input states and W_Q, W_K, W_V are learned parameter matrices.

The attention score between one token and another is determined by the similarity between the query of the first token and the key of the second token. These scores are scaled and normalized:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{3.3}$$

The result is a weighted combination of value vectors, where more relevant tokens receive greater weight.

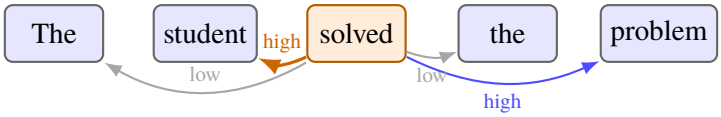


Figure 3.3: An intuitive view of self-attention. The token "solved" may focus strongly on semantically related words such as "student" and "problem".

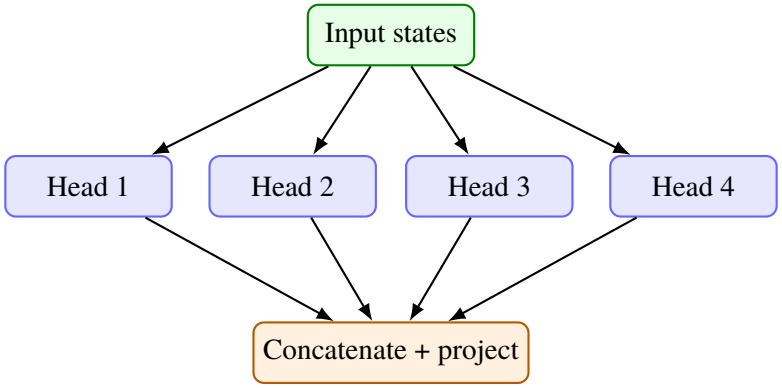


Figure 3.4: Multi-head attention allows the model to learn several complementary patterns of attention at the same time.

3.3.2 Intuition

The key idea is that a token does not have to rely only on nearby context or a compressed hidden state. Instead, it can directly attend to the most relevant parts of the sequence. A pronoun may attend to a noun earlier in the sentence, a verb may attend to its subject, and a summarization model may attend broadly across the entire input.

Figure 3.3 shows this intuition. Attention is not a fixed grammar rule but a learned mechanism that adapts to the data and the task.

3.4 Multi-Head Attention

A single attention operation can learn useful dependencies, but one pattern of comparison is often not enough. Transformers therefore use *multi-head attention*, in which several attention mechanisms operate in parallel.

If there are H heads, the model projects the input into H different query, key, and value spaces. Each head can specialize in different relationships, such as local structure, long-distance dependency, entity reference, or semantic role. The outputs of all heads are then concatenated and projected back into the model dimension.

Formally, if each head is indexed by i, then:

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i) \tag{3.4}$$

and the final output is:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)W_O \tag{3.5}$$

where W_O is a learned output projection.

This parallelism is one of the reasons transformers are expressive. Rather than learning one notion of

relevance, the model can learn many.

3.5 The Transformer Block

The transformer is built by stacking repeated layers, often called *transformer blocks*. Each block contains two major sublayers:

1. a self-attention module,
2. a position-wise feed-forward network.

These are combined with residual connections and layer normalization to stabilize optimization.

3.5.1 Residual Connections

A residual connection adds the input of a sublayer to its output. If $f(\mathbf{x})$ is the transformation performed by a sublayer, the residual update takes the form:

$$\mathbf{y} = \mathbf{x} + f(\mathbf{x}) \tag{3.6}$$

This helps gradients flow through deep networks and reduces optimization difficulty.

3.5.2 Layer Normalization

Layer normalization rescales activations in a way that improves training stability. In deep transformer stacks, this is essential for maintaining numerical consistency and allowing the model to learn effectively across many layers.

3.5.3 Feed-Forward Networks

After self-attention, each position is passed independently through a small feed-forward neural network. This network usually expands the hidden dimension, applies a non-linear activation, and projects back down. Although simple, it greatly increases the representational power of the block.

Figure 3.5 shows the internal structure of a typical block. In practice, an LLM consists of many such blocks stacked on top of one another.

3.6 Causal Masking and Autoregressive Generation

Not all transformers are used in the same way. For generative LLMs, the most important variant is the *decoder-only transformer*. Its defining feature is *causal masking*.

3.6.1 Why Causal Masking Is Needed

When predicting the next token, the model must not see future tokens. Otherwise, training would become trivial and generation would not reflect the real inference setting. Causal masking enforces this constraint by ensuring that position t can only attend to positions 1 through t.

In other words, the model can look backward but not forward.

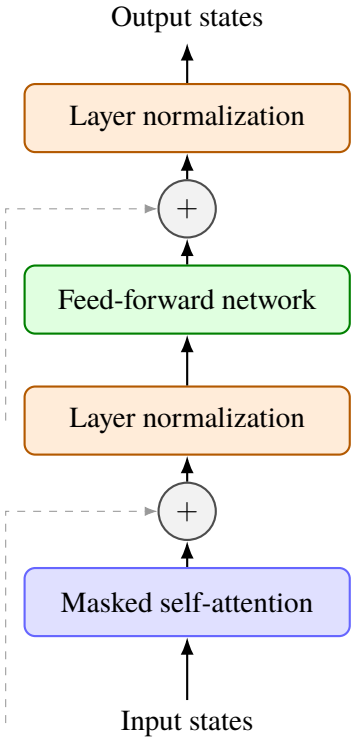


Figure 3.5: A simplified decoder-style transformer block with self-attention, feed-forward transformation, residual connections, and normalization.

3.6.2 Autoregressive Factorization

A decoder-only transformer models the probability of a token sequence as:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}) \tag{3.7}$$

At training time, the model learns all next-token predictions in parallel under the mask. At inference time, it generates one token at a time, appending each new token to the context and repeating the process.

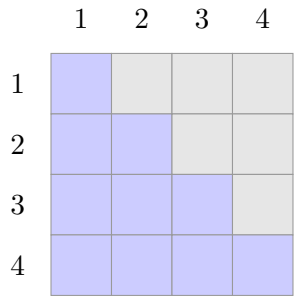
Figure 3.6 illustrates this masking pattern. This constraint is central to autoregressive language modeling and explains why decoder-only transformers are well suited to open-ended text generation.

3.7 Encoder-Only, Decoder-Only, and Encoder–Decoder Models

Transformers can be organized into three broad design families.

3.7.1 Encoder-Only Models

Encoder-only models process the full sequence bidirectionally. Every token can attend to every other token. This makes such models effective for understanding tasks such as classification, retrieval, and tagging. BERT is a well-known example of this design.



Blue cells indicate allowed attention; gray cells indicate masked future positions.

Figure 3.6: A causal attention mask for a four-token sequence. Each position can attend only to itself and earlier positions.

3.7.2 Decoder-Only Models

Decoder-only models use causal masking and are trained to predict the next token. This makes them naturally suited for generation. Most modern chat-oriented LLMs belong to this category because it supports dialogue, code generation, summarization, and free-form reasoning through a unified objective.

3.7.3 Encoder–Decoder Models

Encoder–decoder models combine both designs. The encoder processes the input sequence, and the decoder generates an output sequence while attending to the encoder states. This architecture is especially effective for sequence-to-sequence tasks such as translation and abstractive summarization.

For many agentic systems, decoder-only models are especially important because they can generate actions, plans, tool calls, and explanations using the same autoregressive mechanism.

3.8 Depth, Width, and Model Capacity

A transformer's behavior depends not only on its architecture but also on its scale. Several design dimensions influence capacity:

- **Depth:** the number of transformer layers
- **Width:** the hidden dimensionality of each layer
- **Number of heads:** how many parallel attention mechanisms are used
- **Context length:** how many tokens the model can process at once

Increasing these dimensions usually improves performance, but it also increases memory usage, training cost, and inference latency. In practice, model design involves balancing capability, efficiency, and deployment constraints.

This trade-off is especially relevant when comparing small language models, large language models, and frontier systems. A larger transformer can generally model richer patterns, but only if sufficient data and optimization are available.

3.9 Why Transformer Architecture Matters for Agents

The transformer is more than a technical curiosity; it directly shapes what later agent systems can do. Several properties are particularly relevant.

First, self-attention enables the model to integrate information across long prompts. This matters when an agent must reason over instructions, memory, tool descriptions, and retrieved documents in the same context window.

Second, autoregressive generation allows the model to produce structured outputs step by step. These outputs may include natural language explanations, plans, code, JSON tool calls, or action sequences.

Third, the deep stack of transformer layers supports highly expressive internal computation. Although the model is still fundamentally a token predictor, the learned representations can support behaviors that appear as planning, decomposition, or intermediate reasoning.

Finally, the modularity of the architecture makes it easy to integrate into larger systems. The transformer can be paired with retrieval mechanisms, external memory, function-calling interfaces, or execution environments. These combinations are central to LLM agents.

3.10 Limitations of the Transformer

Despite its success, the transformer has important limitations.

The most well-known limitation is the computational cost of attention. Standard self-attention scales quadratically with sequence length, since every token can attend to every other token. This becomes expensive for long contexts.

A second limitation is that transformers do not possess persistent memory by default. They operate over a finite context window, and once that window is exceeded, earlier information may be lost unless the surrounding system stores and reintroduces it.

A third limitation is that transformer-based generation is probabilistic rather than guaranteed to be correct. Fluency can mask uncertainty, and the architecture alone does not ensure truthfulness, faithfulness, or sound reasoning.

These limitations help explain why LLM agents often require additional components such as retrieval, memory modules, verifiers, tool use, and control logic.

3.11 Summary

This chapter examined the transformer architecture, the core design behind modern LLMs. We began by explaining why transformers replaced recurrent models, then described how text is represented through token embeddings and positional information. We explored self-attention, multi-head attention, the structure of a transformer block, and the importance of causal masking in decoder-only language models. We also compared encoder-only, decoder-only, and encoder–decoder variants and discussed the connection between transformer design and agent behavior.

Understanding the transformer is essential because it explains how LLMs process context, generate outputs, and scale to modern capability levels. In the next chapter, we move from architecture to training and examine how transformer models become LLMs through pretraining, fine-tuning, alignment, and evaluation.

References

[1] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

Part II

Core Agent Patterns

Chapter 4 Study of Single Agents

In this chapter we study the simplest class of agents: single agents with a single execution process. Such systems form the conceptual and architectural foundation for the more complex multi-agent systems introduced in later chapters. We begin by constructing simple agents using LangGraph and progressively extend the framework toward richer computational structures.

To formalize the computational structure underlying LangGraph, we first recall the mathematical notion of a graph.

Definition 4.1

*A **graph** is an ordered pair $G = (\mathcal{V}, \mathrm{E})$ where*

- *$\mathcal{V}$ is a finite set whose elements are called* vertices *(or* nodes*), and*
- *E is a set of edges describing relationships between vertices.*

♣

In the case of an *undirected graph*, each edge is represented as a two-element subset of vertices. Formally,

$$\mathrm{E} \subseteq \{\{u, v\} \mid u, v \in \mathcal{V},\ u \neq v\}.$$

Thus, if $\{u, v\} \in \mathrm{E}$, we say that vertices u and v are connected by an edge.

Within the context of LangGraph, vertices correspond to computational units such as LLM invocations, tool calls, or control logic. Edges represent the possible transitions between these units during execution. Consequently, an agent implemented in LangGraph may be viewed as a computational graph whose structure determines the flow of information and control between nodes.

Figure 4.1 illustrates several simple graphs together with their corresponding vertex and edge sets.

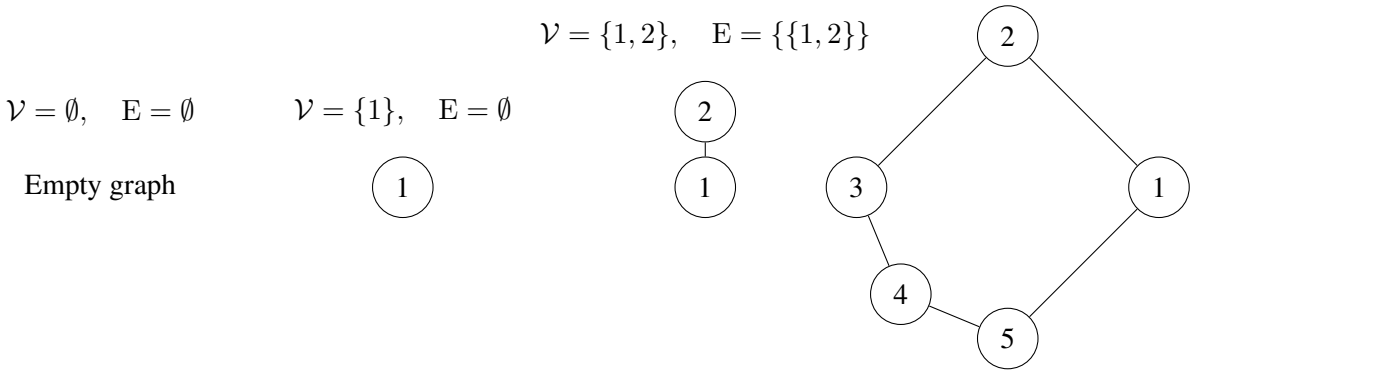


Figure 4.1: Examples of graphs together with their vertex and edge sets.

These examples illustrate several fundamental cases: the empty graph, a single-vertex graph, a graph containing a single edge, and a cyclic graph consisting of five vertices. In subsequent sections, similar graph structures will be used to model increasingly complex execution flows for LangGraph agents.

Before constructing such graphs programmatically, we first introduce the minimal set of Python libraries required for implementing a LangGraph agent. Listing 4.1 shows the essential imports used throughout this chapter.

Listing 4.1: Minimal imports for a LangGraph agent

```
1 from typing import TypedDict, Annotated
```

```
2
3 from langgraph.graph import StateGraph, START, END
4 from langgraph.graph.message import add_messages
5
6 from langchain_core.messages import HumanMessage
7 from langchain_openai import ChatOpenAI
```

4.1 Our First Graph in LangGraph

We now construct our first executable graph using LangGraph. As discussed in the previous section, an agent can be represented as a computational graph in which vertices correspond to computational processes and edges describe the possible transitions between these processes.

The simplest possible LangGraph consists of three components:

- a distinguished starting vertex, denoted by `START`,
- a single computational node that performs some processing, and
- a terminal vertex, denoted by `END`.

Execution of the graph begins at the `START` vertex. Control is then transferred to the computational node, which performs some operation on the current state of the system. Once the node finishes its computation, execution proceeds to the `END` vertex, terminating the process.

This structure forms the simplest possible execution pipeline and may be viewed as a directed path consisting of three vertices.

Figure 4.2 illustrates this minimal computational graph.

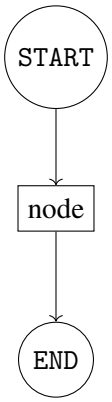


Figure 4.2: The simplest LangGraph execution pipeline.

To implement this graph in Python, we first define the *state* of the agent. In LangGraph, the state represents the information passed between nodes during execution. A convenient way to define such a state is by using a `TypedDict`, which provides a structured representation of the data.

Next, we define the computational node. A node is simply a function that takes the current state as input and returns an updated state. Finally, we construct the graph by specifying its nodes and edges, compile it, and execute it.

Listing 4.2 shows a minimal "Hello World" example of a LangGraph agent.

Listing 4.2: A LangGraph Hello World example

```
1 from typing import TypedDict
```

```python
from langgraph.graph import StateGraph, START, END

class AgentState(TypedDict):
    message: str

def node(state: AgentState) -> AgentState:
    return {"message": f"Hello, {state['message']}!"}

graph = (
    StateGraph(AgentState)
    .add_node("node", node)
    .add_edge(START, "node")
    .add_edge("node", END)
)

app = graph.compile()

app.invoke({"message": "Sourena"})
```

The execution proceeds as follows:

1. The initial state {`message` : "Sourena"} is passed to the graph.
2. Execution begins at the `START` vertex.
3. Control transitions to the node labeled `node`.
4. The node processes the state and returns the updated message `"Hello, Sourena!"`.
5. Execution proceeds to the `END` vertex and the graph terminates.

Although extremely simple, this example demonstrates the essential components of a LangGraph program: a state definition, computational nodes, and a directed execution structure. More sophisticated agents can be constructed by adding additional nodes, branching logic, and tool integrations.

4.2 Graphs with Multiple Processes

In the previous section we considered the simplest possible execution graph, consisting of a single computational node. In practice, however, most agents require multiple processing stages. Each stage performs a specific computation and passes the resulting state to the next stage in the pipeline.

A natural extension of the previous example is therefore a graph containing multiple computational nodes arranged in sequence. Formally, this structure corresponds to a directed path

$$\texttt{START} \to v_1 \to v_2 \to \cdots \to v_k \to \texttt{END},$$

where each vertex v_i represents a computational process that transforms the agent state.

During execution, the state flows through the graph sequentially. Each node receives the current state, performs a transformation, and returns an updated state that becomes the input for the next node.

Figure 4.3 illustrates a simple example consisting of four processing nodes connected in a linear pipeline.

We now implement this graph using LangGraph. As before, we define a state representation using a `TypedDict`. Each node receives the state, modifies the message, and returns the updated state.

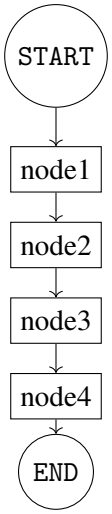


Figure 4.3: LangGraph execution pipeline with multiple sequential processes.

Listing 4.3 shows a simple implementation containing four nodes. Each node gradually transforms the message until the final output is produced.

Listing 4.3: A LangGraph with multiple sequential processes

```
from typing import TypedDict
from langgraph.graph import StateGraph, START, END

class AgentState(TypedDict):
    message: str

def node1(state: AgentState) -> AgentState:
    return {"message": f"Hello {state['message']}"}

def node2(state: AgentState) -> AgentState:
    return {"message": f"{state['message']}!"}

def node3(state: AgentState) -> AgentState:
    return {"message": f"{state['message']} my"}

def node4(state: AgentState) -> AgentState:
    return {"message": f"{state['message']} friend"}

graph = (
    StateGraph(AgentState)
    .add_node("node1", node1)
    .add_node("node2", node2)
    .add_node("node3", node3)
    .add_node("node4", node4)
    .add_edge(START, "node1")
    .add_edge("node1", "node2")
    .add_edge("node2", "node3")
    .add_edge("node3", "node4")
    .add_edge("node4", END)
```

```
)

app = graph.compile()

app.invoke({"message": "Sourena"})
```

The execution proceeds sequentially through the pipeline:

1. The graph receives the initial state {`message` : ”Sourena”}.
2. The first node produces the message `"Hello Sourena"`.
3. The second node appends punctuation, producing `"Hello Sourena!"`.
4. The third node extends the phrase to `"Hello Sourena! my"`.
5. The fourth node completes the sentence `"Hello Sourena! my friend"`.
6. Execution then reaches the `END` vertex and the graph terminates.

This example illustrates an important conceptual idea: each node performs a local transformation of the state, while the overall graph determines the global structure of the computation. More complex agent architectures can be constructed by introducing branching, conditional routing, loops, and tool invocations.

4.3 Conditional Graphs and Branching

In the previous sections we considered graphs whose execution followed a fixed linear path. While such pipelines are useful for simple workflows, many practical agent systems require the ability to choose between multiple possible actions. This motivates the introduction of *conditional graphs*, in which the next node executed by the graph is determined dynamically during runtime.

In a conditional graph, the execution flow is no longer predetermined. Instead, the graph evaluates a condition and selects one of several possible successors. This mechanism introduces *control flow* into the execution structure.

More formally, suppose a node v has outgoing edges

$$(v, u_1), (v, u_2), \ldots, (v, u_k).$$

A routing function

$$r : S \to \{u_1, u_2, \ldots, u_k\}$$

maps the current state S to the next node that should be executed. The choice of successor therefore depends on the information contained in the state rather than being statically defined in the graph structure.

Conditional graphs are particularly useful in agent systems because they allow the program to adapt to changing circumstances. For example, a node may select different tools, perform classification, trigger fallback logic, or terminate early depending on the current state of the computation.

Figure 4.4 illustrates a simple branching structure. Execution begins at `START`. A condition is evaluated, and depending on the result the graph transitions either to `node1` or to `node2`. After either branch completes, the execution converges again at `END`. Such branching patterns occur frequently in workflow orchestration and decision-making pipelines.

In LangGraph, conditional execution is implemented using `add_conditional_edges`. Instead of connecting a node directly to a fixed successor, we provide a routing function that examines the current state and determines which node should be executed next.

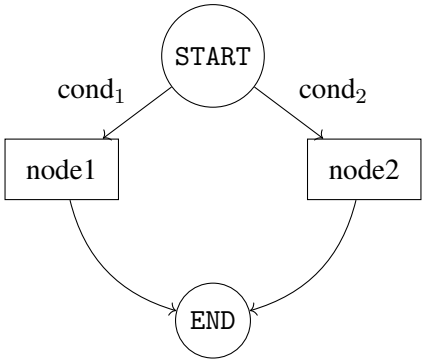


Figure 4.4: A conditional graph with two possible execution branches.

Listing 4.4 demonstrates a simple example in which the graph selects between two possible nodes. For illustrative purposes the routing decision is random, although in practice such routing functions usually depend on properties of the state or the output of an LLM.

Listing 4.4: LangGraph conditional branching

```
from typing import TypedDict, Literal
from langgraph.graph import StateGraph, START, END
import random


class AgentState(TypedDict):
    message: str


def node1(state: AgentState) -> AgentState:
    return {"message": f"node1 processed: {state['message']}"}


def node2(state: AgentState) -> AgentState:
    return {"message": f"node2 processed: {state['message']}"}


def conditional_route(state: AgentState) -> Literal["node1", "node2"]:
    if random.random() > 0.5:
        return "node1"
    return "node2"


graph = (
    StateGraph(AgentState)
    .add_node("node1", node1)
    .add_node("node2", node2)
    .add_conditional_edges(
        START,
        conditional_route,
        {
            "node1": "node1",
```

```
            "node2": "node2",
        },
    )
    .add_edge("node1", END)
    .add_edge("node2", END)
    .compile()
)

result = graph.invoke({"message": "hello"})
print(result)
```

In Listing 4.4, execution begins at `START`. The routing function `conditional_route` is invoked immediately and returns either `"node1"` or `"node2"`. LangGraph then follows the corresponding edge and executes the selected node. Once that node finishes, the graph proceeds to the terminal vertex `END`, at which point execution terminates.

Conditional routing is a central mechanism for building intelligent agent systems. By combining branching logic with language model outputs, tool calls, and memory updates, developers can construct complex decision-making workflows while maintaining the clear structural representation provided by graph-based execution.

4.4 Loops and Iterative Graphs

So far we have considered graphs whose execution flows either sequentially or through conditional branches. However, many computational processes require *iteration*, where a particular step is repeated until some condition is satisfied. In graph-theoretic terms, such behavior corresponds to the presence of a *cycle* in the directed graph.

A cycle occurs when a node has a path that eventually leads back to itself. This allows the execution to revisit previously executed nodes and perform repeated computation. Iterative structures are fundamental in programming and often correspond to classical constructs such as `while` loops or `retry` mechanisms.

Figure 4.5 illustrates a simple iterative graph. Execution begins at the `START` vertex and proceeds to `node1`. After `node1` completes its computation, the graph evaluates a condition. If the condition indicates that the task should continue, execution returns to `node1`, forming a loop. Otherwise, the graph transitions to the terminal vertex `END`.

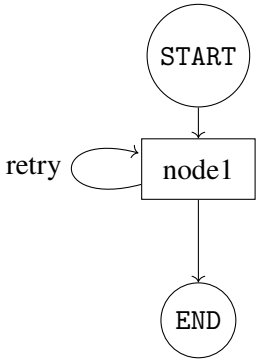


Figure 4.5: An iterative graph containing a loop.

In LangGraph, such behavior is implemented by combining a normal edge with a conditional routing function. The node performs some computation and then a routing function decides whether execution should con-

tinue looping or terminate.

Listing 4.5 demonstrates a minimal example of this pattern. The graph maintains a counter in the state and repeatedly increments it until the counter reaches a fixed threshold.

Listing 4.5: LangGraph loop example

```python
from typing import TypedDict, Literal
from langgraph.graph import StateGraph, START, END


class AgentState(TypedDict):
    count: int


def node1(state: AgentState) -> AgentState:
    return {
        "count": state["count"] + 1,
    }


def route(state: AgentState) -> Literal["node1", "END"]:
    return "node1" if state["count"] < 3 else "END"


graph = StateGraph(AgentState)

graph.add_node("node1", node1)
graph.add_edge(START, "node1")

graph.add_conditional_edges(
    "node1",
    route,
    {
        "node1": "node1",
        "END": END,
    },
)

app = graph.compile()

result = app.invoke({"count": 0})
print(result)
```

The execution proceeds as follows:

1. The graph receives the initial state $\{\texttt{count} : 0\}$.
2. Execution begins at the `START` vertex and transitions to `node1`.
3. The node increments the counter and returns the updated state.
4. The routing function `route` inspects the new value of `count`.
5. If the counter is less than 3, execution returns to `node1`, forming a loop.

6. Once the counter reaches 3, the routing function returns `END`, and execution terminates.

Iterative graphs are particularly important in agent systems because many reasoning processes require repeated interaction between components. For example, a language model may repeatedly generate thoughts, call tools, and update its internal state until it determines that a final answer has been reached. Such iterative reasoning patterns can be naturally expressed using graph cycles in LangGraph.

By combining sequential pipelines, conditional branching, and iterative loops, LangGraph provides a flexible framework for representing complex agent workflows while preserving a clear and structured computational model.

4.5 Graphs with Tools and LLMs

The previous sections introduced the structural components of LangGraph, including sequential pipelines, conditional branching, and iterative loops. While these patterns describe the control flow of a graph, real agent systems often require interaction with external computational resources such as databases, APIs, or specialized algorithms. These resources are commonly referred to as *tools*.

Modern language-model-based agents frequently combine a large language model (LLM) with a collection of tools that extend the model's capabilities. The language model performs reasoning and decides when a tool should be invoked, while the tool performs deterministic computation. The result of the tool call is then returned to the model, which can incorporate the result into further reasoning.

Figure 4.6 illustrates a simple interaction pattern between a graph node, a language model, and an external tool. The node sends a prompt to the LLM, which may call a tool if necessary. The tool executes its computation and returns the result to the model. Finally, the node produces an updated state and the graph terminates.

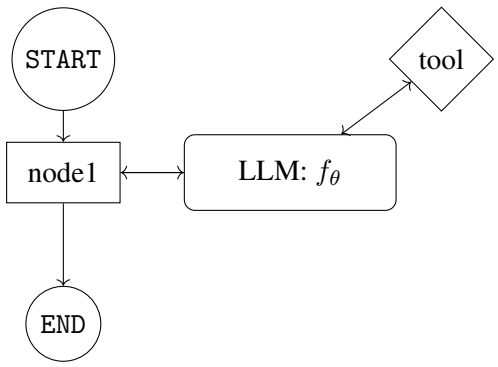


Figure 4.6: Interaction between a LangGraph node, an LLM, and an external tool.

To demonstrate this pattern, we construct a simple agent that uses a language model together with a custom tool. The tool computes the sum of a list of numbers, while the language model determines when the tool should be used.

Listing 4.6 shows a minimal LangGraph program implementing this interaction.

Listing 4.6: LangGraph example using an LLM with tools

```
from typing import TypedDict, List
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.tools import tool
from langgraph.graph import StateGraph, START, END
import os
```

```python
import getpass


if "GOOGLE_API_KEY" not in os.environ:
    os.environ["GOOGLE_API_KEY"] = getpass.getpass("Enter API Key: ")


class AgentState(TypedDict):
    message: str


@tool
def sum_all(numbers: List[int]) -> int:
    """Return the sum of a list of integers."""
    return sum(numbers)


llm = ChatGoogleGenerativeAI(
    model="gemini-1.5-pro",
    temperature=1.0,
)

llm_with_tools = llm.bind_tools([sum_all])


def node1(state: AgentState) -> AgentState:
    response = llm_with_tools.invoke(state["message"])
    return {"message": response}


graph = (
    StateGraph(AgentState)
    .add_node("node1", node1)
    .add_edge(START, "node1")
    .add_edge("node1", END)
)

app = graph.compile()

result = app.invoke({
    "message": "What is the sum of 1, 2, 3, and 4?"
})

print(result)
```

The execution proceeds as follows:

1. The graph receives the initial message from the user.
2. The node sends the message to the language model.

3. The language model determines whether a tool should be used.
4. If a tool call is required, the model invokes the appropriate tool.
5. The tool executes its computation and returns the result.
6. The model incorporates the tool output into its response.
7. The node updates the state and the graph transitions to `END`.

This example illustrates the fundamental architecture of many modern agent systems: a language model responsible for reasoning and planning, combined with tools that perform precise computation or access external information. By embedding this interaction within a graph structure, LangGraph enables developers to build complex agent workflows that remain modular, interpretable, and easy to extend.

Chapter 5 State, Messages, and Memory in LangGraph

In the previous chapter we introduced the structural foundations of LangGraph, showing how agents can be modeled as graphs whose nodes represent computational processes and whose edges determine the flow of execution. While the graph structure defines *how* computation proceeds, an equally important question is *what information* flows through the graph during execution.

This chapter therefore focuses on three closely related concepts that determine how information is represented and preserved within LangGraph agents:

- **State**, which represents the data passed between nodes.
- **Messages**, which provide a structured format for communication between users, language models, and tools.
- **Memory**, which allows agents to persist information across multiple steps or interactions.

Together, these components form the informational backbone of an agent system. The graph defines the computational structure, while the state and memory mechanisms determine how knowledge evolves during execution.

5.1 The Concept of State

In LangGraph, the *state* of an agent represents the collection of data that is passed between nodes during execution. Each node receives the current state as input, performs some computation, and returns an updated state that is then forwarded to subsequent nodes.

Formally, if we denote the state space by $\mathcal{S}$, then each node in the graph implements a function

$$f_i : \mathcal{S} \to \mathcal{S},$$

mapping an input state to an updated output state. The overall execution of the graph can therefore be viewed as the composition of these state-transforming functions along the edges of the graph.

In practice, the state is typically represented using a structured Python object such as a `TypedDict`. This representation provides both flexibility and clarity, allowing developers to specify exactly which pieces of information are available during execution.

Listing 5.1 shows a simple example of a state definition.

Listing 5.1: Defining a simple LangGraph state

```
from typing import TypedDict

class AgentState(TypedDict):
    message: str
    step: int
```

In this example the state consists of two fields: a message string and a counter indicating the current step of the computation. Nodes may update either field, depending on the logic of the agent.

Because the state is explicitly defined, LangGraph ensures that each node receives and returns a consistent structure. This design makes the flow of information through the graph easier to reason about and debug.

5.1.1 State Fields and Their Roles

In realistic agent systems, states tend to carry a variety of fields that serve different roles during execution. Table 5.1 lists several common categories.

Table 5.1: Common categories of state fields in LangGraph agents.

Category	Example fields	Purpose
Conversational	`messages`, `history`	Track dialogue context
Control	`step`, `iteration`	Guide routing decisions
Domain data	`query`, `results`	Hold task-specific payloads
Metadata	`user_id`, `timestamp`	Provide contextual information
Flags	`needs_review`, `is_done`	Signal conditions to edges

A well-designed state captures exactly the information required by the nodes in the graph — neither more (which introduces unnecessary coupling) nor less (which forces nodes to re-derive information).

5.2 State Transformation Across Nodes

Once a state representation has been defined, each node in the graph becomes a function that modifies some portion of that state. Nodes may add new information, update existing fields, or derive new values based on previous computation.

Suppose we have a state space $\mathcal{S}$ and three nodes implementing functions

$$f_1 : \mathcal{S} \to \mathcal{S}, \qquad f_2 : \mathcal{S} \to \mathcal{S}, \qquad f_3 : \mathcal{S} \to \mathcal{S}.$$

If the graph contains the path

$$\texttt{START} \to v_1 \to v_2 \to v_3 \to \texttt{END},$$

then the resulting computation corresponds to the composition

$$s_{\text{final}} = f_3\big(f_2\big(f_1(s_0)\big)\big),$$

where s_0 denotes the initial state provided to the graph.

Figure 5.1 illustrates the state as it evolves across three nodes.

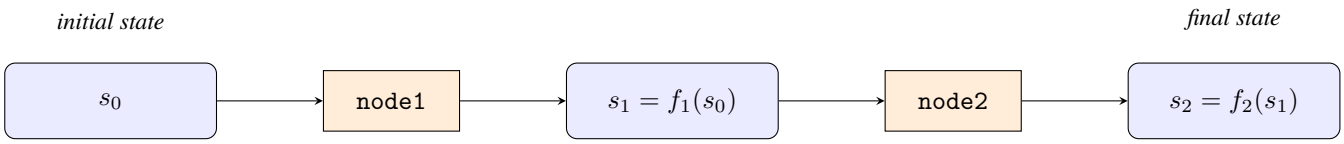


Figure 5.1: State evolution across two nodes. Each node receives the current state and produces an updated version that is forwarded to the next node.

Listing 5.2 illustrates a concrete example in which three nodes progressively build up a research summary.

Listing 5.2: Multi-step state transformation: a research pipeline

```
from typing import TypedDict
from langgraph.graph import StateGraph, START, END


class ResearchState(TypedDict):
```

```
    topic: str
    raw_data: str
    analysis: str
    summary: str
    step: int


def gather(state: ResearchState) -> ResearchState:
    """Simulate data retrieval."""
    return {
        "raw_data": f"Data about '{state['topic']}' ...",
        "step": state["step"] + 1,
    }


def analyze(state: ResearchState) -> ResearchState:
    """Derive insights from raw data."""
    return {
        "analysis": f"Key findings from: {state['raw_data'][:40]}",
        "step": state["step"] + 1,
    }


def summarize(state: ResearchState) -> ResearchState:
    """Produce a final summary."""
    return {
        "summary": f"Summary: {state['analysis']}",
        "step": state["step"] + 1,
    }


graph = (
    StateGraph(ResearchState)
    .add_node("gather", gather)
    .add_node("analyze", analyze)
    .add_node("summarize", summarize)
    .add_edge(START, "gather")
    .add_edge("gather", "analyze")
    .add_edge("analyze", "summarize")
    .add_edge("summarize", END)
)

app = graph.compile()

result = app.invoke({
    "topic": "quantum computing",
    "raw_data": "",
    "analysis": "",
```

```
    "summary": "",
    "step": 0,
})
print(result)
# {'topic': 'quantum computing',
#  'raw_data': "Data about 'quantum computing' ...",
#  'analysis': "Key findings from: Data about 'quantum c",
#  'summary': "Summary: Key findings from: Data about ...",
#  'step': 3}
```

During execution, each node operates only on the information available in the state. Note that nodes need not return *all* fields — LangGraph merges the returned dictionary into the existing state, so omitted fields retain their previous values.

5.2.1 Partial Updates and Merge Semantics

An important design choice in LangGraph is that node functions do not need to return the entire state. Instead, they return a dictionary containing only the fields they wish to update. LangGraph *merges* the returned dictionary into the current state using the following rule:

> For each key k in the returned dictionary, the state field $s[k]$ is replaced by the new value. All other fields remain unchanged.

This behaviour can be summarised by the merge operator $\oplus$:

$$s' = s \oplus \Delta s, \quad \text{where} \quad s'[k] = \begin{cases} \Delta s[k] & \text{if } k \in \Delta s, \\ s[k] & \text{otherwise.} \end{cases}$$

This partial-update model greatly simplifies node implementations, because each node only needs to concern itself with the fields it is responsible for.

5.3 State Reducers

The default merge semantics described above — overwriting the existing value — work well for scalar fields such as strings and integers. However, there are situations where *accumulating* values is more appropriate. For example, a list of messages should *grow* over time rather than being replaced at each step.

LangGraph addresses this through **state reducers**. A reducer is a function

$$r : (V_{\text{old}}, V_{\text{new}}) \to V_{\text{merged}},$$

that determines how a new value is combined with the existing value of a given field. By annotating a state field with a reducer, developers can customise the merge behaviour on a per-field basis.

The most commonly used built-in reducer is `add_messages`, which appends new messages to an existing list. However, the mechanism is fully general: any binary function with the appropriate signature can serve as a reducer.

Figure 5.2 contrasts the default overwrite behaviour with reducer-based accumulation.

Listing 5.3 demonstrates a state with two reducer-annotated fields.

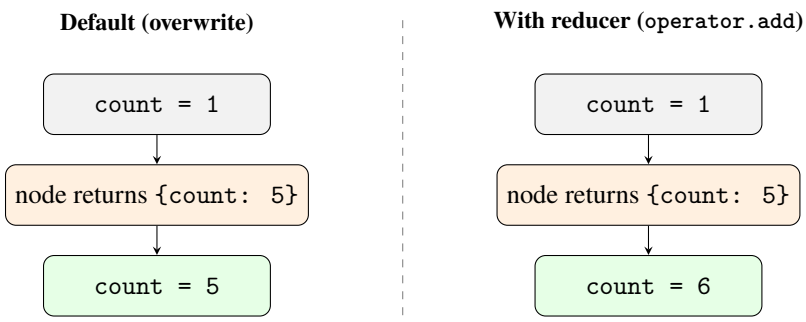


Figure 5.2: Default overwrite versus reducer-based accumulation. The left column shows the standard behaviour; the right column shows the effect of annotating the field with `operator.add`.

Listing 5.3: State fields with custom reducers

```python
import operator
from typing import TypedDict, Annotated
from langgraph.graph import StateGraph, START, END


class CounterState(TypedDict):
    total: Annotated[int, operator.add] # accumulate
    labels: Annotated[list, operator.add] # concatenate
    status: str                         # overwrite


def step_a(state: CounterState) -> CounterState:
    return {
        "total": 10,
        "labels": ["step_a"],
        "status": "running",
    }


def step_b(state: CounterState) -> CounterState:
    return {
        "total": 20,
        "labels": ["step_b"],
        "status": "done",
    }


graph = (
    StateGraph(CounterState)
    .add_node("step_a", step_a)
    .add_node("step_b", step_b)
    .add_edge(START, "step_a")
    .add_edge("step_a", "step_b")
    .add_edge("step_b", END)
)
```

```
36
37 app = graph.compile()
38 result = app.invoke({"total": 0, "labels": [], "status": ""})
39 print(result)
40 # {'total': 30, 'labels': ['step_a', 'step_b'], 'status': 'done'}
```

After execution, the `total` field contains $0 + 10 + 20 = 30$ and `labels` contains the concatenation of the two lists. The `status` field, which has no reducer, simply reflects the last value written.

5.4 Messages as Structured Communication

While simple scalar fields are useful for representing small pieces of data, agent systems frequently need to exchange complex conversational information. To support this, LangGraph adopts the message abstractions provided by LangChain.

Messages provide a structured representation of communication between different participants in the system. Rather than representing conversation as a single string, messages maintain a sequence of structured objects describing who produced each piece of content. This structure is particularly important for language models that rely on conversation history.

Figure 5.3 shows the principal message types and their typical roles in an agent conversation.

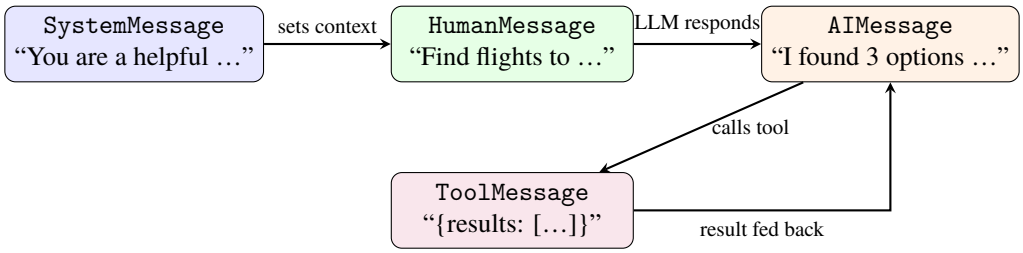


Figure 5.3: The four principal message types in a LangGraph agent conversation. Arrows indicate the typical flow of control.

Table 5.2 summarises the common message types and their purposes.

Table 5.2: LangChain message types used in LangGraph.

Type	Source	Description
`SystemMessage`	Developer	Sets the behaviour or persona of the model.
`HumanMessage`	User	Represents input from the end user.
`AIMessage`	Model	A response generated by the language model; may include tool-call requests.
`ToolMessage`	Tool	The result returned by an external tool after the model invoked it.

Listing 5.4 illustrates how messages of different types are created.

Listing 5.4: Creating structured messages

```
1 from langchain_core.messages import (
2     SystemMessage, HumanMessage, AIMessage, ToolMessage,
3 )
```

```
conversation = [
    SystemMessage(content="You are a travel planning assistant."),
    HumanMessage(content="Find flights from NYC to London."),
    AIMessage(
        content="Let me search for flights.",
        tool_calls=[{
            "id": "call_001",
            "name": "search_flights",
            "args": {"origin": "NYC", "destination": "LHR"},
        }],
    ),
    ToolMessage(
        content='[{"airline": "BA", "price": 450}, ...]',
        tool_call_id="call_001",
    ),
    AIMessage(content="I found several options. The cheapest ..."),
]
```

By representing conversations in this structured format, LangGraph can preserve dialogue history, maintain context, and enable more complex interaction patterns such as multi-step tool use.

5.5 Message-Based State in LangGraph

In many agent architectures the state primarily consists of a sequence of messages representing the conversation history. LangGraph provides built-in support for this pattern through the `add_messages` reducer.

Instead of replacing the message history at each step, nodes *append* new messages to the existing list. This ensures that the conversation context grows over time as the agent interacts with the user or with tools.

The `add_messages` reducer also provides a powerful feature: if a returned message has the same `id` as an existing message in the list, it *replaces* rather than duplicates the original. This makes it possible for nodes to update earlier messages (for example, to revise a draft response) without polluting the history.

Listing 5.5 demonstrates a conversational agent that invokes a language model at each step.

Listing 5.5: A minimal chatbot using message-based state

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, START, END
from langchain_core.messages import HumanMessage, AIMessage


class ChatState(TypedDict):
    messages: Annotated[list, add_messages]


def chatbot(state: ChatState) -> ChatState:
    """In production this would call an LLM; here we echo."""
    last = state["messages"][-1]
```

```
    reply = AIMessage(
        content=f"You said: {last.content}"
    )
    return {"messages": [reply]}


graph = (
    StateGraph(ChatState)
    .add_node("chatbot", chatbot)
    .add_edge(START, "chatbot")
    .add_edge("chatbot", END)
)

app = graph.compile()

result = app.invoke({
    "messages": [HumanMessage(content="Hello, LangGraph!")]
})

for msg in result["messages"]:
    role = msg.__class__.__name__
    print(f"{role}: {msg.content}")
# HumanMessage: Hello, LangGraph!
# AIMessage: You said: Hello, LangGraph!
```

The key observation is that the node returns only the *new* messages. The `add_messages` reducer handles appending them to the existing conversation, preserving the full dialogue history.

5.5.1 Combining Messages with Other State Fields

In practice, agent states rarely consist of messages alone. Listing 5.6 shows a richer state that pairs the conversation history with additional metadata used for routing and control.

Listing 5.6: A state combining messages with control fields

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages


class AgentState(TypedDict):
    messages: Annotated[list, add_messages] # conversation
    iteration: int                          # loop counter
    tool_output: str                        # latest tool result
    should_continue: bool                   # routing flag
```

This pattern is common in agentic workflows where the conversation history drives LLM calls, while scalar fields guide conditional edges or store intermediate results.

5.6 Conditional Routing with State

One of the most powerful features of LangGraph is the ability to route execution based on the *current state*. Rather than following a fixed sequence, the graph can branch dynamically at runtime.

A routing function inspects the state and returns the name of the next node to execute. When combined with conditional edges, this allows the graph to implement patterns such as loops, retries, and multi-path branching.

Figure 5.4 illustrates a common pattern: a node that decides whether to invoke a tool or return a final answer.

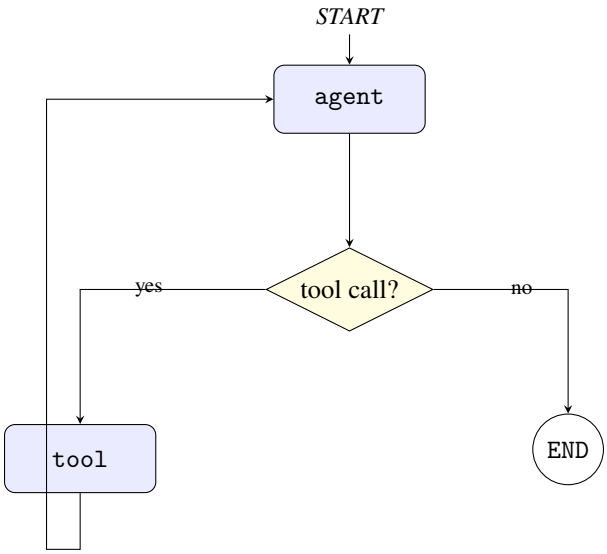


Figure 5.4: Conditional routing: the agent node produces a response; a routing function inspects the state to decide whether a tool should be called or execution should end. Tool results loop back to the agent.

Listing 5.7 provides a concrete implementation.

Listing 5.7: Conditional routing based on state

```
from typing import TypedDict, Annotated, Literal
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, START, END
from langchain_core.messages import AIMessage, ToolMessage


class AgentState(TypedDict):
    messages: Annotated[list, add_messages]


def agent(state: AgentState) -> AgentState:
    """Simulate an LLM that may request a tool call."""
    # In production, this calls an LLM.
    last = state["messages"][-1]
    if "weather" in last.content.lower():
        return {"messages": [AIMessage(
            content="",
            tool_calls=[{
                "id": "call_w",
```

```
                "name": "get_weather",
                "args": {"city": "London"},
            }],
        )]}
    return {"messages": [AIMessage(content="How can I help?")]}


def tool_node(state: AgentState) -> AgentState:
    """Execute the requested tool."""
    return {"messages": [ToolMessage(
        content="Sunny, 22C",
        tool_call_id="call_w",
    )]}


def should_call_tool(
    state: AgentState,
) -> Literal["tool_node", "__end__"]:
    last = state["messages"][-1]
    if hasattr(last, "tool_calls") and last.tool_calls:
        return "tool_node"
    return "__end__"


graph = (
    StateGraph(AgentState)
    .add_node("agent", agent)
    .add_node("tool_node", tool_node)
    .add_conditional_edges("agent", should_call_tool)
    .add_edge(START, "agent")
    .add_edge("tool_node", "agent")
)

app = graph.compile()
```

5.7 Memory and Persistent Context

So far we have discussed state as information that flows through the graph during a *single* execution. However, many agent systems require the ability to remember information across multiple interactions.

This capability is referred to as *memory*.

5.7.1 Short-Term vs. Long-Term Memory

It is useful to distinguish two broad categories of memory:

Short-term memory corresponds to the conversation history maintained within a single session. In LangGraph this is naturally represented by the message list in the state.

Long-term memory refers to information that persists across sessions or graph invocations. This requires an external storage backend.

Figure 5.5 illustrates the relationship between these two categories and the agent execution graph.

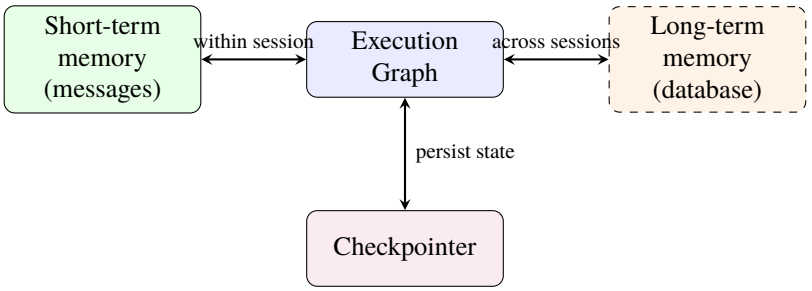


Figure 5.5: Short-term memory lives inside the state and lasts for a single session. Long-term memory and checkpointers provide persistence across sessions.

5.7.2 Checkpointers: Persisting State Across Invocations

LangGraph provides **checkpointers** as the primary mechanism for persisting the full graph state between invocations. A checkpointer serialises the state after each step (or at the end of an invocation) and restores it when the same *thread* is resumed.

Listing 5.8 shows how to enable checkpointing with an in-memory backend.

Listing 5.8: Enabling memory with a checkpointer

```python
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import MemorySaver
from langchain_core.messages import HumanMessage, AIMessage


class ChatState(TypedDict):
    messages: Annotated[list, add_messages]


def chatbot(state: ChatState) -> ChatState:
    last = state["messages"][-1]
    return {"messages": [
        AIMessage(content=f"Echo: {last.content}")
    ]}


graph = (
    StateGraph(ChatState)
    .add_node("chatbot", chatbot)
    .add_edge(START, "chatbot")
    .add_edge("chatbot", END)
)

```

```python
# Compile with a checkpointer
memory = MemorySaver()
app = graph.compile(checkpointer=memory)

# First invocation
config = {"configurable": {"thread_id": "user-123"}}
app.invoke(
    {"messages": [HumanMessage(content="Hi!")]},
    config=config,
)

# Second invocation -- previous messages are restored
result = app.invoke(
    {"messages": [HumanMessage(content="What did I say?")]},
    config=config,
)

for msg in result["messages"]:
    print(f"{msg.__class__.__name__}: {msg.content}")
# HumanMessage: Hi!
# AIMessage: Echo: Hi!
# HumanMessage: What did I say?
# AIMessage: Echo: What did I say?
```

The `thread_id` parameter acts as a session identifier. All invocations sharing the same thread ID operate on the same persisted state, enabling multi-turn conversations with full memory.

For production systems, LangGraph provides persistent backends such as `SqliteSaver` and `PostgresSaver` that store checkpoints in a database.

5.7.3 Long-Term Memory with External Stores

Beyond checkpointing, agents sometimes need to store and retrieve information that transcends any single conversation thread — for example, user preferences, knowledge base entries, or learned facts. This is typically achieved by connecting nodes to an external store such as a vector database or key–value store.

Figure 5.6 illustrates a full memory architecture combining checkpointing and external storage.

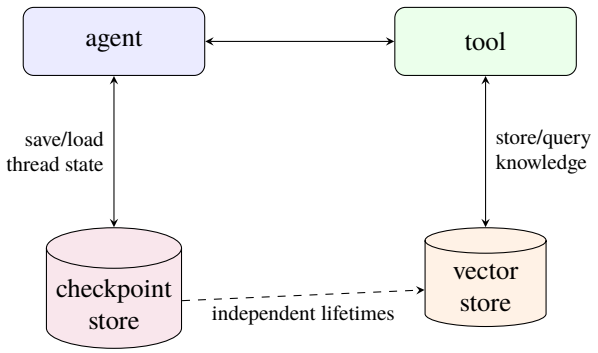


Figure 5.6: Full memory architecture. The checkpointer persists the conversational state per thread, while the vector store provides long-term knowledge accessible across all threads.

5.8 Putting It All Together: A Stateful Agent

To consolidate the concepts introduced in this chapter, Listing 5.9 presents a complete agent that combines typed state, messages, conditional routing, and checkpointed memory.

Listing 5.9: A complete stateful agent with routing and memory

```python
from typing import TypedDict, Annotated, Literal
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import MemorySaver
from langchain_core.messages import (
    HumanMessage, AIMessage, ToolMessage,
)


class AgentState(TypedDict):
    messages: Annotated[list, add_messages]
    tool_calls_made: int


def assistant(state: AgentState) -> AgentState:
    last = state["messages"][-1]
    if isinstance(last, HumanMessage) and "calc" in last.content:
        return {"messages": [AIMessage(
            content="",
            tool_calls=[{
                "id": "call_calc",
                "name": "calculator",
                "args": {"expr": "2+2"},
            }],
        )]}
    return {"messages": [
        AIMessage(content=f"Reply: {last.content}")
    ]}


def calculator(state: AgentState) -> AgentState:
    return {
        "messages": [ToolMessage(
            content="4",
            tool_call_id="call_calc",
        )],
        "tool_calls_made": state["tool_calls_made"] + 1,
    }


def route(
    state: AgentState,
```

```
) -> Literal["calculator", "__end__"]:
    last = state["messages"][-1]
    if hasattr(last, "tool_calls") and last.tool_calls:
        return "calculator"
    return "__end__"


graph = (
    StateGraph(AgentState)
    .add_node("assistant", assistant)
    .add_node("calculator", calculator)
    .add_conditional_edges("assistant", route)
    .add_edge(START, "assistant")
    .add_edge("calculator", "assistant") # loop back
)

app = graph.compile(checkpointer=MemorySaver())

config = {"configurable": {"thread_id": "demo"}}

# Turn 1
r1 = app.invoke(
    {"messages": [HumanMessage(content="calc 2+2")],
     "tool_calls_made": 0},
    config=config,
)
print(r1["messages"][-1].content)
# "Reply: 4"

# Turn 2 -- memory preserves the full history
r2 = app.invoke(
    {"messages": [HumanMessage(content="Thanks!")]},
    config=config,
)
print(len(r2["messages"])) # includes all previous turns
```

5.9 Summary

In this chapter we introduced the informational components that support LangGraph agents.

- The **state** represents the structured data passed between nodes. Each node receives the state, transforms it, and passes it forward. LangGraph merges partial updates using configurable *reducers*.
- **Messages** provide a standardized format for conversational interactions between users, models, and tools. The `add_messages` reducer enables incremental growth of the conversation history.
- **Conditional routing** allows the graph to branch dynamically based on the current state, enabling patterns such as tool-calling loops and multi-path decision making.
- **Memory** allows agents to persist information beyond a single execution. *Checkpointers* save and restore

the full state per thread, while external stores provide long-term knowledge that spans threads.

Together, these mechanisms complement the graph-based execution model introduced in the previous chapter. While the graph determines the structure of computation, the state and memory determine how information evolves throughout the agent's operation.

Table 5.3 summarises the key concepts and their LangGraph implementations.

Table 5.3: Summary of concepts and their LangGraph implementations.

Concept	**Mechanism**	**Key API**
State definition	`TypedDict`	Define fields with types
Partial updates	Merge semantics	Return only changed fields
Accumulation	Reducers	`Annotated[T, reducer]`
Conversation	Messages	`add_messages`
Dynamic routing	Conditional edges	`add_conditional_edges`
Short-term memory	Checkpointing	`MemorySaver`
Long-term memory	External stores	Vector DB, key–value store

In the next chapter we will build upon these ideas to construct more advanced agent architectures that combine structured state management with language model reasoning and tool usage.

Chapter 6 Tool-Augmented Agents in LangGraph

The previous chapter showed how state, messages, and memory provide the informational foundation for LangGraph agents. With these mechanisms in place, we can now address a central question in agent design: how does a language model *act* on the world?

The answer, in most modern architectures, is **tools**. A tool is any external capability that a language model can invoke — a web search engine, a calculator, a database query, an API call, or even another model. By giving the model access to tools, we transform it from a passive text generator into an agent that can gather information, perform computations, and produce side effects.

This chapter develops the theory and practice of tool-augmented agents in LangGraph. We begin with the conceptual foundations of tool use, then show how tools are defined, bound to models, and integrated into graph-based agent architectures. We conclude with the **ReAct** pattern, which combines reasoning and acting in an iterative loop.

6.1 Why Tools Matter

Language models, despite their impressive capabilities, have well-known limitations. They cannot reliably perform arithmetic, they lack access to real-time information, and they cannot interact with external systems. Table 6.1 summarises several common failure modes and the tools that address them.

Table 6.1: Common LLM limitations and the tools that compensate for them.

Limitation	Example failure	Compensating tool
Arithmetic errors	$397 \times 28 = ?$	Calculator
Stale knowledge	"Who won the match today?"	Web search
No system access	"Send an email to Alice"	Email API
Hallucinated facts	Fabricated citations	Database lookup
No file I/O	"Summarise this PDF"	File reader

The key insight is that a language model does not need to *be* good at arithmetic or web search — it only needs to know *when* to delegate these tasks and *how* to interpret the results. This separation of concerns is the foundation of tool-augmented agent design.

Formally, if we denote the set of available tools by $\mathcal{T} = \{t_1, t_2, \ldots, t_k\}$, then at each step the model must solve a decision problem:

1. **Select**: choose a tool $t_i \in \mathcal{T}$ (or choose to respond directly).
2. **Parameterise**: construct the input arguments for the selected tool.
3. **Interpret**: incorporate the tool's output into its ongoing reasoning.

Modern language models solve this problem through a mechanism called *function calling* (or *tool calling*), in which the model produces structured output that specifies which tool to invoke and with what arguments.

6.2 Defining Tools in LangChain

LangChain provides a unified abstraction for defining tools. At its simplest, a tool is a Python function decorated with `@tool`, which automatically extracts the function's name, docstring, and type annotations to

produce a schema that the language model can reason about.

Listing 6.1 shows how to define two simple tools.

Listing 6.1: Defining tools with the `@tool` decorator

```python
from langchain_core.tools import tool


@tool
def multiply(a: float, b: float) -> float:
    """Multiply two numbers together."""
    return a * b


@tool
def web_search(query: str) -> str:
    """Search the web for current information."""
    # In production, this calls a search API.
    return f"Search results for: {query}"


# Inspect the generated schema
print(multiply.name)      # "multiply"
print(multiply.description) # "Multiply two numbers together."
print(multiply.args_schema.schema())
# {'properties': {'a': {'type': 'number'},
#                 'b': {'type': 'number'}},
#  'required': ['a', 'b'],
#  'type': 'object'}
```

The decorator transforms the function into a `StructuredTool` object that carries its own JSON schema. This schema is later passed to the language model so that it can generate correctly structured tool calls.

6.2.1 Tool Anatomy

Every LangChain tool exposes the following components, which together allow the language model to decide when and how to use it.

Table 6.2: Components of a LangChain tool.

Component	Source	Role
`name`	Function name	Identifier used by the model in tool-call requests.
`description`	Docstring	Natural-language explanation that helps the model decide when to use the tool.
`args_schema`	Type annotations	JSON schema describing the expected input arguments.
`invoke()`	Function body	Actually executes the tool and returns a result.

The quality of the `description` is particularly important: it is the primary signal the model uses to select among available tools. A vague or misleading description will cause the model to invoke the tool at inappropriate times.

6.2.2 Tools with Complex Inputs

For tools that require structured or nested inputs, Pydantic models can be used to define the argument schema explicitly.

Listing 6.2: A tool with a Pydantic input schema

```python
from pydantic import BaseModel, Field
from langchain_core.tools import tool


class FlightSearch(BaseModel):
    """Input schema for the flight search tool."""
    origin: str = Field(description="IATA code of departure airport")
    destination: str = Field(description="IATA code of arrival airport")
    date: str = Field(description="Travel date in YYYY-MM-DD format")
    max_stops: int = Field(default=1, description="Maximum layovers")


@tool(args_schema=FlightSearch)
def search_flights(
    origin: str,
    destination: str,
    date: str,
    max_stops: int = 1,
) -> str:
    """Search for available flights between two airports."""
    return (
        f"Found 3 flights from {origin} to {destination} "
        f"on {date} with at most {max_stops} stop(s)."
    )
```

The `Field` descriptions are included in the schema and help the model populate each argument correctly.

6.3 Binding Tools to Language Models

Defining tools is only the first step. To enable a language model to use them, the tools must be *bound* to the model. Binding informs the model of the available tools and their schemas, so that it can choose to invoke them during generation.

Listing 6.3: Binding tools to a chat model

```python
from langchain_openai import ChatOpenAI
from langchain_core.tools import tool


@tool
```

```python
def get_weather(city: str) -> str:
    """Get the current weather for a city."""
    return f"Sunny, 22C in {city}"


@tool
def get_population(city: str) -> int:
    """Get the population of a city."""
    return 8_799_000 # placeholder


llm = ChatOpenAI(model="gpt-4o")
llm_with_tools = llm.bind_tools([get_weather, get_population])
```

After binding, invoking `llm_with_tools` with a message may produce an `AIMessage` that contains one or more `tool_calls` instead of (or in addition to) a text response.

6.3.1 The Tool-Call Protocol

When a model decides to use a tool, it returns an `AIMessage` whose `tool_calls` field contains a list of structured requests. Each request specifies the tool name, the arguments, and a unique call identifier.

Figure 6.1 illustrates the full round-trip protocol.

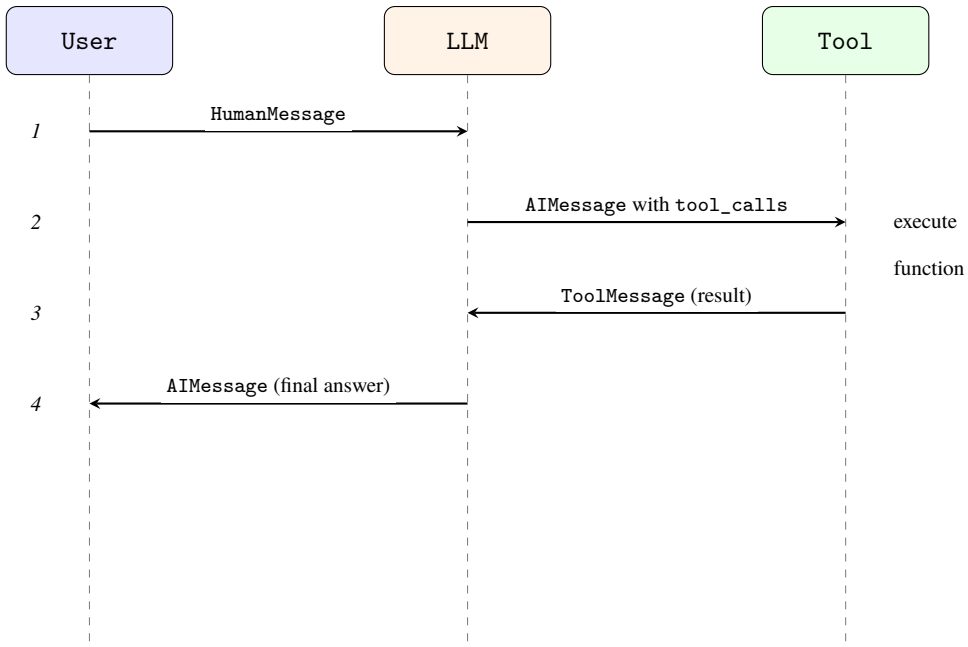


Figure 6.1: The tool-call protocol. (1) The user sends a message. (2) The model responds with a tool-call request. (3) The tool executes and returns its result. (4) The model incorporates the result and produces a final answer.

Listing 6.4 shows how to inspect the tool-call fields of an `AIMessage`.

Listing 6.4: Inspecting tool calls in an AIMessage

```python
from langchain_core.messages import HumanMessage

```

```python
response = llm_with_tools.invoke([
    HumanMessage(content="What's the weather in Paris?")
])

# The model may decide to call a tool
if response.tool_calls:
    for call in response.tool_calls:
        print(f"Tool: {call['name']}")
        print(f"Args: {call['args']}")
        print(f"ID:  {call['id']}")
# Tool: get_weather
# Args: {'city': 'Paris'}
# ID:   call_abc123
```

6.4 The ToolNode Utility

Manually dispatching tool calls — matching each call to the correct function, invoking it, and wrapping the result in a `ToolMessage` — is tedious and error-prone. LangGraph provides the `ToolNode` class to automate this process.

A `ToolNode` receives a list of tool objects, inspects the most recent `AIMessage` in the state for tool-call requests, executes the corresponding functions, and appends the results as `ToolMessage` objects.

Listing 6.5: Using `ToolNode` to automate tool dispatch

```python
from langgraph.prebuilt import ToolNode
from langchain_core.tools import tool


@tool
def calculator(expression: str) -> str:
    """Evaluate a mathematical expression."""
    return str(eval(expression)) # simplified


@tool
def lookup(term: str) -> str:
    """Look up a term in the knowledge base."""
    return f"Definition of '{term}': ..."


# ToolNode handles dispatch automatically
tool_node = ToolNode([calculator, lookup])
```

When added to a LangGraph graph, the `ToolNode` acts as a single node that can execute any of the registered tools based on the model's requests.

Figure 6.2 shows the internal logic of a `ToolNode`.

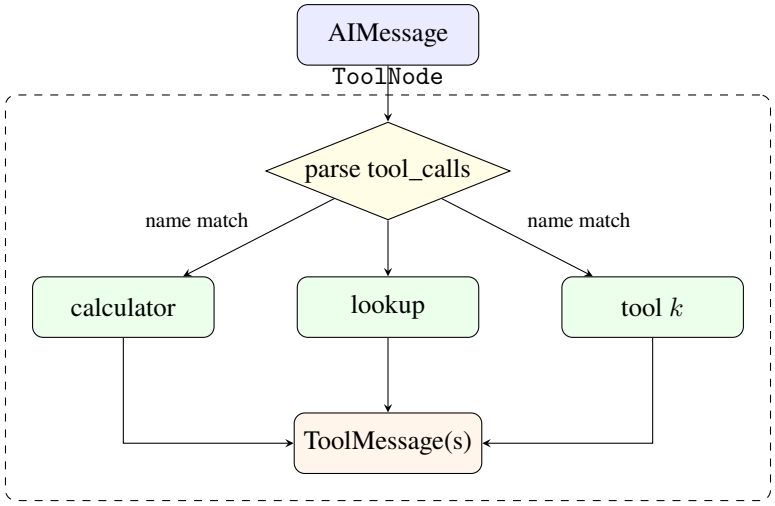


Figure 6.2: Internal dispatch logic of `ToolNode`. The node parses tool-call requests from the latest `AIMessage`, routes each call to the matching function, and collects the results as `ToolMessage` objects.

6.5 The ReAct Pattern

With the building blocks now in place — typed state, messages, tools, and conditional routing — we can implement the most widely used agent architecture: the **ReAct** (Reasoning and Acting) pattern [3].

The core idea of ReAct is simple: the agent alternates between *thinking* (reasoning about what to do next) and *acting* (invoking a tool). After each tool result, the agent reasons again, deciding whether to call another tool or produce a final response.

6.5.1 The ReAct Loop

The ReAct loop can be expressed as a cycle in a LangGraph graph with two nodes:

1. **Agent node** — calls the language model, which may produce either a text response or a tool-call request.
2. **Tool node** — executes the requested tool and returns the result.

A conditional edge after the agent node inspects the response: if it contains tool calls, execution routes to the tool node; otherwise, it routes to `END`.

Figure 6.3 shows the graph structure.

6.5.2 Full Implementation

Listing 6.6 provides a complete, runnable implementation of the ReAct pattern in LangGraph.

Listing 6.6: A complete ReAct agent in LangGraph

```
from typing import TypedDict, Annotated, Literal
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, START, END
from langgraph.prebuilt import ToolNode
from langgraph.checkpoint.memory import MemorySaver
from langchain_openai import ChatOpenAI
from langchain_core.tools import tool
from langchain_core.messages import HumanMessage

```

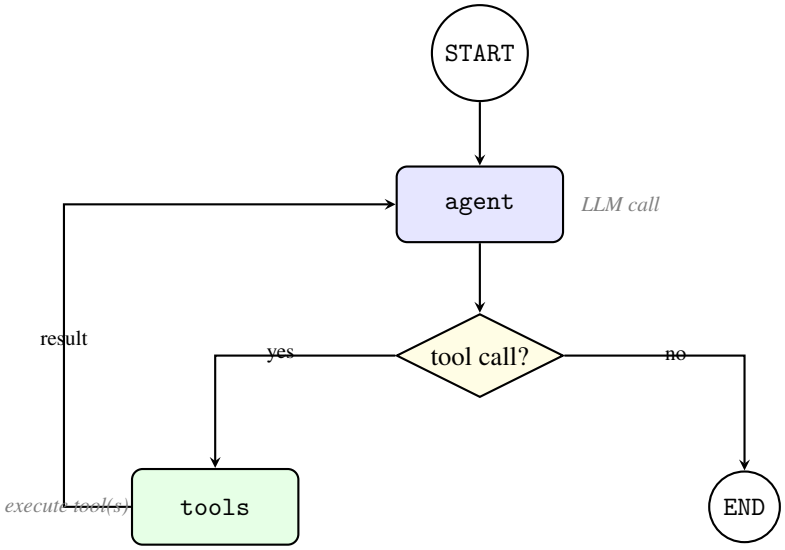


Figure 6.3: The ReAct graph. The agent node calls the LLM. A conditional edge inspects the response: tool calls route to the tool node, which loops back; otherwise execution ends.

```
10
11 # ---- State ----
12 class AgentState(TypedDict):
13     messages: Annotated[list, add_messages]
14
15
16 # ---- Tools ----
17 @tool
18 def get_weather(city: str) -> str:
19     """Get the current weather for a city."""
20     data = {"London": "Rainy, 14C", "Tokyo": "Sunny, 28C"}
21     return data.get(city, f"No data for {city}")
22
23
24 @tool
25 def calculate(expression: str) -> str:
26     """Evaluate a mathematical expression safely."""
27     allowed = set("0123456789+-*/(). ")
28     if all(c in allowed for c in expression):
29         return str(eval(expression))
30     return "Invalid expression"
31
32
33 tools = [get_weather, calculate]
34 tool_node = ToolNode(tools)
35
36
37 # ---- Model ----
38 llm = ChatOpenAI(model="gpt-4o")
39 llm_with_tools = llm.bind_tools(tools)
40
```

```python

# ---- Agent node ----
def agent(state: AgentState) -> AgentState:
    response = llm_with_tools.invoke(state["messages"])
    return {"messages": [response]}


# ---- Routing ----
def should_continue(
    state: AgentState,
) -> Literal["tools", "__end__"]:
    last = state["messages"][-1]
    if last.tool_calls:
        return "tools"
    return "__end__"


# ---- Graph assembly ----
graph = (
    StateGraph(AgentState)
    .add_node("agent", agent)
    .add_node("tools", tool_node)
    .add_edge(START, "agent")
    .add_conditional_edges("agent", should_continue)
    .add_edge("tools", "agent")
)

app = graph.compile(checkpointer=MemorySaver())


# ---- Run ----
config = {"configurable": {"thread_id": "react-demo"}}

result = app.invoke(
    {"messages": [HumanMessage(
        content="What's the weather in London? "
                "Also, what is 365 * 24?"
    )]},
    config=config,
)

# Print the conversation
for msg in result["messages"]:
    role = msg.__class__.__name__
    content = msg.content or "(tool call)"
    print(f"{role}: {content}")
```

6.5.3 Execution Trace

To build intuition for how the ReAct loop operates, let us trace the execution of the agent from Listing 6.6 on the input `"What's the weather in London? Also, what is 365 * 24?"`.

Figure 6.4 shows the sequence of states.

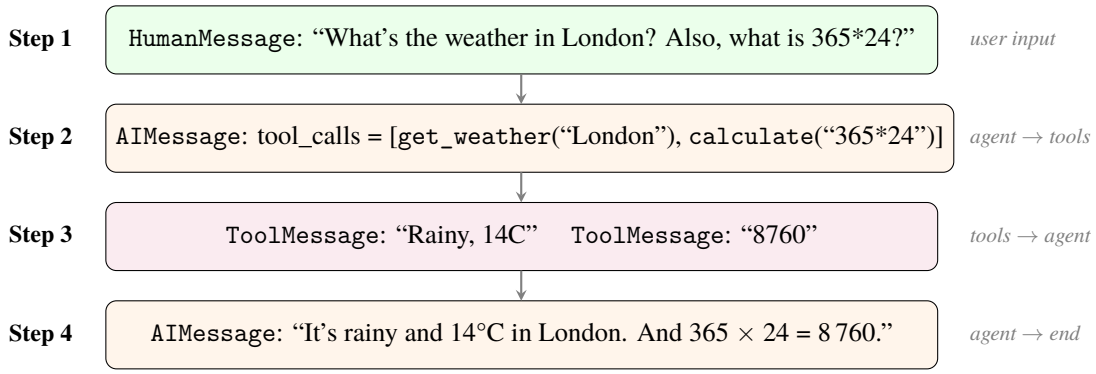


Figure 6.4: Execution trace of the ReAct agent. The model issues two parallel tool calls in step 2, receives both results in step 3, and synthesises a final answer in step 4.

Notice that the model issued *two* tool calls in a single turn. This is called **parallel tool calling** and is supported by most modern language models. The `ToolNode` executes all requested calls and returns a `ToolMessage` for each one.

6.5.4 Formal Characterisation

The ReAct loop can be characterised as a fixed-point computation over the state space $\mathcal{S}$. Let $a : \mathcal{S} \to \mathcal{S}$ denote the agent function, $t : \mathcal{S} \to \mathcal{S}$ denote the tool function, and $\phi : \mathcal{S} \to \{\texttt{continue}, \texttt{stop}\}$ denote the routing predicate. Then the ReAct execution is the sequence

$$s_0,\ s_1 = a(s_0),\ s_2 = t(s_1),\ s_3 = a(s_2),\ \ldots$$

which terminates at step n when $\phi(s_n) = \texttt{stop}$. This corresponds to the least fixed point of the operator

$$F(s) = \begin{cases} s & \text{if } \phi(a(s)) = \texttt{stop}, \\ F\big(t(a(s))\big) & \text{otherwise.} \end{cases}$$

In practice, a maximum iteration limit is imposed to guarantee termination even when the model enters an infinite loop of tool calls.

6.6 Parallel and Sequential Tool Calls

Modern language models can issue multiple tool calls in a single response. This raises an important design question: should the calls be executed *in parallel* or *sequentially*?

The default behaviour of `ToolNode` is parallel execution. This is generally desirable when the tools are independent — for example, fetching weather for two different cities. However, when one tool call depends on the result of another, sequential execution is necessary.

Figure 6.5 contrasts the two patterns.

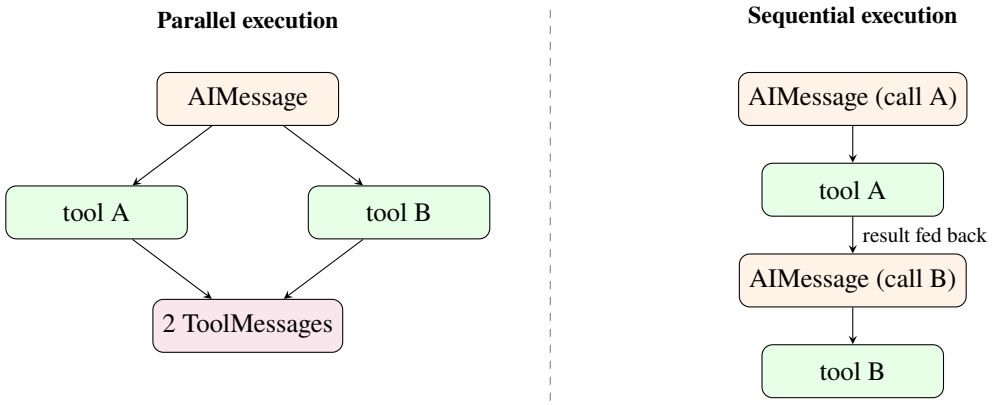


Figure 6.5: Parallel tool execution (left) handles independent calls in a single step. Sequential execution (right) requires multiple agent–tool cycles, allowing later calls to depend on earlier results.

The language model itself typically decides whether to issue parallel or sequential calls. If the model determines that the calls are independent, it emits them simultaneously; if one depends on the other, it issues them in successive turns.

6.7 Error Handling in Tool Execution

Tools interact with the external world, and external calls can fail. Robust agents must handle these failures gracefully. There are two complementary strategies.

6.7.1 Letting the Model Recover

The simplest strategy is to return the error message as a `ToolMessage` and let the model decide how to proceed. Many models can diagnose the issue (for example, a malformed argument) and retry with corrected input.

Listing 6.7: Returning errors to the model for self-correction

```
from langchain_core.tools import tool
from langchain_core.messages import ToolMessage


@tool
def divide(a: float, b: float) -> float:
    """Divide a by b."""
    if b == 0:
        raise ValueError("Division by zero")
    return a / b


def safe_tool_node(state):
    """Execute tools, catching and reporting errors."""
    last = state["messages"][-1]
    results = []
    for call in last.tool_calls:
```

```
        try:
            # Find and invoke the matching tool
            result = tool_map[call["name"]].invoke(call["args"])
            results.append(ToolMessage(
                content=str(result),
                tool_call_id=call["id"],
            ))
        except Exception as e:
            results.append(ToolMessage(
                content=f"Error: {e}",
                tool_call_id=call["id"],
            ))
    return {"messages": results}
```

6.7.2 Retry with Backoff

For transient failures (network timeouts, rate limits), a retry mechanism is more appropriate. Listing 6.8 shows a simple retry wrapper.

Listing 6.8: Retrying tool calls with exponential backoff

```
import time
from functools import wraps


def with_retry(max_attempts=3, base_delay=1.0):
    """Decorator that retries a tool on transient errors."""
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(max_attempts):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == max_attempts - 1:
                        raise
                    delay = base_delay * (2 ** attempt)
                    time.sleep(delay)
        return wrapper
    return decorator


@tool
@with_retry(max_attempts=3)
def fetch_price(ticker: str) -> str:
    """Fetch the current stock price for a ticker symbol."""
    # May raise on network errors
    ...
```

6.8 The Prebuilt ReAct Agent

Because the ReAct pattern is so common, LangGraph provides a convenience function that constructs the entire graph in a single call.

Listing 6.9: Using the prebuilt `create_react_agent` helper

```python
from langgraph.prebuilt import create_react_agent
from langgraph.checkpoint.memory import MemorySaver
from langchain_openai import ChatOpenAI
from langchain_core.tools import tool
from langchain_core.messages import HumanMessage


@tool
def get_weather(city: str) -> str:
    """Get the current weather for a city."""
    return "Sunny, 22C"


llm = ChatOpenAI(model="gpt-4o")
agent = create_react_agent(
    llm,
    tools=[get_weather],
    checkpointer=MemorySaver(),
)

config = {"configurable": {"thread_id": "quick-demo"}}

result = agent.invoke(
    {"messages": [HumanMessage(content="Weather in Rome?")]},
    config=config,
)
print(result["messages"][-1].content)
```

Under the hood, `create_react_agent` builds exactly the same graph structure as Listing 6.6: an agent node, a tool node, and a conditional edge that loops until no more tool calls are issued.

6.9 Design Considerations

Building effective tool-augmented agents involves several design decisions beyond the basic graph wiring. This section highlights the most important considerations.

6.9.1 Tool Granularity

Tools can range from coarse-grained (“search the internet”) to fine-grained (“fetch the temperature at these coordinates”). Coarse tools are simpler to define but give the model less control; fine tools offer precision but increase the cognitive load on the model.

As a rule of thumb, each tool should correspond to a single, well-defined capability. If a tool requires the model to make complex choices among internal options, consider splitting it into multiple tools.

6.9.2 Limiting Tool Access

Not every tool should be available at every step. In some architectures, the set of tools is dynamically adjusted based on the current state. For example, a "confirm purchase" tool might only be available after a "search products" tool has been called.

This can be implemented by binding different tool sets to different nodes in the graph, or by using a routing function that filters the available tools before each LLM call.

6.9.3 Recursion Limits

Without safeguards, a ReAct agent can loop indefinitely if the model repeatedly issues tool calls that never converge on a final answer. LangGraph provides a `recursion_limit` parameter (set in the `config` dictionary) that caps the total number of node executions per invocation.

Listing 6.10: Setting a recursion limit

```
config = {
    "configurable": {"thread_id": "safe-demo"},
    "recursion_limit": 25, # max 25 node steps
}

result = app.invoke(
    {"messages": [HumanMessage(content="...")]},
    config=config,
)
```

6.10 Summary

This chapter introduced the mechanisms by which LangGraph agents interact with the external world through tools.

- **Tools** are defined as decorated Python functions with type annotations and docstrings. LangChain automatically generates the JSON schema that the language model needs to invoke them.
- Tools are **bound** to a language model via `bind_tools`, enabling the model to produce structured `tool_calls` in its responses.
- The **ToolNode** utility automates the dispatch of tool calls, matching each request to the correct function and wrapping results in `ToolMessage` objects.
- The **ReAct pattern** combines reasoning and acting in an iterative loop: the agent calls the model, optionally executes tools, and repeats until a final answer is produced.
- Practical agents must handle **parallel tool calls**, implement **error recovery**, and enforce **recursion limits** to ensure robustness.

Table 6.3 summarises the key components.

In the next chapter we will extend these ideas to *multi-agent* systems, where multiple specialised agents collaborate to solve complex tasks that exceed the capability of any single agent.

Table 6.3: Key components of tool-augmented agents in LangGraph.

Component	API	Role
Tool definition	`@tool`	Wrap a function as a tool with auto-generated schema
Tool binding	`llm.bind_tools()`	Inform the model of available tools
Tool dispatch	`ToolNode`	Execute tool calls and return results
Routing	`add_conditional_edges`	Direct flow based on presence of tool calls
ReAct agent	`create_react_agent`	One-line construction of the full ReAct graph
Recursion limit	`recursion_limit`	Cap the number of loop iterations

Part III

Planning and Reasoning Patterns

Chapter 7 Planning and Reasoning Strategies for Agents

In Part II we established the core patterns for building single agents: graph-based execution, typed state with message passing, and tool-augmented action. The ReAct pattern, in particular, showed how an agent can interleave reasoning with action in an iterative loop.

However, the ReAct loop reasons only one step at a time. At each iteration the model decides what to do *next*, without forming a broader plan for the entire task. This myopic strategy works well for simple queries but struggles with complex, multi-step problems that require foresight, decomposition, or backtracking.

This chapter introduces **planning and reasoning strategies** that give agents the ability to think ahead. We progress from lightweight prompting techniques to full architectural patterns:

1. **Chain-of-thought reasoning**, which encourages the model to externalise its intermediate steps.
2. **Plan-and-execute architectures**, which separate high-level planning from step-by-step execution.
3. **Reflection and self-critique**, which allow agents to evaluate and revise their own outputs.
4. **Tree-of-thought search**, which explores multiple reasoning paths in parallel.

Together, these strategies form a spectrum of increasing sophistication, each trading additional computation for improved reliability on harder tasks.

7.1 The Limits of Reactive Agents

Before introducing planning mechanisms, it is worth understanding precisely *why* reactive agents struggle with complex tasks. Consider an agent asked to "plan a three-day trip to Tokyo covering temples, food, and nightlife." A ReAct agent would proceed as follows:

1. Call a search tool for "temples in Tokyo."
2. Receive results, then call another search for "restaurants in Tokyo."
3. Receive results, then call another search for "nightlife in Tokyo."
4. Attempt to synthesise everything into a coherent itinerary.

The problem is that steps 1–3 are executed without any awareness of the overall structure. The agent does not allocate time across days, does not ensure geographic coherence between activities, and cannot revisit earlier decisions if later constraints conflict.

More formally, reactive agents operate as a *greedy* strategy over the state space: at each step they choose the locally best action without considering future consequences. Planning mechanisms address this by introducing a form of *lookahead*.

Figure 7.1 contrasts the two approaches.

7.2 Chain-of-Thought Reasoning

The simplest planning mechanism is not an architectural change at all, but a *prompting strategy*. Chain-of-thought (CoT) reasoning [2] encourages the language model to produce intermediate reasoning steps before arriving at a final answer.

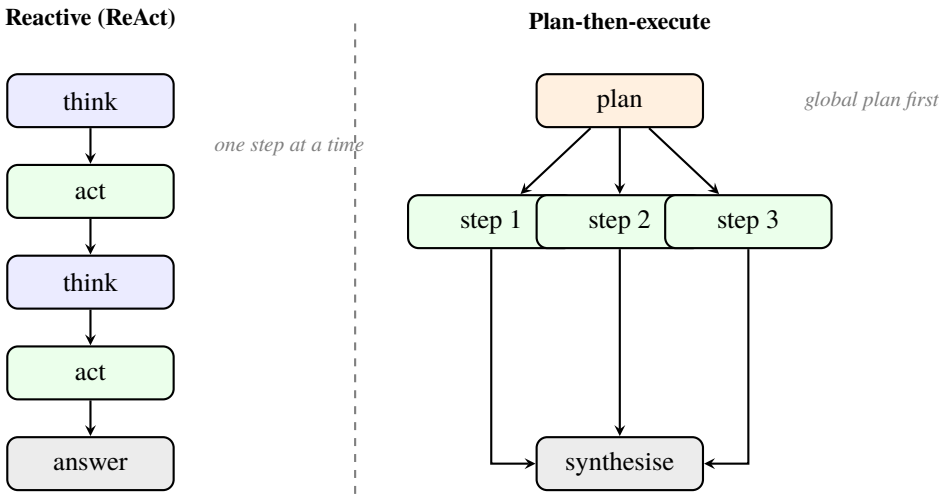


Figure 7.1: Reactive agents (left) interleave single-step reasoning with action. Planning agents (right) form a global plan before executing individual steps, enabling better coordination and resource allocation.

7.2.1 The Core Idea

When presented with a complex question, a standard prompt might produce a direct (and often incorrect) answer. A chain-of-thought prompt instead instructs the model to "think step by step," which causes it to externalise its reasoning as a sequence of intermediate conclusions.

Formally, let q denote the input question and a the desired answer. Standard prompting models the conditional distribution $P(a \mid q)$ directly. Chain-of-thought prompting instead models

$$P(a \mid q) = \sum_{r} P(a \mid r, q)\, P(r \mid q),$$

where r denotes a chain of reasoning steps. In practice the sum is replaced by a single sample: the model generates r autoregressively, then conditions on r to produce a.

7.2.2 Implementation in LangGraph

In a LangGraph agent, chain-of-thought reasoning is typically induced through the system message. The agent node wraps the model call with a system prompt that requests explicit reasoning.

Listing 7.1: Inducing chain-of-thought reasoning via the system message

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, START, END
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage


class AgentState(TypedDict):
    messages: Annotated[list, add_messages]


COT_SYSTEM = SystemMessage(content=(
    "You are an analytical assistant. Before answering any "
    "question, think through the problem step by step. "
```

```
    "Write out your reasoning explicitly, then provide "
    "your final answer on a new line starting with "
    "'ANSWER:'."
))

llm = ChatOpenAI(model="gpt-4o")


def reason(state: AgentState) -> AgentState:
    messages = [COT_SYSTEM] + state["messages"]
    response = llm.invoke(messages)
    return {"messages": [response]}


graph = (
    StateGraph(AgentState)
    .add_node("reason", reason)
    .add_edge(START, "reason")
    .add_edge("reason", END)
)

app = graph.compile()

result = app.invoke({"messages": [
    HumanMessage(content=(
        "A store sells apples in bags of 6. I need at least "
        "50 apples. What is the minimum number of bags?"
    ))
]})
print(result["messages"][-1].content)
# Step 1: I need at least 50 apples.
# Step 2: Each bag has 6 apples.
# Step 3: 50 / 6 = 8.33...
# Step 4: Since I can only buy whole bags, I round up to 9.
# Step 5: 9 * 6 = 54, which is >= 50.
# ANSWER: 9 bags
```

7.2.3 Structured Chain-of-Thought

A more controlled variant separates the reasoning phase from the answer phase using two distinct model calls. The first call produces a *scratchpad* of reasoning; the second reads the scratchpad and produces a concise final answer.

Listing 7.2: Structured chain-of-thought with separate reasoning and answering

```
class CotState(TypedDict):
    messages: Annotated[list, add_messages]
    scratchpad: str
    final_answer: str
```

```python


def think(state: CotState) -> CotState:
    """Generate reasoning steps."""
    messages = [
        SystemMessage(content="Think step by step. Output "
                      "ONLY your reasoning, not a final answer."),
        *state["messages"],
    ]
    response = llm.invoke(messages)
    return {"scratchpad": response.content}


def answer(state: CotState) -> CotState:
    """Produce a concise answer from the scratchpad."""
    messages = [
        SystemMessage(content="Given the reasoning below, "
                      "provide a concise final answer."),
        HumanMessage(content=state["scratchpad"]),
    ]
    response = llm.invoke(messages)
    return {"final_answer": response.content}


graph = (
    StateGraph(CotState)
    .add_node("think", think)
    .add_node("answer", answer)
    .add_edge(START, "think")
    .add_edge("think", "answer")
    .add_edge("answer", END)
)
```

This two-phase design is particularly useful when the final answer must conform to a strict format (JSON, a single number, a boolean) that would be difficult to extract reliably from a free-form chain-of-thought.

7.3 Plan-and-Execute Architecture

Chain-of-thought reasoning improves the quality of individual steps, but it does not provide an explicit *plan* that can be inspected, modified, or re-ordered. The **plan-and-execute** architecture [1] addresses this by introducing two distinct phases:

1. A **planner** decomposes the original task into an ordered list of sub-tasks.
2. An **executor** carries out each sub-task one at a time, feeding results back into the state.

After all sub-tasks have been executed, a final **synthesiser** combines the results into a coherent response.

7.3.1 Architecture Overview

Figure 7.2 shows the graph structure of a plan-and-execute agent.

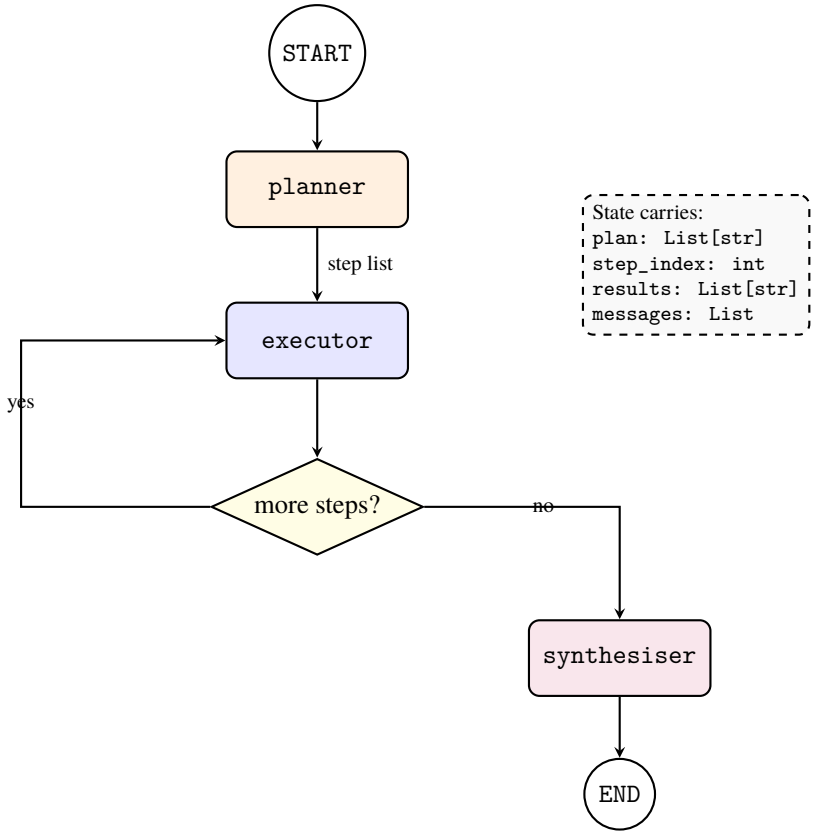


Figure 7.2: The plan-and-execute architecture. The planner produces an ordered list of sub-tasks. The executor processes one sub-task per iteration, looping until all steps are complete. The synthesiser merges all results into a final response.

7.3.2 State Design

The plan-and-execute pattern requires a state that tracks both the plan and the execution progress.

Listing 7.3: State definition for the plan-and-execute agent

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages


class PlanExecuteState(TypedDict):
    messages: Annotated[list, add_messages]
    plan: list[str]       # ordered list of sub-tasks
    step_index: int       # which step to execute next
    step_results: list[str] # result of each completed step
    final_response: str   # synthesised output
```

7.3.3 Implementation

Listing 7.4 provides a complete implementation.

Listing 7.4: A plan-and-execute agent in LangGraph

```python
from typing import Literal
from langgraph.graph import StateGraph, START, END
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

llm = ChatOpenAI(model="gpt-4o")


def planner(state: PlanExecuteState) -> PlanExecuteState:
    """Decompose the task into sub-steps."""
    messages = [
        SystemMessage(content=(
            "You are a planning agent. Given the user's request, "
            "break it down into 3-5 concrete sub-tasks. "
            "Return ONLY a JSON list of strings, e.g. "
            '["step 1", "step 2", "step 3"].'
        )),
        *state["messages"],
    ]
    response = llm.invoke(messages)
    import json
    plan = json.loads(response.content)
    return {"plan": plan, "step_index": 0, "step_results": []}


def executor(state: PlanExecuteState) -> PlanExecuteState:
    """Execute the current sub-task."""
    idx = state["step_index"]
    current_step = state["plan"][idx]

    messages = [
        SystemMessage(content=(
            "You are an execution agent. Carry out the "
            "following task and return the result.\n\n"
            f"Task: {current_step}\n\n"
            "Previous results:\n"
            + "\n".join(
                f"- Step {i+1}: {r}"
                for i, r in enumerate(state["step_results"])
            )
        )),
    ]
    response = llm.invoke(messages)
    new_results = state["step_results"] + [response.content]
    return {
        "step_results": new_results,
        "step_index": idx + 1,
```

```
    }


def synthesiser(state: PlanExecuteState) -> PlanExecuteState:
    """Combine all step results into a final response."""
    results_text = "\n".join(
        f"Step {i+1} ({state['plan'][i]}): {r}"
        for i, r in enumerate(state["step_results"])
    )
    messages = [
        SystemMessage(content=(
            "Synthesise the following results into a coherent, "
            "well-structured final response for the user."
        )),
        HumanMessage(content=results_text),
    ]
    response = llm.invoke(messages)
    return {"final_response": response.content}


def should_continue(
    state: PlanExecuteState,
) -> Literal["executor", "synthesiser"]:
    if state["step_index"] < len(state["plan"]):
        return "executor"
    return "synthesiser"


graph = (
    StateGraph(PlanExecuteState)
    .add_node("planner", planner)
    .add_node("executor", executor)
    .add_node("synthesiser", synthesiser)
    .add_edge(START, "planner")
    .add_edge("planner", "executor")
    .add_conditional_edges("executor", should_continue)
    .add_edge("synthesiser", END)
)

app = graph.compile()
```

7.3.4 Advantages and Limitations

The plan-and-execute architecture has several advantages over reactive agents. Table 7.1 summarises the key trade-offs.

The rigidity of the plan is the primary limitation. If the result of step 2 reveals that step 3 is unnecessary or that a new step is required, the basic plan-and-execute agent has no mechanism to adapt. The next sections

Table 7.1: Trade-offs of the plan-and-execute architecture.

Advantages	Limitations
Explicit, inspectable plan	Upfront planning may be inaccurate
Better resource allocation across steps	Rigid plan cannot adapt to surprises
Easier to debug (each step is isolated)	Additional LLM calls increase latency
Can be parallelised when steps are independent	Planner must anticipate all contingencies

introduce strategies that address this shortcoming.

7.4 Adaptive Replanning

A natural extension of plan-and-execute is to allow the agent to *revise its plan* after each step. After every execution, a **replanning node** inspects the accumulated results and the remaining plan, then decides whether to continue as planned, modify the remaining steps, or add entirely new ones.

Figure 7.3 shows the modified graph.

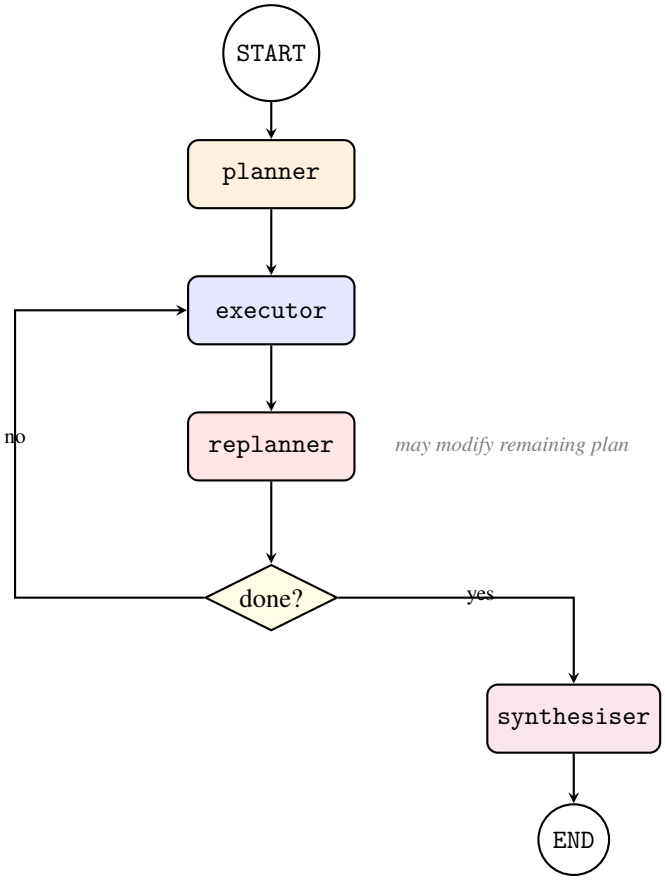


Figure 7.3: Adaptive replanning. After each execution step, the replanner inspects the results and may modify, reorder, or extend the remaining plan before the next iteration.

Listing 7.5 shows the replanner node.

Listing 7.5: A replanner node that adapts the plan after each step

```
def replanner(state: PlanExecuteState) -> PlanExecuteState:
    """Review progress and optionally revise the plan."""
    completed = [
        f"Step {i+1}: {state['plan'][i]} -> {r}"
        for i, r in enumerate(state["step_results"])
    ]
    remaining = state["plan"][state["step_index"]:]

    messages = [
        SystemMessage(content=(
            "You are a replanning agent. Given what has been "
            "completed and what remains, decide whether the "
            "remaining plan is still appropriate.\n\n"
            "Completed:\n" + "\n".join(completed) + "\n\n"
            "Remaining plan:\n"
            + "\n".join(f"- {s}" for s in remaining) + "\n\n"
            "Return a JSON object with two keys:\n"
            '  "revised_plan": [...], // new remaining steps\n'
            '  "done": true/false   // true if no more steps needed'
        )),
    ]
    response = llm.invoke(messages)
    import json
    result = json.loads(response.content)

    if result["done"]:
        return {"plan": state["plan"][:state["step_index"]]}

    full_plan = (
        state["plan"][:state["step_index"]]
        + result["revised_plan"]
    )
    return {"plan": full_plan}
```

The replanning mechanism transforms the agent from a rigid pipeline into an adaptive system that can respond to unexpected results, handle errors, and take advantage of information that was unavailable at planning time.

7.5 Reflection and Self-Critique

Planning helps agents decide *what* to do; **reflection** helps them evaluate *how well* they did it. A reflective agent examines its own output and uses self-critique to identify errors, omissions, or areas for improvement.

7.5.1 The Reflection Loop

The simplest reflection architecture consists of two nodes arranged in a loop:

1. **Generator**: produces a draft output.
2. **Critic**: evaluates the draft and either approves it or provides specific feedback.

If the critic provides feedback, the generator revises its output in the next iteration. The loop continues until the critic approves or a maximum number of iterations is reached.

Figure 7.4 shows the graph structure.

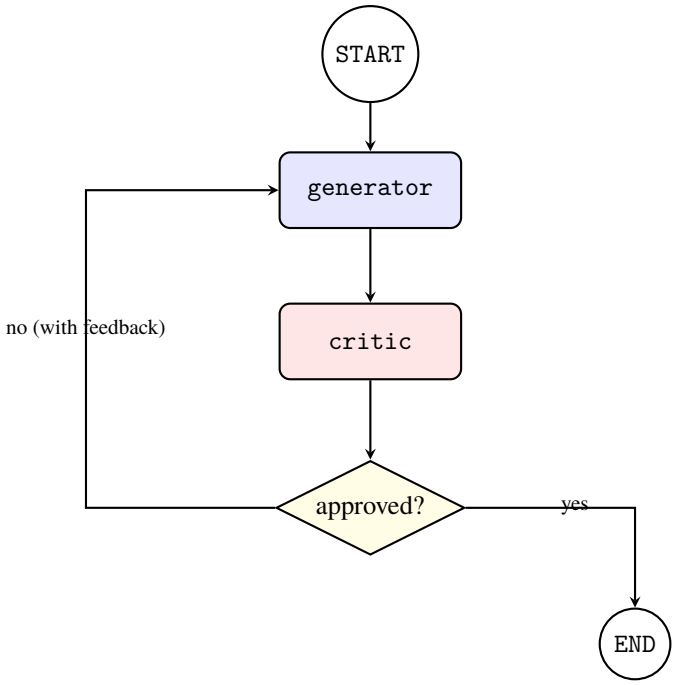


Figure 7.4: The reflection loop. The generator produces output, the critic evaluates it, and if not approved, feedback is fed back to the generator for revision.

7.5.2 Implementation

Listing 7.6 shows a complete reflection agent for writing tasks.

Listing 7.6: A reflection agent with generator and critic

```
from typing import TypedDict, Annotated, Literal
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, START, END
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

llm = ChatOpenAI(model="gpt-4o")


class ReflectionState(TypedDict):
    messages: Annotated[list, add_messages]
    draft: str
    critique: str
    iteration: int
    approved: bool

```

```python

MAX_ITERATIONS = 3


def generator(state: ReflectionState) -> ReflectionState:
    """Generate or revise a draft."""
    if state["iteration"] == 0:
        prompt = (
            "Write a concise response to the user's request."
        )
    else:
        prompt = (
            f"Revise your previous draft based on this "
            f"feedback:\n\n{state['critique']}\n\n"
            f"Previous draft:\n{state['draft']}"
        )

    messages = [
        SystemMessage(content=prompt),
        *state["messages"],
    ]
    response = llm.invoke(messages)
    return {
        "draft": response.content,
        "iteration": state["iteration"] + 1,
    }


def critic(state: ReflectionState) -> ReflectionState:
    """Evaluate the draft and provide feedback."""
    messages = [
        SystemMessage(content=(
            "You are a critical reviewer. Evaluate the "
            "following draft for accuracy, completeness, "
            "and clarity. If it is satisfactory, respond "
            "with exactly 'APPROVED'. Otherwise, provide "
            "specific, actionable feedback."
        )),
        HumanMessage(content=state["draft"]),
    ]
    response = llm.invoke(messages)
    is_approved = "APPROVED" in response.content.upper()
    return {
        "critique": response.content,
        "approved": is_approved,
    }


```

```python
def should_revise(
    state: ReflectionState,
) -> Literal["generator", "__end__"]:
    if state["approved"]:
        return "__end__"
    if state["iteration"] >= MAX_ITERATIONS:
        return "__end__"
    return "generator"


graph = (
    StateGraph(ReflectionState)
    .add_node("generator", generator)
    .add_node("critic", critic)
    .add_edge(START, "generator")
    .add_edge("generator", "critic")
    .add_conditional_edges("critic", should_revise)
)

app = graph.compile()

result = app.invoke({
    "messages": [HumanMessage(
        content="Explain the CAP theorem in two paragraphs."
    )],
    "draft": "",
    "critique": "",
    "iteration": 0,
    "approved": False,
})
print(f"Iterations: {result['iteration']}")
print(f"Approved: {result['approved']}")
print(result["draft"])
```

7.5.3 Reflection with External Validation

Self-critique is limited by the model's own blind spots: the same model that produced an error may fail to detect it during reflection. A stronger approach uses *external validators* — tools or deterministic checks — to provide objective feedback.

Listing 7.7: Reflection with an external code validator

```python
import subprocess


def code_critic(state: ReflectionState) -> ReflectionState:
    """Validate generated code by actually running it."""
    code = state["draft"]

```

```
    # Write code to a temp file and run it
    with open("/tmp/test_code.py", "w") as f:
        f.write(code)

    result = subprocess.run(
        ["python", "/tmp/test_code.py"],
        capture_output=True, text=True, timeout=10,
    )

    if result.returncode == 0:
        return {
            "critique": "APPROVED: code runs successfully.",
            "approved": True,
        }
    return {
        "critique": (
            f"Code failed with error:\n{result.stderr}\n"
            "Please fix the issue."
        ),
        "approved": False,
    }
```

This pattern is especially effective for code generation, mathematical proofs, and any domain where correctness can be verified programmatically.

7.6 Tree of Thoughts

Chain-of-thought reasoning follows a single linear path. If the model makes an error at step 3, all subsequent steps are contaminated. **Tree of Thoughts** (ToT) [4] addresses this by exploring *multiple* reasoning paths simultaneously and selecting the most promising one.

7.6.1 Conceptual Framework

The key idea is to treat reasoning as a search problem over a tree, where each node represents a partial solution and each edge represents a reasoning step. At each level of the tree, the model generates several candidate continuations, evaluates them, and expands only the most promising branches.

Figure 7.5 illustrates the structure.

7.6.2 Search Strategies

Two search strategies are commonly used with Tree of Thoughts:

Breadth-first search (BFS) expands all nodes at depth d before proceeding to depth $d + 1$. At each level, a scoring function evaluates all candidates and retains only the top k (the *beam width*). This is equivalent to *beam search* over reasoning paths.

Depth-first search (DFS) explores one branch fully before backtracking. This uses less memory but may spend time on unproductive branches.

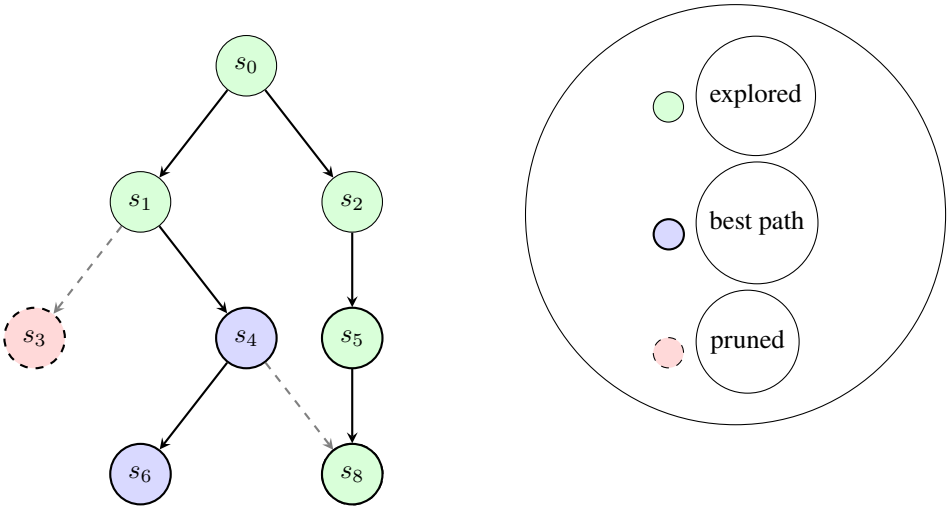


Figure 7.5: Tree of Thoughts. Each node is a partial solution. The model generates multiple candidates at each level, evaluates them, and prunes unpromising branches. The best path (shaded) leads to the final answer.

7.6.3 Implementation Sketch

Listing 7.8 outlines a breadth-first Tree of Thoughts implementation in LangGraph.

Listing 7.8: Breadth-first Tree of Thoughts

```
from typing import TypedDict
from langgraph.graph import StateGraph, START, END
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

llm = ChatOpenAI(model="gpt-4o")

BEAM_WIDTH = 3
MAX_DEPTH = 3


class ToTState(TypedDict):
    question: str
    candidates: list[str] # current beam of partial solutions
    depth: int
    best_answer: str


def expand(state: ToTState) -> ToTState:
    """Generate continuations for each candidate."""
    new_candidates = []
    for candidate in state["candidates"]:
        messages = [
            SystemMessage(content=(
                "Continue the following reasoning with one "
                "additional step. Provide 2 different "
                "continuations, each on a separate line "
```

```
            "prefixed with 'OPTION:'."
        )),
        HumanMessage(content=(
            f"Question: {state['question']}\n"
            f"Reasoning so far: {candidate}"
        )),
    ]
    response = llm.invoke(messages)
    options = [
        line.replace("OPTION:", "").strip()
        for line in response.content.split("\n")
        if line.strip().startswith("OPTION:")
    ]
    for opt in options:
        new_candidates.append(f"{candidate} -> {opt}")
return {"candidates": new_candidates}


def evaluate_and_prune(state: ToTState) -> ToTState:
    """Score candidates and keep the top-k."""
    scored = []
    for cand in state["candidates"]:
        messages = [
            SystemMessage(content=(
                "Rate the following reasoning on a scale of "
                "1-10 for correctness and progress toward "
                "answering the question. Reply with ONLY "
                "the numeric score."
            )),
            HumanMessage(content=(
                f"Question: {state['question']}\n"
                f"Reasoning: {cand}"
            )),
        ]
        response = llm.invoke(messages)
        try:
            score = float(response.content.strip())
        except ValueError:
            score = 0.0
        scored.append((score, cand))

    scored.sort(key=lambda x: x[0], reverse=True)
    top_k = [c for _, c in scored[:BEAM_WIDTH]]
    return {
        "candidates": top_k,
        "depth": state["depth"] + 1,
    }

```

```python

def finalise(state: ToTState) -> ToTState:
    """Select the best candidate as the final answer."""
    return {"best_answer": state["candidates"][0]}


def should_continue(state: ToTState):
    if state["depth"] >= MAX_DEPTH:
        return "finalise"
    return "expand"


graph = (
    StateGraph(ToTState)
    .add_node("expand", expand)
    .add_node("evaluate", evaluate_and_prune)
    .add_node("finalise", finalise)
    .add_edge(START, "expand")
    .add_edge("expand", "evaluate")
    .add_conditional_edges("evaluate", should_continue)
    .add_edge("finalise", END)
)

app = graph.compile()
```

7.7 Comparing Reasoning Strategies

The strategies introduced in this chapter occupy different points in the trade-off space between computational cost and reasoning quality. Figure 7.6 and Table 7.2 summarise the landscape.

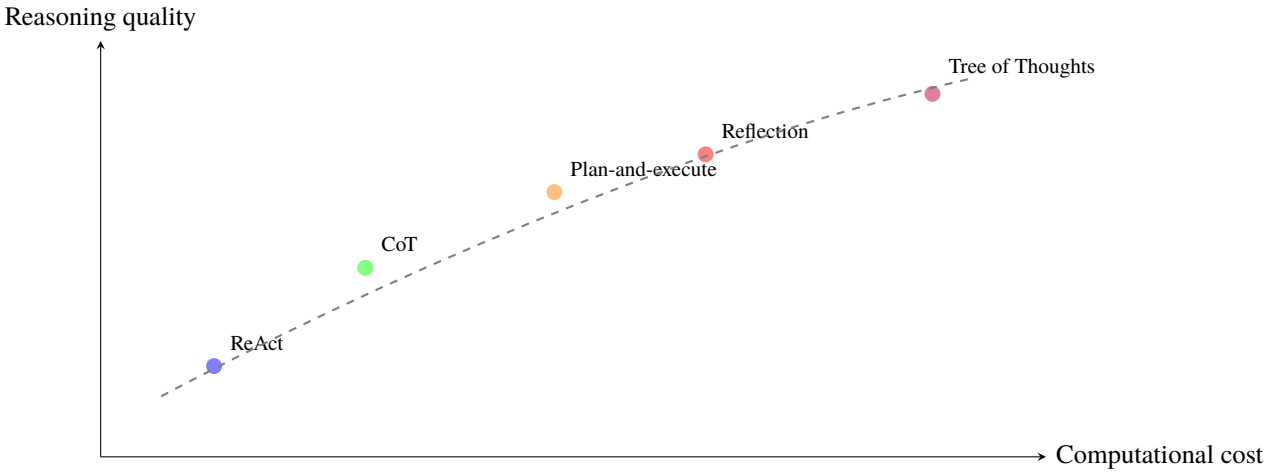


Figure 7.6: The reasoning strategy spectrum. More sophisticated strategies improve reasoning quality but require additional LLM calls.

The choice of strategy depends on the task requirements:

Simple factual queries need only ReAct or basic chain-of-thought.

Table 7.2: Comparison of reasoning strategies.

Strategy	LLM calls	Lookahead	Self-correction	Exploration
ReAct	n (steps)	None	No	Single path
Chain-of-thought	1–2	Implicit	No	Single path
Plan-and-execute	$k + 2$	Full plan	Via replan	Single plan
Reflection	$2n$ (gen+crit)	None	Yes	Single draft
Tree of Thoughts	$O(b^d)$	d levels	Via pruning	b branches

Multi-step research tasks benefit from plan-and-execute with adaptive replanning.
Creative or writing tasks improve significantly with reflection loops.
Complex reasoning problems (mathematics, logic puzzles) may justify the cost of Tree of Thoughts.
In practice, these strategies are often *combined*. For example, a plan-and-execute agent might use chain-of-thought reasoning within each execution step and a reflection loop to validate the final synthesis.

7.8 Summary

This chapter introduced planning and reasoning strategies that extend the reactive agents of Part II with the ability to think ahead, decompose problems, and improve their own outputs.

- **Chain-of-thought reasoning** encourages the model to externalise intermediate steps, improving accuracy on complex questions with minimal architectural change.
- **Plan-and-execute architectures** separate planning from execution, enabling better coordination across sub-tasks. **Adaptive replanning** allows the plan to evolve as new information becomes available.
- **Reflection and self-critique** add an evaluation loop that enables agents to iteratively refine their outputs. External validators provide objective feedback that complements self-critique.
- **Tree of Thoughts** explores multiple reasoning paths in parallel, using evaluation and pruning to converge on the best solution.

These strategies form a toolkit that can be mixed and matched depending on task complexity. The key insight is that *more computation at inference time* — in the form of planning, reflection, or search — can systematically improve the quality of agent outputs.

In Part IV we will shift from single-agent reasoning to **multi-agent systems**, where multiple specialised agents collaborate to tackle tasks that exceed the capability of any individual agent.

Chapter 8 Verification and Reliable Reasoning for Agents

Chapter 7 introduced planning-oriented reasoning patterns such as chain-of-thought, plan-and-execute, reflection, and Tree of Thoughts. These methods improve the *generation* of reasoning paths by encouraging the agent to think ahead, decompose difficult tasks, and search across alternatives.

However, better reasoning generation does not automatically imply reliable reasoning outcomes. A language model may produce a detailed plan that is internally coherent yet factually wrong. It may confidently execute a flawed sub-task, or follow a reasoning path that appears plausible but contains a subtle mistake. This problem becomes especially important in agentic systems, where generated outputs may trigger tool use, external actions, or downstream automation.

For this reason, advanced agents require not only reasoning strategies but also **verification strategies**. Verification introduces a second dimension of reliability: instead of asking only *"What should the agent think?"*, we also ask *"How can the agent know whether its reasoning is good enough to trust?"*

This chapter examines several verification-oriented patterns for LLM agents. We begin with self-consistency, where multiple reasoning traces are sampled and aggregated. We then introduce verifier-guided reasoning, in which a separate evaluation component scores intermediate or final outputs. Next, we study tool-based verification using deterministic programs and external validators. Finally, we discuss confidence-aware routing, abstention, and human escalation. Together, these patterns form the basis of *reliable reasoning* in agent systems.

8.1 Why Reasoning Needs Verification

Planning improves coordination, but it does not eliminate uncertainty. Even when a model is prompted to reason step by step, the following failure modes remain common:

- **Arithmetic drift:** the model follows the correct procedure but makes a numerical error.
- **Search path dependence:** one early mistake corrupts all later reasoning steps.
- **Fluent hallucination:** the model produces a convincing but unsupported explanation.
- **Overconfidence:** the model returns a strong answer even when the available evidence is incomplete.
- **Format correctness without semantic correctness:** the output is syntactically valid JSON, code, or prose, but the content is wrong.

This means that agent design must distinguish between at least three stages:

1. **Generation**: produce a candidate solution or reasoning path.
2. **Verification**: evaluate that candidate for correctness, adequacy, or consistency.
3. **Control**: decide whether to accept, revise, retry, or escalate.

Figure 8.1 shows this broader perspective.

The remainder of this chapter explores different ways to implement this pattern.

8.2 Self-Consistency

One of the simplest and most effective verification patterns is **self-consistency** [1]. Instead of relying on a single reasoning trace, the agent samples multiple reasoning paths and selects the answer that appears most frequently.

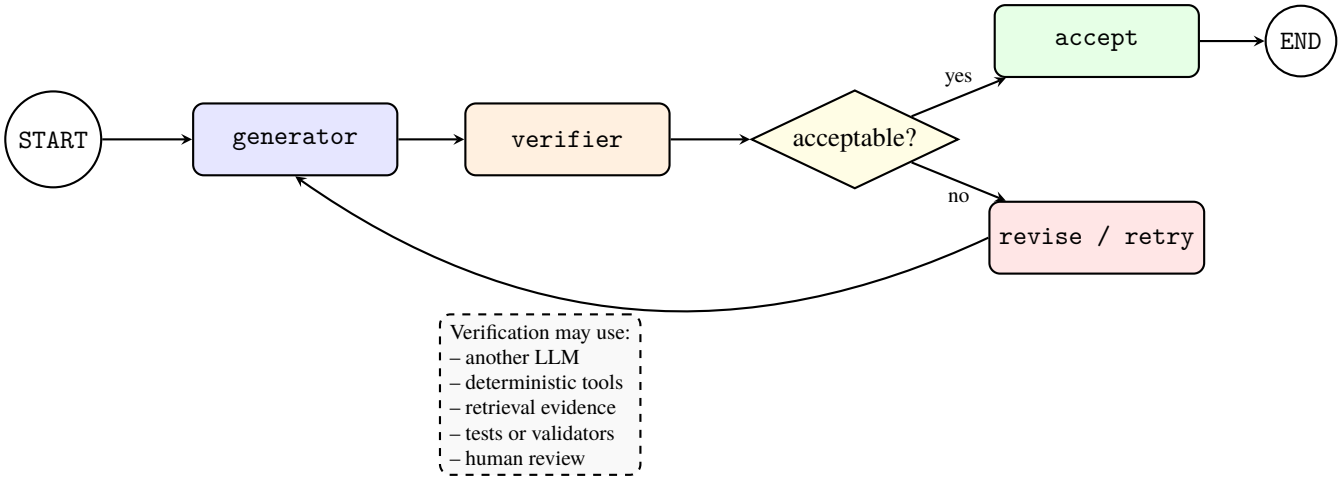


Figure 8.1: Reliable reasoning requires more than generation alone. A practical agent typically combines generation, verification, and control.

8.2.1 Core Idea

Suppose the model is asked a question q. Under chain-of-thought prompting, the model may generate different reasoning traces $r_1, r_2, \ldots, r_n$, each producing an answer $a_1, a_2, \ldots, a_n$. Self-consistency chooses the final answer by aggregation rather than by trusting the first sample:

$$\hat{a} = \arg\max_{a} \sum_{i=1}^{n} \mathbf{1}[a_i = a] \tag{8.1}$$

where $\mathbf{1}[\cdot]$ is the indicator function.

The intuition is that correct answers tend to be more stable across multiple independent reasoning samples, while incorrect answers are more diverse and less likely to dominate the vote.

8.2.2 Why It Works

Self-consistency addresses a key weakness of single-path reasoning: *path dependence*. A single chain-of-thought may make an early mistake and never recover. By sampling multiple paths, the model gets several opportunities to approach the problem correctly.

This strategy is particularly effective for:

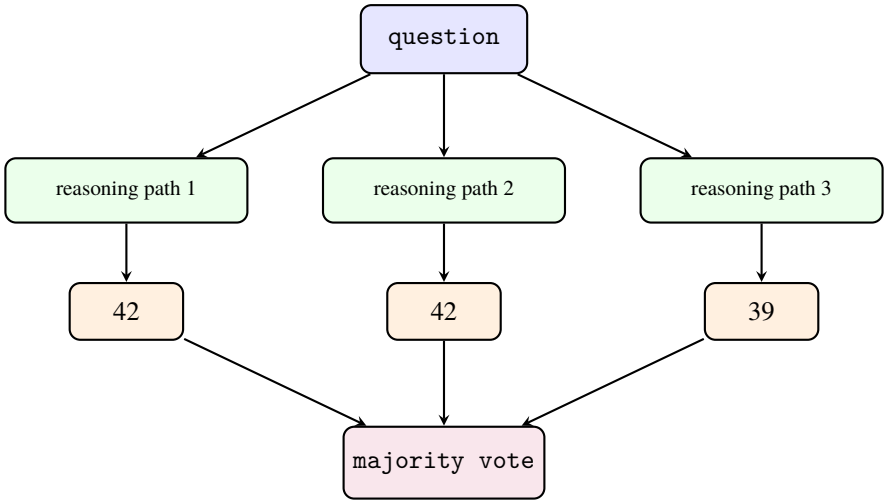
- arithmetic and symbolic reasoning,
- short logical problems,
- classification tasks with ambiguous prompts,
- structured decision tasks where one answer should dominate.

However, it is less effective when all samples share the same misconception, or when answer extraction is ambiguous.

8.2.3 Implementation in LangGraph

A self-consistency agent can be implemented as a small graph with three phases:

1. sample multiple reasoning traces,
2. extract candidate answers,



final answer: 42

Figure 8.2: Self-consistency samples multiple reasoning traces and aggregates their answers. The final answer is chosen by voting or another aggregation rule.

3. aggregate them into a final decision.

Listing 8.1: State definition for self-consistency reasoning

```python
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages


class SelfConsistencyState(TypedDict):
    messages: Annotated[list, add_messages]
    samples: list[str]       # raw reasoning traces
    extracted_answers: list[str] # parsed answers from each trace
    final_answer: str
```

Listing 8.2: A self-consistency agent in LangGraph

```python
from collections import Counter
from langgraph.graph import StateGraph, START, END
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

llm = ChatOpenAI(model="gpt-4o")
NUM_SAMPLES = 5


def sample_reasoning(state: SelfConsistencyState) -> SelfConsistencyState:
    traces = []
    for _ in range(NUM_SAMPLES):
        messages = [
            SystemMessage(content=(
                "Solve the user's question step by step. "
                "End with a line starting exactly with "
```

```
                "'ANSWER:'."
            )),
            *state["messages"],
        ]
        response = llm.invoke(messages)
        traces.append(response.content)
    return {"samples": traces}


def extract_answers(state: SelfConsistencyState) -> SelfConsistencyState:
    answers = []
    for trace in state["samples"]:
        answer = ""
        for line in trace.splitlines():
            if line.strip().startswith("ANSWER:"):
                answer = line.replace("ANSWER:", "").strip()
                break
        answers.append(answer)
    return {"extracted_answers": answers}


def aggregate(state: SelfConsistencyState) -> SelfConsistencyState:
    counts = Counter(state["extracted_answers"])
    final_answer, _ = counts.most_common(1)[0]
    return {"final_answer": final_answer}


graph = (
    StateGraph(SelfConsistencyState)
    .add_node("sample_reasoning", sample_reasoning)
    .add_node("extract_answers", extract_answers)
    .add_node("aggregate", aggregate)
    .add_edge(START, "sample_reasoning")
    .add_edge("sample_reasoning", "extract_answers")
    .add_edge("extract_answers", "aggregate")
    .add_edge("aggregate", END)
)

app = graph.compile()
```

8.2.4 Variants

Several variants of self-consistency are common in practice:

Majority vote choose the most frequent answer.

Weighted vote weight each answer by a confidence score.

Best-of-n generate multiple candidates and ask a judge model to rank them.

Reasoning-path clustering group semantically similar answers before voting.

Self-consistency is often the cheapest way to improve reliability when a task can tolerate multiple LLM calls.

8.3 Verifier-Guided Reasoning

Self-consistency treats agreement as a proxy for correctness. A stronger approach is to introduce an explicit **verifier** that evaluates the quality of candidate outputs. In this pattern, generation and evaluation are separated into distinct roles.

8.3.1 Generator and Verifier

A verifier-guided system contains at least two components:

1. a **generator**, which proposes an answer, plan, or reasoning trace,
2. a **verifier**, which scores that proposal against some criterion such as correctness, completeness, factual support, or policy compliance.

The verifier may be:

- another LLM prompted as a critic,
- a reward model trained from preference data,
- a programmatic checker,
- a hybrid system combining retrieval and rules.

The control policy then decides whether the candidate should be accepted, revised, or rejected.

8.3.2 Scoring Formulation

Let y denote a candidate output and $v(y)$ a verifier score. A simple decision rule is:

$$\text{accept}(y) = \begin{cases} 1 & \text{if } v(y) \geq \tau \\ 0 & \text{otherwise} \end{cases} \tag{8.2}$$

where τ is a threshold chosen by the system designer.

More advanced systems may compare several candidates and select the one with highest score:

$$\hat{y} = \arg\max_{y_i \in \mathcal{Y}} v(y_i) \tag{8.3}$$

where $\mathcal{Y}$ is a set of sampled candidates.

8.3.3 Implementation with a Judge Model

The simplest implementation uses the same base LLM twice: once as a generator and once as a judge. Although this does not remove all shared blind spots, it often improves robustness because the two roles are prompted differently.

Listing 8.3: State for verifier-guided reasoning

```
from typing import TypedDict


```

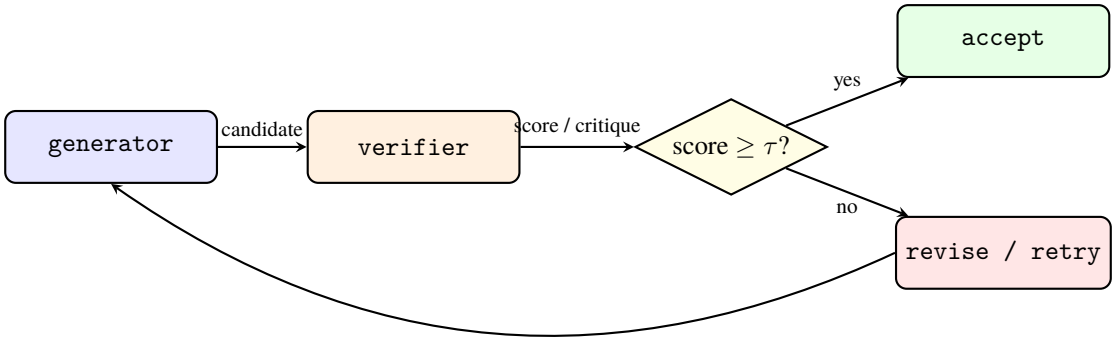


Figure 8.3: Verifier-guided reasoning separates proposal generation from evaluation. Low-scoring candidates can be revised, resampled, or escalated.

```python
class VerifyState(TypedDict):
    question: str
    candidate: str
    score: float
    critique: str
    attempts: int
    final_answer: str
```

Listing 8.4: A generator-verifier loop in LangGraph

```python
from typing import Literal
from langgraph.graph import StateGraph, START, END
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

llm = ChatOpenAI(model="gpt-4o")
MAX_ATTEMPTS = 3
THRESHOLD = 8.0


def generate(state: VerifyState) -> VerifyState:
    if state["attempts"] == 0:
        prompt = (
            "Answer the following question carefully and clearly."
        )
    else:
        prompt = (
            "Revise your previous answer using the critique below.\n\n"
            f"Critique:\n{state['critique']}\n\n"
            f"Previous answer:\n{state['candidate']}"
        )

    response = llm.invoke([
        SystemMessage(content=prompt),
        HumanMessage(content=state["question"]),
    ])
```

```
    return {
        "candidate": response.content,
        "attempts": state["attempts"] + 1,
    }


def verify(state: VerifyState) -> VerifyState:
    response = llm.invoke([
        SystemMessage(content=(
            "You are a strict verifier. Score the candidate answer "
            "from 1 to 10 for correctness, completeness, and clarity. "
            "Then provide a short critique. Format exactly as:\n"
            "SCORE: <number>\nCRITIQUE: <text>"
        )),
        HumanMessage(content=(
            f"Question:\n{state['question']}\n\n"
            f"Candidate answer:\n{state['candidate']}"
        )),
    ])

    score = 0.0
    critique = ""
    for line in response.content.splitlines():
        if line.startswith("SCORE:"):
            try:
                score = float(line.replace("SCORE:", "").strip())
            except ValueError:
                score = 0.0
        if line.startswith("CRITIQUE:"):
            critique = line.replace("CRITIQUE:", "").strip()

    return {"score": score, "critique": critique}


def decide(state: VerifyState) -> Literal["accept", "generate", "__end__"]:
    if state["score"] >= THRESHOLD:
        return "accept"
    if state["attempts"] >= MAX_ATTEMPTS:
        return "__end__"
    return "generate"


def accept(state: VerifyState) -> VerifyState:
    return {"final_answer": state["candidate"]}


graph = (
```

```
    StateGraph(VerifyState)
    .add_node("generate", generate)
    .add_node("verify", verify)
    .add_node("accept", accept)
    .add_edge(START, "generate")
    .add_edge("generate", "verify")
    .add_conditional_edges("verify", decide)
    .add_edge("accept", END)
)

app = graph.compile()
```

8.3.4 Benefits and Risks

Verifier-guided reasoning improves reliability in several ways:

- it separates creation from evaluation,
- it makes acceptance criteria explicit,
- it supports best-of-n search and reranking,
- it provides critiques that can drive revision.

However, verifier quality is crucial. A weak verifier may approve wrong answers or reject correct ones. In practice, the verifier often becomes the new bottleneck in system quality.

8.4 Tool-Based Verification

The strongest form of verification does not rely on another language model alone, but on **external tools** that provide deterministic or evidence-grounded checks. This pattern is especially important in agents, because agents often already have access to search, code execution, calculators, databases, or APIs.

8.4.1 Deterministic Checkers

Some tasks permit exact validation:

- arithmetic can be checked with a calculator,
- code can be executed against test cases,
- SQL queries can be run against a schema,
- JSON outputs can be validated against a schema,
- graph plans can be checked against type constraints.

These checks are far more reliable than free-form self-critique because they do not depend on the model's subjective judgment.

8.4.2 Evidence-Based Verification

Other tasks do not admit a single exact test, but can still be verified against evidence. For example:

- a factual answer can be checked against retrieved documents,
- a citation can be validated against source text,
- a travel recommendation can be verified against an official schedule,

- a legal or policy answer can be checked against authoritative rules.

In such settings, the verifier operates not on the answer alone, but on the pair (y, E) where E is the evidence set.

8.4.3 Pattern: Execute-Then-Verify

A robust design pattern is therefore:

1. generate a candidate,
2. use a tool to test or ground it,
3. transform tool output into feedback,
4. either accept or revise.

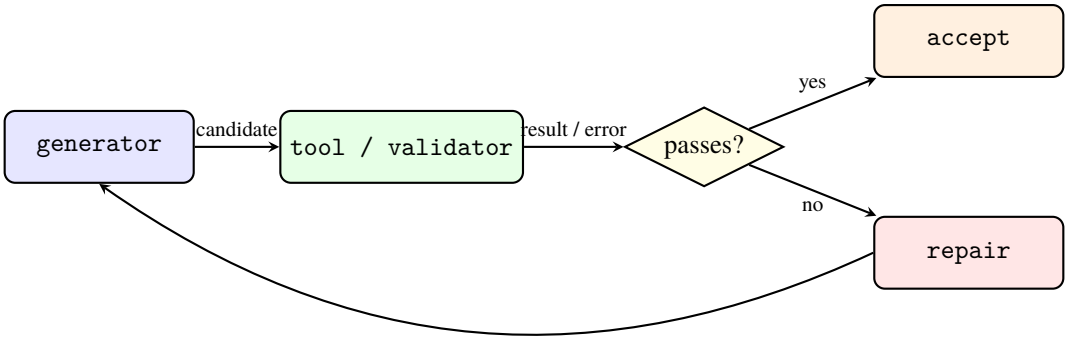


Figure 8.4: Tool-based verification is often more reliable than LLM-only verification because it uses external evidence or deterministic tests.

8.4.4 Example: Code Validation

The following pattern uses Python execution as a verifier for generated code.

Listing 8.5: Tool-based verification for code generation

```
import subprocess
from typing import TypedDict


class CodeState(TypedDict):
    task: str
    code: str
    passed: bool
    feedback: str
    attempts: int


def generate_code(state: CodeState) -> CodeState:
    prompt = (
        "Write Python code for the following task.\n\n"
        f"Task: {state['task']}\n\n"
    )
```

```
    if state["feedback"]:
        prompt += (
            "The previous attempt failed with this feedback:\n"
            f"{state['feedback']}\n\n"
            "Revise the code accordingly."
        )

    response = llm.invoke(prompt)
    return {
        "code": response.content,
        "attempts": state["attempts"] + 1,
    }


def run_tests(state: CodeState) -> CodeState:
    with open("/tmp/generated_code.py", "w") as f:
        f.write(state["code"])

    result = subprocess.run(
        ["python", "/tmp/generated_code.py"],
        capture_output=True,
        text=True,
        timeout=10,
    )

    if result.returncode == 0:
        return {"passed": True, "feedback": "All tests passed."}

    return {
        "passed": False,
        "feedback": result.stderr,
    }
```

The same idea applies beyond code. A planning agent can validate whether a workflow satisfies dependency constraints, whether required fields are present, or whether a tool call conforms to an API schema.

8.5 Confidence, Abstention, and Escalation

Verification does not always produce a binary outcome. In many realistic settings the agent remains uncertain even after self-consistency, reranking, or tool checks. In such cases, a reliable system should support **abstention** and **escalation**.

8.5.1 Why Abstention Matters

A common failure mode of LLM agents is forced answering. The model produces an answer because the interface expects one, even when available evidence is weak. In high-stakes settings, this is often worse than saying *"I am not confident enough to answer without more evidence."*

A reliable agent therefore needs a policy for deciding among at least four actions:

1. accept the answer,
2. gather more evidence,
3. retry with a different reasoning strategy,
4. escalate to a human.

8.5.2 Confidence Signals

Confidence can be estimated from multiple signals:

- agreement across self-consistency samples,
- verifier score,
- tool success or failure,
- retrieval coverage and source quality,
- consistency with prior state,
- explicit uncertainty language from the model.

A simple confidence heuristic may combine several of these into a single score. For example:

$$c = \alpha s + \beta a + \gamma t \tag{8.4}$$

where s is a verifier score, a is answer agreement, t is tool success, and α, β, γ are weights.

8.5.3 Routing Policy

The control layer can then use confidence thresholds to decide the next step:

$$\text{route}(c) = \begin{cases} \text{accept} & c \geq \tau_{\text{high}} \\ \text{retry / retrieve} & \tau_{\text{low}} \leq c < \tau_{\text{high}} \\ \text{escalate} & c < \tau_{\text{low}} \end{cases} \tag{8.5}$$

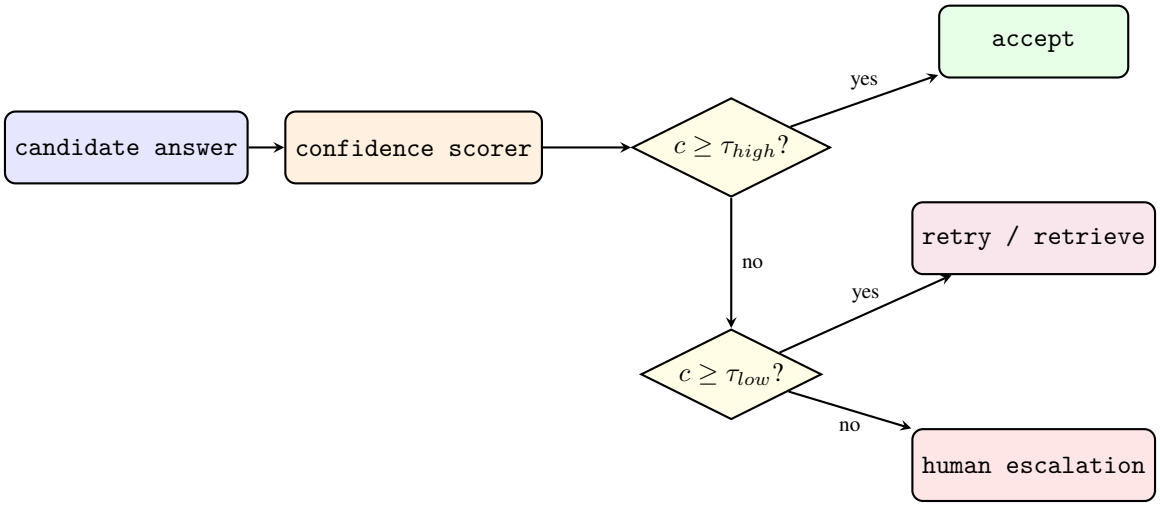


Figure 8.5: Confidence-aware routing enables the agent to accept, retry, or escalate depending on verification outcomes.

8.5.4 Implementation Sketch

Listing 8.6: Confidence-aware routing in LangGraph

```python
from typing import TypedDict, Literal


class ConfidenceState(TypedDict):
    answer: str
    verifier_score: float
    agreement: float
    tool_passed: bool
    confidence: float
    route: str


def compute_confidence(state: ConfidenceState) -> ConfidenceState:
    tool_score = 1.0 if state["tool_passed"] else 0.0
    confidence = (
        0.5 * (state["verifier_score"] / 10.0)
        + 0.3 * state["agreement"]
        + 0.2 * tool_score
    )
    return {"confidence": confidence}


def choose_route(
    state: ConfidenceState,
) -> Literal["accept", "retry", "human_review"]:
    c = state["confidence"]
    if c >= 0.80:
        return "accept"
    if c >= 0.50:
        return "retry"
    return "human_review"
```

This pattern is especially important when agents operate in support, healthcare, finance, or legal workflows, where uncertainty should trigger guardrails rather than confident improvisation.

8.6 Best-of-n and Reranking

A closely related pattern is **best-of-n generation**. Instead of returning the first answer, the agent generates multiple candidates and then uses a ranking function to choose the best. In practice, this ranking function may be:

- a judge LLM,
- a reward model,
- a task-specific metric,
- a hybrid score combining quality and cost.

This pattern is widely used because it balances exploration and control:

1. generation provides diversity,
2. reranking imposes selectivity,
3. only the best candidate is returned or executed.

Best-of-n differs from self-consistency in an important way. In self-consistency, the final answer emerges from aggregation. In best-of-n, the system explicitly ranks candidates and selects one. This makes it particularly useful for open-ended tasks such as writing, planning, tool-use formatting, and code generation, where correct outputs may not match exactly in wording.

8.7 Verification for Tool-Using Agents

The need for verification is even greater when agents interact with tools. In such systems, the agent's output is not merely text; it may be an *action proposal*. Errors can therefore propagate into the external environment.

Several additional checks become important:

- **Argument validation:** are tool arguments complete and well-typed?
- **Precondition checks:** is the action permitted in the current state?
- **Postcondition checks:** did the tool outcome match expectations?
- **Side-effect review:** should this action require human approval?

A robust tool-using agent should therefore verify both its reasoning *and* its action schema. This leads to a general design principle:

> *The closer an output is to an irreversible external action, the stronger the verification policy should be.*

For example, drafting an email may only need self-consistency or a light verifier. Sending the email, modifying a database record, or triggering a purchase should typically require stronger checks and possibly human approval.

8.8 Comparison of Verification Strategies

Table 8.1 summarises the main strategies introduced in this chapter.

These methods are complementary rather than exclusive. A production agent may combine several of them:

1. generate multiple candidates,
2. rank them with a verifier,
3. validate the top candidate with a tool,
4. route uncertain cases to retrieval or human review.

This layered design is often much more reliable than any single pattern in isolation.

8.9 Design Guidelines

The following practical guidelines help determine which verification strategy to use.

Table 8.1: Comparison of verification strategies for LLM agents.

Strategy	Cost	Strength	Best use cases
Self-consistency	Medium	Reduces single-path errors through aggregation	Math, logic, short reasoning tasks
Judge / verifier model	Medium–High	Makes acceptance criteria explicit; supports revision	General QA, planning, summarisation, writing
Tool-based verification	Low–High depending on tool	Objective and often deterministic	Code, calculations, schemas, structured workflows
Confidence-aware routing	Low	Supports abstention, retry, and human escalation	High-stakes systems, uncertain evidence, operational agents
Best-of-n reranking	Medium–High	Improves output quality through search and selection	Planning, code generation, creative tasks

8.9.1 Use Self-Consistency When

- the answer space is small or easily normalised,
- the task is reasoning-heavy but cheap enough to sample,
- voting is a reasonable proxy for correctness.

8.9.2 Use Verifier Models When

- outputs are open-ended,
- critique is useful for revision,
- you need explicit scoring or ranking.

8.9.3 Use Tool-Based Verification When

- correctness can be checked deterministically,
- external evidence is available,
- the cost of being wrong is high.

8.9.4 Use Abstention and Escalation When

- the domain is high stakes,
- evidence is incomplete,
- action consequences are significant,
- system confidence remains low after verification.

In practice, reliable agents are rarely built from a single reasoning prompt. They are built from *reasoning plus verification plus control*.

8.10 Summary

This chapter extended the reasoning patterns of the previous chapter by showing how agents can *verify* and *trust* their outputs.

- **Self-consistency** improves reliability by sampling multiple reasoning paths and aggregating their answers.
- **Verifier-guided reasoning** separates generation from evaluation, enabling scoring, critique, and iterative improvement.
- **Tool-based verification** uses deterministic programs or external evidence to check outputs more objectively than free-form self-critique alone.
- **Confidence-aware routing** supports abstention, retry, and human escalation when uncertainty remains too high.
- **Best-of-n reranking** improves output quality by combining exploration with selective acceptance.

The main insight is that reliable agent behavior does not come only from better prompts or larger models. It emerges from a system architecture that treats reasoning as something to be *generated, checked, and controlled*. Planning determines what the agent should try; verification determines what it should trust.

In the next chapter, we extend these ideas further by examining how agents can reason over **long-horizon workflows and decomposed tasks**, where coordination across many intermediate states becomes as important as the quality of any single reasoning step.

References

[1] Xuezhi Wang et al. “Self-consistency improves chain of thought reasoning in language models”. In: *arXiv preprint arXiv:2203.11171* (2022).

Part IV

Multi-Agent Systems

Chapter 9 Multi-Agent Architectures

The previous part showed how a single agent can be made more capable through planning, reflection, and structured reasoning. Yet even the most sophisticated single agent faces fundamental scaling limits: its context window is finite, its expertise is necessarily general, and its ability to self-correct is bounded by its own blind spots.

Multi-agent systems address these limits by distributing a complex task across several specialised agents, each responsible for a well-defined sub-problem. Rather than building one monolithic agent that must handle research, analysis, writing, and quality assurance simultaneously, we compose a team of focused agents that collaborate through structured communication.

This chapter introduces the foundational concepts of multi-agent design in LangGraph. We begin with the motivation and core principles, survey the major communication topologies, and then build progressively more sophisticated architectures: sequential pipelines, parallel fan-out, supervisor–worker hierarchies, and collaborative debate.

9.1 Why Multiple Agents?

The move from single-agent to multi-agent design is motivated by several converging factors.

9.1.1 Specialisation

A single prompt cannot simultaneously optimise for research, creative writing, code generation, and critical review. By assigning each capability to a dedicated agent with its own system prompt, tool set, and even model choice, we achieve a form of *functional specialisation* analogous to a team of human experts.

9.1.2 Context Management

Language models have finite context windows. A single agent working on a large task risks filling its context with intermediate results, leaving little room for new reasoning. In a multi-agent system each agent operates within its own context, receiving only the information it needs.

9.1.3 Reliability Through Redundancy

When multiple agents independently tackle the same sub-problem, their outputs can be compared, voted on, or merged. This redundancy provides a natural mechanism for error detection that is unavailable in single-agent systems.

9.1.4 Modularity and Reuse

Each agent in a multi-agent system is a self-contained component that can be tested, improved, or replaced independently. A research agent developed for one project can be reused in another without modification.

Table 9.1 summarises the key differences.

Table 9.1: Single-agent versus multi-agent design.

Dimension	Single agent	Multi-agent
Prompt design	One prompt covers all tasks	Each agent has a focused prompt
Tool access	All tools available always	Tools scoped per agent
Context usage	Shared, may overflow	Partitioned across agents
Error handling	Self-correction only	Cross-agent verification
Scalability	Limited by model capacity	Scales with agent count
Complexity	Low	Higher orchestration cost

9.2 Communication Topologies

The defining characteristic of a multi-agent system is *how agents communicate*. The communication topology determines which agents can exchange information, in what order, and under whose control.

We identify four fundamental topologies, each suited to different task structures.

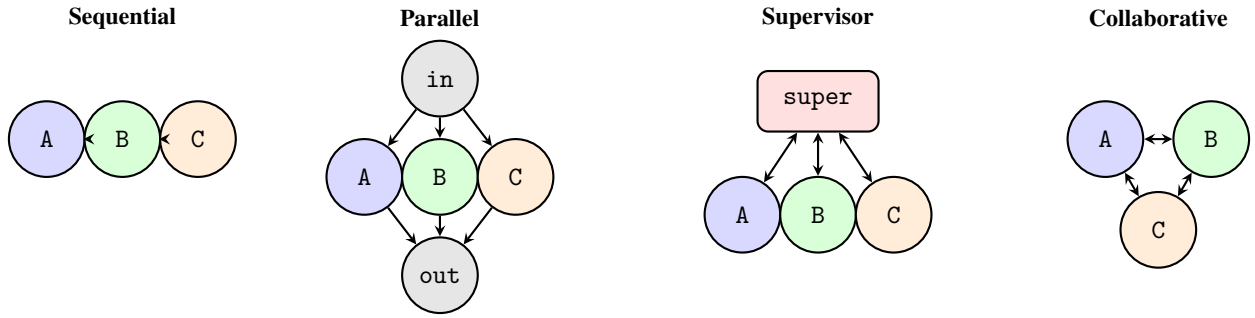


Figure 9.1: The four fundamental multi-agent communication topologies. Sequential agents form a pipeline. Parallel agents work independently on the same input. Supervisor–worker agents are centrally coordinated. Collaborative agents communicate freely with each other.

The following sections implement each topology in LangGraph.

9.3 Sequential Pipeline

The simplest multi-agent topology is the **sequential pipeline**: a chain of agents in which each agent receives the output of the previous one, processes it, and passes the result forward.

This topology is well suited to tasks that decompose into ordered stages — for example, *research* → *draft* → *edit* → *format*.

9.3.1 Architecture

Figure 9.2 shows the graph structure.

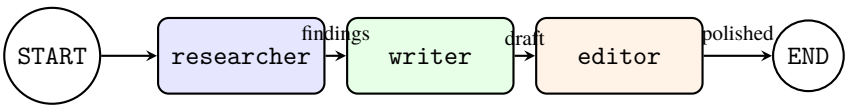


Figure 9.2: A sequential pipeline of three specialised agents: researcher, writer, and editor. Each agent's output becomes the next agent's input.

9.3.2 Implementation

Listing 9.1 provides a complete implementation.

Listing 9.1: A sequential multi-agent pipeline

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, START, END
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

llm = ChatOpenAI(model="gpt-4o")


class PipelineState(TypedDict):
    messages: Annotated[list, add_messages]
    research: str
    draft: str
    final: str


def researcher(state: PipelineState) -> PipelineState:
    messages = [
        SystemMessage(content=(
            "You are a research analyst. Given the user's "
            "topic, produce a concise set of key findings "
            "with supporting evidence. Output only facts."
        )),
        *state["messages"],
    ]
    response = llm.invoke(messages)
    return {"research": response.content}


def writer(state: PipelineState) -> PipelineState:
    messages = [
        SystemMessage(content=(
            "You are a professional writer. Using the "
            "research below, write a well-structured article "
            "of 3-4 paragraphs.\n\n"
            f"Research:\n{state['research']}"
        )),
    ]
    response = llm.invoke(messages)
    return {"draft": response.content}


def editor(state: PipelineState) -> PipelineState:
    messages = [
```

```
        SystemMessage(content=(
            "You are a meticulous editor. Improve the "
            "following draft for clarity, grammar, and "
            "flow. Preserve the original meaning.\n\n"
            f"Draft:\n{state['draft']}"
        )),
    ]
    response = llm.invoke(messages)
    return {"final": response.content}


graph = (
    StateGraph(PipelineState)
    .add_node("researcher", researcher)
    .add_node("writer", writer)
    .add_node("editor", editor)
    .add_edge(START, "researcher")
    .add_edge("researcher", "writer")
    .add_edge("writer", "editor")
    .add_edge("editor", END)
)

app = graph.compile()

result = app.invoke({
    "messages": [HumanMessage(
        content="The impact of sleep deprivation on cognition"
    )],
    "research": "",
    "draft": "",
    "final": "",
})
print(result["final"])
```

9.3.3 When to Use

Sequential pipelines work best when the task has a natural ordering of stages and each stage's output is a complete, self-contained artifact. They are simple to reason about and easy to debug, since each agent's contribution can be inspected independently.

The main limitation is rigidity: information flows in one direction only, so downstream agents cannot request clarifications from upstream agents.

9.4 Parallel Fan-Out and Aggregation

When a task can be decomposed into *independent* sub-tasks, executing them in parallel reduces latency and enables each agent to operate with a focused context. The results are then *aggregated* by a merge node.

This pattern is sometimes called **map–reduce** by analogy with the distributed computing paradigm.

9.4.1 Architecture

Figure 9.3 illustrates the fan-out and aggregation pattern.

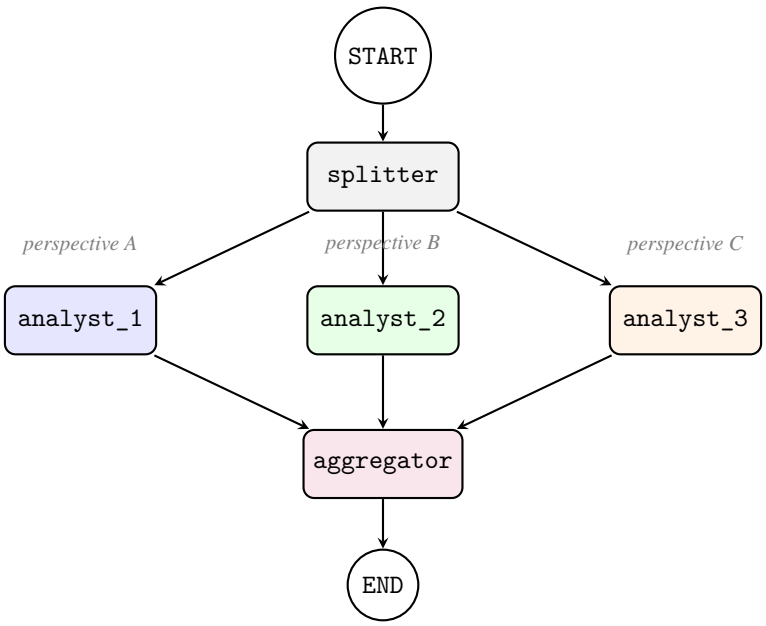


Figure 9.3: Fan-out and aggregation. The splitter distributes the task to three parallel analysts, whose results are collected and synthesised by the aggregator.

9.4.2 Implementation with Send

LangGraph supports dynamic fan-out through the `Send` primitive. A conditional edge can return a list of `Send` objects, each directing execution to a node with a distinct payload.

Listing 9.2 demonstrates this pattern.

Listing 9.2: Parallel fan-out using `Send`

```
import operator
from typing import TypedDict, Annotated
from langgraph.graph import StateGraph, START, END
from langgraph.types import Send
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

llm = ChatOpenAI(model="gpt-4o")


class MainState(TypedDict):
    topic: str
    perspectives: list[str]
    analyses: Annotated[list[str], operator.add]
    synthesis: str


class AnalystInput(TypedDict):
```

```
    topic: str
    perspective: str


def assign_perspectives(state: MainState) -> MainState:
    """Define the perspectives to analyse."""
    return {
        "perspectives": [
            "economic impact",
            "social implications",
            "technological feasibility",
        ]
    }


def fan_out(state: MainState) -> list[Send]:
    """Create a Send for each perspective."""
    return [
        Send("analyst", {
            "topic": state["topic"],
            "perspective": p,
        })
        for p in state["perspectives"]
    ]


def analyst(state: AnalystInput) -> dict:
    """Analyse the topic from one perspective."""
    messages = [
        SystemMessage(content=(
            f"You are an expert in {state['perspective']}. "
            f"Analyse the following topic from your "
            f"perspective in 2-3 sentences."
        )),
        HumanMessage(content=state["topic"]),
    ]
    response = llm.invoke(messages)
    return {"analyses": [
        f"[{state['perspective']}] {response.content}"
    ]}


def aggregator(state: MainState) -> MainState:
    """Synthesise all analyses."""
    combined = "\n\n".join(state["analyses"])
    messages = [
        SystemMessage(content=(
            "Synthesise the following expert analyses into "
```

```
            "a balanced, coherent summary."
        )),
        HumanMessage(content=combined),
    ]
    response = llm.invoke(messages)
    return {"synthesis": response.content}


graph = (
    StateGraph(MainState)
    .add_node("assign", assign_perspectives)
    .add_node("analyst", analyst)
    .add_node("aggregator", aggregator)
    .add_edge(START, "assign")
    .add_conditional_edges("assign", fan_out)
    .add_edge("analyst", "aggregator")
    .add_edge("aggregator", END)
)

app = graph.compile()

result = app.invoke({
    "topic": "Universal basic income",
    "perspectives": [],
    "analyses": [],
    "synthesis": "",
})
print(result["synthesis"])
```

The `analyses` field uses an `operator.add` reducer so that each analyst's output is appended to the list rather than overwriting it. When all `Send` invocations complete, LangGraph automatically triggers the aggregator with the accumulated state.

9.5 Supervisor–Worker Architecture

Sequential pipelines follow a fixed order; parallel fans follow a fixed decomposition. But many real-world tasks require *dynamic orchestration*: the system must decide at runtime which agent to invoke next, how many times, and in what order.

The **supervisor–worker** architecture addresses this by introducing a dedicated orchestration agent — the *supervisor* — that manages a pool of specialised *workers*. The supervisor inspects the current state, decides which worker should act next, and routes execution accordingly.

9.5.1 Architecture

Figure 9.4 shows the architecture.

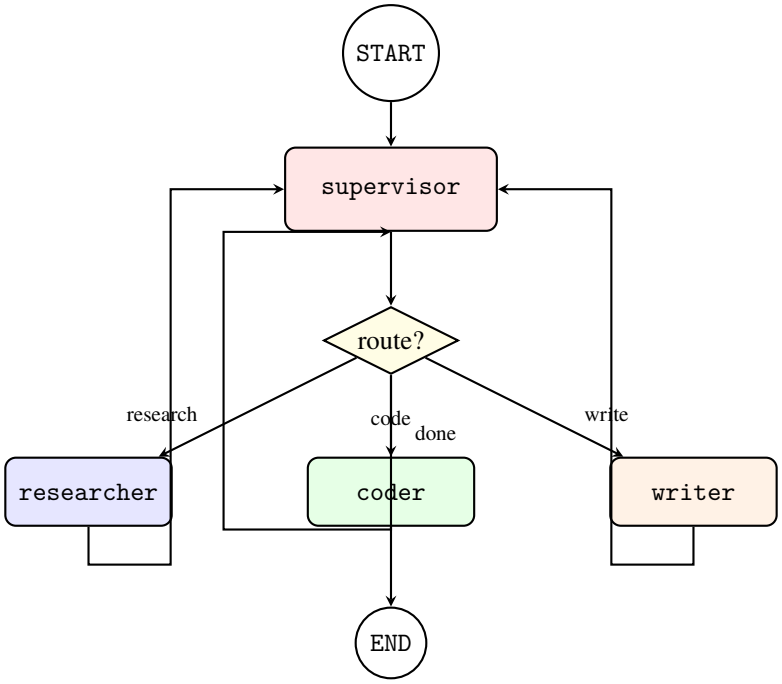


Figure 9.4: Supervisor–worker architecture. The supervisor decides which worker to invoke next. Workers report their results back to the supervisor, which iterates until the task is complete.

9.5.2 Supervisor Design

The supervisor is itself a language model call, but with a carefully crafted system prompt and a restricted output format. It receives the full conversation history and must produce one of a fixed set of routing decisions.

Listing 9.3: Supervisor node with structured routing output

```
from typing import TypedDict, Annotated, Literal
from langgraph.graph.message import add_messages
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage

llm = ChatOpenAI(model="gpt-4o")

WORKERS = ["researcher", "coder", "writer"]
OPTIONS = WORKERS + ["FINISH"]


class SupervisorState(TypedDict):
    messages: Annotated[list, add_messages]
    next_worker: str
    iteration: int


def supervisor(state: SupervisorState) -> SupervisorState:
    """Decide which worker to invoke next."""
    worker_list = ", ".join(WORKERS)
    messages = [
        SystemMessage(content=(
```

```
            f"You are a supervisor managing these workers: "
            f"{worker_list}.\n\n"
            "Given the conversation so far, decide which "
            "worker should act next, or say FINISH if the "
            "task is complete.\n\n"
            f"Respond with ONLY one of: {', '.join(OPTIONS)}"
        )),
        *state["messages"],
    ]
    response = llm.invoke(messages)
    choice = response.content.strip()

    # Validate
    if choice not in OPTIONS:
        choice = "FINISH"

    return {
        "next_worker": choice,
        "iteration": state["iteration"] + 1,
    }
```

9.5.3 Worker Agents

Each worker is a self-contained agent with its own system prompt and (optionally) its own set of tools.

Listing 9.4: Specialised worker agents

```
from langchain_core.messages import AIMessage


def researcher(state: SupervisorState) -> SupervisorState:
    messages = [
        SystemMessage(content=(
            "You are a research assistant. Find relevant "
            "information for the task described in the "
            "conversation. Be concise and factual."
        )),
        *state["messages"],
    ]
    response = llm.invoke(messages)
    return {"messages": [AIMessage(
        content=f"[Researcher]: {response.content}",
        name="researcher",
    )]}


def coder(state: SupervisorState) -> SupervisorState:
    messages = [
        SystemMessage(content=(
```

```
            "You are an expert programmer. Write clean, "
            "well-documented code to solve the task."
        )),
        *state["messages"],
    ]
    response = llm.invoke(messages)
    return {"messages": [AIMessage(
        content=f"[Coder]: {response.content}",
        name="coder",
    )]}


def writer(state: SupervisorState) -> SupervisorState:
    messages = [
        SystemMessage(content=(
            "You are a skilled writer. Produce polished, "
            "reader-friendly text based on the conversation."
        )),
        *state["messages"],
    ]
    response = llm.invoke(messages)
    return {"messages": [AIMessage(
        content=f"[Writer]: {response.content}",
        name="writer",
    )]}
```

9.5.4 Graph Assembly

Listing 9.5 wires the supervisor and workers into a complete graph.

Listing 9.5: Assembling the supervisor–worker graph

```
from langgraph.graph import StateGraph, START, END


def route_worker(
    state: SupervisorState,
) -> Literal["researcher", "coder", "writer", "__end__"]:
    if state["next_worker"] == "FINISH":
        return "__end__"
    return state["next_worker"]


MAX_ITERATIONS = 10


def guard_iterations(
    state: SupervisorState,
) -> Literal["supervisor", "__end__"]:
```

```
    if state["iteration"] >= MAX_ITERATIONS:
        return "__end__"
    return "supervisor"


graph = (
    StateGraph(SupervisorState)
    .add_node("supervisor", supervisor)
    .add_node("researcher", researcher)
    .add_node("coder", coder)
    .add_node("writer", writer)
    # Entry
    .add_edge(START, "supervisor")
    # Supervisor routes to a worker or END
    .add_conditional_edges("supervisor", route_worker)
    # Each worker loops back through an iteration guard
    .add_conditional_edges("researcher", guard_iterations)
    .add_conditional_edges("coder", guard_iterations)
    .add_conditional_edges("writer", guard_iterations)
)

app = graph.compile()
```

9.5.5 Execution Trace

Figure 9.5 illustrates a typical execution trace.

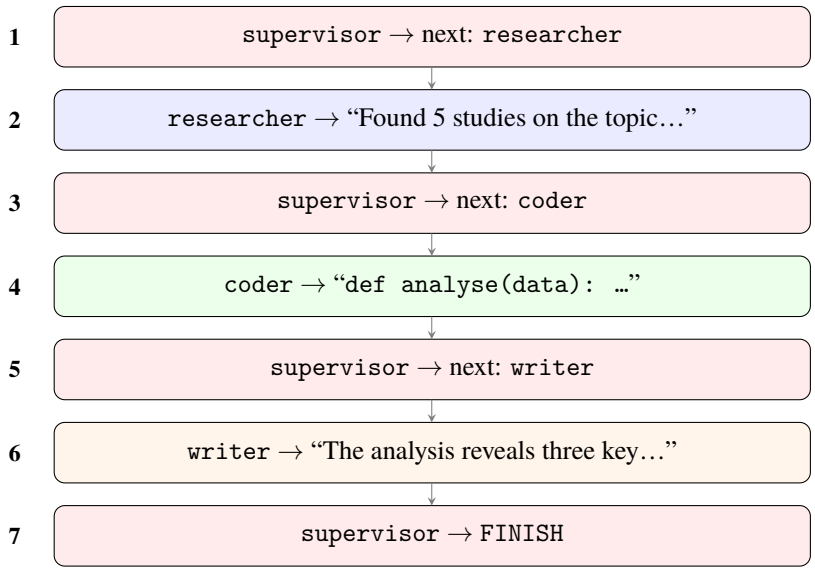


Figure 9.5: Execution trace of a supervisor–worker agent. The supervisor dynamically selects workers based on task progress, invoking research, coding, and writing agents before declaring the task complete.

9.6 Collaborative Debate

In the topologies discussed so far, agents communicate through a shared state but do not directly address each other. **Collaborative debate** introduces a richer form of interaction in which agents take turns responding to each other's arguments, gradually converging on a consensus.

This approach is inspired by research on multi-agent debate [1, 2], which shows that having language models critique each other's reasoning improves factual accuracy and reduces hallucination.

9.6.1 Architecture

The debate architecture arranges agents in a round-robin loop moderated by a judge.

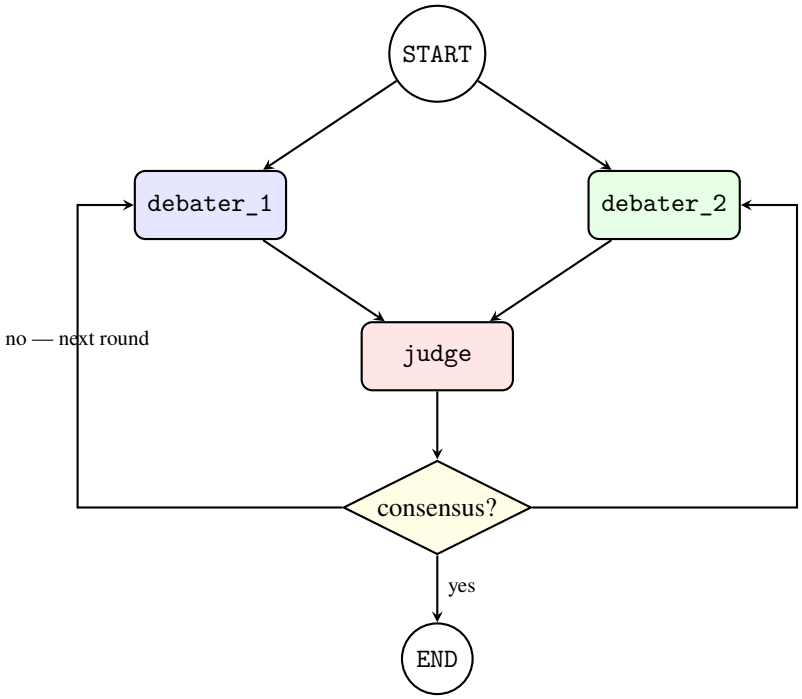


Figure 9.6: Collaborative debate. Two debaters present and refine their arguments. A judge evaluates after each round and either declares consensus or triggers another round.

9.6.2 Implementation

Listing 9.6: A collaborative debate with judge

```
import operator
from typing import TypedDict, Annotated, Literal
from langgraph.graph import StateGraph, START, END
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

llm = ChatOpenAI(model="gpt-4o")


class DebateState(TypedDict):
```

```
    question: str
    history: Annotated[list[str], operator.add]
    round: int
    verdict: str
    consensus: bool


MAX_ROUNDS = 3


def debater_1(state: DebateState) -> DebateState:
    history_text = "\n".join(state["history"]) or "None yet."
    messages = [
        SystemMessage(content=(
            "You are Debater 1. You tend to approach "
            "problems from an empirical, data-driven "
            "perspective. Review the debate history and "
            "present your argument in 2-3 sentences."
        )),
        HumanMessage(content=(
            f"Question: {state['question']}\n\n"
            f"Debate so far:\n{history_text}"
        )),
    ]
    response = llm.invoke(messages)
    return {"history": [
        f"[Debater 1, Round {state['round']+1}]: "
        f"{response.content}"
    ]}


def debater_2(state: DebateState) -> DebateState:
    history_text = "\n".join(state["history"])
    messages = [
        SystemMessage(content=(
            "You are Debater 2. You tend to approach "
            "problems from a theoretical, first-principles "
            "perspective. Review the debate history and "
            "present your argument in 2-3 sentences."
        )),
        HumanMessage(content=(
            f"Question: {state['question']}\n\n"
            f"Debate so far:\n{history_text}"
        )),
    ]
    response = llm.invoke(messages)
    return {"history": [
        f"[Debater 2, Round {state['round']+1}]: "
```

```
        f"{response.content}"
    ]}


def judge(state: DebateState) -> DebateState:
    history_text = "\n".join(state["history"])
    messages = [
        SystemMessage(content=(
            "You are an impartial judge. Review the debate "
            "and determine whether the debaters have reached "
            "a reasonable consensus. If yes, provide a final "
            "verdict starting with 'CONSENSUS:'. If not, "
            "start with 'CONTINUE:' and explain what "
            "disagreements remain."
        )),
        HumanMessage(content=(
            f"Question: {state['question']}\n\n"
            f"Full debate:\n{history_text}"
        )),
    ]
    response = llm.invoke(messages)
    is_consensus = response.content.startswith("CONSENSUS:")
    return {
        "verdict": response.content,
        "consensus": is_consensus,
        "round": state["round"] + 1,
    }


def should_continue(
    state: DebateState,
) -> Literal["debater_1", "__end__"]:
    if state["consensus"] or state["round"] >= MAX_ROUNDS:
        return "__end__"
    return "debater_1"


graph = (
    StateGraph(DebateState)
    .add_node("debater_1", debater_1)
    .add_node("debater_2", debater_2)
    .add_node("judge", judge)
    .add_edge(START, "debater_1")
    .add_edge("debater_1", "debater_2")
    .add_edge("debater_2", "judge")
    .add_conditional_edges("judge", should_continue)
)

```

```
app = graph.compile()

result = app.invoke({
    "question": "Should programming be taught in all high schools?",
    "history": [],
    "round": 0,
    "verdict": "",
    "consensus": False,
})
print(f"Rounds: {result['round']}")
print(f"Verdict:\n{result['verdict']}")
```

9.7 Hierarchical Multi-Agent Systems

For very complex tasks, a single supervisor may itself become a bottleneck. **Hierarchical** architectures address this by composing supervisors into layers: a top-level supervisor delegates to mid-level supervisors, each of which manages its own team of workers.

9.7.1 Architecture

Figure 9.7 illustrates a two-level hierarchy.

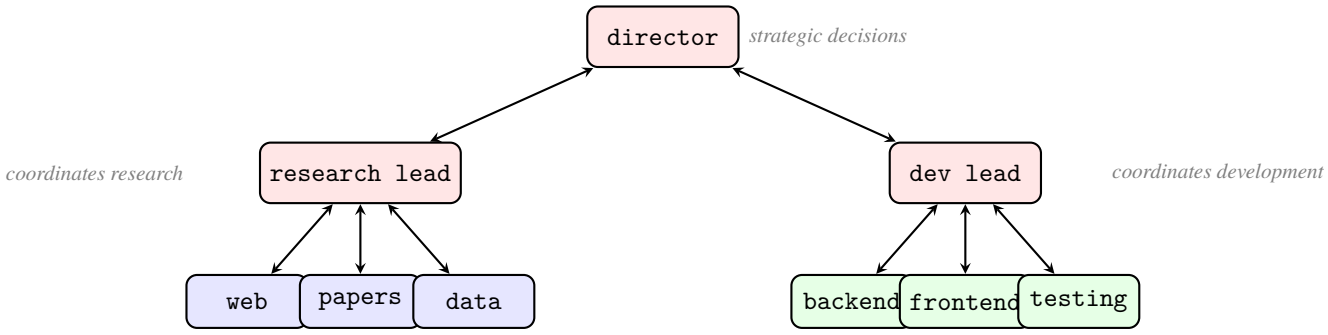


Figure 9.7: A two-level hierarchical multi-agent system. The director delegates to team leads, each of which manages specialised workers.

9.7.2 Subgraphs in LangGraph

LangGraph supports hierarchical composition through **subgraphs**. A compiled graph can be used as a node inside a parent graph, creating a natural nesting of supervisors and teams.

Listing 9.7: Using a compiled subgraph as a node in a parent graph

```
from langgraph.graph import StateGraph, START, END


# --- Research team (inner graph) ---
research_graph = (
    StateGraph(SupervisorState)
    .add_node("research_lead", research_lead)
```

```
    .add_node("web_searcher", web_searcher)
    .add_node("paper_reader", paper_reader)
    .add_edge(START, "research_lead")
    # ... routing edges ...
)
research_team = research_graph.compile()


# --- Development team (inner graph) ---
dev_graph = (
    StateGraph(SupervisorState)
    .add_node("dev_lead", dev_lead)
    .add_node("backend_dev", backend_dev)
    .add_node("frontend_dev", frontend_dev)
    .add_edge(START, "dev_lead")
    # ... routing edges ...
)
dev_team = dev_graph.compile()


# --- Top-level director (outer graph) ---
class DirectorState(TypedDict):
    messages: Annotated[list, add_messages]
    next_team: str


director_graph = (
    StateGraph(DirectorState)
    .add_node("director", director)
    .add_node("research_team", research_team) # subgraph
    .add_node("dev_team", dev_team)       # subgraph
    .add_edge(START, "director")
    .add_conditional_edges("director", route_team)
    .add_edge("research_team", "director")
    .add_edge("dev_team", "director")
)

app = director_graph.compile()
```

Each subgraph runs as an independent unit: it maintains its own internal state, executes its own loop, and returns a result to the parent graph. The parent sees it as a single node.

9.8 Design Patterns and Best Practices

Designing multi-agent systems requires balancing expressiveness against complexity. The following guidelines distil recurring lessons from practice.

9.8.1 Start Simple, Add Agents Incrementally

Begin with a single agent. Add a second agent only when you can identify a clear failure mode that specialisation would address. Many tasks that *seem* to need multiple agents can be handled by a single agent with a better prompt or additional tools.

9.8.2 Define Clear Interfaces

Each agent should have a well-defined input–output contract. The state fields that an agent reads and writes should be documented explicitly, just as function signatures are documented in conventional software.

9.8.3 Limit Shared State

Agents that share too much state become tightly coupled, defeating the purpose of modular design. Prefer narrow interfaces: each agent should read only the state fields it needs and write only the fields it is responsible for.

9.8.4 Monitor and Trace Execution

Multi-agent systems are harder to debug than single agents. LangGraph provides built-in tracing through LangSmith, which records every node invocation, state transition, and LLM call. Always enable tracing during development.

9.8.5 Guard Against Infinite Loops

Any cycle in the graph is a potential infinite loop. Always pair supervisor–worker loops and debate loops with explicit iteration limits. Defensive routing functions should map unexpected outputs to a safe default (typically `END`).

9.8.6 Choosing a Topology

Table 9.2 provides guidance on selecting the right topology for a given task.

9.9 Summary

This chapter introduced the design and implementation of multi-agent systems in LangGraph.

- Multi-agent design is motivated by the need for **specialisation**, **context management**, **reliability through redundancy**, and **modularity**.
- The four fundamental **communication topologies** — sequential, parallel, supervisor–worker, and collaborative — each suit different task structures.
- **Sequential pipelines** chain agents in a fixed order, passing artifacts from one stage to the next.
- **Parallel fan-out** distributes independent sub-tasks across agents using LangGraph's `Send` primitive, then aggregates the results.
- **Supervisor–worker** architectures use a central orchestrator to dynamically route tasks to specialised workers.

Table 9.2: Topology selection guide.

Topology	Best suited for
Sequential pipeline	Tasks with a natural ordering of stages (e.g. research → write → edit).
Parallel fan-out	Tasks decomposable into independent sub-tasks (e.g. multi-perspective analysis).
Supervisor–worker	Open-ended tasks where the required steps cannot be determined in advance (e.g. complex customer requests).
Collaborative debate	Tasks where accuracy is paramount and adversarial cross-examination reduces errors (e.g. fact-checking, legal review).
Hierarchical	Large-scale tasks involving multiple teams with distinct skill sets (e.g. full-stack software projects).

- **Collaborative debate** arranges agents in an adversarial loop moderated by a judge, improving accuracy through structured disagreement.
- **Hierarchical systems** nest subgraphs to manage complexity in large-scale projects.

Table 9.3 provides a quick reference.

Table 9.3: Summary of multi-agent patterns and their LangGraph mechanisms.

Pattern	Key mechanism	When to use
Sequential	`add_edge` chain	Ordered stages
Parallel	`Send` + reducer	Independent sub-tasks
Supervisor	Conditional routing loop	Dynamic orchestration
Debate	Round-robin + judge	Accuracy-critical tasks
Hierarchical	Compiled subgraphs as nodes	Large-scale, multi-team

In the next part we will explore **agent orchestration** patterns that address the operational concerns of deploying multi-agent systems in production: human-in-the-loop control, streaming, fault tolerance, and real-time monitoring.

References

[1] Chi-Min Chan et al. "Chateval: Towards better llm-based evaluators through multi-agent debate". In: *arXiv preprint arXiv:2308.07201* (2023).

[2] Tongxuan Liu et al. "Groupdebate: Enhancing the efficiency of multi-agent debate using group discussion". In: *arXiv preprint arXiv:2409.14051* (2024).

Part V

Agent Orchestration

Chapter 10 Agent Orchestration and Human-in-the-Loop Control

The previous parts of this book focused on the *design* of agent systems: how to structure graphs, manage state, invoke tools, plan ahead, and coordinate multiple agents. These techniques produce agents that are increasingly capable — but capability alone is not sufficient for production deployment.

Real-world agent systems must also be **controllable**. Stakeholders need the ability to inspect what an agent is doing, intervene when it goes wrong, approve high-stakes actions before they execute, and recover gracefully from failures. These operational concerns fall under the umbrella of **agent orchestration**.

This chapter introduces the orchestration primitives provided by LangGraph:

1. **Human-in-the-loop** patterns that pause execution for human approval or input.
2. **Breakpoints** that halt execution at designated nodes for inspection.
3. **Dynamic state modification** that allows operators to edit the agent's state at runtime.
4. **Time travel** that enables replaying or branching from earlier states.
5. **Streaming** that provides real-time visibility into agent execution.
6. **Fault tolerance** patterns that handle errors and enable recovery.

Together, these mechanisms transform an autonomous agent into a system that humans can supervise, steer, and trust.

10.1 The Case for Human Oversight

Fully autonomous agents are appealing in theory but dangerous in practice. Language models hallucinate, tools can produce unintended side effects, and the cost of an unchecked mistake grows with the agent's level of authority.

Consider the following scenarios:

- An agent about to send an email to a client with incorrect pricing.
- A coding agent about to execute a database migration that would delete production data.
- A research agent about to commit to an expensive API call when a cheaper alternative exists.

In each case, a brief human review would prevent a costly error. The challenge is designing the system so that human oversight is seamless: the agent should pause at the right moments, present the right information, and resume efficiently after the human responds.

Figure 10.1 illustrates the spectrum of agent autonomy.

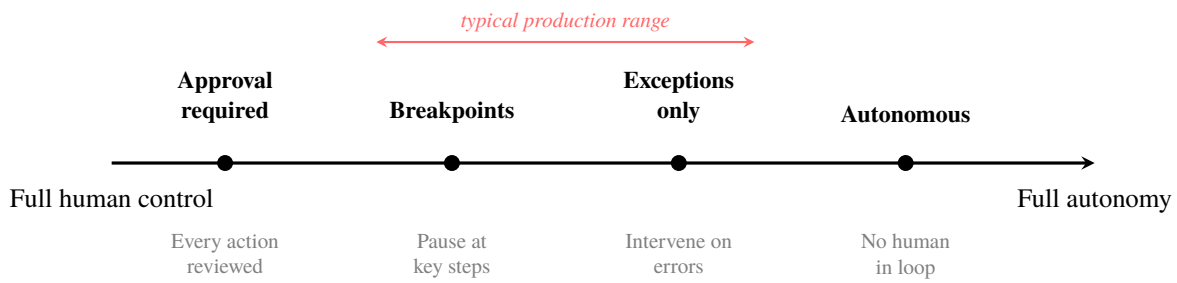


Figure 10.1: The autonomy spectrum. Most production systems operate between breakpoint-based oversight and exception-only intervention, balancing efficiency against safety.

10.2 Breakpoints

A **breakpoint** is the simplest orchestration primitive: it tells LangGraph to pause execution *before* or *after* a specified node. When a breakpoint is hit, the graph state is persisted via the checkpointer, and control returns to the caller. The caller (a human or an external system) can then inspect the state, optionally modify it, and resume execution.

10.2.1 Setting Breakpoints

Breakpoints are configured at compile time by passing `interrupt_before` or `interrupt_after` to the `compile` method.

Listing 10.1: Configuring breakpoints at compile time

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import MemorySaver
from langchain_core.messages import HumanMessage, AIMessage


class AgentState(TypedDict):
    messages: Annotated[list, add_messages]


def research(state: AgentState) -> AgentState:
    return {"messages": [
        AIMessage(content="Found 3 relevant articles.")
    ]}


def write(state: AgentState) -> AgentState:
    return {"messages": [
        AIMessage(content="Draft: The evidence suggests...")
    ]}


def send_email(state: AgentState) -> AgentState:
    # High-stakes: actually sends an email
    return {"messages": [
        AIMessage(content="Email sent to client.")
    ]}


graph = (
    StateGraph(AgentState)
    .add_node("research", research)
    .add_node("write", write)
    .add_node("send_email", send_email)
```

```
    .add_edge(START, "research")
    .add_edge("research", "write")
    .add_edge("write", "send_email")
    .add_edge("send_email", END)
)

# Pause BEFORE the high-stakes node
app = graph.compile(
    checkpointer=MemorySaver(),
    interrupt_before=["send_email"],
)
```

10.2.2 Inspecting and Resuming

When the breakpoint triggers, the graph returns control to the caller. The caller can inspect the current state, and then resume execution by invoking the graph with `None` as the input.

Listing 10.2: Inspecting state at a breakpoint and resuming

```
config = {"configurable": {"thread_id": "review-1"}}

# Run until the breakpoint
result = app.invoke(
    {"messages": [HumanMessage(content="Send Q3 report.")]},
    config=config,
)

# Inspect the state at the breakpoint
state = app.get_state(config)
print("Paused before:", state.next)
# ('send_email',)

# Review the draft
for msg in state.values["messages"]:
    print(f" {msg.__class__.__name__}: {msg.content}")

# --- Human decides to approve ---
# Resume from the breakpoint
final = app.invoke(None, config=config)
print(final["messages"][-1].content)
# "Email sent to client."
```

Figure 10.2 illustrates the execution flow with a breakpoint.

10.3 Human-in-the-Loop Patterns

Breakpoints provide the mechanism for pausing execution; human-in-the-loop (HITL) patterns determine *what the human does* during the pause. We identify four common patterns, each suited to a different level of human involvement.

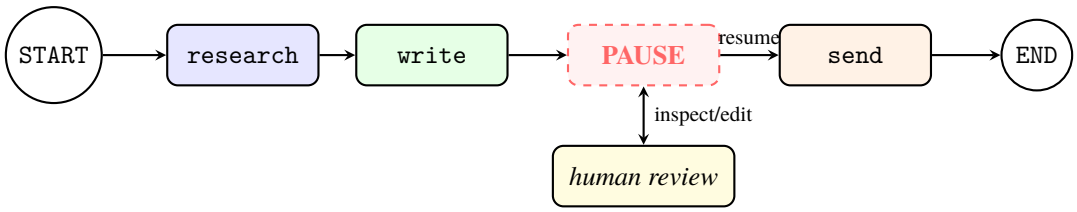


Figure 10.2: Execution flow with a breakpoint before the `send_email` node. The graph pauses, the human inspects (and optionally edits) the state, and then execution resumes.

10.3.1 Pattern 1: Approve or Reject

The simplest pattern: the agent proposes an action, and the human either approves it (execution continues) or rejects it (execution terminates or takes an alternative path).

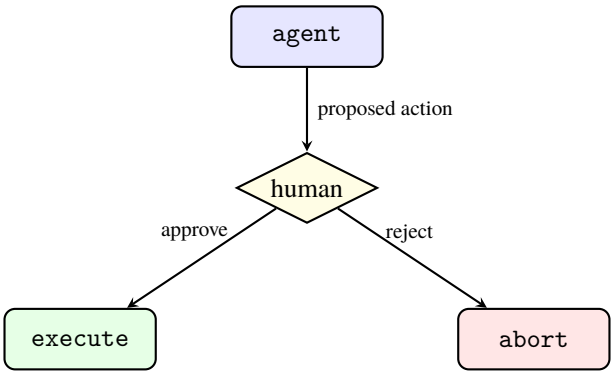


Figure 10.3: Approve-or-reject pattern.

10.3.2 Pattern 2: Edit Before Continuing

The human modifies the agent's proposed action or the state before execution continues. For example, the human might correct the recipient of an email or adjust a parameter in a tool call.

Listing 10.3: Editing state before resuming execution

```
config = {"configurable": {"thread_id": "edit-demo"}}

# Run until breakpoint
app.invoke(
    {"messages": [HumanMessage(content="Email the report")]},
    config=config,
)

# Get the current state
state = app.get_state(config)

# Human edits the last message (e.g., fix the recipient)
from langchain_core.messages import AIMessage

corrected = AIMessage(
    content="Sending report to correct-team@company.com",
    id=state.values["messages"][-1].id, # same ID = replace
```

```
)

# Update the state with the correction
app.update_state(config, {"messages": [corrected]})

# Resume with the corrected state
final = app.invoke(None, config=config)
```

The `update_state` method applies the update through the same reducer logic used by nodes. Because the corrected message has the same `id` as the original, the `add_messages` reducer replaces it rather than appending a duplicate.

10.3.3 Pattern 3: Provide Additional Input

Instead of editing the existing state, the human provides new information that the agent did not have. This is useful when the agent reaches a point where it lacks the knowledge or authority to proceed.

Listing 10.4: Human providing additional input at a breakpoint

```
# After the breakpoint, inject a human message
app.update_state(
    config,
    {"messages": [HumanMessage(
        content="Use the budget code PROJ-2024-Q3 for this."
    )]},
)

# Resume --- the agent now has the budget code
final = app.invoke(None, config=config)
```

10.3.4 Pattern 4: Redirect to a Different Node

The most powerful pattern: the human overrides the graph's routing decision and forces execution to a different node. The `update_state` method accepts an optional `as_node` parameter that specifies which node the update should appear to come from, effectively redirecting the next step.

Listing 10.5: Redirecting execution to a different node

```
# The agent wanted to call send_email, but the human
# decides more research is needed first.
app.update_state(
    config,
    {"messages": [HumanMessage(
        content="Wait --- double-check the pricing first."
    )]},
    as_node="write", # pretend this came from 'write'
)

# Now the graph will route to whatever follows 'write'
# instead of proceeding to 'send_email'
```

```
final = app.invoke(None, config=config)
```

Figure 10.4 summarises all four patterns.

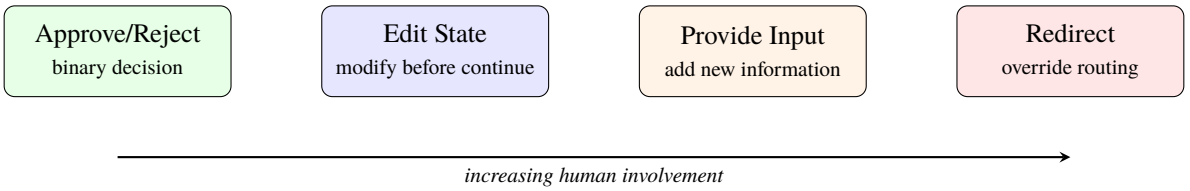


Figure 10.4: The four human-in-the-loop patterns, ordered by the degree of human involvement.

10.4 Time Travel

Breakpoints allow intervention at the *current* point of execution. **Time travel** extends this capability to *any previous point* in the execution history.

Because a checkpointer saves the state after every node execution, the full history of an agent's execution is available as a sequence of snapshots. Time travel allows developers to:

1. **Replay** from an earlier state, re-executing the same path.
2. **Branch** from an earlier state, choosing a different path.
3. **Debug** by stepping through states to locate the point where a mistake was introduced.

10.4.1 Accessing the State History

Listing 10.6: Retrieving and inspecting the full state history

```
config = {"configurable": {"thread_id": "tt-demo"}}

# Run the graph to completion
app.invoke(
    {"messages": [HumanMessage(content="Analyse this data")]},
    config=config,
)

# Retrieve the full history of checkpoints
history = list(app.get_state_history(config))

for i, snapshot in enumerate(history):
    print(f"Step {i}: next={snapshot.next}, "
          f"msgs={len(snapshot.values['messages'])}")
# Step 0: next=('research',), msgs=1
# Step 1: next=('write',), msgs=2
# Step 2: next=('send',), msgs=3
# Step 3: next=(),         msgs=4
```

10.4.2 Branching from a Previous State

To branch, we select a snapshot from the history and resume execution with a modified state and the snapshot's configuration.

Listing 10.7: Branching from an earlier checkpoint

```
# Find the state right after research (step 1)
target = history[2] # history is reverse chronological
branch_config = target.config

# Modify the state at that point
app.update_state(
    branch_config,
    {"messages": [HumanMessage(
        content="Focus on the European market instead."
    )]},
)

# Resume from the branch point
branched_result = app.invoke(None, branch_config)
```

Figure 10.5 illustrates how branching creates an alternative execution path from a past state.

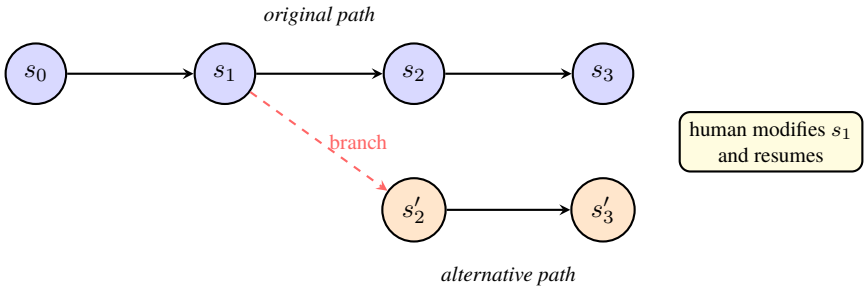


Figure 10.5: Time travel: branching from an earlier checkpoint. The original execution path continues undisturbed while the branch explores an alternative trajectory from state s_1.

10.5 Streaming

In production, users and operators should not have to wait until an agent finishes executing to see what it is doing. **Streaming** provides real-time visibility into the agent's progress by emitting events as each node executes, each token is generated, or each state update occurs.

LangGraph supports several streaming modes that can be used individually or combined.

10.5.1 Streaming Modes

10.5.2 Streaming Node Updates

Listing 10.8: Streaming state updates as nodes execute

```
config = {"configurable": {"thread_id": "stream-1"}}
```

Table 10.1: LangGraph streaming modes.

Mode	Granularity	Use case
`values`	Per-node state snapshot	Display the full state after each node completes.
`updates`	Per-node state delta	Show only what changed at each step.
`messages`	Per-token from LLM	Stream tokens to the user interface in real time.
`events`	Fine-grained event log	Detailed tracing for debugging and monitoring.

```
2
3 for chunk in app.stream(
4     {"messages": [HumanMessage(content="Analyse AI trends")]},
5     config=config,
6     stream_mode="updates",
7 ):
8     # Each chunk is {node_name: state_delta}
9     for node_name, delta in chunk.items():
10         print(f"[{node_name}] updated: {list(delta.keys())}")
11 # [research] updated: ['messages', 'research']
12 # [write]    updated: ['messages', 'draft']
13 # [send]     updated: ['messages']
```

10.5.3 Streaming LLM Tokens

For conversational agents, the most important streaming mode is `messages`, which emits individual tokens from the language model as they are generated.

Listing 10.9: Streaming tokens from the language model

```
1 for event in app.stream(
2     {"messages": [HumanMessage(content="Explain quantum computing")]},
3     config=config,
4     stream_mode="messages",
5 ):
6     message, metadata = event
7     if metadata.get("langgraph_node") == "chatbot":
8         # Print tokens as they arrive
9         print(message.content, end="", flush=True)
```

10.5.4 Combining Streaming Modes

Multiple modes can be activated simultaneously by passing a list.

Listing 10.10: Using multiple streaming modes simultaneously

```
1 for mode, chunk in app.stream(
```

```
    {"messages": [HumanMessage(content="...")]},
    config=config,
    stream_mode=["updates", "messages"],
):
    if mode == "updates":
        print(f"Node completed: {list(chunk.keys())}")
    elif mode == "messages":
        msg, meta = chunk
        print(msg.content, end="")
```

Figure 10.6 shows how streaming fits into the overall execution architecture.

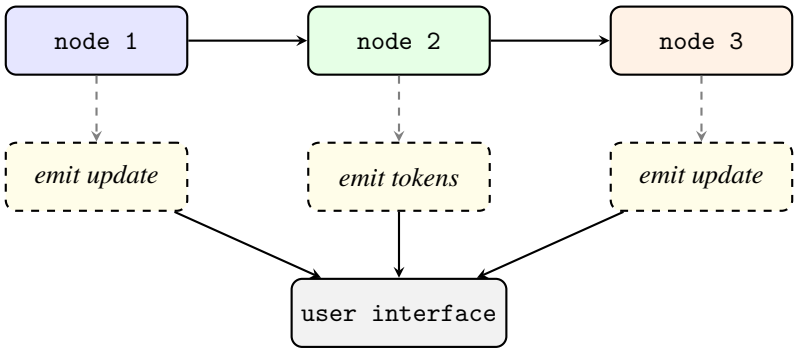


Figure 10.6: Streaming architecture. Each node emits events (state updates, tokens, or both) that are forwarded in real time to the user interface.

10.6 Fault Tolerance and Recovery

Agent systems interact with external services — APIs, databases, language model providers — that can fail at any time. A production system must handle these failures without losing the work already completed.

10.6.1 Checkpoint-Based Recovery

Because checkpointers save the state after every node, a failed execution can be resumed from the last successful checkpoint. This is the same mechanism used for breakpoints and time travel, applied now to error recovery.

Listing 10.11: Resuming execution after a transient failure

```
config = {"configurable": {"thread_id": "fault-demo"}}

try:
    result = app.invoke(
        {"messages": [HumanMessage(content="Process report")]},
        config=config,
    )
except Exception as e:
    print(f"Failed: {e}")

    # The state was checkpointed up to the failing node.
```

```
    state = app.get_state(config)
    print(f"Last successful step before: {state.next}")

    # Fix the issue (e.g., the external API is back up)
    # and resume from where we left off.
    result = app.invoke(None, config=config)
```

10.6.2 Retry Policies

For transient failures, automatic retries with exponential backoff can resolve the issue without human intervention. LangGraph integrates with retry mechanisms at the tool level (as discussed in Chapter 4) and at the node level through custom wrappers.

Listing 10.12: A node-level retry wrapper

```
import time
from functools import wraps


def retry_node(max_attempts=3, backoff=2.0):
    """Decorator that retries a node function on failure."""
    def decorator(func):
        @wraps(func)
        def wrapper(state):
            for attempt in range(max_attempts):
                try:
                    return func(state)
                except Exception as e:
                    if attempt == max_attempts - 1:
                        raise
                    wait = backoff * (2 ** attempt)
                    print(f"Retry {attempt+1}/{max_attempts} "
                          f"in {wait}s: {e}")
                    time.sleep(wait)
        return wrapper
    return decorator


@retry_node(max_attempts=3)
def call_external_api(state: AgentState) -> AgentState:
    """A node that calls a flaky external API."""
    response = external_api.call(state["query"])
    return {"result": response}
```

10.6.3 Fallback Strategies

When retries are exhausted, a **fallback** provides a degraded but acceptable result. Fallback strategies include returning a cached response, switching to a different model or API, or escalating to a human operator.

Figure 10.7 illustrates the fault tolerance hierarchy.

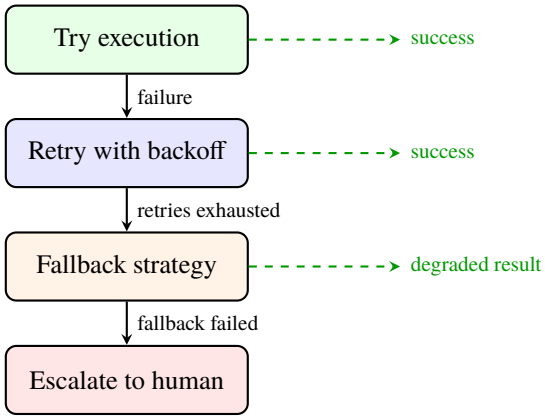


Figure 10.7: The fault tolerance hierarchy. Each level provides a progressively stronger recovery mechanism. Execution exits to the right at the first successful level.

10.7 Dynamic Breakpoints with NodeInterrupt

Compile-time breakpoints pause execution at a fixed set of nodes. For more flexible control, LangGraph provides `NodeInterrupt`, a runtime exception that a node can raise to pause execution conditionally.

This enables scenarios where the decision to pause depends on the *content* of the state rather than the *position* in the graph.

Listing 10.13: Dynamic breakpoints using `NodeInterrupt`

```
from langgraph.types import interrupt


def process_order(state: AgentState) -> AgentState:
    """Process an order, pausing for approval if large."""
    amount = state.get("order_amount", 0)

    if amount > 10_000:
        # Dynamically pause for human approval
        approval = interrupt(
            f"Order of ${amount:,.0f} requires approval. "
            "Reply 'yes' to continue or 'no' to cancel."
        )
        if approval != "yes":
            return {"messages": [
                AIMessage(content="Order cancelled.")
            ]}

    return {"messages": [
        AIMessage(content=f"Order of ${amount:,.0f} processed.")
    ]}
```

When the `interrupt` function is called, LangGraph persists the state, returns control to the caller, and

waits for a response. The caller provides the response by invoking the graph with the response value through `Command(resume=...)`.

Listing 10.14: Resuming from a dynamic interrupt with a response

```python
from langgraph.types import Command

config = {"configurable": {"thread_id": "order-1"}}

# Triggers the interrupt
app.invoke(
    {"messages": [HumanMessage(content="Place order")],
     "order_amount": 15_000},
    config=config,
)

# Human reviews and approves
final = app.invoke(
    Command(resume="yes"),
    config=config,
)
print(final["messages"][-1].content)
# "Order of $15,000 processed."
```

10.8 Orchestration in Production

The primitives introduced in this chapter combine to form a comprehensive orchestration toolkit. Table 10.2 maps common production requirements to the appropriate mechanisms.

Table 10.2: Mapping production requirements to orchestration mechanisms.

Requirement	Mechanism	Key API
Approve risky actions	Breakpoint / interrupt	`interrupt_before`
Correct agent mistakes	State editing	`update_state`
Explore alternatives	Time travel	`get_state_history`
Real-time progress	Streaming	`stream()`
Handle failures	Checkpoint recovery	`invoke(None)`
Conditional pauses	Dynamic interrupt	`interrupt()`
Audit and compliance	Event streaming	`stream_mode="events"`

Figure 10.8 shows how these mechanisms fit together in a typical production deployment.

10.9 Summary

This chapter introduced the orchestration mechanisms that make LangGraph agents suitable for production deployment.

- **Breakpoints** pause execution at designated nodes, enabling human inspection before high-stakes actions proceed.

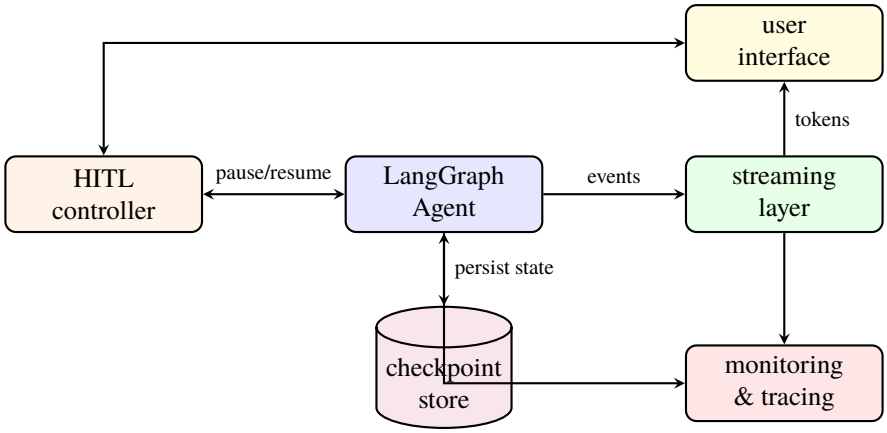


Figure 10.8: Production architecture for an orchestrated LangGraph agent. The checkpoint store enables recovery and time travel; the streaming layer provides real-time visibility; the HITL controller manages human intervention; and the monitoring system captures events for audit.

- **Human-in-the-loop patterns** — approve/reject, edit state, provide input, and redirect — give humans fine-grained control over agent behaviour at runtime.
- **Time travel** leverages the checkpoint history to replay, branch, or debug from any previous state.
- **Streaming** provides real-time visibility through multiple modes: state snapshots, deltas, token-level output, and fine-grained events.
- **Fault tolerance** is achieved through checkpoint-based recovery, automatic retries, and fallback strategies.
- **Dynamic interrupts** with `NodeInterrupt` enable conditional pauses based on state content rather than graph position.

These mechanisms are not alternatives to good agent design; they are complements. A well-designed agent with proper orchestration is more reliable, more debuggable, and more trustworthy than either component alone.

In the next part we will turn to **memory and knowledge management**, exploring how agents can maintain long-term context through vector stores, retrieval-augmented generation, and persistent knowledge bases.

Part VI

Memory and Knowledge

Chapter 11 Memory Systems and Knowledge Management

In Chapter 2 we introduced the distinction between short-term memory (the conversation history within a single session) and long-term memory (information that persists across sessions). We also showed how LangGraph checkpointers save and restore state between invocations, providing a basic form of persistence.

But real-world agents need far more than session replay. A customer service agent must recall a user's past complaints filed months ago. A research agent must draw on a corpus of thousands of documents. A personal assistant must remember that a user prefers morning meetings and dislikes sushi.

This chapter develops a comprehensive framework for agent memory and knowledge management. We begin with a taxonomy of memory types, then implement each layer using LangGraph and its ecosystem: conversation buffers, summarisation, vector-based retrieval, retrieval-augmented generation (RAG), and structured knowledge stores.

11.1 A Taxonomy of Agent Memory

Human cognition distinguishes several forms of memory — sensory, working, episodic, semantic, procedural — each serving a different function. Agent systems benefit from an analogous decomposition. Table 11.1 maps cognitive memory types to their agent counterparts.

Table 11.1: A taxonomy of agent memory, with cognitive analogues.

Memory type	Cognitive analogue	Agent implementation
Conversation buffer	Working memory	Full message history within a session.
Conversation summary	Gist memory	Compressed summary of past dialogue.
Episodic memory	Episodic memory	Retrievable records of past interactions (checkpoints, transcripts).
Semantic memory	Semantic memory	Factual knowledge stored in vector databases or knowledge graphs.
Procedural memory	Procedural memory	Learned patterns encoded in prompts, tools, or fine-tuned weights.

Figure 11.1 illustrates how these layers relate to the agent execution graph.

The remainder of this chapter implements each layer, starting from the bottom (conversation buffer) and working upward.

11.2 Conversation Buffer Memory

The most basic form of memory is the **conversation buffer**: the complete list of messages exchanged in the current session. We implemented this in Chapter 2 using the `add_messages` reducer.

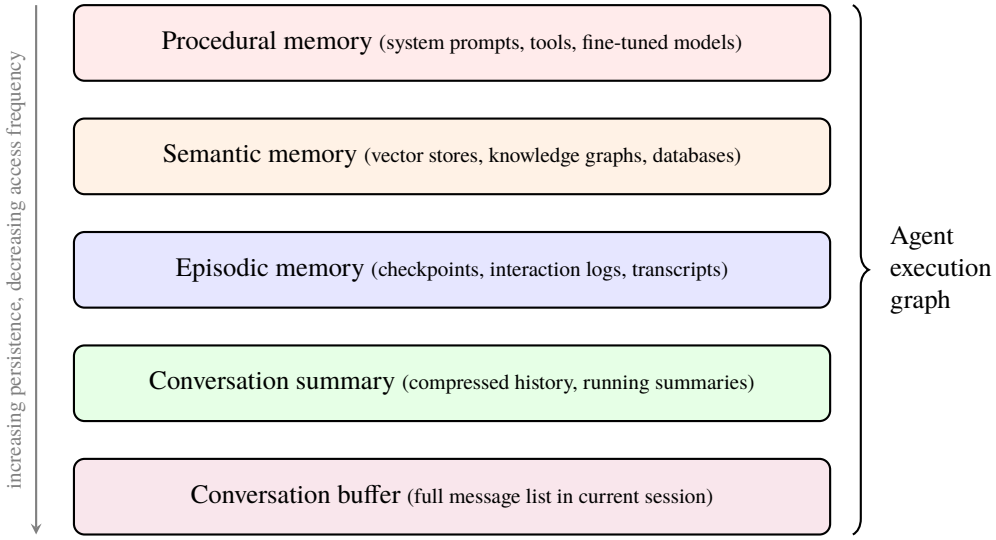


Figure 11.1: Memory layers in an agent system. Lower layers are accessed most frequently but are the most transient. Upper layers persist longer but are consulted less often.

Listing 11.1: Conversation buffer memory (recap)

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages


class AgentState(TypedDict):
    messages: Annotated[list, add_messages]
```

The buffer preserves complete fidelity: every message is available for the model to reference. However, this fidelity comes at a cost.

11.2.1 The Context Window Problem

Language models have finite context windows. As the conversation grows, the message list eventually exceeds the model's capacity. Even before hitting the hard limit, long contexts degrade model performance: the model pays less attention to messages in the middle of the window (the *lost-in-the-middle* effect [4]) and inference becomes slower and more expensive.

Figure 11.2 illustrates the problem.

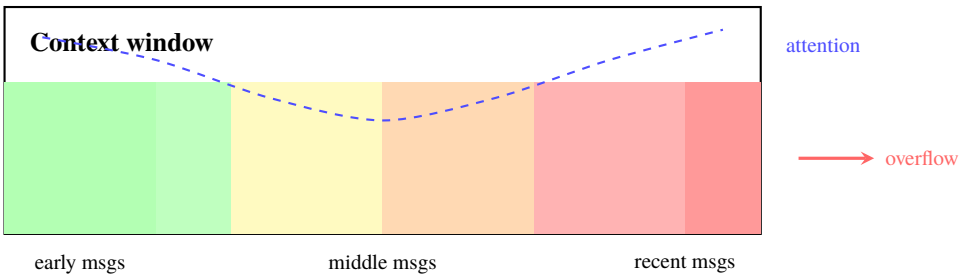


Figure 11.2: The context window problem. As messages accumulate, the context fills up. The model's attention (dashed curve) is strongest at the start and end, weakest in the middle.

The solution is to *manage* the buffer: trim old messages, compress them into summaries, or move important information to long-term storage.

11.3 Conversation Window and Trimming

The simplest buffer management strategy is **windowing**: keeping only the k most recent messages and discarding older ones.

LangGraph provides the `trim_messages` utility for this purpose.

Listing 11.2: Trimming conversation history to a fixed window

```
from langchain_core.messages import (
    trim_messages, SystemMessage, HumanMessage, AIMessage,
)

messages = [
    SystemMessage(content="You are a helpful assistant."),
    HumanMessage(content="What is 2+2?"),
    AIMessage(content="4."),
    HumanMessage(content="And 3+3?"),
    AIMessage(content="6."),
    HumanMessage(content="What about 10+10?"),
]

trimmed = trim_messages(
    messages,
    max_tokens=100,
    strategy="last",       # keep most recent
    token_counter=len,     # simplified; use model counter
    start_on="human",      # ensure window starts on human msg
    include_system=True,   # always keep the system message
)
# Result: [SystemMessage, HumanMessage("And 3+3?"),
#          AIMessage("6."), HumanMessage("What about 10+10?")]
```

11.3.1 Integrating Trimming into the Graph

Trimming can be applied inside the agent node, just before calling the language model.

Listing 11.3: Applying message trimming inside an agent node

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import trim_messages

llm = ChatOpenAI(model="gpt-4o")


def agent(state: AgentState) -> AgentState:
    """Call the LLM with a trimmed context window."""
    trimmed = trim_messages(
        state["messages"],
        max_tokens=4000,
        strategy="last",
```

```
        token_counter=llm.get_num_tokens_from_messages,
        include_system=True,
    )
    response = llm.invoke(trimmed)
    return {"messages": [response]}
```

The key design choice is that trimming affects only the *input to the model*, not the stored state. The full message history remains in the state (and in the checkpointer), so no information is permanently lost.

11.4 Conversation Summary Memory

Windowing discards old messages entirely, losing potentially important context. **Summary memory** offers a middle ground: instead of discarding old messages, it compresses them into a running summary that preserves the essential information in a fraction of the token count.

11.4.1 Architecture

The pattern works as follows:

1. After every n turns (or when the context exceeds a threshold), a **summariser** compresses the oldest messages into a summary.
2. The summary replaces the compressed messages in the state.
3. Future model calls see the summary (as a `SystemMessage`) followed by the recent messages.

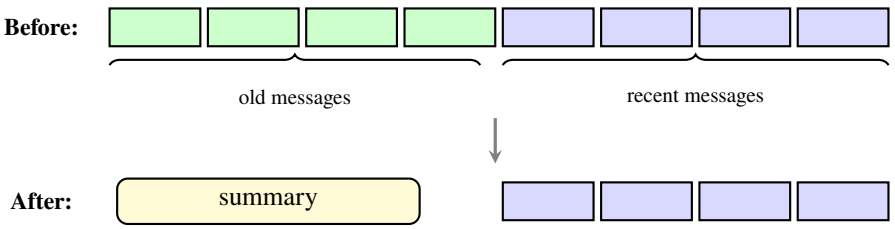


Figure 11.3: Summary memory compresses old messages into a compact summary, preserving context while freeing space in the context window.

11.4.2 Implementation

Listing 11.4 shows a complete implementation.

Listing 11.4: Conversation summary memory in LangGraph

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, START, END
from langchain_openai import ChatOpenAI
from langchain_core.messages import (
    SystemMessage, HumanMessage, AIMessage,
    RemoveMessage,
)

llm = ChatOpenAI(model="gpt-4o")
```

```python

SUMMARY_THRESHOLD = 10 # summarise after this many messages


class AgentState(TypedDict):
    messages: Annotated[list, add_messages]
    summary: str


def chatbot(state: AgentState) -> AgentState:
    """Call the LLM, prepending the summary if available."""
    messages = list(state["messages"])
    if state.get("summary"):
        summary_msg = SystemMessage(
            content=f"Summary of earlier conversation: "
                    f"{state['summary']}"
        )
        messages = [summary_msg] + messages

    response = llm.invoke(messages)
    return {"messages": [response]}


def should_summarise(state: AgentState):
    """Check whether we need to compress the history."""
    if len(state["messages"]) > SUMMARY_THRESHOLD:
        return "summarise"
    return "__end__"


def summarise(state: AgentState) -> AgentState:
    """Compress old messages into a summary."""
    # Keep the last 4 messages; summarise everything else
    old_messages = state["messages"][:-4]
    recent_messages = state["messages"][-4:]

    old_text = "\n".join(
        f"{m.__class__.__name__}: {m.content}"
        for m in old_messages
    )

    prompt = (
        f"Summarise this conversation concisely, preserving "
        f"key facts, decisions, and user preferences:\n\n"
        f"{old_text}"
    )
    if state.get("summary"):
        prompt = (
```

```
            f"Existing summary: {state['summary']}\n\n"
            f"New messages to incorporate:\n{old_text}\n\n"
            f"Produce an updated summary."
        )

    response = llm.invoke([HumanMessage(content=prompt)])

    # Remove the old messages from state
    deletions = [
        RemoveMessage(id=m.id) for m in old_messages
    ]

    return {
        "summary": response.content,
        "messages": deletions,
    }


graph = (
    StateGraph(AgentState)
    .add_node("chatbot", chatbot)
    .add_node("summarise", summarise)
    .add_edge(START, "chatbot")
    .add_conditional_edges("chatbot", should_summarise)
    .add_edge("summarise", END)
)

app = graph.compile()
```

The `RemoveMessage` objects, when processed by the `add_messages` reducer, delete messages with matching IDs from the state. This is how old messages are replaced by the summary.

11.5 Vector Stores and Semantic Search

Conversation summaries compress dialogue history, but agents often need access to knowledge that was never part of the conversation: documents, articles, manuals, databases. **Vector stores** provide the mechanism for storing and retrieving such knowledge based on semantic similarity.

11.5.1 How Vector Stores Work

A vector store operates in two phases:

Indexing. Documents are split into chunks, each chunk is converted into a high-dimensional vector (an *embedding*) using an embedding model, and the vectors are stored in a specialised database.

Retrieval. Given a query, the query is embedded using the same model, and the store returns the k chunks whose embeddings are most similar to the query embedding (typically measured by cosine similarity).

Figure 11.4 illustrates the indexing and retrieval pipeline.

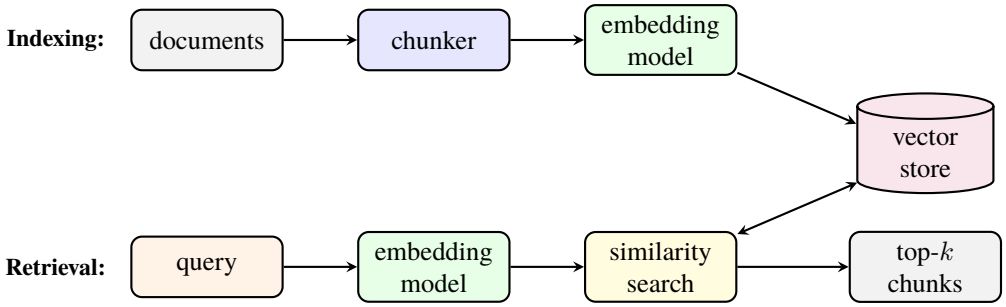


Figure 11.4: Vector store indexing and retrieval. Documents are chunked and embedded during indexing. At query time, the query is embedded and compared against stored vectors to retrieve the most relevant chunks.

11.5.2 Creating a Vector Store

LangChain provides a uniform interface across dozens of vector store backends. Listing 11.5 shows how to create an in-memory store using FAISS.

Listing 11.5: Creating and querying a FAISS vector store

```python
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_core.documents import Document

# Prepare documents
docs = [
    Document(
        page_content="LangGraph uses a graph-based architecture "
                     "for building stateful agents.",
        metadata={"source": "docs", "chapter": 1},
    ),
    Document(
        page_content="Checkpointers persist agent state across "
                     "invocations using thread IDs.",
        metadata={"source": "docs", "chapter": 2},
    ),
    Document(
        page_content="The ReAct pattern alternates between "
                     "reasoning and tool invocation.",
        metadata={"source": "docs", "chapter": 4},
    ),
]

# Create embeddings and index
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
vectorstore = FAISS.from_documents(docs, embeddings)

# Query
results = vectorstore.similarity_search(
    "How does LangGraph save state?", k=2
)
for doc in results:
```

```
    print(f"[{doc.metadata['chapter']}] {doc.page_content}")
```

11.6 Retrieval-Augmented Generation (RAG)

A vector store gives an agent the ability to *retrieve* relevant knowledge; **retrieval-augmented generation** (RAG) [3] integrates this retrieval step into the generation process, so that the language model can ground its responses in factual, up-to-date information.

11.6.1 The RAG Pipeline

The standard RAG pipeline has three stages:

1. **Retrieve**: given the user's query, fetch the top-k relevant chunks from the vector store.
2. **Augment**: inject the retrieved chunks into the model's prompt as additional context.
3. **Generate**: call the language model with the augmented prompt to produce a grounded response.

Figure 11.5 shows the pipeline as a LangGraph graph.

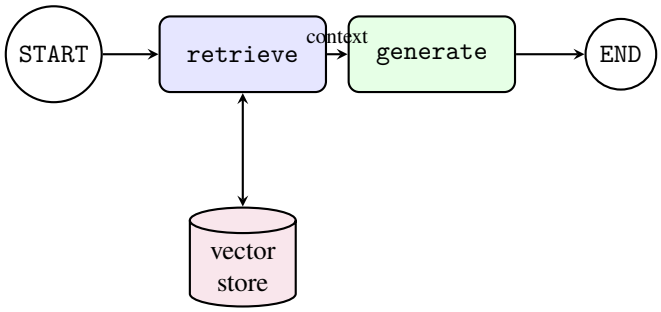


Figure 11.5: The RAG pipeline as a LangGraph graph. The retrieve node fetches relevant chunks from the vector store; the generate node produces a grounded response.

11.6.2 Implementation

Listing 11.6: A RAG agent in LangGraph

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, START, END
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_core.messages import SystemMessage, HumanMessage

llm = ChatOpenAI(model="gpt-4o")


class RAGState(TypedDict):
    messages: Annotated[list, add_messages]
    context: str


```

```python
# Assume vectorstore is already populated
vectorstore = FAISS.load_local(
    "my_index", OpenAIEmbeddings(), allow_dangerous_deserialization=True,
)
retriever = vectorstore.as_retriever(search_kwargs={"k": 4})


def retrieve(state: RAGState) -> RAGState:
    """Retrieve relevant documents for the last query."""
    query = state["messages"][-1].content
    docs = retriever.invoke(query)
    context = "\n\n".join(
        f"[Source: {d.metadata.get('source', '?')}] "
        f"{d.page_content}"
        for d in docs
    )
    return {"context": context}


def generate(state: RAGState) -> RAGState:
    """Generate a grounded response using retrieved context."""
    system = SystemMessage(content=(
        "Answer the user's question based on the following "
        "context. If the context does not contain enough "
        "information, say so. Cite your sources.\n\n"
        f"Context:\n{state['context']}"
    ))
    messages = [system] + state["messages"]
    response = llm.invoke(messages)
    return {"messages": [response]}


graph = (
    StateGraph(RAGState)
    .add_node("retrieve", retrieve)
    .add_node("generate", generate)
    .add_edge(START, "retrieve")
    .add_edge("retrieve", "generate")
    .add_edge("generate", END)
)

app = graph.compile()
```

11.6.3 Advanced RAG Patterns

The basic retrieve-then-generate pipeline can be extended with several advanced techniques. Table 11.2 summarises the most common patterns.

Table 11.2: Advanced RAG patterns.

Pattern	Description
Query rewriting	Rephrase the user's query before retrieval to improve recall (e.g. expand acronyms, resolve pronouns).
Multi-query	Generate multiple query variants, retrieve for each, and merge the results (increases recall at the cost of latency).
Reranking	After initial retrieval, use a cross-encoder model to rerank the results by relevance before passing them to the generator.
Self-RAG	The model decides at each step whether retrieval is needed, retrieves if so, and critiques the relevance of retrieved chunks before using them [1].
Corrective RAG	After generation, a verification step checks whether the response is grounded in the retrieved context; if not, the system re-retrieves or falls back [5].

11.6.4 Agentic RAG in LangGraph

The most powerful RAG variant integrates retrieval as a *tool* within a ReAct agent. Rather than always retrieving before generating, the model decides when retrieval is needed and formulates its own queries.

Listing 11.7: Agentic RAG: retrieval as a tool in a ReAct agent

```
from langchain_core.tools import tool
from langgraph.prebuilt import create_react_agent
from langgraph.checkpoint.memory import MemorySaver


@tool
def search_knowledge_base(query: str) -> str:
    """Search the internal knowledge base for information."""
    docs = retriever.invoke(query)
    return "\n\n".join(d.page_content for d in docs)


@tool
def search_web(query: str) -> str:
    """Search the web for current information."""
    # In production, call a search API
    return f"Web results for: {query}"


agent = create_react_agent(
    llm,
    tools=[search_knowledge_base, search_web],
```

```
    checkpointer=MemorySaver(),
)
```

This design lets the model choose between the knowledge base and the web depending on the nature of the question, and it can issue multiple retrieval calls to gather comprehensive information before responding.

11.7 The LangGraph Store for Long-Term Memory

Vector stores excel at semantic retrieval over large document collections, but agents also need to store and retrieve *structured* information about individual users: preferences, facts, prior decisions. LangGraph provides a built-in **Store** abstraction for this purpose.

The Store is a namespaced key–value system where each entry is a `(namespace, key, value)` triple. Namespaces allow separation of data by user, session, or domain.

11.7.1 Writing and Reading from the Store

Listing 11.8: Using the LangGraph Store for long-term user memory

```
from langgraph.store.memory import InMemoryStore

store = InMemoryStore()

# Store a user preference
store.put(
    namespace=("users", "alice"),
    key="preferences",
    value={
        "timezone": "US/Eastern",
        "meeting_preference": "morning",
        "dietary": "vegetarian",
    },
)

# Store a fact from a past conversation
store.put(
    namespace=("users", "alice"),
    key="fact_project",
    value={
        "content": "Alice is leading the Q3 migration project.",
        "confidence": 0.95,
    },
)

# Retrieve
prefs = store.get(("users", "alice"), "preferences")
print(prefs.value)
# {'timezone': 'US/Eastern', 'meeting_preference': 'morning', ...}

```

```
# Search within a namespace
results = store.search(("users", "alice"))
for item in results:
    print(f" {item.key}: {item.value}")
```

11.7.2 Accessing the Store from Nodes

When a graph is compiled with a store, every node receives access to it through the `store` parameter in its configuration.

Listing 11.9: A node that reads and writes user memory

```
from langchain_core.messages import AIMessage
from langgraph.config import get_store


def personalised_agent(state: AgentState, config) -> AgentState:
    """Use stored preferences to personalise responses."""
    store = get_store(config)
    user_id = config["configurable"].get("user_id", "default")

    # Read preferences
    prefs_item = store.get(("users", user_id), "preferences")
    prefs = prefs_item.value if prefs_item else {}

    # Use preferences in the prompt
    pref_text = ", ".join(
        f"{k}: {v}" for k, v in prefs.items()
    ) or "none recorded"

    messages = [
        SystemMessage(content=(
            f"User preferences: {pref_text}\n"
            "Personalise your response accordingly."
        )),
        *state["messages"],
    ]
    response = llm.invoke(messages)

    # Extract and store any new preferences mentioned
    # (in production, use a structured extraction step)
    if "prefer" in state["messages"][-1].content.lower():
        store.put(
            ("users", user_id),
            "last_preference_msg",
            {"content": state["messages"][-1].content},
        )

    return {"messages": [response]}
```

```
38
39
40 # Compile with the store
41 app = graph.compile(
42     checkpointer=MemorySaver(),
43     store=store,
44 )
```

11.8 Knowledge Graphs

Vector stores retrieve by semantic similarity, which works well for unstructured text but poorly for structured relationships. For queries like "Which tools does the research agent use?" or "What projects is Alice assigned to?", a **knowledge graph** provides a more natural representation.

A knowledge graph stores information as a set of *triples*:

$$(\texttt{subject}, \texttt{predicate}, \texttt{object})$$

For example:

$$(\texttt{Alice}, \texttt{leads}, \texttt{Q3 Migration})$$
$$(\texttt{Q3 Migration}, \texttt{uses}, \texttt{PostgreSQL})$$
$$(\texttt{Alice}, \texttt{reports\_to}, \texttt{Bob})$$

Figure 11.6 visualises a small knowledge graph.

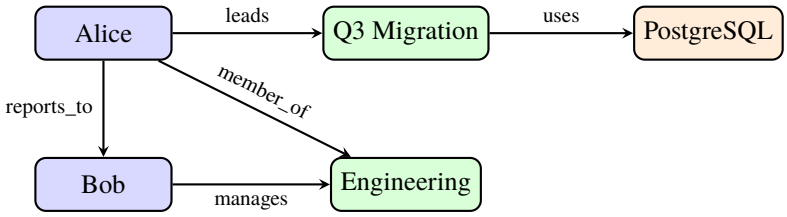


Figure 11.6: A small knowledge graph representing relationships between people, projects, and technologies.

11.8.1 Combining Vector Stores and Knowledge Graphs

In practice, the most effective memory systems combine both approaches:

- The **vector store** handles free-text retrieval over unstructured documents.
- The **knowledge graph** answers structured queries about entities and relationships.
- The **agent** decides which store to query based on the nature of the question.

This hybrid architecture is sometimes called **GraphRAG** [2].

11.9 Memory Architecture Design

Choosing the right memory architecture depends on the agent's requirements. Figure 11.7 provides a decision framework.

Table 11.3 compares the memory mechanisms covered in this chapter across several dimensions.

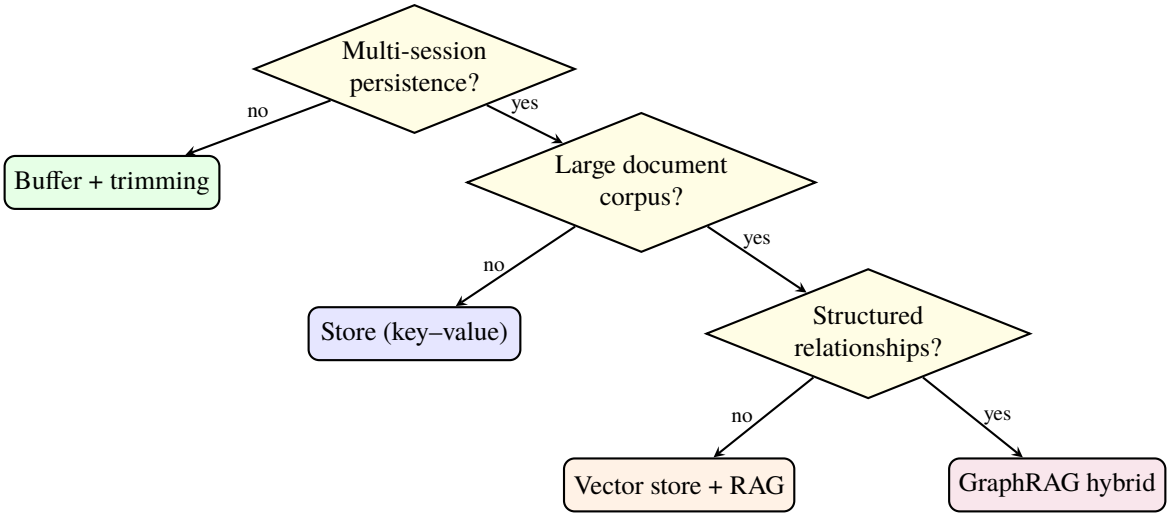


Figure 11.7: Decision framework for selecting a memory architecture.

Table 11.3: Comparison of memory mechanisms.

Mechanism	Persistence	Capacity	Retrieval	Structure	Cost
Buffer	Session	Small	Sequential	None	Minimal
Trimming	Session	Medium	Windowed	None	Minimal
Summary	Session	Medium	Compressed	None	LLM call
Checkpointer	Cross-session	Medium	By thread	Flat	Storage
Store	Permanent	Medium	By key	Key–value	Storage
Vector store	Permanent	Large	Semantic	None	Embedding
Knowledge graph	Permanent	Large	Structured	Triples	Maintenance

11.10 Summary

This chapter developed a comprehensive framework for agent memory and knowledge management.

- A **taxonomy of memory types** maps cognitive memory categories to agent implementations: buffers, summaries, episodic records, semantic knowledge, and procedural memory.
- **Conversation buffers** provide full-fidelity history within a session, but are limited by the model's context window. **Trimming** and **summary memory** manage context pressure by discarding or compressing old messages.
- **Vector stores** enable semantic retrieval over large document collections. Combined with language model generation, they form the **RAG** pipeline that grounds agent responses in factual knowledge.
- **Agentic RAG** integrates retrieval as a tool within a ReAct agent, letting the model decide when and what to retrieve.
- The LangGraph **Store** provides namespaced key–value storage for structured, per-user long-term memory such as preferences and facts.
- **Knowledge graphs** represent structured relationships between entities and complement vector stores for queries that require relational reasoning.

The choice of memory architecture depends on the agent's requirements: session scope, corpus size, query type, and maintenance budget. In practice, production agents often combine multiple mechanisms — buffer memory for the current conversation, a vector store for document retrieval, and a key–value store for user personalisation.

In the final part of this book we will look ahead to the **future of agent systems**, examining emerging paradigms, open research questions, and the trajectory of the field.

References

[1] Akari Asai et al. “Self-rag: Learning to retrieve, generate, and critique through self-reflection”. In: *The Twelfth International Conference on Learning Representations*. 2023.

[2] Darren Edge et al. “From local to global: A graph rag approach to query-focused summarization”. In: *arXiv preprint arXiv:2404.16130* (2024).

[3] Patrick Lewis et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474.

[4] Nelson F Liu et al. “Lost in the middle: How language models use long contexts”. In: *Transactions of the association for computational linguistics* 12 (2024), pp. 157–173.

[5] Shi-Qi Yan et al. “Corrective retrieval augmented generation”. In: (2024).

Part VII

The Future of Agent Systems

Chapter 12 The Road Ahead

The best way to predict the future is to invent it.
— Alan Kay

Over the course of this book we have constructed agent systems from first principles. We began with the graph abstraction that gives LangGraph its name, layered on typed state and structured messages, equipped agents with tools, taught them to plan and reflect, organised them into multi-agent teams, added human oversight and streaming, and grounded their reasoning in long-term memory and retrieved knowledge.

Each chapter added a capability. Each capability made agents more useful. But stepping back from the engineering details, a larger question comes into focus: *where is all of this going?*

This final part of the book attempts an answer — necessarily incomplete, necessarily speculative, but grounded in the technical foundations we have built. We examine the forces that are shaping the evolution of agent systems, the obstacles that stand in the way, and the milestones that will mark meaningful progress.

12.1 Where We Stand

It is useful to begin with an honest assessment of the current state of the art. Agent systems in 2025 are simultaneously impressive and fragile.

On the impressive side, a well-engineered ReAct agent with appropriate tools can browse the web, write and execute code, query databases, draft documents, and hold multi-turn conversations — all within a single interaction. Multi-agent systems can decompose complex research tasks, debate competing hypotheses, and produce polished reports. These capabilities would have been considered science fiction five years ago.

On the fragile side, the same systems hallucinate facts, lose track of long conversations, call the wrong tools, get stuck in infinite loops, and occasionally produce outputs that are confidently wrong. They require careful prompt engineering, extensive tool descriptions, and guardrails at every turn. They work well on demonstrations and benchmarks; they work less well on the messy, ambiguous, adversarial inputs of production environments.

Figure 12.1 illustrates this gap between demonstrated capability and production reliability.

The chapters in this book have been, in large part, an effort to close this gap: checkpointers for fault tolerance, breakpoints for human oversight, reflection loops for self-correction, memory systems for context preservation. But the gap remains, and understanding *why* it persists is the first step toward closing it further.

12.2 The Three Frontiers

The evolution of agent systems is being shaped by three largely independent frontiers, each advancing on its own timeline and with its own set of challenges.

12.2.1 Frontier 1: Foundation Models

The capabilities of agent systems are ultimately bounded by the capabilities of the underlying language models. Several model-level trends have direct implications for agent design.

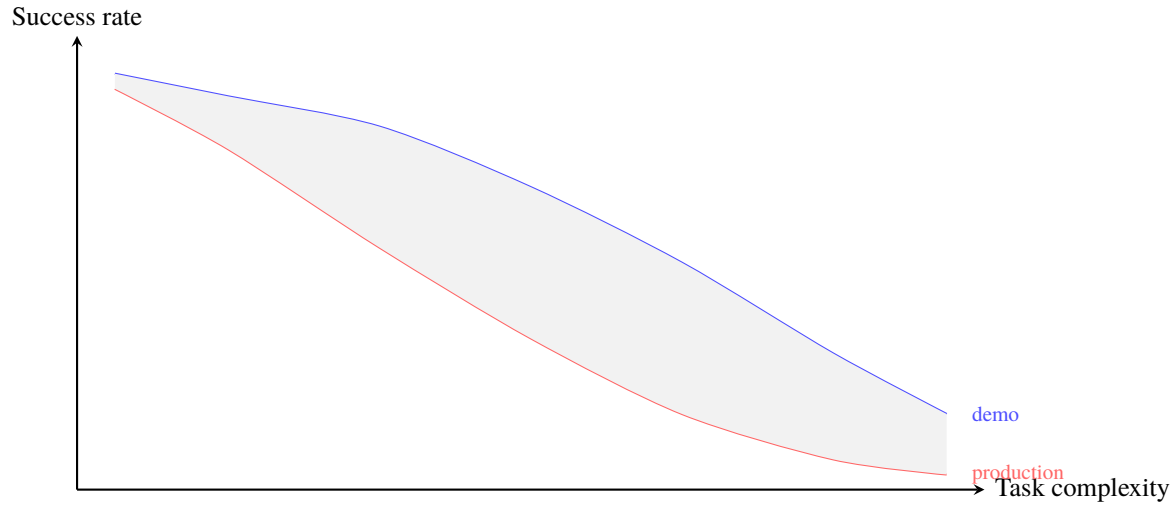


Figure 12.1: The reliability gap. Agent systems perform well on demonstrations and controlled benchmarks (blue) but degrade faster than expected in production environments with adversarial inputs, edge cases, and long horizons (red). Closing this gap is the central engineering challenge.

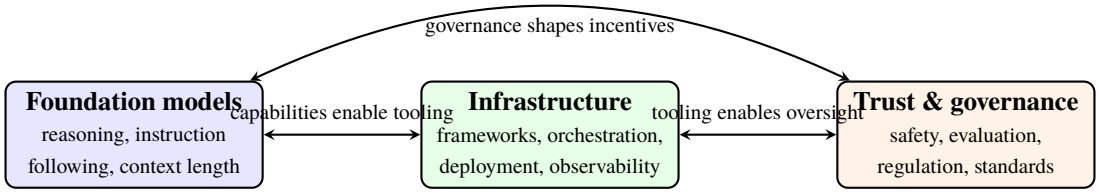


Figure 12.2: The three frontiers shaping the evolution of agent systems. Progress on any one frontier creates both opportunities and demands for progress on the others.

12.2.1.0.1 Longer context windows. Context lengths have grown from 4 096 tokens in early GPT-3.5 to over one million tokens in recent models. Longer contexts reduce the need for aggressive conversation trimming and summarisation (Chapter 8), but they do not eliminate it: attention quality still degrades over very long sequences, and cost scales linearly with context length.

12.2.1.0.2 Improved instruction following. Modern models are better at following complex, multi-part instructions. This directly improves supervisor agents (Chapter 6), which rely on the model parsing structured routing instructions, and planning agents (Chapter 5), which require the model to produce well-formed plans.

12.2.1.0.3 Native tool use. Early tool-calling implementations required careful prompt engineering to coerce models into producing structured output. Newer models have tool use trained into the weights, producing more reliable tool calls with fewer formatting errors. This trend will eventually reduce the need for the defensive parsing and retry logic that production agents currently require.

12.2.1.0.4 Reasoning models. A new class of models — sometimes called "reasoning" or "thinking" models — performs extended internal computation before producing an answer. These models internalise some of the chain-of-thought and tree-of-thought reasoning that agent systems currently implement externally (Chapter 5). The boundary between what the model does internally and what the agent orchestrates externally is shifting, and agent architectures will need to adapt.

12.2.2 Frontier 2: Infrastructure

The gap between a research prototype and a production system is primarily an *infrastructure* gap. Several infrastructure trends are closing it.

12.2.2.0.1 Standardised frameworks. LangGraph, the focus of this book, represents one approach to standardising agent construction. As frameworks mature, they absorb recurring patterns (ReAct loops, RAG pipelines, supervisor routing) into reusable primitives, reducing the engineering effort per agent.

12.2.2.0.2 Observability and tracing. Production agents generate complex execution traces spanning multiple LLM calls, tool invocations, and state transitions. Observability platforms (such as LangSmith) that capture, visualise, and analyse these traces are essential for debugging, performance tuning, and compliance.

12.2.2.0.3 Deployment platforms. Running an agent in a notebook is very different from running it as a scalable, fault-tolerant service. Deployment platforms that handle scaling, load balancing, persistent storage, and authentication are emerging to bridge this gap.

12.2.2.0.4 Interoperability protocols. As agents proliferate, the need for standardised communication between agents (and between agents and tools) grows. Emerging protocols aim to define how agents discover, authenticate with, and invoke external capabilities in a vendor-neutral way.

12.2.3 Frontier 3: Trust and Governance

The most capable agent in the world is useless if no one trusts it. The trust frontier encompasses evaluation, safety, regulation, and the social structures that determine whether agent systems are adopted.

12.2.3.0.1 Evaluation standards. The field lacks universally accepted benchmarks for agent reliability. Existing benchmarks (SWE-bench, WebArena, GAIA) measure narrow capabilities; a comprehensive evaluation standard that covers safety, robustness, and long-horizon performance does not yet exist.

12.2.3.0.2 Regulatory frameworks. Governments are beginning to regulate AI systems, with particular attention to autonomous decision-making. The EU AI Act, for example, classifies systems by risk level and imposes requirements on high-risk applications. Agent systems that take real-world actions will increasingly fall under such regulations, requiring audit trails, explainability, and human oversight — exactly the orchestration mechanisms described in Chapter 7.

12.2.3.0.3 Liability and accountability. When an agent makes a mistake — sending the wrong email, executing a bad trade, providing incorrect medical information — who is responsible? The developer? The deployer? The model provider? These questions are unresolved and will become more pressing as agents gain authority over higher-stakes decisions.

12.3 The Spectrum of Agent Autonomy

A useful framework for thinking about the future of agents is the *autonomy spectrum*, which we introduced briefly in Chapter 7. Here we develop it more fully, defining five levels of agent autonomy analogous to the SAE levels for autonomous vehicles.

Table 12.1: Five levels of agent autonomy.

Level	Name	Description	Human role
0	No autonomy	The system provides information; the human performs all actions.	Operator
1	Action suggestion	The agent proposes actions; the human approves or rejects each one.	Approver
2	Supervised execution	The agent executes routine actions autonomously; the human reviews exceptions and high-risk decisions.	Supervisor
3	Monitored autonomy	The agent operates independently within defined boundaries; the human monitors dashboards and intervenes on anomalies.	Monitor
4	Full autonomy	The agent operates without human oversight, handling all contingencies within its domain.	Absent

Figure 12.3 maps these levels to the LangGraph mechanisms that enable each one.

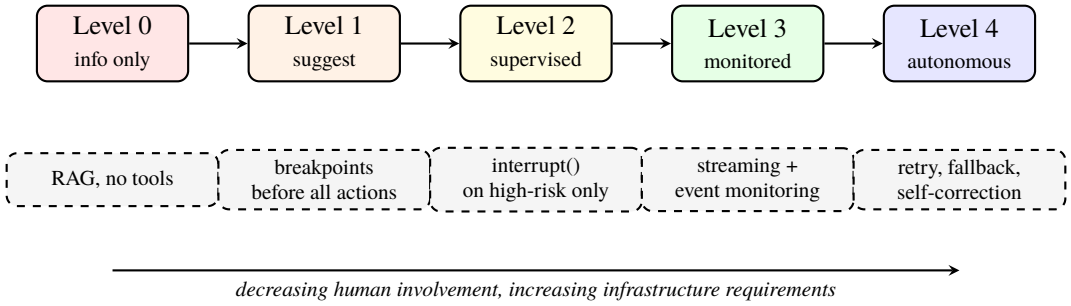


Figure 12.3: Levels of agent autonomy mapped to LangGraph mechanisms. Higher autonomy levels require more sophisticated infrastructure to compensate for the reduction in human oversight.

Most production agent systems today operate at Levels 1–2. The trajectory of the field is toward Level 3 for routine domains and, eventually, Level 4 for well-bounded tasks with strong safety guarantees. But each step up the ladder demands not just better models but better infrastructure, better evaluation, and better governance.

12.4 The Economics of Agent Systems

Technical capability is necessary but not sufficient for adoption. Agent systems must also be *economically viable*: the value they create must exceed the cost of building, running, and maintaining them.

12.4.1 Cost Structure

The cost of an agent system has four primary components.

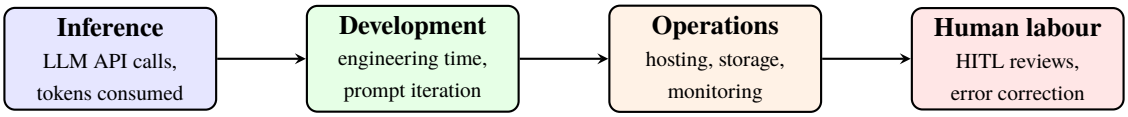


Figure 12.4: The four cost components of an agent system.

Of these, **inference cost** is the most visible and the most variable. A complex multi-agent system with planning, reflection, and retrieval may make 20–50 LLM calls per user interaction, each consuming thousands of tokens. At current API prices, this can amount to several dollars per complex task — a cost that is acceptable for high-value workflows (legal research, software engineering) but prohibitive for high-volume, low-value interactions (casual chatbots, simple FAQ).

12.4.2 Cost Optimisation Strategies

Several strategies can reduce agent costs without proportionally reducing quality.

Table 12.2: Strategies for reducing agent inference costs.

Strategy	Mechanism	Trade-off
Model routing	Use a small, cheap model for simple tasks and a large, expensive model for hard tasks.	Requires a classifier to judge task difficulty.
Prompt caching	Cache the system prompt and common prefixes to avoid reprocessing them on every call.	Effective only when prompts are reused across calls.
Fewer agent turns	Invest in better planning to reduce the number of reasoning–action cycles.	May reduce flexibility for novel tasks.
Summarisation	Compress conversation history to reduce input tokens on each call.	Lossy; may discard important context.
Distillation	Train a smaller model on the larger model's agent traces to replicate behaviour at lower cost.	Requires training infrastructure and evaluation.

12.4.3 The Value Equation

The economic viability of an agent system depends on the ratio of value created to total cost. Formally:

$$\text{ROI} = \frac{\text{Value of tasks completed} - \text{Total cost}}{\text{Total cost}}$$

The value side of this equation is task-dependent. An agent that saves a software engineer two hours of debugging per day creates far more value than one that answers a question the user could have Googled. The most successful agent deployments target *high-value, time-consuming tasks* where the alternative is expensive human labour.

12.5 Governance and Responsible Deployment

As agent systems move from research labs to production, the question of *governance* becomes inescapable. Who decides what an agent is allowed to do? How is compliance verified? What happens when things go wrong?

12.5.1 Organisational Governance

Before external regulation, organisations must govern their own agent deployments. Figure 12.5 outlines a governance framework with four layers.

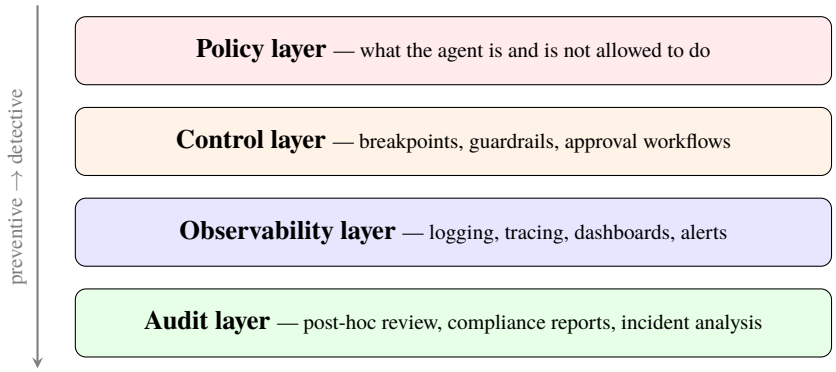


Figure 12.5: A four-layer governance framework for agent systems. Upper layers are preventive (stopping bad actions before they happen); lower layers are detective (identifying problems after the fact).

12.5.1.0.1 Policy layer. Define explicit rules for what the agent may and may not do. These rules should be encoded both in system prompts (soft constraints) and in code (hard constraints such as action allowlists and output validators).

12.5.1.0.2 Control layer. Implement the orchestration mechanisms from Chapter 7: breakpoints before high-stakes actions, dynamic interrupts for edge cases, guardrail models that screen outputs before execution.

12.5.1.0.3 Observability layer. Log every LLM call, tool invocation, state transition, and human interaction. Make these logs searchable and visualisable through dashboards that surface anomalies in real time.

12.5.1.0.4 Audit layer. Periodically review agent behaviour against the defined policies. Conduct incident post-mortems when failures occur. Use the findings to tighten policies, add guardrails, or retrain models.

12.5.2 The Regulatory Landscape

External regulation is arriving. Table 12.3 summarises the key regulatory developments that are likely to affect agent deployments.

Table 12.3: Key regulatory developments relevant to agent systems.

Regulation	Implications for agent systems
EU AI Act	Classifies AI systems by risk level. High-risk systems (e.g. those making decisions in employment, finance, health) require conformity assessments, human oversight, and detailed documentation.
Sector-specific rules	Financial services, healthcare, and legal domains impose their own requirements on automated decision-making, data handling, and accountability.
Data protection (GDPR, CCPA)	Agents that process personal data must comply with consent, purpose limitation, data minimisation, and right-to-erasure requirements. Long-term memory systems (Chapter 8) are directly affected.
Transparency requirements	Users may need to be informed when they are interacting with an AI system, and may have the right to request a human alternative.

The practical implication is clear: the orchestration and observability mechanisms from Chapter 7 are not luxuries but *regulatory necessities* for many deployment contexts. Organisations that build these capabilities early will have an advantage as regulation tightens.

12.6 Toward General-Purpose Agents

The agents in this book are *special-purpose*: each is designed for a specific task or domain, with hand-crafted prompts, curated tools, and bespoke graph structures. The long-term aspiration of the field is *general-purpose* agents that can handle arbitrary tasks with minimal task-specific engineering.

12.6.1 What General-Purpose Means

A general-purpose agent would need to:

1. **Understand** arbitrary tasks expressed in natural language.
2. **Discover** and learn to use tools it has not seen before.
3. **Plan** multi-step strategies for novel problems.
4. **Learn** from experience, improving its performance over time.
5. **Know its limits**, asking for help or refusing tasks it cannot reliably perform.

Current systems achieve items 1 and 3 with moderate reliability, item 2 with significant engineering support, and items 4 and 5 only in limited settings.

12.6.2 The Path Forward

Figure 12.6 sketches a plausible trajectory, organised around the three frontiers from Section 2.

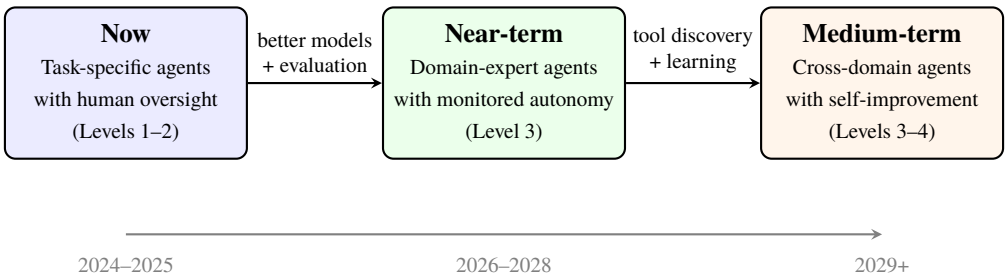


Figure 12.6: A plausible trajectory toward more general agent systems. Each era builds on the foundations of the previous one and requires advances on all three frontiers.

It is worth noting what this trajectory does *not* include: a sudden leap to artificial general intelligence. The path to more capable agents is more likely to be incremental — better models, better tools, better orchestration, better evaluation — than revolutionary. And at every step, the design principles from this book apply: transparency, controllability, composability.

12.7 Closing Thoughts

We began this book with a simple graph: two nodes, one edge, a typed state dictionary. From that foundation we built systems of remarkable sophistication — agents that plan, reflect, debate, remember, and collaborate.

The techniques in these pages will evolve. The specific APIs will change. New patterns will emerge that we cannot anticipate today. But we believe the underlying principles will endure:

Make the structure explicit. Agents built on explicit graphs with typed state are easier to understand, debug, and maintain than agents buried in opaque chains of prompts.

Keep humans in the loop. The most capable agent is the one that knows when to ask for help. Orchestration mechanisms that enable human oversight are not a concession to the limitations of current models; they are a permanent feature of responsible system design.

Build incrementally. Start with the simplest agent that could work. Measure its performance. Add complexity only where the measurements indicate a need. This discipline prevents over-engineering and keeps the system understandable.

Invest in evaluation. You cannot improve what you cannot measure. Trajectory-level evaluation, LLM-as-judge, and human review are not afterthoughts; they are the foundation on which all other improvements rest.

The field of LLM-powered agents is young, moving fast, and full of open questions. That is what makes it exciting. We hope this book has given you both the practical tools to build agents today and the conceptual framework to adapt as the field evolves tomorrow.

The future of agent systems will not be built by any single researcher, company, or framework. It will be built by the collective effort of practitioners who take these ideas, apply them to real problems, discover what works and what doesn't, and share what they learn.

We look forward to seeing what you build.